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

Lab Session IV

3.1 More on Structures

Let us recap structures (`struct`) in C; they group multiple related variables under a single identifier or simply a name. You can also make a structure a member of another structure to form a nested `struct`. See below for an example of a `struct` and a nested `struct` definition:

Code 3.1: `struct_recap.c`

```c
// Define a structure type College
struct College {
    int college_id;
    char college_name[50];
};

// Define a nested structure type Student
struct Student {
    int student_id;
    char student_name[50];
    float gpa;
    struct College college;
};
```

This section will walk you through some new patterns with structures that you may use a lot in writing practical C.

3.1.1 Typedef and designated initialisation

In C, `typedef struct ...` is used to create an alias (i.e., a new name) for a structure. This simplifies the syntax for declaring variables of that structure type, helps avoid repeating `struct ...` everywhere, and allows explicit field naming. Code 3.2 presents an example of this usage.

Code 3.2: typedef_designated.c

```c
// Define a new struct and assign it to the alias
typedef struct employee {
    int id;
    char name[50];
    char position[50];
} Employee;

struct student {
  int id;
  char name[50];
  float gpa;
};

typedef struct student Student; // Alias an existing struct

int main() {
    // Designated initialisation, better readability and portability
    Student a = {.id=1, .name="Alice", .gpa=2.1};
    Employee b = {.id=5, .name="Bob", .position="Engineer"};
    return 0;
}
```

3.1.2 Pass by pointer (to modify) and `const` (read-only)

Passing a `struct` to a function **by value** copies all fields of the `struct`; changes inside the function do not affect the caller. If you wish to modify the original object, pass a `pointer`. For read-only helpers, consider `const` pointers. Let us walk through an example with the following `struct` for Student.

Code 3.3: pass_by_pointer.c

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

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

```

```c
/* The following function takes a copy and cannot do in-place
    modifications (caller won't see changes)*/
void rename_by_value(Student s, const char *new_name){
    strncpy(s.name, new_name, sizeof s.name);
    s.name[sizeof s.name - 1] = '\0';
}

// The following takes a pointer to modify in place
void rename_by_pointer(Student *s, const char *new_name){
    strncpy(s->name, new_name, sizeof s->name);
    s->name[sizeof s->name - 1] = '\0';
}

void set_gpa_value(Student s, float g){ s.gpa = g;} // modifies the
    copy, no effect on the original
void set_gpa_pointer(Student *s, float g){ s->gpa = g; } // directly
    modifies the original

void print_student(const Student *s){ // read-only
    printf("ID=%d Name=%s GPA=%.2f\n", s->id, s->name, s->gpa);
}

int main(){
    Student a = {.id=1, .name="Alice", .gpa=3.1f };
    rename_by_value(a, "Alicia"); // no effect (worked on a copy)
    set_gpa_value(a, 3.5f);
    print_student(&a); // still "Alice", with a GPA of 3.10

    rename_by_pointer(&a, "Alicia"); // modifies caller's object
    set_gpa_pointer(&a, 3.5f);
    print_student(&a); // now "Alicia", GPA becomes 3.50
    return 0;
}
```

Some tips to remember:

- Use `.` with variable values, `->` with pointers.
- Add `const` to the pointer parameters of functions that do not modify the original `struct`.
- Passing by a pointer avoids copies for larger structs (performance + clarity); However, such an in-place modification is NOT always the best practice in general.

3.1.3 Arrays of structs and simple lookups

Many programs keep a small "database" as an **array of structs**. This enables some common operations, such as searching, updating, and printing.

Code 3.4: `array_of_structs.c`

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

typedef struct {
    int id;
    char title[64];
    int year;
} Book;

int find_by_id(const Book *arr, int n, int id){
    for (int i = 0; i < n; ++i)
    if (arr[i].id == id) return i;
    return -1;
}

int update_year_by_id(Book *arr, int n, int id, int new_year){
    int k = find_by_id(arr, n, id);
    if (k < 0) return 0; // not found
    arr[k].year = new_year; // modify in place
    return 1;
}

void print_books(const Book *arr, int n){
    for (int i = 0; i < n; ++i)
    printf("%d %s (%d)\n", arr[i].id, arr[i].title, arr[i].year);
}

int main(void){
    Book shelf[3] = {
    {.id=101, .title="C Primer", .year=2016 },
    {.id=102, .title="Algorithms", .year=2009 },
    {.id=103, .title="Signals", .year=2013 }
    };
    update_year_by_id(shelf, 3, 102, 2020);
    print_books(shelf, 3);
    return 0;
}
```

Note that when you write functions to handle arrays of structures, it is almost always better to pass the length of the array `n` if possible (unless there is a sentinel value or some other information), and you should keep the look-ups small and linear to not complicate your implementation. We strongly encourage you to develop this habit. If you need to modify or update the elements inside the array, you can write pointer-taking functions to handle it.

