
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 II

## 2.1 Loop

Like many conventional and popular programming languages, the C programming language is known as a procedural programming language that supports the structured programming paradigm. From the standpoint of a programmer, what this means is that a C program is composed of three general control structures:

- **Sequential Structure**: Executing a sequence of statements one by one.
- **Selective Structure**: This allows the program to decide between alternative sequences.
- **Iterative Structure**: It repeatedly executes a code block as long as a specified condition holds, allowing iterative processing of tasks or data.

This section introduces three ways of looping, i.e., three forms of iterative structure, in a C program: the `while` loop, the `do...while` loop, and `for` loop.

### 2.1.1 The `while` loop

Generally, the simplest loop structure in C may be the `while` loop, whose syntax is:

```
while(condition){ /*Code to execute iteratively */}
```

An analogy of a `while` loop is that it can be regarded as an `if` statement with repetitive execution. In a `while` loop, `condition` is checked whether it is **true** or **false** before each iteration, and the loop will continue as long as this `condition` is **true**. An example usage of `while` loops is given in Code 2.1.

**Code 2.1:** `whileSimple.c`

```c
#include <stdio.h>

int main() {
    int i = 1;
    while(i <= 5) {
        printf("The loop has repeated %d times.\n", i);
        i++;
    }
    return 0;
}
```

```
The loop has repeated 1 times.
The loop has repeated 2 times.
The loop has repeated 3 times.
The loop has repeated 4 times.
The loop has repeated 5 times.
```

Note that you can make infinite `while` loops that never terminate if `condition` always evaluates as **true**, e.g., writing `while(1)`. To get out of a program with this behaviour, use `ctrl + C` on your keyboard. Can you think of situations where a never-ending loop becomes useful?

### 2.1.2 The `do...while` loop

The next loop structure in C is closely related to the `while` loop. The syntax is as follows:

```
do { /*Code to go through iterative execution*/} while(condition)
```

Examine Code 2.2. Is there any difference except for the slightly different structure?

**Code 2.2:** `whileLoop.c`

```c
#include <stdio.h>

int main() {
    int i = 0;
    while(i < 5) {
        printf("The loop has repeated %d times.\n", i);
        i++;
    }

    i = 0;
    do {
        printf("The loop has repeated %d times.\n", i);
        i++;
    } while(i < 5);
    return 0;
}
```

```
The loop has repeated 1 times.
The loop has repeated 2 times.
The loop has repeated 3 times.
The loop has repeated 4 times.
The loop has repeated 5 times.
The loop has repeated 0 times.
```

```
The loop has repeated 1 times.
The loop has repeated 2 times.
The loop has repeated 3 times.
The loop has repeated 4 times.
The loop has repeated 5 times.
```

The `do...while` loop checks the `condition` **after** the execution of the loop body, in contrast to that of the `while` loop. This means that code inside a `do...while` loop will at least be executed once even if the `condition` does not hold. Such a difference is demonstrated in Code 2.3.

**Code 2.3:** `whileLoopDiff.c`

```c
#include <stdio.h>

int main() {
    /*when the looping condition is not met*/
    int i = 10;
    do {
        printf("i is: %d.\n", i);
        i++;
    } while (i < 5);

    int j = 10;
    while (j < 5) {
        printf("j is: %d.\n", j); // This will not print anything
        j++;
    }
    return 0;
}
```

```
i is: 0.
```

### 2.1.3 The `for` loop

The third and final loop structure here is the `for` loop, with the code syntax:

```c
for(initialisation; condition; post-loop operation){/*Code to be executed */}
```

A `for` loop has three components:

- `initialisation`: Sets up a starting point for the loop. This is executed once at the beginning, before entering the loop, e.g., `i=0` or `i=8`.
- `condition`: This is evaluated **before** each iteration; the loop continues if true. e.g.,`i<=10`.

- `post-loop operation`: The computer executes this statement after each loop. Usually, but not always, this is to update the variable that appears in the `initialisation` (`i` in this case) after each iteration, e.g., `i++` or `i--`.

The `for` loop is generally the most commonly used loop in many programming languages because of its compact syntax, convenience to iterate over data (e.g., arrays), and high flexibility.

