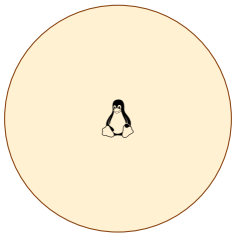
Assignment 1

Essential Development Tools



OwlTech Industries

CS 3502: Operating Systems

Department of Computer Science
College of Computing and Software Engineering
Kennesaw State University

Welcome to OwlTech Industries!

New Developer Onboarding

Congratulations on joining OwlTech Industries as a Junior Developer! As part of your onboarding process, you'll need to set up your development environment with the essential tools used by our engineering teams. This assignment will guide you through setting up a Linux virtual machine, familiarizing yourself with basic C programming, and establishing version control with Git.
At OwlTech, we believe in building a strong foundation. The tools and skills you'll learn here will be used throughout your time with us and are fundamental to systems programming.

Assignment Overview

This assignment consists of three main parts:

Part 1: Environment Setup - Establishing your Linux development environment

Part 2: C Programming Fundamentals - Writing your first C programs

Part 3: Version Control with Git - Managing your code professionally

Each part includes hands-on tasks and reflection questions to ensure you understand not just the "how" but also the "why" behind these tools.

1 Part 1: Environment Setup

1.1 Why Use a Local VM?

Understanding Sandbox Environments

A **sandbox** is an isolated testing environment that enables you to run programs or execute files without affecting the application, system, or platform on which they run. Think of it as a "safe playground" where you can experiment without breaking your main computer.

Benefits of using a Virtual Machine for this course:

- **Safety**: Mistakes won't harm your main operating system
- **Consistency**: Everyone has the same environment regardless of their host OS
- **Snapshots**: Save your VM state and restore if something goes wrong
- **Multiple OS**: Run Linux on Windows or macOS without dual-booting
- **Resource Control**: Allocate specific CPU, memory, and disk resources

1.2 Understanding Your Architecture

Before choosing a hypervisor, you need to know your system architecture:

Check Your Architecture

Windows:

- Open Command Prompt and run: `wmic cpu get architecture`
- Result meanings: 0 = x86 (32-bit), 9 = x64 (64-bit), 12 = ARM64

macOS:

- Open Terminal and run: `uname -m`
- Results: `x86_64` = Intel Mac, `arm64` = Apple Silicon (M1/M2/M3)

Linux:

- Open Terminal and run: `uname -m` or `arch`

1.3 Pick Your Hypervisor

Important Note

If you already have a Linux system installed (bare metal), you can use it directly for this course! Just ensure you have sudo access and can install packages.

1.4 Virtual Machine Requirements

Based on typical OS course needs, here are the recommended specifications:

Platform	Recommended	Alternative	Notes
Windows (x64)	VMware Workstation Player (Free)	VirtualBox	WSL2 also available
Windows (ARM)	WSL2	Hyper-V	Limited VM options
macOS (Intel)	VMware Fusion (Free for students)	VirtualBox	Parallels (paid)
macOS (Apple Silicon)	UTM	VMware Fusion	Parallels (paid)
Linux (Any)	VirtualBox	KVM/QEMU	Bare metal is fine!

Table 1: Hypervisor Selection Guide

Component	Minimum	Recommended
CPU Cores	1-2	2-4
RAM	2 GB	4 GB
Storage	8 GB	12-15 GB
Network	NAT	NAT + Host-Only

Table 2: VM Resource Allocation

1.5 Virtualization Checklist

Complete these steps in order:

- ☐ Check your system architecture (x86_64, ARM64, etc.)
- ☐ Verify virtualization is enabled in BIOS/UEFI
- ☐ Choose and install hypervisor based on your platform
- ☐ Download Ubuntu LTS ISO (or your preferred Linux distribution)
- ☐ Create new VM with recommended specifications
- ☐ Install Linux and complete initial setup
- ☐ Install Guest Additions/VMware Tools for better performance
- ☐ Take a snapshot of your clean installation

1.6 Tasks

1.6.1 Task 1.1: Virtual Machine Setup

Steps included but are not limited to:
1. **Install your hypervisor** (refer to the table above)
2. **Create a new virtual machine**:
   - Name: "CS3502-DevEnv" (or anything)
   - Version: Ubuntu (64-bit) or your chosen distribution
3. **Configure resources** (see requirements table)
4. **Install Linux**

