

# C the Code

Partha Bhowmick

Professor

CSE Department, IIT Kharagpur

<http://cse.iitkgp.ac.in/~pb>

November 8, 2024

---

# Contents

<b>1</b>	<b>Introduction</b>	<b>7</b>
1.1	Computer . . . . .	7
1.2	Components of a computer . . . . .	7
1.3	Algorithm and flowchart . . . . .	8
1.4	Computer program . . . . .	8
1.5	Exercise problems . . . . .	9
<b>2</b>	<b>Variables and expressions</b>	<b>11</b>
2.1	Variables . . . . .	11
2.2	Data types . . . . .	12
2.3	Constants . . . . .	13
2.4	Statements . . . . .	13
2.5	Operands, operators, expressions . . . . .	14
2.6	Precedence of arithmetic operators . . . . .	14
2.7	Type casting . . . . .	15
2.8	Types of expressions . . . . .	15
2.9	Solved problems . . . . .	17
2.10	Exercise problems . . . . .	24
<b>3</b>	<b>Conditionals</b>	<b>25</b>
3.1	Solved problems . . . . .	26
3.2	Exercise problems . . . . .	32
<b>4</b>	<b>Loops</b>	<b>34</b>
4.1	Syntax of <code>while</code> loop and <code>do-while</code> loop . . . . .	34
4.2	Syntax of <code>for</code> loop . . . . .	35
4.3	<code>break</code> and <code>continue</code> . . . . .	35
4.4	Nested loops . . . . .	36
4.5	Solved problems . . . . .	38
4.6	Exercise problems . . . . .	43
<b>5</b>	<b>One-dimensional arrays</b>	<b>44</b>
5.1	What is array? . . . . .	44
5.2	Why array? . . . . .	44
5.3	Declaring arrays . . . . .	46
5.4	Initializing arrays . . . . .	46
5.5	Accessing and working with arrays . . . . .	47
5.6	Solved problems . . . . .	48
5.7	Exercise problems . . . . .	52
<b>6</b>	<b>Functions</b>	<b>54</b>
6.1	What is function? . . . . .	54
6.2	Defining a function . . . . .	55
6.3	Execution of a Function . . . . .	56

6.4	Prototype versus Definition . . . . .	57
6.5	The <code>return</code> statement . . . . .	59
6.6	Local and global variables . . . . .	60
6.7	Scope of a variable . . . . .	60
6.8	Parameter passing . . . . .	61
6.8.1	Passing by value . . . . .	61
6.8.2	Passing by reference . . . . .	62
6.9	Recursive function . . . . .	63
6.9.1	Activation record and recursion stack . . . . .	64
6.9.2	Tower of Hanoi . . . . .	66
6.9.3	Direct and indirect recursion . . . . .	68
6.9.4	Mutual recursion . . . . .	68
6.10	Passing an array to a function . . . . .	71
6.11	Macros ( <code>#define</code> ) . . . . .	71
6.12	<code>#define</code> with arguments . . . . .	72
6.13	Extra topics . . . . .	74
6.13.1	Generating random input using <code>rand</code> . . . . .	74
6.13.2	<code>main()</code> with arguments . . . . .	76
6.14	Solved problems . . . . .	78
6.15	Exercise problems . . . . .	83
<b>7</b>	<b>Strings</b> . . . . .	<b>85</b>
7.1	Characters and strings . . . . .	85
7.2	Declaring a string . . . . .	86
7.3	Initializing a string . . . . .	86
7.4	Reading strings with <code>%s</code> . . . . .	87
7.5	Reading strings with white spaces . . . . .	88
7.6	String library . . . . .	89
7.6.1	<code>strlen</code> . . . . .	90
7.6.2	<code>strcpy</code> . . . . .	91
7.6.3	<code>strcat</code> . . . . .	92
7.7	Solved problems . . . . .	93
7.8	Exercise problems . . . . .	95
<b>8</b>	<b>Pointers</b> . . . . .	<b>96</b>
8.1	What is a pointer? . . . . .	96
8.2	Types of pointer . . . . .	97
8.3	Use of pointer . . . . .	98
8.4	Operations with pointers and dereferenced pointers . . . . .	100
8.4.1	Dereferencing . . . . .	100
8.5	Pointer to 1D array . . . . .	101
8.5.1	Usefulness . . . . .	101
8.5.2	Indexing with pointer . . . . .	101
8.6	Pointer arithmetic . . . . .	102
8.7	Scale factor: <code>sizeof()</code> . . . . .	103
8.8	Passing pointers to a function . . . . .	104
8.9	Dynamic memory allocation . . . . .	105
8.9.1	Memory Allocation Functions . . . . .	106
8.9.2	Dynamic memory allocation and error handling . . . . .	107
8.10	Solved problems . . . . .	107
8.11	Exercise problems . . . . .	111
<b>9</b>	<b>Two-dimensional arrays</b> . . . . .	<b>113</b>
9.1	Examples . . . . .	113

9.2	Declaration (static) . . . . .	114
9.3	Initialization . . . . .	115
9.4	Operations . . . . .	116
9.5	2D array storage . . . . .	117
9.6	Passing 2D arrays to functions . . . . .	118
9.7	Dynamic memory allocation for 2D array . . . . .	119
9.8	Declaration (dynamic): A summary . . . . .	121
9.9	Arrays with higher dimension . . . . .	122
9.10	Solved problems . . . . .	123
9.11	Exercise problems . . . . .	130
<b>10</b>	<b>Searching in 1D array</b> . . . . .	<b>132</b>
10.1	Linear search . . . . .	132
10.2	Binary search . . . . .	134
10.3	Solved problems . . . . .	137
10.4	Exercise problems . . . . .	140
<b>11</b>	<b>Sorting</b> . . . . .	<b>142</b>
11.1	Bubble Sort: A basic sorting algorithm . . . . .	143
11.2	Selection Sort (Version 1) . . . . .	145
11.3	Selection Sort (Version 2) . . . . .	146
11.4	Insertion Sort: Another basic sorting algorithm . . . . .	148
11.5	Quick Sort . . . . .	150
11.6	Merge Sort . . . . .	155
11.7	Classification of sorting algorithms . . . . .	156
11.8	Recursive vs Iterative Algorithms . . . . .	157
11.9	Exercise problems . . . . .	158
<b>12</b>	<b>Number systems</b> . . . . .	<b>159</b>
12.1	Representation of Integers . . . . .	159
12.1.1	Decimal number system . . . . .	159
12.1.2	Octal number system . . . . .	159
12.1.3	Hexadecimal number system . . . . .	160
12.1.4	Conversion from decimal number system . . . . .	160
12.1.5	Conversion from decimal to binary . . . . .	161
12.1.6	Unsigned binary number system . . . . .	161
12.1.7	Word of CPU . . . . .	161
12.1.8	Signed number . . . . .	162
12.1.9	Sign-magnitude . . . . .	162
12.1.10	1's complement numeral . . . . .	163
12.1.11	2's complement numeral . . . . .	164
12.1.12	Word extension in 2's complement . . . . .	165
12.1.13	Carry-out and overflow in 2's complement . . . . .	165
12.1.14	10's complement . . . . .	166
12.2	Decimal numbers: Standard floating-point representation . . . . .	167
12.2.1	Floating-point representation versus fixed-point representation . . . . .	168
12.3	Binary numbers: Normalized floating-point representation . . . . .	169
12.3.1	Data classes and normalization . . . . .	169
12.3.2	Examples . . . . .	171
12.3.3	<code>nan</code> . . . . .	172
12.3.4	About <code>inf</code> , <code>+0</code> , <code>-0</code> . . . . .	172
12.4	Solved problems . . . . .	173
12.5	Exercise problems . . . . .	178
<b>13</b>	<b>Structures</b> . . . . .	<b>179</b>

13.1	Structure declaration . . . . .	179
13.2	Structure variable declaration . . . . .	180
13.3	The <code>typedef</code> construct . . . . .	180
13.4	Structures and <code>typedef</code> . . . . .	180
13.5	Operations with structure . . . . .	181
	13.5.1 Accessing the members: dot operator . . . . .	181
	13.5.2 Structure initialization . . . . .	182
13.6	Assignment of structure variables . . . . .	182
13.7	Array inside structure . . . . .	182
13.8	Size of a structure . . . . .	183
13.9	Array of structure . . . . .	183
13.10	Nested structure . . . . .	184
13.11	Self-referencing structure . . . . .	185
13.12	Structure as function argument . . . . .	185
13.13	Pointer to structure as function argument . . . . .	186
13.14	Operator Precedence . . . . .	187
13.15	Solved problems . . . . .	187
13.16	Exercise problems . . . . .	190
<b>14</b>	<b>Abstract Data Types</b> . . . . .	<b>192</b>
14.1	List . . . . .	192
14.2	Set . . . . .	192
14.3	Stack . . . . .	193
	14.3.1 Header Files . . . . .	194
14.4	Queue . . . . .	197
	14.4.1 Applications . . . . .	197
	14.4.2 Linear queue . . . . .	198
	14.4.3 Circular queue . . . . .	201
14.5	Conceptual problems . . . . .	203
14.6	Solved problems . . . . .	207
<b>15</b>	<b>Linked Lists</b> . . . . .	<b>215</b>
15.1	Essence of linked lists . . . . .	215
	15.1.1 Scope and programmability . . . . .	215
	15.1.2 Advantages of linked lists . . . . .	216
	15.1.3 Disadvantages of linked lists . . . . .	216
	15.1.4 Comparison with array-based datatypes . . . . .	216
15.2	Node of a linked list . . . . .	217
15.3	Operations on linear linked list . . . . .	218
	15.3.1 Creating a linked list . . . . .	218
	15.3.2 Inserting at the beginning of a linked list . . . . .	218
	15.3.3 Inserting at the end of a linked list . . . . .	220
	15.3.4 Inserting in an ordered linked list . . . . .	221
	15.3.5 Deleting from a linked list . . . . .	222
	15.3.6 Searching in a linked list . . . . .	223
	15.3.7 Displaying a Linked List . . . . .	223
15.4	Stack as a linked list . . . . .	224
15.5	Queue as a linked list . . . . .	225
15.6	Solved problems . . . . .	227
15.7	Exercise problems . . . . .	230

## General note

1. Questions at the advanced level are marked with ♣ (harder) and ♠ (hardest).
-

# 1 | Introduction

## 1.1 Computer

Computers have changed our world, making it easier to work, learn, and play. At the heart of every computer is programming, which tells the computer what to do. Programming turns ideas into instructions that computers can follow, enabling amazing things in science, medicine, and everyday life. As we dive into C programming, we will learn the basics that allow these incredible machines to solve problems, run apps, and power the digital world. Understanding programming is like unlocking a superpower, giving you the skills to create and innovate with technology.

A **computer** is a rapidly evolving machine engineered to process data and extract information. The processing is done according to a given sequence of instructions called a **program** or **code**. These programs enable computers to perform a wide range of computational tasks. More importantly, many of these tasks are performed at lightning speed—much faster than even the most brilliant minds—such as adding a million numbers in less than a millisecond!

## 1.2 Components of a computer

All computations in a computer are carried out through arithmetic and logical operations, based on Boolean logic and the binary number system. And all kinds of data in a computer—whether numbers or text or image or audio-video signal—are stored in a computer just as binary numbers or binary strings. To store data and to process them, the following components are required for manufacturing a **computer system** (Figure 1.1).