**Code 2.4:** `forLoop.c`

```c
#include <stdio.h>

int main() {
    /* An 'ordinary' usage of for loop */
    for(int i = 1; i <= 5; i++) {
        printf("The loop has repeated %d times.\n", i);
    }

    /* You can also do this */
    int j = 0;
    for (int i = 0; i < 5; j++) { // Notice 'j++' instead of 'i++'
        printf("i is %d, j is %d.\n", i, j);
        i++; // You can manually increase the loop control variable 'i'
    }
    return 0;
}
```

```
The loop has repeated 1 times.
The loop has repeated 2 times.
The loop has repeated 3 times.
The loop has repeated 4 times.
The loop has repeated 5 times.
i is 0, j is 0.
i is 1, j is 1.
i is 2, j is 2.
i is 3, j is 3.
i is 4, j is 4.
```

You can choose these loop structures according to your specific needs. For example:

- If the loop condition is difficult to express or requires prolonged execution, the `while` loop may be a good choice.
- If you want to guarantee **at least one execution** of the loop body regardless of the condition, the `do...while` loop is likely the correct choice.
- If the number of iterations is **known** before entering the loop, you can use `for` loops.

## 2.2 Random Number

> *Anyone who attempts to generate random numbers by deterministic means is, of course, living in a state of sin.*
>
> *John von Neumann*

When people refer to a "random number" in the context of a computer (a deterministic system) and related engineering fields, they imply "pseudorandom number." This is because we start with a single value known as the **random seed** and perform a series of **deterministic** algorithmic operations on it to generate "randomness". The problem is that we cannot get an actual random number starting with a predefined seed — every time we restart using the same random seed, we will get the same sequence of pseudorandom numbers.

In this exercise, we investigate pseudorandom numbers. Code 2.5 provides a program to print a pseudorandom number, sampled from a uniform distribution ranging from 0 to 1. Try it on your machine and observe the result!

**Code 2.5:** `rand.c`

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

float getRand() {
    return rand() / (RAND_MAX + 1.0);
}

int main() {
    srand(time(NULL));
    float number = getRand();
    printf("Random number: %f\n", number);
    return 0;
}
```

## 2.3 Arrays

An *array* is composed of a series of elements of one data type stored *contiguously* in memory, i.e., sequentially without any gaps. A variable *declaration* such as `float candy[365]` tells the compiler how many elements the array contains and the type for these elements. An element of an array can be accessed by its subscript number, also called its *index*. The index starts at 0, i.e., `candy[0]` is the first element of the `candy` array, and `candy[364]` is the 365th (the last) element.

### 2.3.1 Initialisation

We can initialise an array with a comma-separated list of values enclosed in braces, as in Code 2.6.

**Code 2.6:** months1.c

```c
#include <stdio.h>
#define MONTHS 12

int main() {
    int days[MONTHS]
        = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
    for(int i = 0; i < MONTHS; i++)
        printf("Month %2d have %2d days.\n", i + 1, days[i]);

    return 0;
}
```

Note that in this example, we use a preprocessor directive, `#define` and create a constant `MONTHS` to represent the array size. This is a common and recommended practice. For example, if the world switched to a 13-month calendar, you would then just have to modify the `#define` statement to update the arrays that should have this size.

Sometimes you might use an array that is intended to be read-only. That is, the program can only retrieve values from the array while disallowing modification of the array's values. In such cases, you can, and should, use the `const` keyword when you declare and initialise the array. Therefore, a better choice for Code 2.6 would be `const int days[MONTHS] = {31, 28, ..., 30, 31};`. This makes the program treat each element in the array as a constant. You must initialise `const` variables when they are declared because you cannot assign values to them later, i.e., the semantics of `const` prevent modification.

We can also let the compiler match the array size to the list by omitting the size from the braces, as shown in Code 2.7.

**Code 2.7:** months2.c

```c
#include <stdio.h>
#define MONTHS 12

int main() {
    const int days[]
        = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
    for(int i = 0; i < MONTHS; i++)
        printf("Month %2d have %2d days.\n", i + 1, days[i]);
    return 0;
}
```

### 2.3.2 Designated Initialisers

The C99 standard added a new capability: designated initialisers. This feature allows you to pick and choose which elements are initialised. For example, you only want to initialise the last element in an array. With traditional C initialisation syntax, you also have to initialise every element preceding the last, like Code 2.6. Since C99, you can use an index in brackets in the initialisation list to specify a particular element, as shown in Code 2.8.

**Code 2.8:** `months3.c`