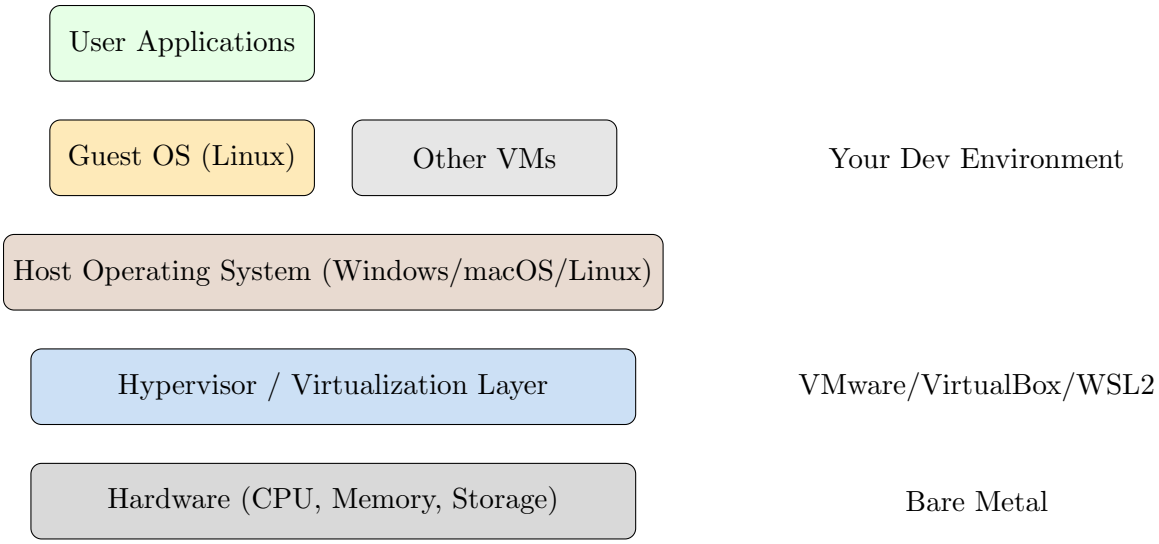


Figure 1: Virtualization Stack Architecture

1.6.2 Task 1.2: Essential Tools Installation

Open a terminal (Ctrl+Alt+T in Ubuntu) and run these commands:

```
# Update package lists (downloads latest package information)
$ sudo apt update

# Upgrade installed packages to latest versions
$ sudo apt upgrade

# Install development tools
$ sudo apt install build-essential git vim nano tree

# Verify GCC installation
$ gcc --version
```

Understanding These Commands

- `sudo` - "Super User DO" - runs commands with administrator privileges
- `apt` - Advanced Package Tool - Ubuntu's package manager
- `build-essential` - includes GCC compiler, make, and other development tools
- `git` - version control system
- `vim/nano` - text editors (vim is powerful but complex, nano is beginner-friendly)
- `tree` - displays directory structure in tree format

1.6.3 Task 1.3: Terminal Customization

Let's customize your shell environment to be more productive:

```
# Open the bashrc file in nano editor
$ nano ~/.bashrc

# Scroll to the bottom and add these lines:
# Custom prompt showing username, hostname, and current directory
export PS1="\[\033[01;32m\]\u@\h\[\033[00m\]:\[\033[01;34m\]\w\[\033[00m\]\$ "

# Helpful aliases (shortcuts for common commands)
alias ll='ls -la'           # List files with details
alias gs='git status'       # Quick git status check
alias cls='clear'           # Clear screen (like Windows)
alias ..='cd ..'            # Go up one directory

# Save and exit: Ctrl+O, Enter, Ctrl+X

# Apply changes
$ source ~/.bashrc
```

What Are Aliases?

An **alias** is a shortcut for a longer command. Instead of typing `ls -la` every time to see detailed file listings, you can just type `ll`. The shell replaces your alias with the full command.