1. **Computer hardware:** Includes the physical parts of a computer, which are as follows.
  - (i) **Central Processing Unit (CPU)** with a microprocessor that carries out all arithmetic and logical operations, and with a control unit to arrange the order of operations as the need be.

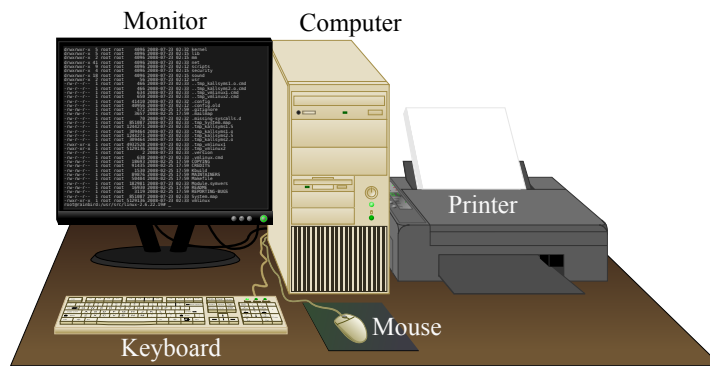


Figure 1.1: An example of a desktop computer system (source: *wiki*).

- (ii) **Memory Unit** consisting of the following types:
    - (a) **Random Access Memory (RAM)**, also known as **primary memory** or **main memory** or simply **memory**. Anywhere if you get just the term ‘memory’, know that it is RAM.
    - (b) **Secondary memory**, also known as **hard disk**.
    - (c) **Tertiary memory**, such as an external disk or a pen-drive.
  - (iii) **Input devices**, such as keyboard, mouse, and joystick.
  - (iv) **Output devices**, such as monitor or display screen, speakers, printers, and plotters.
  - (v) **Motherboard**: The printed circuit board (PCB) within a computer, containing the main circuitry where all other parts of the hardware plug into, in order to create a cohesive whole.
2. **Operating system (OS)** and other software: OS is the system software that manages computer hardware and software resources, and provides necessary services to run computer programs. For hardware functions related to input, output, and memory allocation, the OS acts as an intermediary between programs and the computer hardware. A **software** consists of one or more computer programs along with necessary documentation and other files.

### 1.3 Algorithm and flowchart

An **algorithm** provides a systematic method to solve a problem. It is visually represented by a **flowchart** and practically implemented by a **computer program**. Flowchart aids programmers in grasping the logic quickly, bypassing intricate programming details. Drawing a flowchart before writing the actual program is a very good habit for beginners. Figure 1.2 shows a flowchart with the related algorithm and a part of the C program needed to find and print the larger between two numbers. Another flowchart for the same problem but with three numbers as input, is also shown in Figure 1.2. Notice that in this flowchart, no extra variable is used but we need three comparisons. In §1.5 (Exercise), with the same input, you are asked to construct a flowchart that will use two comparisons and one extra variable.

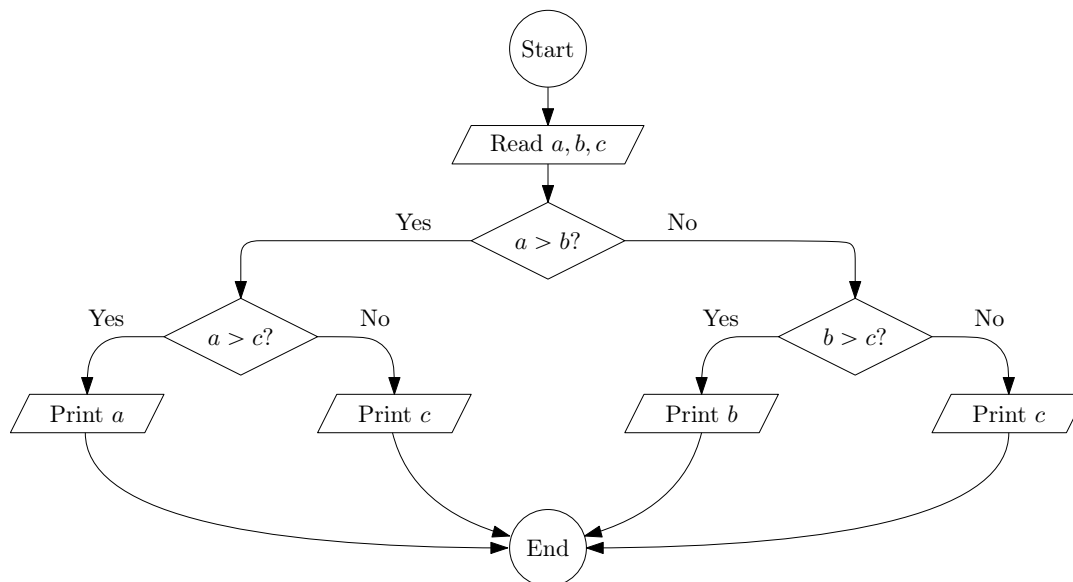
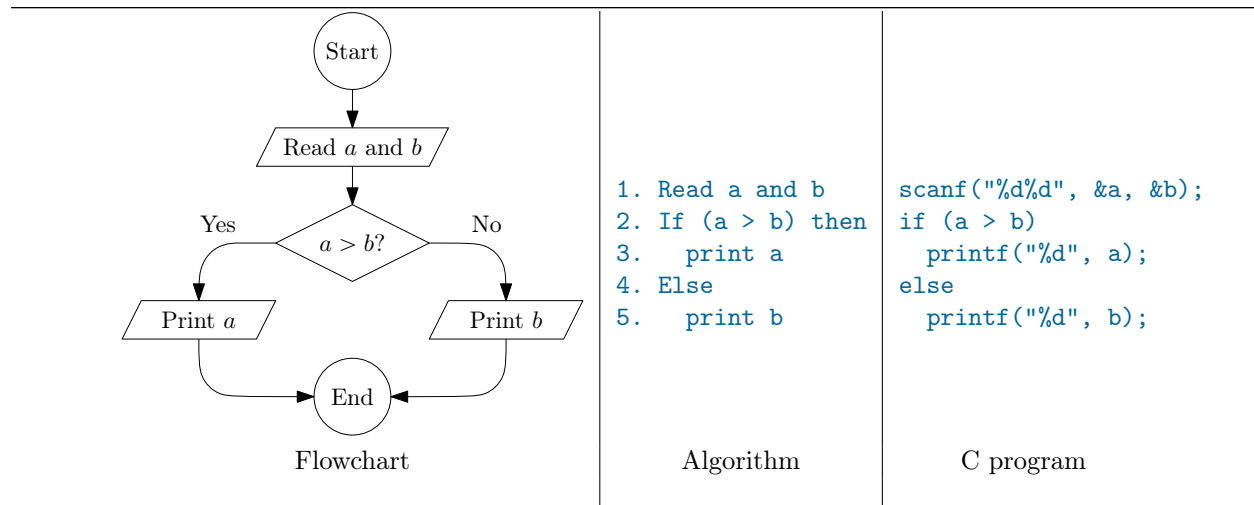
### 1.4 Computer program

A **computer program** is a sequence of instructions for a computer to execute. A computer program in its human-readable form is called **source code**. A source code is written in a **programming language** (also called **high-level language** or **HLL**), such as C, by a programmer. For example, a program `a01-1.c` written by you in C language is the source code. In C language, the source code needs a compiler such as `gcc` to run on it, in order to get the **binary executable code** (also called **executable code**, or simply **executable**, for brevity). For example, the command `gcc a01-1.c` gives you the executable `a.out` corresponding to the source code `a01-1.c`. Any executable code is basically a **machine code**, as it consists of machine-language instructions. The compiler `gcc` that you run on any C code is just a machine code. There are several important stages during the compilation process; the important ones are shown in Figure 1.3.

The source code `a01-1.c` and its executable `a.out` are all stored in the hard disk of your computer. When the executable `a.out` is requested for execution, the operating system loads `a.out` from the hard disk to the RAM and starts a **process**. The CPU switches to this process as soon as possible so that it can fetch, decode, and then execute the machine-language instructions of `a.out`, one by one; we then say that the process is **running** in the computer.

**Library** It contains many files written by experts. We call them **header files** or **library files**. One such header file is `stdio.h`, which must be included in any C code in its first line, using the instruction `#include <stdio.h>`. Any other header file, such as `math.h` or `stdlib.h`, should be included this way in the beginning. Each header file contains detailed instructions for complicated tasks. For example, the functions `scanf` and `printf` are contained in `stdio.h`, and the square-root function `sqrt` is contained in





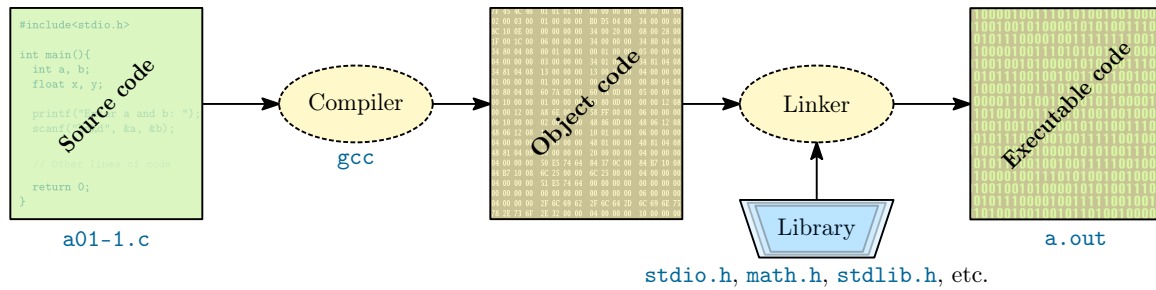
**Figure 1.2: Top:** The flowchart, the related algorithm, and a part of the C program needed to find and print the larger between two numbers,  $a$  and  $b$ .

**Bottom:** The flowchart to find and print the largest among three numbers.

`math.h`. You have to include these library files in your C program to use their functions. For example, you have to include `stdio.h` to use `scanf` or/and `printf`, by writing `#include <stdio.h>` in the beginning of your code. Similarly, you have to include `math.h` to use its `sqrt` function, by writing `#include <math.h>` just after `#include <stdio.h>`.

## 1.5 Exercise problems

1. [Largest among three numbers] Draw a flowchart to read three numbers and determine the largest. You may use an extra variable but must not exceed two comparisons in total.



**Figure 1.3:** The steps of compilation in C using the command `gcc a01-1.c`.

2. **[Sum of 10 numbers]** Draw a flowchart to read 10 numbers one by one and compute their sum. You may use two extra variables and a maximum of 9 additions in total.
3. **[Largest among 10 numbers]** Draw a flowchart to read 10 numbers one by one and determine the largest. You may use two extra variables and a maximum of 9 comparisons in total.
4. **[Largest and smallest among 10 numbers]** Draw a flowchart to read 10 numbers one by one and determine both the largest and the smallest. You may use three extra variables and should aim to minimize the total number of comparisons.

## 2 | Variables and expressions

### 2.1 Variables

The data used by any computer program are stored in **variables**. A variable is identified by its four attributes: **name, type, value, address**. These are explained below.