3.1.4 Enums and Unions - A brief introduction

Beyond `struct`, C also provides other user-defined data types, such as enumerations and unions, which we will briefly walk through here.

Enumerations: `enum` is a specialised type that represents a group of constants (unchangeable). It gives a readable name to a set of integers, making it great for representing statuses, modes, weekdays, etc., and fitting naturally inside structs.

Code 3.5: `enum_brief.c`

```c
#include <stdio.h>

typedef enum {
    PENDING=0,
    PACKED=1,
    SHIPPED=2,
    DELIVERED=3
} OrderStatus;

typedef struct {
    int order_id;
    char customer[32];
    OrderStatus status;
} Order;

const char* get_status(OrderStatus s){
    static const char *N[] = {"PENDING", "PACKED", "SHIPPED", "
        DELIVERED" };
    return N[s];
}

int main(){
    Order o = {.order_id=5001, .customer="Chen", .status=PENDING };
    printf("%d %s %s\n", o.order_id, o.customer, get_status(o.status));
    o.status = SHIPPED; // switch to another named value
    printf("%d %s %s\n", o.order_id, o.customer, get_status(o.status));
    return 0;
}
```

You may ask the question of "Why bother using `enum` when we have the more general `struct`?" A valid answer is that `enum` provides self-documenting code that reads naturally without using "magic numbers" for your designs. It also helps improve control flow, such as with `switch` statements, and makes table lookups easier (arrays can be indexed by `enum`). We encourage you to think of different situations where `enum` becomes useful, and you are free to explore its usage after the labs.

Unions (Optional) : `union` in C is similar to `struct`, as a union can also store members of different data types. However, unlike `struct` whose members have their own memory, all members of a `union` share the same memory, which allows **only one active member at a time** – Only the **last** assigned member holds a valid value. The following example shows this feature of `union`.

Code 3.6: `union_brief.c`

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

union student {
  int ID;
  char name[50];
  float gpa;
};

int main() {
  union student a;

  a.ID = 1;
  printf("ID: %d\n", a.ID); // prints 'ID: 1'

  a.gpa = 3.1f;
  printf("GPA: %.2f\n", a.gpa); // prints 'GPS: 3.10'
  printf("ID: %d\n", a.ID); // The original value of a.ID is now gone

  return 0;
}
```

This may seem inconvenient, but by sharing the same memory, `union` is much more compact than `struct`, assuming both have the same types of members. You can use unions when:

- You need to store different types with the same identifier, like structures.
- You only need one active member at a time.
- **Saving memory is crucial.**

Unions are most important when memory is precious. This situation would be more common for those of you who proceed to embedded systems or other hardware with limited memory in the future. **However, for Introductory Programming, this brief introduction to unions only works as supplemental material**, and we do NOT require you to become familiar with the application of unions.

3.2 Bit Manipulation

Recall from Lab 1 and lecture that all variable values are stored as a sequence of `0`s and `1`s within computer memory, i.e., a bit representation. Consequently, bit manipulation is a powerful tool in C programming, enabling you to directly manipulate individual bits of data. This is particularly useful in scenarios where performance and memory efficiency are critical. In this section, we will explore the basic bitwise operations in C and how to use them effectively.

C provides several bitwise operators that allow you to manipulate bits directly:

- `&` (AND) - The `&` operator performs a bitwise AND operation. Each bit in the result is set to 1 if both corresponding bits in the operand are equal to 1. E.g., `1 & 1` results in `1` while `1 & 0` results in `0`.
- `|` (OR) - The `|` operator performs a bitwise OR operation. Each bit in the result is set to 1 if at least one of the corresponding bits in the operand is 1. E.g. `1 | 0` results in `1`, `0 | 0` results in `0`.
- `^` (XOR) - The `^` operator performs a bitwise XOR operation. Each bit in the result is set to 1 if **only one** of the corresponding bits in the operand is 1. E.g. `1 ^ 0` results in `1` while `1 ^ 1` results in `0`.
- `~` (NOT) - The `~` operator performs a bitwise NOT operation. It inverts all the bits of its operand. E.g., `~1` results in `0` and vice versa.
- `<<` (left shift) - The `<<` operator shifts bits to the left, filling the vacated bits with 0. E.g., `0010 << 1` results in `0100` and `0010 << 2` results in `1000` for a 4-bit value.
- `>>` (right shift) - The `>>` operator shifts bits to the right. The working principle is the same as `<<`. For unsigned types, the vacated bits are filled with 0. For signed types, the behaviour depends on the implementation (arithmetic shift).

Aside

The operands `&` and `~` are very powerful. You can use them to achieve the same function as `|` and `^`. For example, `x ^ y = ~(x & y)& ~(~x & ~y)`. Why is this true?

3.3 Exercise 4A

4A - TDPS: Multi-Robot Grid Simulator

Problme Statement

In your third year at Glasgow College, UESTC, you will experience an interesting and exciting course. The name of the course is Team Design Project Skills (TDPS), which requires you to work in a team and design an automatic robot or drone that must perform various tasks, including navigating a predefined path and communicating wirelessly. The test site is always filled with students from Glasgow College at that time of year, even at night.

This exercise will give you a brief sense of what taking TDPS will be like in the future. Your task is to simulate a small fleet of floor robots in a square arena. Each robot knows its name, position (x, y), and heading direction – North (`N`), East (`E`), South (`S`), and West (`W`). The central control console sends a sequence of single-step commands to individual robots:

- `F` — move one cell forward in the current direction
- `L` — turn left 90° (no movement)
- `R` — turn right 90° (no movement)

Commands are executed one by one in the received order. You must simulate the fleet, report collisions, and report the state of each robot. The specific rules are as follows:

- Arena is sized $N \times N$, 0-indexed: $0 \leq x, y < N$.
- Heading: `N` increases y; `E` increases x; `S` decreases y; `W` decreases x.
- When moving beyond the boundaries: ignore the move (no output).
- Into-robot forward (if the move causes one robot to take another robot's position): 1. Print "COLLISION <moving robot's name> <blocking robot's name>"; 2. Ignore the move (the positions should not change after collision).
- Turning never collides.
- All initial positions are valid, and no robots share the same initial position.
- Robot names are unique alphanumerics with no more than 15 characters.

Clarifications on edge cases:

- A robot can move into a cell vacated by another robot only if that other robot has already moved earlier in the sequence. We assume that there is NO concurrency (i.e., no parallel moves).
- The COLLISION line prints the moving robot first, then the blocking robot.
- Commands target exactly one named robot per line; if an unknown name appears, your program should ignore it.
- Coordinates are always printed as non-negative integers between $[0, N)$.

Input

The complete program with your implemented functions will receive $m + s + 1$ lines, with the **first line** containing the following:

```
N m s
```

where `N` represents the square grid size (`N` $\in [1, 50]$); `m` is the number of robots, which is an integer no larger than 100; `s` denotes the number of commands, an integer no larger than 10000.

The next `m` lines will each contain the initial coordinate (`x` and `y`), and the heading (`D`, which is one of `N`, `E`, `S`, `W`) of `robot` (the name of the robot):

```
robot x y D
```

Finally, the last `s` lines include the commands to be executed in the format of:

```
robot C
```

where `C` is the command to be executed by `robot`, and is one of {`F`, `L`, `R`} as defined above.

Output

The program should During command processing, if a forward move would step a robot outside the grid, do nothing and output nothing.

If a forward move would step onto an occupied cell, the program will output the following line and do nothing.

```
COLLISION moving_robot blocked_robot
```

where `moving_robot` and `blocked_robot` stand for the names of the robot currently in action and the robot that blocks the moving robot, respectively.

After all `s` commands have been executed, the program prints the final states for all robots, in the **same order** they were declared (the `m` lines from the input), each line should be in the same format as the `m` input lines.

```
robot x y D
```

Sample Input 1

```
5 2 6
R1 1 1 N
R2 3 1 E
R1 F
R1 R
R2 F
R1 F
R2 F
R2 L
```

Sample Output 1

```
R1 2 2 E
R2 4 1 N
```

Sample Input 2

```
3 2 1
R1 1 1 N
R2 1 2 W
R1 F
```

Sample Output 2

```
COLLISION R1 R2
R1 1 1 N
R2 1 2 W
```

Base Code

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

/* --- Constants and types (use in your TODOs and do NOT modify) --- */
#define MAXN 50
#define MAXM 100
#define MAXS 10000
#define EMPTY (-1)

typedef enum { DIR_N = 0, DIR_E = 1, DIR_S = 2, DIR_W = 3 } Direction;
```

```c
typedef struct {
    char name[16]; // robot name (null-terminated, up to 15 chars)
    int x, y; // position on grid, 0 <= x,y < N
    Direction dir; // current heading
} Robot;

/* Direction tables for convenience (do NOT modify) */
static const int DX[4] = { 0, 1, 0,-1 }; // step in x for N,E,S,W
static const int DY[4] = { 1, 0,-1, 0 }; // step in y for N,E,S,W
static const Direction LEFT_OF[4] = { DIR_W, DIR_N, DIR_E, DIR_S };
static const Direction RIGHT_OF[4] = { DIR_E, DIR_S, DIR_W, DIR_N };

/* -------- TODOs: implement the following helper functions -------- */

/* Convert a direction character ('N','E','S','W') to a Direction value
   If c is not one of these, return DIR_N by default. */
static Direction dir_from_char(char c){
    /* TODO */
}

/* Convert a Direction value to the corresponding character ('N','E','S
    ','W'). */
static char char_from_dir(Direction d){
    /* TODO */
}

/* Rotate the robot 90 degrees to the left (counterclockwise). */
void turn_left(Robot *r){
    /* TODO */
}

/* Rotate the robot 90 degrees to the right (clockwise). */
void turn_right(Robot *r){
    /* TODO */
}

/* Compute the forward step for a given direction.
   After calling this, (*dx,*dy) should hold the change in (x,y)
   if a robot moves one cell forward while facing d. */
void forward_delta(Direction d, int *dx, int *dy){
    /* TODO */
}
```

```c
/* Return 1 if (x,y) is inside the N x N grid (0 <= x < N, 0 <= y < N),