1.7 Deliverables

Provide screenshots showing:

1. Your VM running with system information - run: `neofetch` or `uname -a && lsb_release -a`
2. GCC version output - run: `gcc --version`
3. Your customized terminal prompt showing the colors and format
4. Output of `alias` command showing your custom aliases

1.8 Reflection Questions

1. What type of hypervisor did you choose and why? What architecture is your host system?
2. Explain the difference between a Type 1 (bare metal) and Type 2 (hosted) hypervisor. Which type are you using?
3. What does the `sudo` command do and why is it important for system security?
4. Explain what happens when you run `sudo apt update` versus `sudo apt upgrade`.
5. What is the purpose of the `.bashrc` file and when is it executed?
6. Describe what the `PS1` environment variable controls and explain one part of the prompt format.

7. **Challenge**: Create your own custom alias for a command you think you'll use often. Explain what it does and why you chose it.

2 Part 2: C Programming Fundamentals

2.1 Overview

C is the language of systems programming. Unlike higher-level languages you may know (Java, Python), C gives you direct control over memory and hardware resources. This power comes with responsibility—there's no garbage collector to clean up after you!

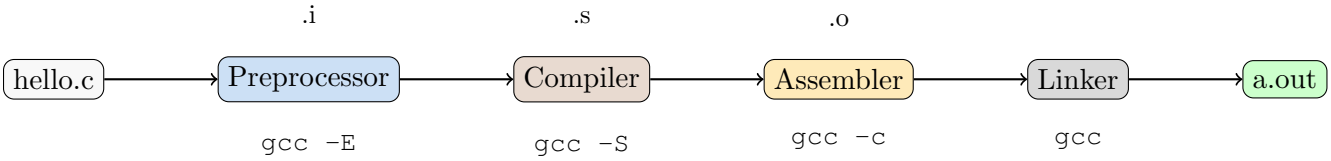


Figure 2: C Compilation Process

2.2 Tasks

2.2.1 Task 2.1: Hello, OwlTech!

First, let's create a directory for our C programs:
NOTE: You are welcome to change the directory names as you see fit.

```
# Navigate to home directory (this is a Wavy Low Line ~)
$ cd ~

# Create a directory for assignment 1
$ mkdir cs3502-a1

# Enter the directory
$ cd cs3502-a1

# Create a subdirectory for C programs
$ mkdir c-programs

# Enter the C programs directory
$ cd c-programs

# Create your first C file using nano
$ nano hello.c
```

Begin with the following program:

```c
#include <stdio.h>  // Include standard input/output library

int main() {
    // Print welcome message
    printf("Welcome to OwlTech Industries!\n");
    printf("Systems Programming Division\n");

    return 0;  // Indicate successful execution
}
```

Listing 1: hello.c

Compile and run:

```
# Compile the program
# gcc: GNU Compiler Collection
# hello.c: input source file
# -o hello: output file name (instead of default a.out)
$ gcc hello.c -o hello

# Run the program (. means current directory)
$ ./hello
```

2.2.2 Task 2.2: User Input and Variables

Create a program that interacts with the user:

```c
#include <stdio.h>
#include <string.h>

int main() {
    // Character array to store name
    // In C, strings are arrays of characters ending with '\0'
    char name[50];
    int employee_id;
    float hours_worked;

    printf("OwlTech Employee Registration\n");
    printf("=============================\n");

    printf("Enter your name: ");
    // fgets reads a line including spaces
    // stdin means "standard input" (keyboard)
    fgets(name, sizeof(name), stdin);

    // Remove newline that fgets includes
    name[strcspn(name, "\n")] = '\0';

    printf("Enter your employee ID: ");
    // & means "address of" - scanf needs to know where to store the value
    scanf("%d", &employee_id);

```

```c
    printf("Hours worked this week: ");
    scanf("%f", &hours_worked);

    printf("\nEmployee Summary:\n");
    printf("Name: %s\n", name);      // %s for string
    printf("ID: %d\n", employee_id);  // %d for integer
    printf("Hours: %.2f\n", hours_worked); // %.2f for float with 2 decimals

    return 0;
}
```

Listing 2: employee.c

Understanding Character Arrays and Strings

In C, there is no built-in string type like in Java or Python. Instead:

- Strings are arrays of characters
- They must end with a null terminator '\0'
- `char name[50]` can hold up to 49 characters plus the null terminator
- String functions like `strcspn` help manipulate these arrays