The **name** of a variable is written by the programmer. It can be *any* word or string composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Uppercase and lowercase letters are treated as distinct in the C language. Here are some examples of variable names: `x, dx, y12, sum_1, _MYvar, realNum, complexNum, point, area, tax_rate, list, Set, Vector`

The **type** of a variable is determined based on the type of data it stores. Accordingly, we have different **types of variables**, which are also termed as **data types**. Further details are given in §2.2.

Depending on the type, a variable has a specific **value** within a specific domain. This value is either assigned or computed when the program runs. To store this value in the main memory (i.e., RAM), a variable needs some space during execution of the program. The amount of space, referred to as **size**, depends on the data type of the variable. Hence, before using any variable, its type must be declared in the program. This is called **variable declaration**. For example, by the declaration `int a`, the variable name is `a` and its type is declared as an integer; hence, a space of 4 bytes will be allocated in the memory during execution of the program. As another example, by the declaration `char c`, the variable `c` is declared as a character, and hence a space of 1 byte will be allocated in the memory.

The **address** of a variable specifies its storage location in the main memory, which is settled during the execution of the code. The address is denoted by the ampersand character, i.e., `&`; for example, for the declaration `int a`, the address of `a` is denoted by `&a`. While reading the value of a variable for taking input using `scanf`, its address is needed to store the value; that's why in `scanf` the address is passed as an argument. For example, to take as input the value of the integer variable `a`, we have to write `scanf("%d", &a)` to tell the compiler that an integer has to be scanned in the decimal number system ("`%d`" means that) and stored at the address `&a`.

Apart from the above-mentioned attributes, every variable has a **scope** and an **extent** in its corresponding code. Its **scope** describes the part or block of code where it can be used. The **extent** describes its lifetime in the execution of the code, i.e., the duration for which it has a meaningful value. The scope of a variable is actually a property of the name of the variable, and the extent is a property of the storage location of the variable. Scope is an important part of the name resolution of a variable. Because two variables (in the same code) may have the same name but with different scopes. For example, a variable with the name `x` can be declared and used in one `for` loop, and another variable with the same name `x` may be declared in another `for` loop. They will be assigned different memory locations and will work therefore without any conflict.

There are certain reserved words, called **keywords**, that have predefined meanings in C. These keywords cannot be used as variable names. They are listed in Table 2.1. Note that the keywords are all written in lowercase only. Since uppercase and lowercase characters are not equivalent in programming language, it is not illegal to write a variable name as an uppercase keyword. However, this is considered a poor programming practice.

**Table 2.1:** List of standard keywords in C language.

<code>auto</code>	<code>break</code>	<code>case</code>	<code>char</code>	<code>const</code>	<code>continue</code>	<code>default</code>	<code>do</code>
<code>double</code>	<code>else</code>	<code>enum</code>	<code>extern</code>	<code>float</code>	<code>for</code>	<code>goto</code>	<code>if</code>
<code>int</code>	<code>long</code>	<code>register</code>	<code>return</code>	<code>short</code>	<code>signed</code>	<code>sizeof</code>	<code>static</code>
<code>struct</code>	<code>switch</code>	<code>typedef</code>	<code>union</code>	<code>unsigned</code>	<code>void</code>	<code>volatile</code>	<code>while</code>

## 2.2 Data types

**Table 2.2:** Basic data types in C language.

Data type	Meaning	Size
<code>char</code>	character	1 byte
<code>short</code>	integer	2 bytes
<code>int</code>	integer	4 bytes
<code>long int</code>	integer	4 or 8 bytes
<code>long long int</code>	integer	8 bytes
<code>float</code>	real	4 bytes
<code>double</code>	real	8 bytes
<code>long double</code>	real	16 bytes

A handful of **basic data types** are defined in C language (Table 2.2). Clearly, all the above data types basically store numbers. It should be understood that the data type `char` also stores just a number, which actually represents a single character.<sup>1</sup> Since the number of characters is limited, one byte is enough to represent all of them in a unique manner. Unless specified as **unsigned**, each of the above data types is **signed**. For example, the declaration `unsigned int a` specifies that the variable `a` is unsigned, i.e., all its 32 bits are used to represent its absolute value. If it is not unsigned, then its leftmost bit is used to represent the sign and hence termed as **sign bit**. If the sign bit is `0`, then it is a positive number, else it is negative. In case of `char`, there are 8 bits; by default, it is considered unsigned, and hence its value ranges from `0000 0000` = 0 to `1111 1111` = 255. If it is declared as **unsigned**, then this range is same. However, if it is declared as **signed**, e.g., `signed char c`, then the leftmost bit is used to fix its sign, and so its value ranges from `1000 0000` = -128 to `0111 1111` = 127. Know that the magnitude of `1000 0000` is given by its **2's complement** as follows: 1's complement of `1000 0000` = `0111 1111` (by reversing each bit); now, the 2's complement is given by adding `1` to the 1's complement, thus giving `0111 1111 + 1` = `1000 0000` = 128; taking the negative sign into account, we get -128 in the decimal number system.

For every other basic data type, the default specifier is signed. Explicit declaration has to be done to make it unsigned. For example, `int a` means `a` can be a positive or negative integer, and since it uses 32 bits, its range is from  $-2^{31}$  to  $2^{31} - 1$ , which is -2,147,483,648 (1 followed by 31 0's in binary) to 2,147,483,647 (0 followed by 31 1's in binary). If during execution of the code, the value of `a` goes out of this range, there could be faulty output. If the declaration is `unsigned int a`, then the variable `a` can take only non-negative integer values from 0 (32 0's in binary) to  $2^{32} - 1$  = 4,294,967,295 (32 1's in binary).

Apart from the above basic data types, there is also a special type specifier named `void`, which indicates that no value is available. It is used to specify the data type returned by a function, which we shall see later.

<sup>1</sup>In this context, it should also be well-understood that all data in the computer are essentially binary numbers or binary strings; the data may be simply a text or an image or an audio or a video or any other entity.

We also have another useful data type called **pointer** that can store the memory-address of any variable. This will also be discussed later.

Using the basic data types, a programmer can obtain new data types called **derived data types**. These include string, array, structure, and union, which will come up in later chapters.

## 2.3 Constants

Constants are broadly of two types: **numeric** and **character**. A numeric character is either integer or real. Character constant means either a single character or a string of characters. A single character is specified within single quotes, e.g., '0', 'y', '+'. A string is specified within double quotes. Some typical examples are as follows.

```
int a = 0;           // a is an integer, initialized with the value 0.
float x = 0.5, y;    // x is a real number in floating-point format.
                    // y is also a floating-point number but not initialized.
float z = 0.123e9    // z is a floating-point number,
                    // initialized with 0.123 times 10 raised to the power 9.
char c = 'y';        // c is a character, initialized with the letter y.
char s[10] = "iit\0" // s is a string, initialized with "iit\0" (we'll study it later).
```

Integer constants can also be written in hexadecimal number system. A hexadecimal integer must begin with `0x` or `0X` and then contain one or more hex digits. The hex digits are `0` through `9` and `a` through `f` (or `A` through `F`). The six letters `a` through `f` (or `A` through `F`) represent the decimal numbers 10 through 15, respectively. Following are some hexadecimal numbers: `0x`, `0xA`, `0X2D`, `0xf5a`; check that their respective values in decimal number system are 0, 10, 45, 3930.

There are also multiple ways of representing a real number. For example,  $5 \times 10^4$  can be represented by any of the following floating-point constants: `50000`, `5e4`, `5e+4`, `5E4`, `5.0e+4`, `.5e5`, `50E3`, `50.E+3`, `500e2`.

## 2.4 Statements

In C language, a **statement** is a complete instruction that tells the computer to perform a specific task. Statements are the building blocks of a program and typically end with a semicolon (`;`). Statements can include operations like assigning values to variables, controlling the flow of execution, or calling functions. Following are some examples.

```
int a = 5, b = 10; // assignment statement
float x;           // declaration statement
if (a < b) {       // control statement
    printf("a is less than b"); // function call statement
    x = a + b/2;    // assignment statement
}
else{
    ; // null statement
}
```

Note that the sign of equality (`=`) is used to assign values to variables. Assignment is an operation, and hence `=` is called the **assignment operator**. The left of `=` is termed as ***l*-value**. In `x = a + b/2`, the *l*-value refers to the value of `x`. An *l*-value has a definite address in memory during execution. On the contrary, whatever occurs at the right side of `=` is referred to as the ***r*-value**. In the last example, it corresponds to the expression `a + b/2`. It has no address of its own, although the variables `a` and `b` have definite addresses in the memory during the execution.

Types of *l*-value and *r*-value should preferably be the same; e.g., either both are integers or both are real. If not, the type of the *r*-value will be converted to the type of the *l*-value, and then assigned to it. Consider the following example.

```
double a;
a = 2*3;
```

In the assignment statement, the type of *r*-value is `int` because 2 and 3 are both integers, and so its value is 6. However, since the type of *l*-value is `double`, the value stored for `a` is real, i.e., 6.0.

Now, consider another example:

```
int a;
a = 2*3.4;
```

Here, the type of *r*-value is real and the value is 6.8. But, since the type of *l*-value is `int`, so its integer part, i.e., 6, is stored in `a`.

## 2.5 Operands, operators, expressions

An **expression** is an appropriate combination of variables, constants, and one or more operators. In general, by the word ‘expression’, we mean an expression with at least one variable; an expression with no variable at all, e.g., `2+3`, is referred to as a ‘constant expression’ and seldom used in a code. An expression gets a single numerical value when its variables are assigned appropriate values and all the necessary operations are performed. Expressions can be of different types, as shown in Table 2.3.

An **operand** may be a constant or a variable or an expression. Its value can be an integer (`int` or `long int` or `long long int`) or a real number (`float` or `double` or `long double`). It can be the ASCII value of a character as well. For example, in `'f'+2`, the operand is `'f'`. Here, the English lowercase character `f` is denoted precisely by `'f'` to indicate that its type is `char`. The value of the operand `'f'` is simply the ASCII value of the character `f`. When this value is added with 2, the result will be the ASCII value of the next-to-next lowercase character in the English alphabet, i.e., `h`. Similarly, `'h'-2` will give the ASCII value of `'f'`.

An **operator** defines the operation either on a single operand or on two operands; in the former case, it is said to be a **unary operator**, while for the latter it is a **binary operator**. For example, `++` is a unary operator, and `+` is a binary operator. In `a++` or `++a`, the value of the operand `a` increases by unity when the operator `++` operates on `a`.

An expression can be used to build larger expressions. As a typical example, consider `(a+b)*(--a) - 5`. It is an expression with two variables and one constant, having the operators for addition, multiplication, decrement, and subtraction. The addition acts on two operands `a` and `b`, each being a variable. The multiplication acts on two operands, each being an expression but not a variable. The decrement, denoted by `--`, is the sole unary operator here and acts on `a` to decrease its value by unity. The subtraction is a binary operation defined on the two operands, `(a+b)*(--a)` and `5`, the first one being a typical expression and the second being a constant. To understand how the value is computed, let us take `a` and `b` as 3 and 2 respectively. Then, the steps of computation are:  $(a+b)*(--a) - 5 = (3+2)*(--3) - 5 = (5)*(2) - 5 = 10 - 5 = 5$ . In the actual process of computation in a computer, there are several other intermediate steps involving computer memory and processing unit.