   otherwise return 0. */
int inside(int N, int x, int y){
    /* TODO */
}

/* Find the index of the robot whose name matches ‘name‘ in rs[0..m-1].
   Return the index if found, or -1 if there is no such robot. */
int find_robot(Robot *rs, int m, const char *name){
    /* TODO */
}

/* Move R[idx] according to the defined rule.
   - occ[y][x] is either EMPTY or a robot's index.*/
void step_forward(int N, int occ[][MAXN], Robot R[], int idx){
    /* TODO */
}
```

Instruction

You only need to implement the helper functions of the robot simulator in this exercise, as in the base code. Recall how you implemented and submitted the matrix transpose function back in Lab 2. A main function is provided in a separate file `local_main.c` for your local testing. You should name your implementation `robot.c` and compile the entire project before testing. All related files are provided as a bundle for you to download.

3.4 Exercise 4B

4B - Dysania

Time limit per test: 1 second

Problem Statement

As a student dealing with Dysania, you often find it hard to get out of bed. To help kickstart your morning, you decide to tackle a math problem.

Consider an infinite binary tree with a single node numbered 1 as the root, as shown in **Figure 3.1**. The root node has two children numbered 2 and 3. For each subsequent vertex starting from vertex 2, two children are added with the smallest unused numbers. This results in a tree where every vertex has exactly two children, and the vertex numbers are arranged sequentially by layers.

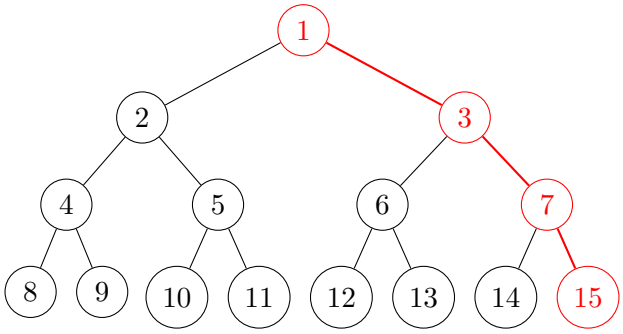

Figure 3.1: Example for Binary Tree

A *path* refers to a sequence of nodes in the tree where:

- Each node in the sequence is connected to the next by a direct edge.
- The path begins at some node v_1 and ends at v_m, passing through intermediate nodes without skipping any connections.

An example for a path from v_1 to v_{15} is marked in red in **Figure 3.1**. Assume that the path includes m nodes $v_1, \cdots, v_{\lfloor \frac{m}{2} \rfloor}, v_m$, where $\lfloor x \rfloor$ is the *floor* of x, i.e., the highest integer less than or equal to x. Then its XOR sum is defined by

$$\text{XOR sum} = v_1 \oplus \cdots \oplus v_{\lfloor \frac{m}{2} \rfloor} \oplus v_m$$

$\oplus$ denotes the bitwise XOR operator.

Your goal is to find the XOR sum of the vertex numbers along the path from the root (vertex 1) to a given vertex numbered n.

Constraints

In the C programming language, `^` denotes the bitwise XOR operator. **However, you cannot use this operator in this assignment!** At the same time, the operators `*` and `/` are **also**

forbidden in this assignment. You should consider using bit shift operations to replace these bitwise operators.

Input

The input contains one integer n that is no greater than 10^{16}:

```
n
```

Output

Output the XOR sum S of the vertex numbers along the path from the root (vertex 1) to a given vertex numbered n.

```
S
```

Sample Input 1

```
15
```

Sample Output 1

```
10
```

The XOR sum when $n = 15$ should be $1 \oplus 3 \oplus 7 \oplus 15 = 10$.

Sample Input 2

```
4093003438735155
```

Sample Output 2

```
3110384177588770
```

Base Code

```
#include <stdio.h>

long long XOR(long long a, long long b) {
    //TODO
}

int main() {
    //TODO
}
```