2.2.3 Task 2.3: File Operations

Create a program that works with files:

```c
#include <stdio.h>
#include <time.h>

int main() {
    FILE *logfile;  // Pointer to file structure
    char message[100];
    time_t current_time;

    // Open file for appending ("a" mode)
    // "w" would overwrite, "r" would read
    logfile = fopen("owltech.log", "a");

    // Always check if file opened successfully
    if (logfile == NULL) {
        printf("Error: Could not open log file\n");
        return 1;  // Return non-zero to indicate error
    }

    printf("Enter log message: ");
    fgets(message, sizeof(message), stdin);

    // Get current time
    time(&current_time);

    // Write to file with timestamp
```

```c
    // fprintf works like printf but writes to a file
    fprintf(logfile, "[%s] %s", ctime(&current_time), message);

    // Always close files when done
    fclose(logfile);
    printf("Log entry saved successfully!\n");

    // Display the log file contents
    printf("\n--- Current Log Contents ---\n");
    system("cat owltech.log");

    return 0;
}
```

Listing 3: logwriter.c

2.3 Deliverables

Provide screenshots showing:

1. Each program successfully compiling (show the gcc command)
2. Each program running with sample input/output
3. The contents of owltech.log after running logwriter multiple times

Note: You do NOT need to submit source code files - screenshots are sufficient!

2.4 Reflection Questions

1. What is the purpose of `#include <stdio.h>` and what happens if you omit it?
2. Explain the difference between `printf()` and `fprintf()`. When would you use each?
3. Why do we check if `fopen()` returns NULL? What could cause this to happen?
4. What is the purpose of `return 0;` versus `return 1;` in the main function?
5. In the employee.c program, why do we use `&employee_id` with scanf but not `&name`?
6. What's the difference between `fgets()` and `scanf()` for reading strings? Why did we use fgets for the name?
7. Explain what the file mode "a" means in `fopen("owltech.log", "a")`. What other modes are available?
8. What does `sizeof(name)` return and why is it useful in `fgets()`?

3 Part 3: Version Control with Git

3.1 Overview

Professional software development requires version control. Git allows you to track changes, collaborate with others, and maintain a history of your project. At OwlTech, all code must be properly versioned and documented.

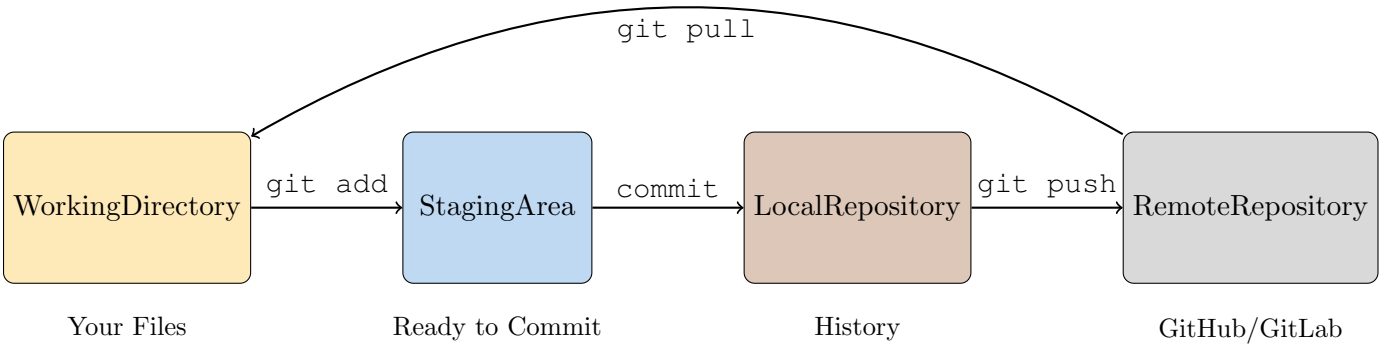


Figure 3: Git Workflow

3.2 Tasks

3.2.1 Task 3.1: Git Configuration

Set up your Git identity:

```
# Set your name (use your real name)
$ git config --global user.name "Your Name"

# Set your email (use a repo account email)
$ git config --global user.email "your.personal.email@example.com"

# Set default branch name to 'main' instead of 'master'
$ git config --global init.defaultBranch main

# Verify your configuration
$ git config --list
```