## 2.6 Precedence of arithmetic operators

It refers to the priority or order in which operations are performed when an expression contains multiple operators. Operators with higher precedence are evaluated before those with lower precedence. The precedence

order for arithmetic operators in C (from highest to lowest) is:

1. Parentheses: (...)
2. Unary minus: e.g., -5
3. Multiplication, Division, and Modulus: \*, /, %
4. Addition and Subtraction: +, -

For operators of the same priority, evaluation is from left to right as they appear. For example, `a*-b+d%e-f` means `a*(-b)+(d%e)-f`. Parenthesis may be used to change the precedence of operator evaluation.

While working with expressions, it is the responsibility of the programmer to write an expression in the correct form. For example, `1.0 / 3.0 * 3.0` will produce the value `0.999999`, while `1.0 * 3.0 / 3.0` will produce the value `1.000000`.

## 2.7 Type casting

While working with numbers, you have to be careful. See the following code.

```
int a=10, b=4, c;
float x;
c = a / b;
x = a / b;
```

The value of `c` will be the integer obtained by dividing 10 by 4. In the integer domain, this value is the quotient of dividing 10 by 4, and so `c` receives the value 2. The value of `x` will be the integer obtained by dividing 10 by 4, which is 2 again, and but mapping it to the real domain gives 2.0; hence, `x` gets the value 2.0. By **type casting**, we can store the value 2.5 to `x`. It can be done by:

```
x = (float)a / b;
```

Since the right side is a mix of real (`(float)a`) and integer (`b`), the operation is done in the real domain. As another example, the following code won't produce correct output when `a+b` is odd.

```
int a, b;
scanf("%d%d", &a, &b);
avg = (a + b)/2;
printf("%f\n", avg);
```

The correct one will be if you write the assignment as:

```
avg = ((float) (a + b))/2;
```

or as:

```
avg = (a + b) / 2.0;.
```

There are some restrictions on typecasting. Everything cannot be typecast to anything. For example, `float` or `double` should not be typecast to `int`, as a variable of type `int` cannot store everything that a `float` or `double` can store. For a similar reason, a variable of type `int` should not be typecast to `char`.

## 2.8 Types of expressions

The type of an expression is determined by the operators used in it. For example, if `b` and `c` are two integers, then `b+c` is an arithmetic expression, `b<c` is a relational expression, `b&& c` is a logical expression, whereas `b&c` is a bitwise expression. Following are some important points about different types of expression.

**Table 2.3:** Different types of expressions.

Type of expression	Operators	Type of operands	Value of expression
1. Arithmetic expression	+ - * / ++ --	integer or real	integer or real
2. Relational expression	< <= > >= == !=	integer or real	0 or 1
3. Logical expression	&&    !	integer or real	0 or 1
4. Bitwise expression	>> << ~ &   ^	integer	integer

1. **Arithmetic expression:**  $b+c$  is an arithmetic expression in which the addition operator  $+$  acts on the operands  $b$  and  $c$ . On assigning the values 2 and 3 to  $b$  and  $c$  respectively, the value of the expression  $b+c$  becomes 5.

Although it may sound strange to one who is new to computer programming,  $a=b+c$  is also an arithmetic expression with two operators, namely addition ( $+$ ) and assignment ( $=$ ). Here, the value of the expression  $b+c$  is computed first, and that value is assigned to  $a$ , which eventually becomes the value of the whole expression  $a=b+c$ . For example, on assigning the values 2 and 3 to  $b$  and  $c$  respectively, the variable  $a$  gets the value 5, and hence the value of the expression  $a=b+c$  becomes 5. Note that the value of  $a=b+c$  is same as the value of  $a$ , because in C language, due to the assignment operator, such an expression always gets the value of the variable (i.e., operand) to the left side of the assignment operator. It is the  $l$ -value and refers to the value of  $a$ , as explained in §2.4.

The value of an expression is important because that expression may be used as an operand in another expression. For example,  $d=(a=b+c)+1$  is an expression composed with the previous expression,  $a=b+c$ . On assigning the respective values 2 and 3 to  $b$  and  $c$ , the value of  $a$  becomes 5, which means the expression  $a=b+c$  receives the value 5, which gives 6 as the value of  $d$ , and this finally sets the value of the whole expression  $d=(a=b+c)+1$  to 6.

2. **Relational expression:** Consider the relational operator  $<$  used in the relational expression  $a<b$ . The value of  $a<b$  will be 1 if  $a$  is less than  $b$ , and 0 otherwise.

A relational operator is used to compare an operand with another. The operand may be a variable or a non-variable expression. For example, in the expression  $a != b$  the operand  $!=$  is used between two variables; it yields 1 if  $a$  and  $b$  are unequal, and yields 0 if not. On the contrary, in the expression  $a != b*b+1$ , the same operand works between a variable and an expression.

3. **Logical expression:** Here we need to be careful—any nonzero integer means TRUE, and only the integer 0 means FALSE. For example,  $a&&b$  evaluates to 0 if and only if at least one of  $a$  and  $b$  is 0. On the contrary,  $a||b$  evaluates to 0 if and only if both  $a$  and  $b$  are 0. Equivalently,  $a&&b$  evaluates to 1 if and only if both  $a$  and  $b$  are nonzero, whereas  $a||b$  evaluates to 1 if and only if at least one of  $a$  and  $b$  is nonzero. The notation  $&&$  denotes the logical AND operator, whereas  $||$  denotes the logical OR.

The notation  $!$  denotes the logical NOT; it is a unary operator and hence acts on a single operand. For example, the expression  $!a$  evaluates to 0 if and only if  $a$  is nonzero. That is, if  $a$  is any number other than 0, then  $!a$  evaluates to 0, and it evaluates to 1 only if  $a$  is 0.

Clearly, since a logical expression basically works with the truth value of the associated operands, the operands can be any real number.

Consider a practical expression:  $((!weekday \&\& hobby) || (weekday \&\& study))$ . It has three variables working as operands, namely `weekday`, `hobby`, and `study`; and it has four logical operators in total. With `weekday = 0`, `hobby = 1`, `study = 0`, it evaluates to  $(1 || 0) = 1$ . With `weekday = 1, 2, \dots, 6`, it evaluates to 0 if `study = 0`, no matter the value of `hobby`—can you check and argue?

4. **Bitwise expression:** An expression where bitwise operators are used for computation at the bit level. It makes the computation fast. Bitwise operators work with integer-valued operands and cannot be applied to non-integer real numbers such as `float` or `double`. The bitwise binary operators are  $\&$  (AND),  $|$  (OR),  $\wedge$  (XOR),  $\ll$  (left shift), and  $\gg$  (right shift); the bitwise unary operator is  $\sim$  (1's complement).



For example,  $a \ll k$  means a left shift of the bits of  $a$  by  $k$  places. Say  $a$  has the value 5; so, its 8-bit representation is `00000101`; then  $a \ll 3$  means a left shift of the bits of  $a$  by 3 places, appending three 0's at the right side so that it again has 8 bits. This gives `00101000`, thereby changing the value of  $a$  to  $5 \times 2^3 = 40$ . On the contrary,  $a \gg 3$  does a right shift by 3 bits, changing the value of  $a$  to `00000000` = 0. Similarly,  $a \gg 1$  applies a right shift by 1 bit, changing the value to `00000010` = 2; and  $a \gg 2$  applies a right shift by 2 bits, changing the value to `00000001` = 1. That is, bitwise left shift by  $k$  bits is equivalent to multiplying by  $2^k$ , and bitwise right shift by  $k$  bits is equivalent to dividing by  $2^k$ .

The following code shows uses of the bitwise operators. Note that if the leftmost bit is 1, then it is a negative number. For example, the 1's complement of  $a$  (i.e.,  $\sim a$ ) is a negative number. To print its unsigned value, the format is `%u`. To print its signed value, the format is `%d`; and that value is given by the 2's complement of  $\sim a$ , which, in turn, is given by the 1's complement of  $\sim a$  (i.e.,  $a$ ) plus 1.

```

1 #include <stdio.h>
2
3 int main() {
4     int a = 20;           // 00...0 0001 0100 = 20
5     int b = 13;          // 00...0 0000 1101 = 13
6
7     printf("Bitwise AND: a&b = %d\n", a&b ); // 00...0 0000 0100 = 4
8     printf("Bitwise OR: a|b = %d\n", a|b ); // 00...0 0001 1101 = 45
9     printf("Bitwise XOR: a^b = %d\n", a^b ); // 00...0 0001 1001 = 25
10    printf("1's Complement: ~a = %u\n", ~a ); // 11...1 1110 1011
11                                           // = 4294967275 (unsigned)
12    printf("1's Complement: ~a = %d\n", ~a ); // 11...1 1110 1011
13                                           // = -21 (2's complement)
14    printf("Left Shift: a<<2 = %d\n", a<<2); // 00...0 0101 0000 = 80
15    printf("Right Shift: a>>2 = %d\n", a>>2); // 00...0 0000 0101 = 5
16
17    return 0;
18 }
```

## 2.9 Solved problems

1. [One-variable integer expressions] Given an integer  $a$ , compute and print the values of the following expressions:  $-a$ ,  $2a - 3$ ,  $2a^2 - 3a - 4$ .

```

1 #include <stdio.h>
2
3 int main(){
4     int a;
5
6     printf("Enter the value of a: ");
7     scanf("%d", &a);
8
9     printf("-a = %d\n", -a);
10    printf("2a-3 = %d\n", 2*a-3);
11    printf("2a^2-3a-4 = %d\n", 2*a*a-3*a-4);
12
13    return 0;
14 }
```

2. **[Two-variable integer expressions]** Given two integers  $a$  and  $b$  as input, compute and print the values of the following expressions:  $a + b$ ,  $-a - 2b + 3$ ,  $-2ab$ ,  $1 - 2a(b - 3)$ .

```

1 #include <stdio.h>
2
3 int main(){
4     int a,b;
5
6     printf("Enter the values of a and b: ");
7     scanf("%d%d", &a, &b);
8
9     printf("a+b = %d\n", a+b);
10    printf("-a-2b+3 = %d\n", -a-2*b+3);
11    printf("-2ab = %d\n", -2*a*b);
12    printf("1-2a(b-3) = %d\n", -2*a*(b-3));
13
14    return 0;
15 }
```

3. **[Left shift]** Given two integers  $a$  and  $b$  as input, compute and print the value of  $2a + 4b$  without using any multiplication.

```

1 #include <stdio.h>
2
3 int main(){
4     int a,b;
5
6     printf("Enter the values of a and b: ");
7     scanf("%d%d", &a, &b);
8
9     a <<= 1;
10    b <<= 2;
11
12    printf("Answer = %d.\n", a+b);
13
14    return 0;
15 }
```

4. **[Real-domain expressions]** Compute and print the values of the  $\frac{x}{y}$  and  $\frac{1}{x} + \frac{1}{y}$  in floating point, where  $x, y$  are nonzero real numbers given as input. The value of the 1st expression should be printed up to the 6th decimal place, and that of the 2nd expression up to the 3rd decimal place. For example, if  $x = 2$  and  $y = 3$ , then the printed values should be 0.666667 and 0.833, respectively.

```

1 #include <stdio.h>
2
3 int main(){
4     float x, y, z;
5
6     printf("Enter the values of x and y: ");
7     scanf("%f%f", &x, &y);
8
9     printf("x/y = %f, 1/x + 1/y = %0.3f\n", x/y, 1/x + 1/y);
```

```

10 |
11 |     return 0;
12 | }

```

5. **[Integer-to-real map]** Compute and print the values of the following expressions in floating point (rounded off to 3rd decimal place), where  $a, b$  are positive integers given as input.

$$a + b, \quad \frac{a}{b}, \quad \left( \sqrt{\frac{1}{a} + \frac{1}{b}} \right)^{\frac{1}{a+b}}.$$

For example, if  $a = 2$  and  $b = 3$ , then the respective printed values will be 5.000, 0.667, 0.982.

