电子科技大学
格拉斯哥学院
Glasgow College, UESTC

UESTC(HN) 1005
Introductory Programming
Lab Manual

S1 2025–26

Contents

3 Lab Session III **1**
3.1 Strings 1
3.2 Structures 1
3.3 Sorting Algorithm 3
3.4 Exercise 3A 4
3.5 Exercise 3B 6

Lab Session III

3.1 Strings

In C, a string is a special array of characters that ends with a null character (`\0`). This null character indicates the end of the string and distinguishes it from other arrays. For example, the word "Hello" is stored in memory as {'H', 'e', 'l', 'l', 'o', '`\0`'}.

The standard C library provides many built-in functions for operating on strings in the header file `string.h`. Some of the most commonly used string handling functions include:

`gets`: Reads a string from standard input, `stdin`, typically from user input on the keyboard.

`puts`: Outputs a string to standard output, `stdout`, typically the display screen.

`strlen`: Returns the length of a string, excluding the null terminating character (`\0`).

`strcat`: Appends (concatenates/joins) one string to the end of another.

`strncat`: Appends one string to another, but only up to a specified number of characters.

`strcmp`: Compares two strings *lexicographically* (similar to alphabetical order, but considering ASCII values). This function returns a negative value if the first string comes before the second, `0` if the strings are equal, and a positive value if the first string comes after the second.

`strcpy`: Copies the contents of one string to another.

`strncpy`: Copies up to a specified number of characters from one string to another.

If you want to explore more about these functions, refer to the lecture notes or this link.

3.2 Structures

Structures (or *structs*) are one of the many data types in C that allow us to group multiple related variables under a single identifier. These variables, referred to as "members" of a `struct`, can be of different data types, making structures incredibly useful for modelling real-world entities and creating more complex data types. In this lab session, we will explore how to define and use structures in C, unlocking their potential to simplify and organise your code.

Let us start by defining a simple structure to represent a student.

Code 3.1: `student.c`

```c
struct Student {
    int id;
    char name[50];
    float gpa;
};
```

In the `Student` structure, `id` represents the student's ID as an integer, `name` stores the student's name as a string, and `gpa` stores the student's GPA (grade point average) as a floating-point number.

Next, we implement functions to perform basic operations on a `Student` structure, such as creating an instance of it (i.e., a new student) and printing this new student's details.

Code 3.2: `testStudent.c`

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

// Define the structure
struct Student {
    int id;
    char name[50];
    float gpa;
};

// Function to create a new student
struct Student createStudent(int id, const char* name, float gpa) {
    struct Student s;
    s.id = id; // note how we assign value to a member of the structure
    strcpy(s.name, name);
    s.gpa = gpa;
    return s;
}

// Function to print student details
void printStudent(struct Student s) {
    printf("Student ID: %d\n", s.id);
    printf("Name: %s\n", s.name);
    printf("GPA: %.2f\n", s.gpa);
}

int main() {
    // Create a new student
    struct Student s1 = createStudent(1, "Glas Now", 20.8);
```

```c

    // Print the student's details
    printf("Student Details:\n");
    printStudent(s1);

    return 0;
}
```

3.3 Sorting Algorithm

This section introduces a classic sort algorithm: Bubble Sort. Bubble Sort is a simple sorting algorithm that repeatedly traverses a list of values, comparing adjacent elements, and swapping them if they are out of order. This process continues until the entire list is sorted. The algorithm gets its name because smaller elements gradually "bubble" up to the beginning of the list (or, equivalently, larger elements "sink" to the end) with each pass. Code 3.3 is an example implementation of bubble sort in C.

Code 3.3: `bubbleSort.c`