3.2.2 Task 3.2: Repository Creation

Create a repository for your assignments:

```
# Go to home directory
$ cd ~

# Create and enter project directory
$ mkdir cs3502-assignments
$ cd cs3502-assignments
```

```
# Initialize Git repository
$ git init

# Create a README file
$ echo "# CS 3502 Operating Systems Assignments" > README.md
$ echo "This repository contains my work for CS 3502" >> README.md

# Stage the README file
$ git add README.md

# Make your first commit
$ git commit -m "Initial commit: Add README"

# View the commit history
$ git log --oneline
```

3.2.3 Task 3.3: Project Structure

Organize your work:

```
# Create directory structure
$ mkdir -p assignment1/{part1,part2,part3}

# Copy your C files to the appropriate directory
$ cp ~/cs3502-a1/c-programs/*.c assignment1/part2/

# Move screenshots to part1 (if you have them)
$ mv ~/Desktop/screenshot*.png assignment1/part1/ 2>/dev/null || echo "No
    screenshots yet"

# Create a .gitignore file
$ nano .gitignore
```

Add this content to .gitignore:

```
# Compiled files
*.o
*.out
a.out
hello
employee
logwriter

# Log files
*.log

# Editor backup files
*.swp
*~
*.bak

# OS files
.DS_Store
```

```
19 Thumbs.db
```

3.2.4 Task 3.4: Commit Your Work

```
# See what Git detects as changes
$ git status

# Add all files (respecting .gitignore)
$ git add .

# Review what will be committed
$ git status

# Commit with descriptive message
$ git commit -m "Add Assignment 1: VM setup and C programs"

# View your commit history
$ git log --oneline --graph
```

3.2.5 Task 3.5: Remote Repository (Optional but Recommended)

Tips

Use a personal email address for your Git configuration and GitHub/GitLab account, NOT your KSU email. You'll want to keep access to your portfolio after graduation! If you don't have a GitHub account yet, now is a great time to create one for your professional portfolio.

If you want to push to GitHub:

```
# Generate SSH key (if needed)
$ ssh-keygen -t ed25519 -C "your.personal.email@example.com"
# Press Enter for default location, add passphrase if desired

# Display your PUBLIC key (NEVER share private key!)
$ cat ~/.ssh/id_ed25519.pub

# Copy this key to GitHub: Settings -> SSH and GPG keys -> New SSH key

# Add remote repository (replace with your GitHub repo URL)
$ git remote add origin git@github.com:yourusername/cs3502.git

# Push your code
$ git push -u origin main
```

SSH Key Security

NEVER share your private SSH key (id_ed25519) with anyone! Only utilize the public key (id_ed25519.pub). Your private key is like a password - keep it secret!
If the push fails, that's okay! Local version control still provides benefits. You can always push later when you're ready.

3.3 Deliverables

Provide screenshots showing:

1. Output of `git config --list` showing your name and email
2. Output of `git log --oneline` showing your commits
3. Output of `tree assignment1` or `ls -la assignment1/*` showing your repository structure
4. Output of `git status` showing a clean working directory

3.4 Reflection Questions

1. What is the difference between `git add` and `git commit`? Why are they separate commands?
2. Explain what the staging area (index) is and why Git has this intermediate step.
3. Why is a `.gitignore` file important? Give three examples of files that should never be committed.
4. What does "atomic commits" mean and why is it considered a best practice?
5. Explain the difference between a local repository and a remote repository.
6. What information does `git status` provide? Name at least three different states a file can be in.
7. If you generated an SSH key, explain the difference between the public and private key. Where does each one go?
8. What would happen if you accidentally committed a large binary file? How could you fix this?

Submission Guidelines

Important

Create a single PDF document containing all screenshots and answers to reflection questions. Organize your submission clearly with section headers matching the assignment structure. **DO NOT** submit source code files or log files - screenshots are sufficient!