As the values should be real, the computations should be in the real domain. You should use the math library (`math.h`) and compile your code as follows: `gcc <input file> -lm`

```

1  #include <stdio.h>
2  #include <math.h>
3
4  int main(){
5      int a, b;
6
7      printf("Enter the values of a and b: ");
8      scanf("%d%d", &a, &b);
9
10     printf("1st expression: a+b = %0.3f\n", (float)(a+b));
11     printf("2nd expression: a/b = %0.3f\n", (float)a/b);
12     printf("3rd expression      = %0.3f\n", pow(sqrt(1.0/a + 1.0/b), 1.0/(a+b)));
13
14     return 0;
15 }

```

6. **[Pre-increment and post-increment]** Write the new values obtained after the following statements are executed one after the other.

```

int a, b, x;      → a, b, x all have 'garbage values'
a = 10, b = 20, x; → a gets 10, b gets 20
x = 50 + ++a;    → a = 11, x = 61 (a is first incremented and then added)
x = 50 + a++;    → x = 61, a = 12 (a is first added and then incremented)
x = a++ + --b;   → b = 19, x = 31, a = 13 (b is first decremented and then added)
x = a++ - ++a;   → x = -2, a = 15 (a++ is 13 as operand, ++a is 15 as operand)

```

In the last statement, there is a **side effect**: while calculating some values, something else get changed. It is always better to avoid such complicated statements.

7. **[One-variable integer expressions]** Given an integer  $a$ , compute and print the values of the following expressions:  $-a$ ,  $2a - 3$ ,  $2a^2 - 3a - 4$ .

```

1  #include <stdio.h>
2
3  int main(){
4      int a;
5
6      printf("Enter the value of a: ");
7      scanf("%d", &a);
8
9      printf("-a = %d\n", -a);

```

```

10 |     printf("2a-3 = %d\n", 2*a-3);
11 |     printf("2a^2-3a-4 = %d\n", 2*a*a-3*a-4);
12 |
13 |     return 0;
14 | }

```

8. **[Two-variable integer expressions]** Given two integers  $a$  and  $b$  as input, compute and print the values of the following expressions:  $a + b$ ,  $-a - 2b + 3$ ,  $-2ab$ ,  $1 - 2a(b - 3)$ .

```

1 | #include <stdio.h>
2 |
3 | int main(){
4 |     int a,b;
5 |
6 |     printf("Enter the values of a and b: ");
7 |     scanf("%d%d", &a, &b);
8 |
9 |     printf("a+b = %d\n", a+b);
10 |    printf("-a-2b+3 = %d\n", -a-2*b+3);
11 |    printf("-2ab = %d\n", -2*a*b);
12 |    printf("1-2a(b-3) = %d\n", -2*a*(b-3));
13 |
14 |    return 0;
15 | }

```

9. **[Minimum multiplications]** Given two integers  $a$  and  $b$  as input, compute and print the values of  $a^2b$ ,  $a^2b^2$ , and  $a^2b^4$ , using at most 6 multiplications for all three of them in total, without using any extra variable, and without using the math library.

```

1 | #include <stdio.h>
2 |
3 | int main(){
4 |     int a,b;
5 |
6 |     printf("Enter the values of a and b: ");
7 |     scanf("%d%d", &a, &b);
8 |
9 |     a = a*a;
10 |    printf("Answers = %d, ", a*b);
11 |    b = b*b;
12 |    printf("%d, ", a*b);
13 |    b = b*b;
14 |    printf("%d.\n", a*b);
15 |
16 |    return 0;
17 | }

```

10. **[Left shift]** Given two integers  $a$  and  $b$  as input, compute and print the value of  $2a + 4b$  without using any multiplication.

```

1 | #include <stdio.h>
2 |
3 | int main(){
4 |     int a,b;

```

```

5 |
6 | printf("Enter the values of a and b: ");
7 | scanf("%d%d", &a, &b);
8 |
9 | a <<= 1;
10 | b <<= 2;
11 |
12 | printf("Answer = %d.\n", a+b);
13 |
14 | return 0;
15 | }

```

11. **[Real-domain expressions]** Compute and print the values of the  $\frac{x}{y}$  and  $\frac{1}{x} + \frac{1}{y}$  in floating point, where  $x, y$  are nonzero real numbers given as input. The value of the 1st expression should be printed up to the 6th decimal place, and that of the 2nd expression up to the 3rd decimal place. For example, if  $x = 2$  and  $y = 3$ , then the printed values should be 0.666667 and 0.833, respectively.

```

1 | #include <stdio.h>
2 |
3 | int main(){
4 |     float x, y, z;
5 |
6 |     printf("Enter the values of x and y: ");
7 |     scanf("%f%f", &x, &y);
8 |
9 |     printf("x/y = %f, 1/x + 1/y = %0.3f\n", x/y, 1/x + 1/y);
10 |
11 |     return 0;
12 | }

```

12. **[Average]** Read in three integers and print their average as a real number.

```

1 | #include <stdio.h>
2 |
3 | int main(){
4 |     int a, b, c;
5 |     printf("Enter three integers: ");
6 |     scanf("%d%d%d", &a, &b, &c);
7 |     printf("Average = %f\n", ((float)(a+b+c))/3);
8 |     return 0;
9 | }

```

13. **[Area of a triangle]** Read in the coordinates (as double-precision real numbers) of three points on  $xy$ -plane, and print the area of the triangle formed by them. You can use the *Heron's formula*  $\sqrt{s(s-a)(s-b)(s-c)}$ , where  $s = \frac{1}{2}(a+b+c)$ , and  $a, b, c$  denote the lengths of three sides. You can use `sqrt` function of `math.h` library to compute the square root, but should not use any other function.

```

1 | #include <stdio.h>
2 | #include <math.h>
3 |
4 | int main(){

```

```

5 | double area, x1, y1, x2, y2, x3, y3, a, b, c, s;
6 |
7 | printf("Enter the coordinates of 1st vertex: ");
8 | scanf("%lf%lf", &x1, &y1); // read in double precision
9 | printf("Enter the coordinates of 2nd vertex: ");
10 | scanf("%lf%lf", &x2, &y2); // read in double precision
11 | printf("Enter the coordinates of 3rd vertex: ");
12 | scanf("%lf%lf", &x3, &y3); // read in double precision
13 |
14 | a = sqrt((x2-x3)*(x2-x3) + (y2-y3)*(y2-y3));
15 | b = sqrt((x1-x3)*(x1-x3) + (y1-y3)*(y1-y3));
16 | c = sqrt((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2));
17 |
18 | s = (a+b+c)/2;
19 | area = sqrt(s*(s-a)*(s-b)*(s-c));
20 | printf("Area = %f\n", area);
21 |
22 | return 0;
23 | }

```

14. **[Compound interest]** Read in the principal amount  $P$ , the interest rate  $R$  in percentage, the number of years  $N$ , and print the compound interest  $C$  earned after  $N$  years. Read  $P$  and  $R$  as floating-point numbers, and  $N$  as an integer. The compound interest  $C$  should be printed as the nearest whole number. For example, if  $P = 100$ ,  $R = 10$ ,  $N = 7$ , then the value of  $C$  is Rs. 94.871712, which should be printed as Rs. 95.

You can use `pow` function of `math.h`.

```

1 | #include <stdio.h>
2 | #include <math.h>
3 |
4 | int main(){
5 |     float P, R, C;
6 |     int N;
7 |     printf("Enter P: ");
8 |     scanf("%f", &P);
9 |     printf("Enter R: ");
10 |    scanf("%f", &R);
11 |    printf("Enter N: ");
12 |    scanf("%d", &N);
13 |
14 |    C = P*pow((double)(1+R/100.0), (double)N) - P;
15 |    printf("Compound interest = Rs. %0.0f\n", C);
16 |
17 |    return 0;
18 | }

```

15. **[One-variable logical expressions]** Compute and print the logical value of `!a`, where `a` is any integer given as input. You should check (and know) that `!a = 1` if and only if `a = 0`. That is, `!a = 1` when `a = 0`, and `!a = 0` when `a = 1, -1, 2, -2, ...`

```

1 | #include <stdio.h>
2 |
3 | int main(){

```

```

4 |   int a;
5 |
6 |   printf("Enter the value of a: ");
7 |   scanf("%d", &a);
8 |
9 |   printf("Logical value of !a = %d\n", !a);
10 |
11 |   return 0;
12 | }

```

16. [Multi-variable logical expressions] Compute and print the logical values of the following expressions, where *a*, *b*, *c* are the integers given as input.

```

(!a) && (!b) && c
(a && b) || (!c)

```

```

1 | #include <stdio.h>
2 |
3 | int main(){
4 |     int a, b, c;
5 |
6 |     printf("Enter the values of a, b, c: ");
7 |     scanf("%d%d%d", &a, &b, &c);
8 |
9 |     printf("Logical value of (!a) && (!b) && c = %d\n", (!a) && (!b) && c);
10 |    printf("Logical value of (a && b) || (!c) = %d\n", (a && b) || (!c));
11 |
12 |    return 0;
13 | }

```

17. [Sum of series] Without using any loop, compute and print the value of  $\sum_{i=1}^n i$ , where *n* is a positive integer given as input.

```

1 | #include <stdio.h>
2 |
3 | int main(){
4 |     int n;
5 |
6 |     printf("Enter the value of n: ");
7 |     scanf("%d", &n);
8 |     printf("Sum = %d.\n", n*(n+1)/2);
9 |
10 |    return 0;
11 | }

```

18. [Number of digits] Read in a positive integer *n*. Assume that it has at most 4 digits. Without using any conditional such as **if-else** or **switch-case**, print its number of digits. Also, print the expression you have used to get the answer.

```

1 | #include <stdio.h>
2 |
3 | int main(){

```

```

4 |     int n;
5 |     printf("Enter the value of n: ");
6 |     scanf("%d", &n);
7 |     printf("Number of digits in n = %d\n", 1 + (n>=10) + (n>=100) + (n>=1000));
8 |     printf("Expression used: 1 + (n>=10) + (n>=100) + (n>=1000)\n");
9 |     return 0;
10| }

```

## 2.10 Exercise problems

1. **[Multi-variable logical expressions]** Compute and print the logical values of the following expressions, where  $a$ ,  $b$ ,  $c$  are the integers given as input.

```

a && (b || c)
a && ((b || c)==0)
!(a && (b || c))
(a && (b || c)) == 0
(a && (b || c)) == 1
(!a) && (!(b || c))
((a == 1) || ((b == 0) && (c == 1))) || ((a == 0) || ((b == 1) && (c == 0)))
((0 <= a) && (a >= 10) && (11 <= b) && (b <= 20) && (15 <= a+b) && (c >= 5))

```

2. **[Right shift]** Given two integers  $a$  and  $b$  as input, compute and print the value of  $\lfloor \frac{a}{2} \rfloor + \lfloor \frac{b}{4} \rfloor$  without using division. (Observe that  $\lfloor \frac{a}{2} \rfloor$  and  $\lfloor \frac{b}{4} \rfloor$  are the respective quotients obtained when  $a$  and  $b$  are divided by 2 and 4 respectively.)
3. **[Integer-to-real map]** Compute and print the values of the following expressions in floating point (rounded off to 3rd decimal place), where  $a, b, c$  are integers given as input. You can use the math library.

$$a + 2b + 3c, \quad \frac{a}{2} + \frac{b}{2} + \frac{c}{3}, \quad (1.25a + 2.75a^2 + 5.625a^3)(b^2/(1.25 + c^3)), \quad \sqrt{2}a + \sqrt{3}b^2 + \left(\frac{5}{4}\right)^{\frac{1}{\sqrt{5}}} c^3.$$

As the values should be real, the computations should be in the real domain. For example, if  $a = 3, b = 2, c = 1$ , then the value of  $\frac{a}{2} + \frac{b}{2} + \frac{c}{3}$  would be  $1.500 + 1.000 + 0.333 = 2.833$ .