```c
#include <stdio.h>
#define MONTHS 12

int main() {
    int days[MONTHS]
        = {31, [2] = 28, 31, [1] = 30, 31, 30, 31, 31, 30};
    for(int i = 0; i < MONTHS; i++)
        printf("Month %2d have %2d days.\n", i + 1, days[i]);

    return 0;
}
```

### 2.3.3 Variable Length Array

A variable-length array (VLA) is an array in which the length is determined at **runtime** rather than at **compile time**. Unlike standard arrays, which require a fixed size defined by a constant, VLAs in C can be created with sizes that depend on variable values. VLAs were introduced in the C99 standard, allowing more flexible memory use in certain situations without the need for dynamic memory allocation functions like `malloc()` and `free()`.

However, VLAs are generally discouraged in C for several reasons. First, they can lead to undefined behaviour if the requested size is too large for the memory stack, potentially causing a stack overflow and program crashes. This is because VLAs are often allocated to the stack, which has a limited space. Moreover, C11 made VLAs optional, meaning that some compilers may not support them, which impacts portability. Finally, compilers are generally better at optimising code with arrays that have explicitly defined sizes. Due to these limitations and risks, **we recommend that you use dynamic memory allocation or fixed-size arrays instead of VLAs**.

**Code 2.9:** `vla.c`

```c
// A VLA way to define a small array
int n;
scanf("%d", &n);
int array[n];
```

**Aside**

We do not recommend using VLA in the IP course. So when compiling your program, we add the compile flag `-Werror=vla` to forbid the use of VLAs.

## 2.4 Pointers

A *pointer* is a data type that stores the memory address of another variable. A pointer is a powerful tool in C, enabling efficient array handling, dynamic memory allocation, and the implementation of complex data structures. However, the pointer mechanism must be used carefully, as it allows the programmer to modify *any* value in computer memory. Most modern programming languages disallow direct manipulation of values in memory by address and prefer more structured and safe mechanisms. Be warned: *"With great power comes great responsibility."* — a simple bug in your program can result in unintentionally modifying memory outside the desired scope, causing a *segmentation fault*, i.e., access to restricted memory outside the bounds reserved for your program and termination of your program by the operating system.

### 2.4.1 The Relationship Between Arrays and Pointers

In C, the relationship between arrays and pointers is fundamental. The array identifier, i.e., its name, acts as a constant pointer to its first element, and pointer arithmetic can be applied to access all elements. Understanding this relationship is crucial for effective C programming.

Consider the example below that demonstrates the connection between arrays and pointers.

**Code 2.10:** `arrayPointer.c`

```c
#include <stdio.h>

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    int *ptr = arr; // Pointer to the first element of the array

    printf("Array elements using pointer:\n");
    for (int i = 0; i < 5; i++) {
        printf("%d ", *(ptr + i));
    }

    printf("\nArray elements using array name:\n");
    for (int i = 0; i < 5; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

In Code 2.10, `ptr` is a pointer that points to the first element of the array `arr`. Both `*(ptr + i)` and `arr[i]` can be used to access the $i-th$ element of the array.

### 2.4.2 Multidimensional Arrays

Pointers are also useful when working with multidimensional arrays. Understanding how to use pointers with multidimensional arrays can help in writing more efficient and readable code.

For example, Code 2.11 summarises operating on a two-dimensional array using pointers.

**Code 2.11:** `multiArrayPointer.c`

```c
#include <stdio.h>

#define ROWS 3
#define COLS 4

// Function to print and modify the matrix
void print_modMatrix(int (*matrix)[COLS], int rows) {
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < COLS; ++j) {
            // Modify each element to demonstrate passing by reference
            matrix[i][j] += 1; // Example modification: increment by 1
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
}

int main(int argc, char **argv) {
    int matrix[ROWS][COLS] = {
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12}};
    printf("Original Matrix elements:\n");
    // Print the original matrix
    for (int i = 0; i < ROWS; ++i) {
        for (int j = 0; j < COLS; ++j) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
    printf("\nModified Matrix elements in print_modMatrix:\n");
    // Call the function to print and modify the matrix
    print_modMatrix(matrix, ROWS);
    printf("\nMatrix elements after calling print_modMatrix:\n");
    // Print the matrix again to show that it was modified
```

```
36     for (int i = 0; i < ROWS; ++i) {
37         for (int j = 0; j < COLS; ++j) {
38             printf("%d ", matrix[i][j]);
39         }
40         printf("\n");
41     }
42     return 0;
43 }
```

Here, `print_modMatrix` is a function that takes a pointer to an array of `COLS` integers with its number of rows, increments each element, and then prints the matrix.

