# CSCI-304-02 Computer Organization: Assignment 2

## Due: 11:55pm 10/20/2024, Monday

### At a Glance

| Part | Pts | What you produce | Filename |
|---|---|---|---|
| 1. VS Code Remote SSH via GenAI | 10 | Prompt/response transcript, final check | `lab2p1.doc/lab2p1.pdf` |
| 2. Binary accumulator | 25 | printf/scanf practice | `lab2p2.c`, `README` |
| 3. Simple cipher | 20 | Pointer-only string transform program | `lab2p3.c`, `README` |
| 4. Linked list | 25 | List ops (insert, find, delete, display) | `lab2p4.c`, `README` |
| 5. Using AI tools for coding | 20 | Prompts, code, analysis write-up | `lab2p5.doc/lab2p5.pdf` |

### Key Policies

**Required doc/pdf for question 1 and 5.** Include:

- Prompt and response from GenAI
- Your answers to the questions

**Required README for question 2-4.** Include:

- Total time spent in this question
- Short notes on concerns, interesting problems, discoveries, and general comments.
- Directions on how to run your solution(s), as needed.

**Must compile/run on th121 server.** Non-compiling submissions earn an automatic 0.

## Part 1 (10%) — Setup VS Code Remote SSH *with GenAI*

**Goal**

Use a GenAI assistant (e.g., ChatGPT, Gemini, Claude, perplexity, etc) to figure out how to use VS Code Remote SSH to code on the department Linux server (th121-x.wm.cs.edu). You'll learn how to ask good questions, verify answers, and document decisions—skills you'll use all semester.

**What you must achieve (milestones)**

1. Terminal login: `ssh <username>@<server-hostname>` → enter password → confirm login.
2. VS Code Remote-SSH: add the host, connect with password, open a folder.
3. Verify: create `hello.c`, compile with `gcc hello.c -o hello`, run `./hello`, show `whoami` and `hostname`.
4. Document the process using a short conversation with GenAI (planning → doing → verifying → troubleshooting if anything fails).

**Example conversation starters/prompts**

- "I'm an undergraduate learning C. I need to use *VS Code Remote-SSH with password* to code on `<username>@<server-hostname>`. Give me a short *plan* (terminal login, VS Code connection, verification with `hello.c`)."
- "I'm on **Windows 11**. Please provide Windows-specific steps; label macOS/Linux separately."
- "How do I *prove* I'm on the remote server? Provide commands and expected outputs (e.g., `whoami`, `hostname`, build/run `hello.c`)."
- "I got: `ssh: connect to host ... port 22: Connection timed out`. Give the top 3 likely causes and quick tests/fixes."

**Deliverable**

Submit a single report named `lab2p1.pdf`. Include:

- **Prompts you used:** List each prompt you sent to GenAI (copy/paste). Add one short line after each prompt explaining *why* you asked it.
- **Verification:**
  - If success, adding a **screenshot** of the terminal on the *remote* host showing: `gcc hello.c -o hello`, `./hello`, `whoami`, and `hostname`. An example screenshot is shown in Figure 1.
  - If you did *not* succeed, write 1–2 sentences explaining what failed instead of the second screenshot.
- **Problems you met & how you changed your prompt:** For each problem,
  - **Symptom:** 1–2 lines with the exact error text.
  - **Original prompt → Revised prompt:** show both.
  - **Outcome:** what fixed it (or note that it still failed).
- **Comments / thoughts (3–5 bullets):** what was confusing, what GenAI did well/poorly, what you'd try next time.