4. **[Largest fraction]** Given as input six positive integers  $a, b, c, d, e, f$ , find a/the largest among  $\frac{a}{d}, \frac{b}{e}, \frac{c}{f}$ , using only integer computations.
5. **[Sum of series]** Without using any loop, compute and print the values of the following sums, where  $n$  is a positive integer given as input.

$$\sum_{i=1}^n i^2, \quad \sum_{i=1}^n i^3, \quad \sum_{i=1}^n (i+1)(i+2)(i+3).$$



## 3 | Conditionals

**Conditionals** or **conditional expressions** are basically expressions used in codes for taking required decisions. They have two possible constructs: (i) **if** or **if-else** and (ii) **switch-case**. The **if** construct means if some condition is true, then do something. The **if-else** construct means if some condition is true, then do something; otherwise do whatever is mentioned after **else**. The **if-else** construct can be extended to build a logical chain of **if-else**, but should be carefully coded. The two curly braces, i.e., **{** and **}**, are used to fix the logic.

Complications in the coding logic are often handled using **switch-case** or a combination of **if-else** and **switch-case** constructs. In the **switch-case** construct, out of multiple cases, every particular case is handled based on the value of a single expression in the argument of **switch**. The value of that expression is computed only once and it is compared with the value of each **case** one by one. If there is a match, then the block of code associated with only that **case** is executed, rest are not. The **break** statement breaks out of the entire **switch** block. The **default** statement is optional, and its code is executed only if there is no match with any **case** in that **switch** block.

### Examples

1. Suppose **weekday** is an integer in  $[0, 6]$ , where 0 represents Sunday, 1 represents Monday, and so on. Suppose that on Sunday, everyone wants to enjoy a movie. Its construct will be:

```
if (!weekday)
    printf("Go for a movie!\n");
```

Now, suppose that movies are restricted other than on Sunday. Then its construct will be:

```
if (weekday)
    printf("Try to avoid movies!\n");
```

Now, suppose that movies are prescribed on Sunday but music is prescribed for everyday. Then its construct will be:

```
if (!weekday)
    printf("Go for a movie or listen to music!\n");
else
    printf("Avoid movie and listen to music!\n");
```

2. To complicate the above example, suppose that you need to write a code that will suggest for seeing a movie on Sunday only, playing some sports on Monday, Wednesday, and Friday, going to a restaurant on Tuesday, and visiting the library on Thursday and Saturday. Since there are several cases, they are a little difficult to code using the **if-else** construct, but quite easier using **switch-case**, as shown below.

```
switch (weekday){
    case 0:{
        printf("It's Sunday! Go for a movie!\n");
        break;
```

```

};
case 1: case 3: case 5:{
    printf("Go for sports!\n");
    break;
};
case 2: {
    printf("Enjoy your favorite food in some restaurant!\n");
    break;
};
default:{ // all other cases
    printf("Visit the library!\n");
    break;
};
} // end switch

```

### 3.1 Solved problems

1. **[Larger fraction]** Given as input four positive integers  $a, b, c, d$ , find a/the larger between  $\frac{a}{c}$  and  $\frac{b}{d}$ , using only integer computations.

```

1 #include <stdio.h>
2
3 int main(){
4     int a, b, c, d;
5
6     printf("Enter the values of a, b, c, d: ");
7     scanf("%d%d%d%d", &a, &b, &c, &d);
8
9     if(a*d < b*c)
10        printf("%d/%d is larger.\n", b, d);
11    else
12        printf("%d/%d is larger.\n", a, c);
13
14    return 0;
15 }
16
17 /* Examples:
18
19 Enter the values of a, b, c, d: 1 2 3 4
20 2/4 is larger.
21
22 Enter the values of a, b, c, d: 4 2 3 5
23 4/3 is larger. */

```

2. **[Prime digits]** User supplies a positive integer having value less than 100. Find and print its prime digits if any. For example, in 47, the prime digit is 7.

```

1 #include <stdio.h>
2
3 int main(){
4     int n, a, b, k = 0; // k = no. of primes

```


```

5
6 printf("Enter the value of n: ");
7 scanf("%d", &n);
8
9 a = n/10;
10 b = n - 10*a;
11
12 printf("Prime digits:");
13
14 if ((a==2) || (a==3) || (a==5) || (a==7)){
15     k++;
16     printf(" %d", a);
17 }
18
19 if ((b==2) || (b==3) || (b==5) || (b==7)){
20     k++;
21     printf(" %d", b);
22 }
23
24 if(k>0)
25     printf(".\n");
26 else
27     printf("None.\n");
28
29 return 0;
30 }

```

3. **[Line intersection]** There are two straight lines  $ax + by + c = 0$  and  $px + qy + r = 0$ , where  $a, b, c, p, q, r$  are all integers and given as input. (a) Check whether they are parallel, and if not, (b) find their point of intersection. Its coordinates should be printed up to the 3rd decimal place (using `%0.3f` instead of `%f` in `printf`). 2024S

```


1 #include<stdio.h>
2
3 int main(){
4     int a, b, c, p, q, r;
5     float x, y;
6
7     printf("Enter a, b, c: ");
8     scanf("%d%d%d", &a, &b, &c);
9     printf("Enter p, q, r: ");
10    scanf("%d%d%d", &p, &q, &r);
11
12    if(q*a == b*p)
13        printf("Lines are parallel.\n");
14    else{
15        x = (float)(-q*c+b*r)/(q*a-b*p);
16        y = (float)(-p*c+a*r)/(p*b-a*q);
17        printf("Point of intersection = (%0.3f, %0.3f)\n", x, y);
18    }
19
20    return 0;
21 }

```

4. **[One is sum of two]** Read in three integers and print a message if any one of them is equal to the sum of the other two.

```

1 #include <stdio.h>
2
3 int main() {
4     int a, b, c;
5
6     printf("Enter three integers: ");
7     scanf("%d %d %d", &a, &b, &c);
8
9     if (a == b + c) {
10        printf("a is equal to the sum of b and c.\n");
11    } else if (b == a + c) {
12        printf("b is equal to the sum of a and c.\n");
13    } else if (c == a + b) {
14        printf("c is equal to the sum of a and b.\n");
15    } else {
16        printf("None of the integers is equal to the sum of the other two.\n");
17    }
18
19    return 0;
20 }
```

5. **[Roots of quadratic equation]** Read in the coefficients  $a, b, c$  of the quadratic equation  $ax^2 + bx + c = 0$ , and print its roots nicely. For complex roots, print in  $x + iy$  form.

```

1 #include <stdio.h>
2 #include <math.h>
3
4 int main() {
5     float a, b, c;
6     float discriminant, realPart, imaginaryPart, root1, root2;
7
8     // Read coefficients a, b, c
9     printf("Enter coefficients a, b, and c: ");
10    scanf("%f %f %f", &a, &b, &c);
11
12    // Calculate the discriminant
13    discriminant = b*b - 4*a*c;
14
15    // Check for real and different roots
16    if (discriminant > 0) {
17        root1 = (-b + sqrt(discriminant)) / (2*a);
18        root2 = (-b - sqrt(discriminant)) / (2*a);
19        printf("Root 1 = %.2f\n", root1);
20        printf("Root 2 = %.2f\n", root2);
21    }
22    // Check for real and equal roots
23    else if (discriminant == 0) {
24        root1 = root2 = -b / (2*a);
25        printf("Root 1 = Root 2 = %.2f\n", root1);
26    }
27 }
```

```
27 // If roots are complex
28 else {
29     realPart = -b / (2*a);
30     imaginaryPart = sqrt(-discriminant) / (2*a);
31     printf("Root 1 = %.2f + %.2fi\n", realPart, imaginaryPart);
32     printf("Root 2 = %.2f - %.2fi\n", realPart, imaginaryPart);
33 }
34
35 return 0;
36 }
```

6. **[Operation selection]** Create a simple calculator program that takes two numbers and an operator (+, -, \*, /, %) as input and performs the corresponding operation using a `switch-case` statement.

```
1 #include <stdio.h>
2
3 int main() {
4     char operator;
5     float num1, num2, result;
6
7     // Read operator and two numbers
8     printf("Enter operator (+, -, *, /, %c): ", '%');
9     scanf(" %c", &operator);
10    printf("Enter two numbers: ");
11    scanf("%f%f", &num1, &num2);
12
13    // Perform operation based on the operator
14    switch (operator) {
15        case '+':
16            result = num1 + num2;
17            break;
18        case '-':
19            result = num1 - num2;
20            break;
21        case '*':
22            result = num1 * num2;
23            break;
24        case '/':
25            if (num2 != 0)
26                result = num1 / num2;
27            else {
28                printf("Error! Division by zero.\n");
29                return 1;
30            }
31            break;
32        case '%':
33            if ((int)num2 != 0)
34                result = (int)num1 % (int)num2;
35            else {
36                printf("Error! Division by zero.\n");
37                return 1;
38            }
39            break;
40    default:
```

```

41     printf("Invalid operator!\n");
42     return 1;
43 }
44
45 printf("Result: %.2f\n", result);
46 return 0;
47 }

```

#### Mind the gap while scanning with %c

While coding in C, you may have experienced that scanning a single character using the `%c` specifier can cause annoying issues during runtime, even though there are no compilation errors. Specifically, `scanf("%c", &choice);` can lead to unexpected behavior, while `scanf(" %c", &choice);` works correctly. Let's explore why this happens.

When you read input using `scanf`, it may leave a newline character in the input buffer from previous input, such as when the user presses `ENTER` after entering the character `y`. If you do not include a space before `%c`, `scanf` will interpret that `ENTER` (i.e., `\n`) as the next input for the `char` variable, resulting in unintended behavior.

The space before `%c` in the `scanf` function is used to ignore **any** leading whitespace characters, including spaces, tabs, and newline characters. This ensures that the user's intended input is correctly identified.