**Aside**

**This definition of a matrix is not suitable for this lab's exercises** because it requires using a one-dimensional array to represent a two-dimensional matrix. Multi-dimensional arrays are stored linearly in memory, i.e., as a contiguous block of elements, which means you can use a one-dimensional array to store any-dimensional data.

### 2.4.3 Functions and Pointers

Functions and pointers are deeply interconnected in C, allowing powerful ways to manipulate data and create flexible code structures. When a pointer is used with functions, it enables modifying variables directly by their addresses rather than by value. This is essential when passing data to functions that need to modify the caller's variables, as it avoids the need for return values and allows multiple values to be updated at once. By passing pointers to functions, C enables efficient memory usage for faster code execution, particularly with larger data structures.

Consider Code 2.12 for a `swap` function that exchanges the values of two variables. If we define `swap` using pointers, it can directly modify the values of the variables passed to it. The function declaration `void swap(int *a, int *b)` takes two pointers to integers. Inside the function, the values at these memory addresses are swapped using a temporary variable:

**Code 2.12:** `swap.c`

```
1  void swap(int *a, int *b) {
2      int temp = *a;
3      *a = *b;
4      *b = temp;
5  }
6
7  void swap_int(int *a, int *b) {
8      *a = *a ^ *b;
9      *b = *a ^ *b;
10     *a = *a ^ *b;
11 }
```

By calling `swap(&a, &b)` with the addresses of two integers, `a` and `b`, the above function modifies the values of `a` and `b` directly in the calling environment. This example demonstrates the concept of passing references and illustrates the power of pointers in function calls, allowing for in-place updates that are central to C's efficiency and flexibility in low-level programming.

## 2.5 Exercise 2A

### 2A - Guessing Game

**This problem is an interactive game!**

#### Problem Statement

One day, Glasgow College designed a prized puzzle. This puzzle is related to an **arithmetic progression**, which is defined by a sequence of integers such that the difference of adjacent sequence elements ($x_i - x_{i-1}$) is constant. This difference is referred to as the **common difference** of the sequence. For example, $\{3, 6, 9, 12, 15\}$ is an arithmetic progression with a common difference of 3.

The puzzle is defined as follows. There is a secret list of $n$ integers $\{a_0, a_1, \ldots, a_{n-1}\}$. This list is special: if sorted in **ascending** order, it will form an arithmetic progression with a positive common difference ($d > 0$). For example, the list $\{17, 8, 11, 5, 14\}$ satisfies this requirement, as sorting it makes the list $\{5, 8, 11, 14, 17\}$.

Glasgow College has also provided you with a device capable of doing the following: for an input of an index $i$ ($0 \leq i \leq n-1$), the device will show the corresponding value $a_i$. As an additional challenge, the device has a limited battery, so you can only use it to perform a maximum of $N$ queries.

In your case, the battery will allow you to perform a **maximum** of $N = 30$ queries. Your task is to determine the common difference of the secret sequence given the specified constraints.

#### Interaction

The first input is a couple of integers $n$ and $a_0$ ($2 \leq n, a_0 \leq 10^6$), where $n$ represents the size of the integer list, and $a_0$ represents the first element of this list.

Then, you input integer queries `i` ($0 \leq i \leq n-1$) to get the value of $a_i$. ($1 \leq a_i \leq 10^9$) and output the result as `? a_i`. When you find out the common difference $d$, output `! d`.

For example, let us walk through the process with a secret sequence of $\{17, 8, 11, 5, 14\}$.

#### Sample Input

```
5 5

17

8

11

5

14
```

### Sample Output

```
? 0

? 1

? 2

? 3

? 4

! 3
```

### Instructions and Hints

Try 30 random queries for $a_i$, which means that $i$ should be **generated randomly**. Then calculate the greatest common factor of all the sequence differences $a_i - a_0$. The obtained number is almost certainly the correct common difference, with high probability. This can be proved after you have learned Probability and Mathematical Statistics in the second year.

**Please add `fflush(stdout);` after every `printf()`.** For performance and to reduce latency, most C standard library implementations *buffer* input and output, i.e., input and output are performed on "chunks" of text rather than on individual characters. The execution of the statement `fflush(stdout);`, in effect, disables this behaviour and clears the output buffer after each call of `printf()` to *immediately* print the output on the screen. Documentation for the `fflush` command can be found here: https://en.cppreference.com/w/c/io/fflush.