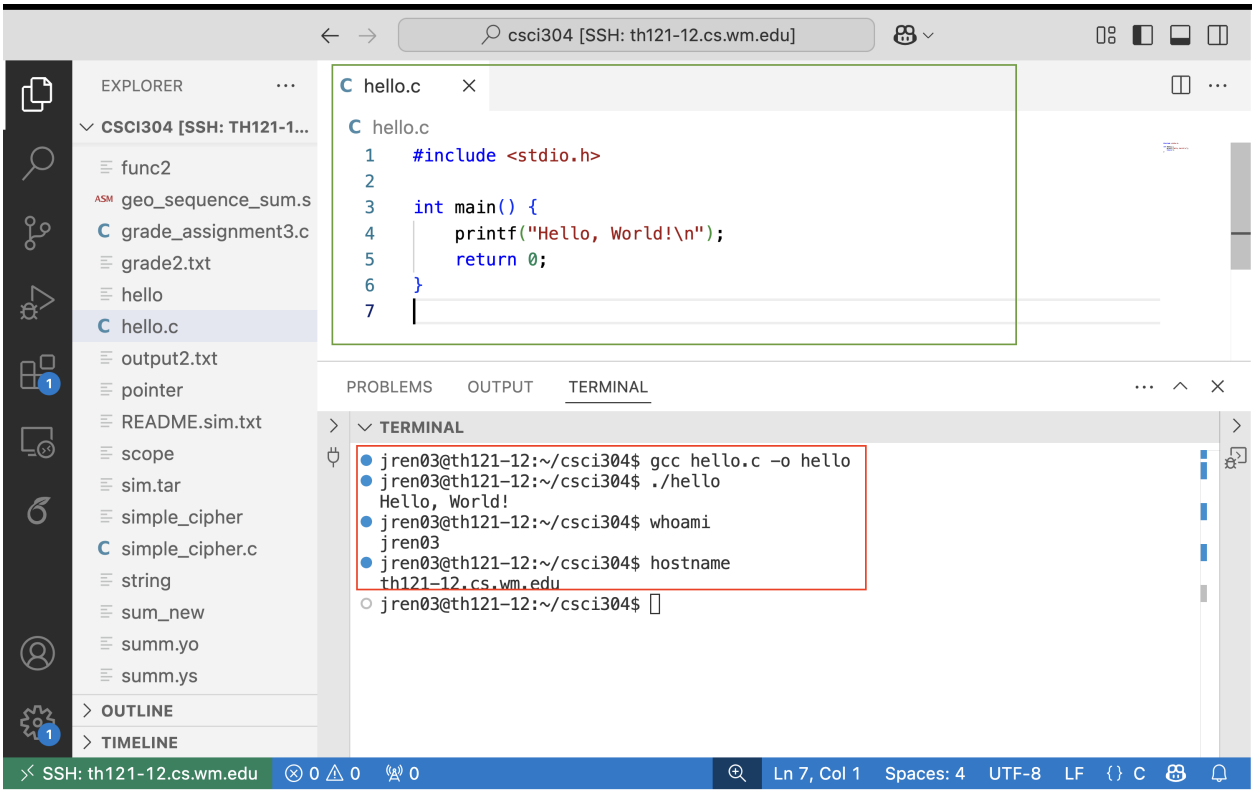


Figure 1: Example verification: VS Code set up with Remote SSH. The red box marks the remote terminal used to compile and execute code; the green box marks the editor; the left pane shows the remote server's directory tree.

## Part 2 (25%) — Binary Accumulator (standard I/O practice), (Mandatory filename: `lab2p2.c`)

**Goal.** Read two *16-bit binary strings*, compute their sum, and print:

- **RESULT (B):** 16-bit binary , zero-padded to 16 characters, derived from the final `short` accumulator.
- **RESULT (O):** 6 digits octal representation of the accumulator.
- **RESULT (H):** `0x` + 4 hex digits of the accumulator.
- **RESULT (D):** decimal representation of the accumulator.
- **MEMORY STORE:** the two bytes of the accumulator `short` printed *in the machine's actual memory order*, lowest address to highest address, each as two hex digits (case-insensitive), separated by a space.