7. [Vowel or consonant] Develop a program that takes a single character as input and determines whether it is a vowel or a consonant using a `switch-case` statement.

```

1  #include <stdio.h>
2
3  int main() {
4      char ch;
5
6      printf("Enter a character: ");
7      scanf(" %c", &ch);
8
9      switch (ch) {
10         case 'a':
11         case 'e':
12         case 'i':
13         case 'o':
14         case 'u':
15         case 'A':
16         case 'E':
17         case 'I':
18         case 'O':
19         case 'U':
20             printf("%c is a vowel.\n", ch);
21             break;
22         default:
23             printf("%c is a consonant.\n", ch);
24     }
25
26     return 0;
27 }

```

8. [Day of the week] Write a program that takes the day of the week as an input (1 for Monday, 2 for Tuesday, etc.). If the day is Saturday or Sunday, print `Weekend`. For other days, print the name of

the day. Additionally, check if the day is a special day (e.g., Wednesday) using `if-else`, and print an appropriate message.

```

1 #include <stdio.h>
2
3 int main() {
4     int day;
5
6     printf("Enter day of the week (1 for Monday, 7 for Sunday): ");
7     scanf("%d", &day);
8
9     switch (day) {
10        case 1:
11            printf("Monday\n");
12            break;
13        case 2:
14            printf("Tuesday\n");
15            break;
16        case 3:
17            printf("Wednesday\n");
18            break;
19        case 4:
20            printf("Thursday\n");
21            break;
22        case 5:
23            printf("Friday\n");
24            break;
25        case 6:
26        case 7:
27            printf("Weekend\n");
28            return 0;
29        default:
30            printf("Invalid day\n");
31            return 0;
32    }
33
34    // Additional check for special day
35    if (day == 3) {
36        printf("It's Wednesday, halfway through the week!\n");
37    } else if (day >= 1 && day <= 5) {
38        printf("It's a weekday.\n");
39    }
40
41    return 0;
42 }
```

9. [Marks to grade] Suppose that you have to print the grade of a student, with 90–100 marks getting EX, 80–89 getting A, 70–79 getting B, 60–69 getting C, 50–59 getting D, 40–49 getting P, and less than 40 getting F. Read in the marks of a student and print his/her grade. You should use `switch-case`.

```

1 #include <stdio.h>
2
3 int main() {
```

```
4   int marks;
5   char grade;
6
7   // Read the student's marks
8   printf("Enter the student's marks: ");
9   scanf("%d", &marks);
10
11  // Determine the grade based on marks
12  switch (marks / 10) {
13      case 10: // Handles marks 100
14          case 9:
15              grade = 'E'; // For EX
16              break;
17          case 8:
18              grade = 'A';
19              break;
20          case 7:
21              grade = 'B';
22              break;
23          case 6:
24              grade = 'C';
25              break;
26          case 5:
27              grade = 'D';
28              break;
29          case 4:
30              grade = 'P';
31              break;
32          default:
33              grade = 'F';
34      }
35
36  // Print the corresponding grade
37  if (grade == 'E') {
38      printf("Grade: EX\n");
39  } else {
40      printf("Grade: %c\n", grade);
41  }
42
43  return 0;
44 }
```

## 3.2 Exercise problems

1. **[Largest fraction]** Given as input six positive integers  $a, b, c, d, e, f$ , find a/the largest among  $\frac{a}{d}, \frac{b}{e}, \frac{c}{f}$ , using only integer computations.
2. **[Largest element]** Given as input five numbers  $a, b, c, d, e$ , find a/the largest among them using at most four comparisons. You can use an extra variable.
3. **[Temperature alert system]** Create a program that takes the current temperature as input and provides a message based on the following conditions:



- Above 100°F: "Heat Alert!"
- 85°F to 100°F: "Warm Weather"
- 60°F to 84°F: "Pleasant Weather"
- 32°F to 59°F: "Cool Weather"
- Below 32°F: "Cold Alert!"

Use `if-else` to evaluate the temperature and display the corresponding message.

4. [**Traffic light simulation**] Create a program that simulates a traffic light. The program should take a character input (`'r'` or `'R'` for Red, `'y'` or `'Y'` for Yellow, `'g'` or `'G'` for Green) and display a message indicating whether to stop, slow down, or go. Use a `switch-case` to handle the different light colors.

## 4 | Loops

Loops are needed to perform repeated tasks of same nature. For example, to compute the value of  $1 + \sqrt{2} + \sqrt{3} + \dots + \sqrt{99}$ , we have to compute the square root of a number, 98 times, and so a loop will be useful. A loop essentially executes a block of code as long as a specified expression is true. The specified expression is treated as a **Boolean condition**.

Loops are of three types: **while** loop, **do-while** loop, and **for** loop. They are all equivalent in the sense that one can be used in place of another, but sometimes one is a little more convenient than the others while writing the code.

### 4.1 Syntax of **while** loop and **do-while** loop

The general syntax of the **while** loop and the **do-while** loop are as follows. As mentioned above, the **expression** written inside **while(...)** is treated as a Boolean condition. A loop iterates till the condition is true.

```
// declare and initialize variables
while (expression){
    // code block to be executed
}
```

```
// declare and initialize variables
do{
    // code block to be executed
} while (expression);
```

The above two loops for the specific problem of computing the value of  $1 + \sqrt{2} + \sqrt{3} + \dots + \sqrt{99}$  are written below. Notice that the condition for either of these two loops is  **$i < 100$** , which is just an expression (as explained earlier in Section 2.5). The value of this expression is nonzero (i.e., true) if and only if  **$i$**  is less than 100. In every iteration, the value of  **$i$**  is increased by unity. So, here the variable  **$i$**  acts as the **loop variable** and determines how many times the loop will iterate. Check that in this example, the loop iterates for 98 times; and when the value of  **$i$**  becomes 100, it terminates. This happens for both the **while** loop and the **do-while** loop.

```
int i = 1;
double sum = 1.0;
while (i < 100){
    i++;
    sum += sqrt((double)i);
}
```

```
int i = 1;
double sum = 1.0;
do{
    i++;
    sum += sqrt((double)i);
} while (i < 100);
```

## 4.2 Syntax of for loop

The general syntax of a `for` loop is given below. Here, `expression 2` is treated as a Boolean condition. `expression 1` is used for initialization of the loop variable and of other variables if needed, while `expression 3` is usually used for modifying the value of the loop variable. During each iteration, `expression 3` is evaluated first, followed by `expression 2`. None of these three expressions is mandatory; in fact, `for( ; ; )` is also a valid syntax.

```
// declare and initialize variables if any
for (expression 1; expression 2; expression 3){
    // code block to be executed
}
```

The `for` loop to compute  $1 + \sqrt{2} + \sqrt{3} + \dots + \sqrt{99}$  is written below. Notice that the condition here is `i < 100`, which is just same as in the `while` and the `do-while` loops. So, here also the variable `i` acts here as the loop variable. Check that this loop iterates 98 times, as in the case of `while` or `do-while` loop.

```
int i;
double sum;
for (i=2, sum=1.0; i < 100; i++){
    sum += sqrt((double)i);
}
```

With the 1st expression empty in `for(...)`, we can rewrite the code as follows.

```
int i = 2;
double sum = 1.0;
for ( ; i < 100; i++){
    sum += sqrt((double)i);
}
```

With all three expressions empty, we can rewrite the code as follows.

```
int i = 2;
double sum = 1.0;
for ( ; ; ){
    sum += sqrt((double)i);
    i++;
    if(i==100)
        break;
}
```

All three are correct, but it is a bad practice to write a `for` loop as the last one.

## 4.3 break and continue

A `break` statement makes the program immediately exit from a `while` loop, `do-while` loop, `for` loop, or a `switch-case` block. Its use in `switch-case` is already shown in Chapter 3. On encountering a `break`, the program execution resumes with the next statement following the loop or `switch-case`. Note that the `break` statement is not applicable within `if` or `if-else` constructs.

A `continue` statement makes the program skip the remaining statements in the body of a `while`, `for`, or `do-while` loop in the ongoing iteration. Afterward, program execution proceeds with the next iteration of the loop. The `continue` statement is not used in `switch-case`.

A `break` statement or `continue` statement has to be used judiciously, when it is really needed. The logical flow of the program should be well-understood for using such statements.

**Examples:** The following code finds the smallest number  $n$  such that  $n!$  exceeds 1000. See how it uses a `break` statement.

```
int fact = 1, n = 1;
while(1){
    fact = fact * n;
    if (fact > 1000){
        printf ("Factorial of %d exceeds 1000", n)
        break; // breaks the while loop
    }
    i++ ;
}
```

The following code goes on adding positive integers until a 0 is encountered, ignoring all negative numbers that appear in between. See how it uses `break` and `continue` statements.

```
int sum = 0, next;
while (1){
    scanf("%d", &next);
    if (next < 0)
        continue; // scans the next element
    if (next == 0)
        break; // breaks the while loop
    sum = sum + next;
}
printf ("Sum = %d\n", sum) ;
```

## 4.4 Nested loops

Nested loops are loops that operate within another loop, allowing for more complex iteration patterns. In a nested loop structure, the inner loop completes all its iterations for each iteration of the outer loop. This means that the inner loop's execution depends on the number of iterations specified by the outer loop. For example, if the outer loop runs  $m$  times and the inner loop runs  $n$  times, then the total number of iterations will be  $mn$ . Nested loops are often used for tasks that require multi-dimensional iteration, such as traversing a 2D array or matrix, where the outer loop handles the rows and the inner loop handles the columns. While nested loops can be powerful, they can also lead to significant increase in computational complexity, particularly if the loops are deeply nested or the iteration ranges are large. Therefore, it is important to manage them carefully to avoid performance issues.

The following program prints a multiplication table for numbers from 10 to 20. It uses nested loops to generate the table.

```
#include <stdio.h>
int main(){
    int i, j;
    for (i = 10; i <= 20; i++){ // outer loop
        for (j = 10; j <= 20; j++){ // inner loop
            printf("%3d\t", i * j);
        }
        printf("\n");
    }
    return 0;
}
```

In the outer loop, the variable `i` iterates from 10 to 20, representing the rows of the multiplication table. In the inner loop, for each iteration of the outer loop, the variable `j` iterates from 10 to 20, representing

the columns. After the multiplication, the product of `i` and `j` is calculated and printed, with each value separated by a tab (`\t`) to format the table neatly. After each row is printed, the program moves to the next line using `\n`. The output looks as follows.