### Base Code

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

int gcd(int a, int b) {//This function will return the greatest common
    divisor between two integers
    if(b == 0) return a;
    return gcd(b, a % b);
}

/* HINT: Your program should have an interactive flow of
[ output '? i' -> flush -> read -> output '! d' -> flush ]
NOTE: input -> scanf(), flush -> fflush(stdout), output -> printf() */
int main() {
    //TODO
}
```

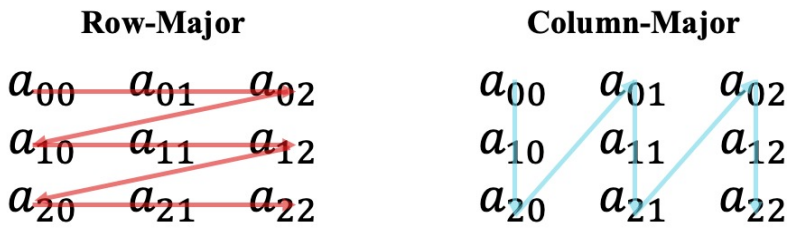
## 2.6 Exercise 2B

### 2B - Row- and column-major matrix representation

**Time limit per test: 3 seconds**

#### Problem Statement

In computing, row-major order and column-major order are methods used to store multidimensional arrays in memory. The following figure shows how to arrange linear memory in row-major order and column-major order for a two-dimensional array.




In **row-major order** (used by languages like C), the elements of an array are stored row by row, with each successive row placed immediately after the previous one in memory. For example, a 2D array `arr[2][3] = { {1, 2, 3}, {4, 5, 6} }` would be stored as an equivalent 1D array `{1, 2, 3, 4, 5, 6}`. In contrast, **column-major order** (used by languages like FORTRAN and MATLAB) stores elements column by column, so the same array would be stored as a 1D array, `{1, 4, 2, 5, 3, 6}`. The following figure shows graphically how multidimensional arrays are stored in memory for this example.


Depending on how the array is accessed during computation, these different storage schemes can affect performance in terms of memory access patterns and cache utilization.

In this exercise, you are given a 2D matrix of size $M \times N$ stored in row-major order as a 1D array. Your goal is to output the matrix elements in column-major order **without** allocating any additional memory. This means that you need to rearrange the matrix directly within the provided memory constraints.

### Constraints

- $0 < M, N \leq 1024$.
- For some (30%) test cases, additional memory allocations are restricted. Any attempt to allocate new memory in these cases will result in failure.
- **The only thing you need to do in this exercise is to complete the `transpose` function. Please delete the `main` function when you submit your code; otherwise, it will lead to a compilation error.**

### Input

The input consists of two lines.

- The first line contains two integers $M$ and $N$, representing the number of rows and columns in the matrix.
- The subsequent $M$ lines each contain $N$ integer numbers, indicating each element of the matrix in row-major order.

### Output

One line containing the elements of the matrix in column-major order.

### Sample Input 1

```
2 3
1 2 3 4 5 6
```

### Sample Output 1

```
1 4 2 5 3 6
```

### Sample Input 2

```
3 2
1 2 3 4 5 6
```

### Sample Output 2

```
1 3 5 2 4 6
```

### Base Code

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

#define max(a, b) ((a) > (b)) ? (a) : (b)

void transpose(int *matrix, int m, int n, int ld) { /* TODO */ }

int main() { //Please delete the main function when you hand in.
    int m, n;
    scanf("%d%d", &m, &n);
    int ld = max(m, n); //leading dimension
    int *matrix;
    matrix = (int *)malloc(ld * ld * sizeof(int));

    /* Sample input 1:
    2 3
    1 2 3 4 5 6
    matrix (row-major order):
    1 2 3
    4 5 6
    0 0 0

    Sample input 2:
    3 2
    1 2 3 4 5 6
    matrix (row-major order):
    1 2 0
    3 4 0
    5 6 0 */

    for(int i = 0; i < m; ++i) {
        for(int j = 0; j < n; ++j) {
            scanf("%d", &matrix[i * ld + j]);
        }
    }

    transpose(matrix, m, n, ld);

    for(int j = 0; j < n; ++j) {
        for(int i = 0; i < m; ++i) {
            printf("%d ", matrix[j * ld + i]);
        }
    }
    return 0;
}
```