**Computation semantics.** Parse both inputs as 16-bit values `a` and `b`. Compute the accumulator exactly as C would for `short`:

```
short acc = (short)(a + b);
```

If the mathematical sum exceeds `0xFFFF`, the accumulator shows the wrapped value (two's-complement wrap). All **RESULT** lines must be printed from `acc` (not from a wider "full" sum).

**Input (interactive via `scanf`)** Prompt *exactly*:

```
First binary is
Second binary is
```

Each input is *exactly 16 characters* of `'0'` or `'1'` (no spaces).

**Output (stdout)** Print exactly five lines, in this order:

```
RESULT (B): <16-bit-binary>
RESULT (O): <octal>
RESULT (H): 0x<4-hex-digits>    % hex letters may be upper or lower case
RESULT (D): <signed-decimal>
MEMORY STORE: <BB0> <BB1>       % bytes from (unsigned char*)&acc: p[0] p[1]
```

**Notes.**

- **Binary:** always 16 bits binary of acc.
- **Octal/Hex:** print `(unsigned short)acc`; hex case is not enforced.
- **Decimal:** print `acc` as a signed `short`.
- **Memory store:** let `p = (unsigned char *)&acc;` then print `p[0]` and `p[1]` as two-digit hex. Output may differ by machine (e.g., `0B 00` vs `00 0B`); either is correct if it reflects actual memory layout.

*Main function is provided (also available on Blackboard)*

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


/* ====== YOU MUST IMPLEMENT THESE ====== */
short get_binary_op(char *bin);
/* Convert a 16-char '0'/'1' string into a short (16-bit).
   Assumption: input is exactly 16 chars (no spaces). */

void convert_to_binary(short acc, char *bin);
/* Write the binary form of 'acc' as a 16-character string into 'bin'.
   'bin' must have space for 17 chars including '\0'. */

void print_memory(short *acc);
/* Print: MEMORY STORE: BB0 BB1
   where bytes are read from the address of *acc upward
   (lowest address first, highest address second), each as two hex digits.
   Example format: "MEMORY STORE: 0B 00"  */
/* HINT:
   unsigned char *p = (unsigned char *)acc;
   // Print p[0], then p[1], as two-digit hex:
   // printf("MEMORY STORE: %02x %02x\n", p[0], p[1]);
*/


void print_results(short acc);
/* Print RESULT lines in all modes:
   - RESULT (B): use convert_to_binary() for a 16-bit, zero-padded string
   - RESULT (O): zero-padded octal for full 16-bit range
   - RESULT (H): 0x + 4 hex digits
   - RESULT (D): signed decimal from 'acc' */


/* ====== INTERNAL: validate exactly 16 bits of '0'/'1' ====== */
static int is_valid_bin16(const char *s) {
    if (strlen(s) != 16) return 0;
    for (int i = 0; i < 16; ++i) {
        if (s[i] != '0' && s[i] != '1') return 0;
    }
    return 1;
}

/* ====== DO NOT CHANGE main() ====== */
int main(void) {
    char str_a[17], str_b[17];

    while (1) {
        printf("First␣binary␣is␣\n");
        if (scanf("%s", str_a) != 1) return 0;

        printf("Second␣binary␣is␣\n");
        if (scanf("%s", str_b) != 1) return 0;
```

```
        if (!is_valid_bin16(str_a) || !is_valid_bin16(str_b)) {
            printf("Inputs are not valid, please try again\n");
            continue;
        }
        break;
    }

    // Parse operands as 16-bit (student function)
    short a = get_binary_op(str_a);
    short b = get_binary_op(str_b);

    // Accumulator: mimic native short addition (wrap as needed)
    short acc = (short)(a + b);

    // Print all representations from the 16-bit accumulator
    print_results(acc);

    // Print actual byte layout of the accumulator in memory (machine-dependent)
    print_memory(&acc);

    return 0;
}
```

*Examples*

**Example 1 (no overflow)**

```
Input:
First binary is
0000000000000010
Second binary is
0000000000001001

Output:
RESULT (B): 0000000000001011
RESULT (O): 000013
RESULT (H): 0x000B
RESULT (D): 11
MEMORY STORE: 00 0B
```

**Example 2 (17-bit sum; accumulator wraps to 0x8000)**