```c
// File name: bubbleSort.c
// Function to perform Bubble Sort
void bubbleSort(int arr[], int n) {
    int i, j;
    for (i = 0; i < n-1; i++) {
        // Last i elements are already in place
        for (j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                // Swap arr[j] and arr[j+1]
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}
```

Other sorting algorithms exist besides Bubble Sort, including merge sort and quicksort. Different sorting algorithms have different performance and code complexity trade-offs. The C standard library comes with an implementation of the quicksort algorithm with the function `qsort` and is provided as part of `stdlib.h`. **However, this is for your own exploration and you are NOT allowed to use this function to complete the lab exercises.**

3.4 Exercise 3A

3A - Morse Code Encoder

Problem Statement

The campus amateur radio club has built a simple Morse beacon that accepts a single line of text and transmits it as Morse code. Morse code is a telecommunications method that encodes text characters as standardised sequences of two different signal durations called dots (.) and dashes (-).

As the lead programmer of the club, your task is to write a C program that reads one line of input text from the user and prints the equivalent Morse code encoding on the screen. To keep the firmware small and robust, we need to follow a strict set of rules. The beacon signal supports only letters, digits, and spaces, and expects a specific spacing convention between letters and words. The implementation guidelines are as follows:

- **Supported input:** A-Z (case-insensitive), 0-9, and spaces. Common punctuation characters should be ignored: `, . ; : - ! ? ' "`.
- **Character separator:** Exactly one space between encoded characters.
- **Word separator:** A single slash `/` (no spaces around it). Any run of one or more spaces in the input should cause your program to generate exactly one `/`.
- **Invalid characters:** For any other character other than the above punctuations (e.g., `@ # $ % ^ & * ( ) [ ] { } \ | < > ~`), print `ERROR`.
- **Empty or space-only input:** Output a blank line.

We have provided you with a lookup table, `MORSE_AZ`, `MORSE_09` as strings in the code snippet below to use. Note the data types.

Input

A single line containing the text to be encoded.

Output

A single line that is the encoded Morse code of the input text, ERROR (invalid input), or blank (empty input or input with only spaces).

Sample Input 1

```
SOS
```

Sample Output 1

```
... --- ...
```

Sample Input 2

```
Hello World
```

Sample Output 2

```
.... . .-.. .-.. ---/.-- --- .-. .-.. -..
```

Sample Input 3

```
Hi_there!
```

Sample Output 3

```
ERROR
```

Base Code

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

// The following definitions are for your Morse code, do NOT modify
static const char *MORSE_AZ[26] = {
    ".-","-...","-.-.","-..",".","..-.","--.","....","..",".---",
    "-.-",".-..","--","-.","---",".--.","--.-",".-.","...","-",
    "..-","...-",".--","-..-","-.--","--.."
};
static const char *MORSE_09[10] = {
"-----",".----","..---","...--","....-",".....","-....","--...","---.."
    ,"----."
};

int main() {
    char inText[256];
    char outMorse[4096] = ""; // plenty for worst-case line

    // TODO

    puts(outMorse);
    return 0;
}
```

3.5 Exercise 3B

3B - Who Won the Kelvin Bicentenary Programming Competition

Problem Statement

Last year, Glasgow College held the Kelvin Bicentenary Programming Competition to celebrate the life and legacy of Lord Kelvin. As part of the challenge, each of the n participants submitted their programs to control a robot to achieve the highest possible score, in the shortest possible time.

The final ranking is based on two main metrics: the score s and the time t spent solving the task. The submission of each participant is evaluated as follows:

- s: The final score the participant's program achieved, representing how well it performed on the task (**higher means better**).
- t: The time that the participant submit their programs (**earlier means better**). The competition begins at 14:00.

The ranking criteria are three-fold:

1. The participants are first sorted by s in **descending** order.
2. If two participants have the same score s, they are then sorted by t in **ascending** order.
3. If two participants have the same score s and the same time t, they are then sorted by their name in **lexicographic** order.

Your program is required to input the name, score s, and time t of each participant. The program should then sort the participants following the ranking criteria and output the final ranking of participants by their names only.

Constraint

- For 20% of the test cases, $n \leq 100$; For 60% of the test cases, $n \leq 300$; For 100% of the test cases, $n \leq 1000$.
- $10000 \leq s \leq 200000$, 14:00 $\leq t \leq$ 17:00.
- You are **NOT allowed to use** `qsort` function to complete the lab exercises. This will be detected in the structure test. If you fail to pass the structure test, the logic test will not be executed.

Input

The input contains $n + 1$ lines.

- The first line contains an integer n which represents the number of participants.
- Each of the following n lines contains a string (the participant's name) and an integer s and a time t (representing the score and time for that participant).

Output

Print the names of the participants in the final ranking order, one name per line.

Sample Input

```
5
Alice 74,287 16:16
Bob 74,287 16:43
Charlie 13,864 15:00
David 13,864 15:00
Eve 79,874 16:15
```

Sample Output

```
Eve
Alice
Bob
Charlie
David
```

Explanation

Recall the ranking criteria: participants are first sorted by their score s in descending order. If two participants have the same score, they are sorted by their time t in ascending order. If two participants have the same score and time, they are sorted by their name.

- Alice and Bob both have a score of $74,287$, but Alice ranks higher because she took less time (as 16:16 $<$ 16:43).
- David and Charlie both have a score of $13,864$ and the same time 15:00, but Charlie ranks higher because his name ranks higher in the lexicographic order.
- Eve ranks first with the highest score of $79,874$ (so her time does not need to be considered).

Base Code

```c
#include <stdio.h>

//Define your structure first
struct {
    //TODO
} Record;

// Then your program. Remember to define helper functions
int main() {
    //TODO
}
```