Your submission should include:

1. **Part 1: Environment Setup**
   - VM/System information screenshot
   - GCC version screenshot
   - Customized terminal screenshot
   - Alias command output screenshot
   - Answers to all 7 reflection questions
2. **Part 2: C Programming**
   - Screenshot of each program compiling
   - Screenshot of each program running
   - Screenshot of log file contents
   - Answers to all 8 reflection questions
3. **Part 3: Version Control**
   - Git configuration screenshot
   - Git log screenshot
   - Repository structure screenshot
   - Git status screenshot
   - Answers to all 8 reflection questions

Submission Tips

- Use meaningful screenshot names IF submitting separately. You are encouraged to submit images in your document(e.g., "part1-gcc-version.png").
- Answer questions in complete sentences
- If something didn't work, explain what you tried and what error you encountered
- Make sure all text in screenshots is readable

Additional Resources

3.5 Virtualization Resources

- **VMware Workstation Player**: https://www.vmware.com/products/workstation-player.html
- **VirtualBox**: https://www.virtualbox.org
- **WSL2 Documentation**: https://docs.microsoft.com/en-us/windows/wsl/
- **UTM for Mac**: https://mac.getutm.app
- **Hyper-V**: https://learn.microsoft.com/en-us/windows-server/virtualization/hyper-v/get-started/install-hyper-v?tabs=powershell&pivots=windows

3.6 Linux Resources

- **Ubuntu Desktop Download**: https://ubuntu.com/download/desktop
- **Linux Command Reference**: https://linuxcommand.org
- **Bash Guide**: https://mywiki.wooledge.org/BashGuide

3.7 C Programming Resources

- **C Programming Tutorial**: https://www.learn-c.org
- **GCC Documentation**: https://gcc.gnu.org/onlinedocs
- **C Reference**: https://en.cppreference.com/w/c

3.8 Git Resources

- **Pro Git Book**: https://git-scm.com/book
- **GitHub Guides**: https://guides.github.com
- **Atlassian Git Tutorial**: https://www.atlassian.com/git/tutorials

A Terminology Reference

Virtual Machine	A software implementation of a computer system that executes programs like a physical machine
Hypervisor	Software that creates and manages virtual machines
Sandbox	An isolated testing environment for safe experimentation
Shell	Command-line interface for interacting with the operating system
Terminal	The program that provides access to the shell
Package Manager	System for installing, updating, and removing software (apt, yum, etc.)
Compiler	Program that translates source code into machine code
Header File	File containing C declarations and macro definitions (.h files)
Standard Library	Collection of pre-written functions available in C
String	In C, an array of characters terminated by '\0'
Pointer	A variable that stores a memory address
File Pointer	A pointer to a FILE structure used for file operations
Repository	Storage location for a project's files and revision history
Commit	A snapshot of changes in version control
Branch	An independent line of development in Git
Remote	A version of your repository hosted on a server
SSH Key	Cryptographic key pair used for secure authentication

B Common C Functions Reference

B.1 Input/Output Functions

`printf()` Print formatted output to stdout

```c
printf("Hello %s, you are %d years old\n", name, age);
```

`scanf()` Read formatted input from stdin

```c
scanf("%d", &number);  // Note the & for address
```

`fgets()` Read a line from a stream (safer than gets)

```c
fgets(buffer, sizeof(buffer), stdin);
```

fprintf() Print formatted output to a file

```
fprintf(fileptr, "Data: %d\n", value);
```

fopen() Open a file

```
FILE *fp = fopen("file.txt", "r");  // r=read, w=write, a=
    append
```

fclose() Close a file

```
fclose(fp);
```

B.2 String Functions

strlen() Get string length (not including '\0')

strcpy() Copy a string (destination must have enough space)

strcmp() Compare two strings (returns 0 if equal)

strcspn() Find length until a character is found

C Essential Linux Commands

Command	Purpose	Example
ls	List directory contents	ls -la
cd	Change directory	cd /home/user
pwd	Print working directory	pwd
mkdir	Create directory	mkdir newfolder
rm	Remove files/directories	rm file.txt
cp	Copy files	cp source.txt dest.txt
mv	Move/rename files	mv old.txt new.txt
cat	Display file contents	cat file.txt
nano	Simple text editor	nano file.txt
gcc	C compiler	gcc program.c -o program
man	Manual pages	man ls

D Basic Git Commands

Command	Purpose
`git init`	Initialize a new repository
`git add <file>`	Stage changes for commit
`git add .`	Stage all changes
`git commit -m "message"`	Commit staged changes
`git status`	Show working tree status
`git log`	Show commit history
`git log --oneline`	Show condensed commit history
`git diff`	Show unstaged changes
`git remote add origin <url>`	Add remote repository
`git push -u origin main`	Push to remote (first time)
`git pull`	Fetch and merge from remote
`git clone <url>`	Copy a repository