```
Input:
First binary is
0111111111111111
Second binary is
0000000000000001

Output:
RESULT (B): 1000000000000000
RESULT (O): 200000
RESULT (H): 0x8000
RESULT (D): -32768
MEMORY STORE: 80 00
```

## Part 3 (20%) — SIMPLE CIPHER (Mandatory filename: `lab2p3.c`)

**Problem**

- You have a message that you want to encode. The scheme is to **reverse each alphabetic "word"** in the phrase, where words are separated by one or more *non-alphabetic* characters; keep each non-alphabetic character in its *original position*. Thus, the output string and the input string should be the same length.
- Use the underscore character _ instead of a blank in the *input* to read it as a single token. However, when you create your encoded *output*, convert each underscore to a blank (space).
- Output **all alphabetic characters as uppercase** letters.

**Input / Output**

- **Input file name:** `lab2p3in.txt`
- **Output file name:** `lab2p3out.txt`
- **Example Input → Output:**

```
War_eagle -> RAW ELGAE
Reading_records_of_variable_length? -> GNIDAER SDROCER FO ELBAIRAV HTGNEL?
Have_fun._Doing_this_lab_:) -> EVAH NUF. GNIOD SIHT BAL :)
Words%end*with^non-alpha... characters!!! -> SDROW%DNE*HTIW^NON-AHPLA... SRETCARAHC!!!
```

**Constraints**

- The length of the input message to be encoded will **not exceed 100 characters**, including the newline character.
- **Do not use array subscripts** for this lab; use **pointers** to manipulate the location of each character in the array(s).

**Sample runs**

```
$ ./lab2p3 < lab2p3in.txt
```

## Part 4 (40%) — Linked list (Mandatory filename: `lab2p4.c`)

**Structs**

- `struct Node { char *data; struct Node *next; };`
- `struct Linkedlist { struct Node *head; };`

**Program behavior**

- Read command-line arguments.
- Insert words starting with *uppercase* at the *beginning* (arrival order) via `insertAtBeginning`.
- Insert words starting with *lowercase* at the *end* (arrival order) via `insertAtTheEnd`.
- Before inserting, if `findNode` returns true, first `deleteNode` that value.
- Print the list via `displayLinkedList` (comma-separated).

**Functions**

```
void insertAtBeginning(struct Linkedlist* LL, char ele[]);
void insertAtTheEnd   (struct Linkedlist* LL, char ele[]);
void deleteNode       (struct Linkedlist* LL, char ele[]);
int  findNode         (struct Linkedlist  LL, char ele[]);
void displayLinkedList(struct Linkedlist  LL);
```

**Sample runs**

```
$ ./lab2p4
ERROR: The program must read at least an argument.
$ ./lab2p4 my name is Marwan and my car is White
The list:- White, Marwan, name, and, car
```

**Constraint** Input is split by spaces as shown.

## Part 5 (20%) — Use AI tools for coding

**Objective** Explore a mainstream AI coding tool; write prompts to implement Part 4; analyze the generated code.

**Step 1 (2%):** pick a tool (ChatGPT, Copilot, Cursor, Tabnine) and justify.

**Step 2 (10%):** craft prompts from Part 4; save prompts and AI code.

**Step 3 (8%):** compile/run; document errors and fixes (with or without AI). Include debugging prompts and whether they worked.

**Deliverable** one text file `lab2p5.txt or lab2p5.doc or lab2p5.pdf`: tool+rationale; prompts+code; results+bug analysis.

## Submission

zip all your files (.c, ,h, README, doc/pdf) as assignment2.zip and submit on Blackboard. Your code must compile/run on `th121-x.cs.wm.edu`.