100	110	120	130	140	150	160	170	180	190	200
110	121	132	143	154	165	176	187	198	209	220
120	132	144	156	168	180	192	204	216	228	240
130	143	156	169	182	195	208	221	234	247	260
140	154	168	182	196	210	224	238	252	266	280
150	165	180	195	210	225	240	255	270	285	300
160	176	192	208	224	240	256	272	288	304	320
170	187	204	221	238	255	272	289	306	323	340
180	198	216	234	252	270	288	306	324	342	360
190	209	228	247	266	285	304	323	342	361	380
200	220	240	260	280	300	320	340	360	380	400

## 4.5 Solved problems

1. **[ASCII of English alphabet]** Print as integers (in decimal number system) the ASCII values of 'a' to 'z' and those of 'A' to 'Z'. Don't directly use in your code the value of any character.

```

1 #include <stdio.h>
2
3 int main(){
4
5     for (int i=0; i<26; i++)
6         printf("%c: %3d\t ", 'a'+i, 'a'+i);
7     printf("\n");
8
9     for (int i=0; i<26; i++)
10        printf("%c: %3d\t ", 'A'+i, 'A'+i);
11    printf("\n");
12
13    return 0;
14 }
```

See that in the expression 'a'+i, 'a' is a character, whereas i is an integer. The operator + is used to add the ASCII value (which is always an integer) of 'a' with the value of i. Consequently, the value of 'a'+i is also an integer. The same holds for the expression 'A'+i as well. The overall output will be as follows:

```

a:  97  b:  98  c:  99  d: 100  e: 101  f: 102  g: 103  h: 104  i: 105
j: 106  k: 107  l: 108  m: 109  n: 110  o: 111  p: 112  q: 113  r: 114
s: 115  t: 116  u: 117  v: 118  w: 119  x: 120  y: 121  z: 122

A:  65  B:  66  C:  67  D:  68  E:  69  F:  70  G:  71  H:  72  I:  73
J:  74  K:  75  L:  76  M:  77  N:  78  O:  79  P:  80  Q:  81  R:  82
S:  83  T:  84  U:  85  V:  86  W:  87  X:  88  Y:  89  Z:  90
```

2. **[Sum]** Given  $n$  real numbers from the keyboard, find their sum. User input is  $n$  and the numbers.

```

1 #include <stdio.h>
2
3 int main(){
4     int i, n;
5     float x, sum;
6
7     printf("Enter n: ");
8     scanf("%d", &n);
9
10    printf("Enter the numbers: ");
11    for (i=0, sum=0.0; i<n; i++){
12        scanf("%f", &x);
13        sum += x;
14    }
15
16    printf("Sum = %f\n", sum);
17
18    return 0;
19 }
```

3. **[Reading characters]** Read in characters until the `n` character is typed. Count and print the number of lowercase letters, the number of uppercase letters, and the number of digits entered.

```

1 #include <stdio.h>
2
3 int main() {
4     char ch;
5     int lower_count = 0, upper_count = 0, digit_count = 0;
6
7     while ((ch = getchar()) != '\n') {
8         if (ch >= 'a' && ch <= 'z') {
9             lower_count++;
10        } else if (ch >= 'A' && ch <= 'Z') {
11            upper_count++;
12        } else if (ch >= '0' && ch <= '9') {
13            digit_count++;
14        }
15    }
16
17    printf("Lowercase letters: %d\n", lower_count);
18    printf("Uppercase letters: %d\n", upper_count);
19    printf("Digits: %d\n", digit_count);
20
21    return 0;
22 }
```

4. **[5th-power sum]** Given a positive integer  $n$ , compute the value of  $\sum_{k=1}^n k^5$ , without using `math.h`.

```

1 #include <stdio.h>
2
3 int main(){
4     int k, n;
5     long long int sum;
6
7     printf("Enter n: ");
8     scanf("%d", &n);
9
10    for (k=1, sum=0; k<=n; k++){
11        sum += k*k*k*k*k;
12    }
13
14    printf("Sum = %lld\n", sum);
15
16    return 0;
17 }
```

5. **[Bit-length]** Given a positive integer  $n$ , determine the integer  $\ell$  such that  $2^{\ell-1} \leq n < 2^\ell$ . Here,  $\ell$  is called the *bit-length* of  $n$ .

```

1 #include <stdio.h>
2
3 int main(){
4     int n, i=1, len=0;
5 }
```

```

6 | printf("Enter an integer: ");
7 | scanf("%d", &n);
8 |
9 | while(i<=n){
10 |     i <= 1; len++;
11 | }
12 | printf("Bit-length = %d\n", len);
13 |
14 | return 0;
15 | }

```

6. [Decimal to binary, 8 bits] Given an integer  $n \in [0, 255]$ , print its 8-bit binary representation. You can use at most 3 integer variables and no character variables.

```

1 | #include <stdio.h>
2 |
3 | int main(){
4 |     int n, i, j;
5 |
6 |     printf("Enter an integer in [0,255]: ");
7 |     scanf("%d", &n);
8 |     printf("Binary: ");
9 |
10 |    for (j=128; j>0; j/=2){
11 |        if(n/j == 1)
12 |            printf("1"),
13 |            n-=j;
14 |        else
15 |            printf("0");
16 |    }
17 |    printf("\n");
18 |
19 |    return 0;
20 | }

```

7. [GCD] Given two positive integers  $a, b$ , compute their GCD. Use the fact that if  $a \leq b$ , then

$$\gcd(a, b) = \begin{cases} b & \text{if } a = 0 \\ \gcd(b \bmod a, a) & \text{otherwise.} \end{cases} \quad (4.1)$$

Since no function other than `main()` is allowed in this section, you have to do it iteratively using a loop.

```

1 | #include <stdio.h>
2 |
3 | int main(){
4 |     int a, b, c;
5 |     printf("\nEnter two positive integers: ");
6 |     scanf("%d%d", &a, &b);
7 |     while(a!=0){
8 |         c = a;
9 |         a = b%a;
10 |        b = c;
11 |        printf("a b c = %7d %7d %7d\n", a, b, c);
12 |    }
13 |    printf("GCD = %d\n\n", b);
14 |    return 0;
15 | }

```



8. [ $e^x$  as a finite sum] We know that in the infinite sum  $e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$ , the trailing terms have negligible values. Given a positive value of  $x$ , compute the sum up to its  $n$ th term and return that value as the approximate value of  $e^x$  as soon as the  $n$ th term is found to be less than  $\epsilon$ . Also print the value of  $n$ . Needless to say, you should not use the math library.

```

1  /* e^x approximate value */
2
3  #include <stdio.h>
4
5  int main(){
6      double x, epsilon, ex = 1.0, current = 1.0, previous = 0.0;
7      int n = 1;
8
9      printf("Enter x: ");
10     scanf("%lf", &x);
11     printf("Enter epsilon: ");
12     scanf("%lf", &epsilon);
13
14     while(current > epsilon){
15         n++;
16         previous = current;
17         current = previous*x/(double)(n-1);
18         printf("%lf ", current);
19         ex += current;
20     }
21
22     printf("\nApproximate e^x = %lf (up to term %d)\n", ex, n);
23     return 0;
24 }

```

9. [Infinite sum] With  $b \in [3, 7]$  as the only input, do the following.

- (i) Compute and print the values of  $\sum_{i=0}^{\infty} (-1)^i \left(\frac{a}{b}\right)^i$  as fractions in increasing order, for  $a \in [1, b-1]$ . Note that there are  $b-1$  fractions, each one less than 1 because  $a < b$ .
- (ii) Also compute and print the sum of the above fractions, as a simple fraction. A *simple fraction* is one in which the GCD of the numerator and the denominator is 1.

```

1  /* sum = 1/(1+x) = 1/(1+(a/b)) = b/(a+b).*/
2
3  #include <stdio.h>
4
5  int main(){
6      int a, b, p, q, r, s, t;
7      printf("Enter b: ");
8      scanf("%d", &b);
9
10     p = b, q = 2*b-1;
11     printf("%d/%d ", b, 2*b-1);
12
13     for(a=b-2; a>0; a--){
14         printf("%d/%d ", b, a+b);
15         p = p*(a+b)+q*b; q = q*(a+b);
16     }
17 }

```

```

18 | //To make p/q simple, divide p and q by gcd(p,q)
19 | r = p, s = q;
20 | while(r!=0){ // find GCD(r,s)
21 |     t = r;
22 |     r = s%r;
23 |     s = t;
24 | } // GCD = s
25 | printf("\nSum = %d/%d\n", p/s, q/s);
26 | return 1;
27 | }
28 |

```

10. **[Print square]** Print a  $5 \times 5$  square filled up with '\*'.

```

1 #include <stdio.h>
2
3 int main(){
4     const int SIZE = 5;
5     int row, col;
6     for (row = 0; row < SIZE; ++row){
7         for (col = 0; col < SIZE; ++col){
8             printf("* ");
9         }
10        printf("\n");
11    }
12    return 0;
13 }

```

Note that `const` indicates that the value of `SIZE` cannot be modified after it is initialized. It is a constant, so trying to change its value later in the code will result in a compilation error.

The output is shown in Figure 4.1(a).

11. **[Print lower triangle]** Print the lower triangle of a  $5 \times 5$  square filled up with '\*'. The output is shown in Figure 4.1(b).

```

1 #include <stdio.h>
2
3 int main(){
4     const int SIZE = 5;
5     int row, col;
6     for (row = 0; row < SIZE; ++row){
7         for (col = 0; col <= row; ++col){
8             printf("* ");
9         }
10        printf("\n");
11    }
12    return 0;
13 }

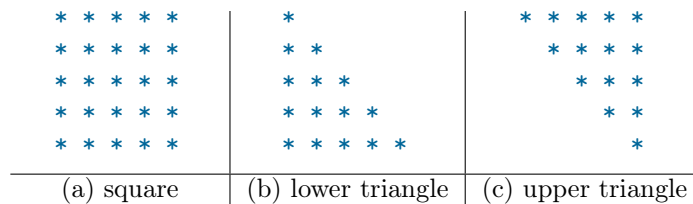
```

12. **[Print upper triangle]** Print the upper triangle of a  $5 \times 5$  square filled up with '\*'. The output is shown in Figure 4.1(c).

```

1 #include <stdio.h>
2
3 int main(){
4     const int SIZE = 5;

```



**Figure 4.1:** Output of the codes for printing a square, its lower triangle, and its upper triangle.

```

5   int row, col;
6   for (row = 0; row < SIZE; ++row){
7       for (col = 1; col <= row; ++col)
8           printf(" ");
9       for (col = 1; col <= SIZE-row; ++col)
10          printf("* ");
11          printf ("\n");
12      }
13      return 0;
14  }
```

## 4.6 Exercise problems

1. [ASCII] Print all the characters with ASCII values from 0 to 127.
2. [max and 2nd max] Read in an integer  $n$ . Then read in  $n$  numbers and print their maximum and second maximum. Assume that all numbers are distinct.
3. [Power sum] Given a positive integer  $n$ , compute  $\sum_{k=1}^n k^k$ , without using `math.h`.
4. [Digit count] Given any integer in decimal number system, compute its number of digits. For example, for 3180, it is 4.
5. [Primes] Find all the prime numbers less than a given positive integer  $n$ .
6. [Co-primes] Two positive integers are said to be co-prime with each other if their GCD is 1. Given a positive integer  $n$ , find the co-primes that are all less than  $n$ . For example, for 9, they are 2, 4, 5, 7, 8.
7. [Two-prime sum] Given a positive integer  $n$ , determine whether it can be written as the sum of two primes.
8. [sin, cos] Using  $x$  as input in the double-precision format, compute and print the values of  $\sin x$  and  $\cos x$  using the following equations.

$$\sin x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1}, \quad \cos x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n}. \quad (4.2)$$

Use `#define DIFF 0.00001` to terminate the summation when the sum up to  $n$ th term differs by less than `DIFF` from the sum up to  $(n-1)$ st term. Needless to say, you should not use the math library.

## 5 | One-dimensional arrays



Suppose that you have lots of storybooks, poetry books, comic books, etc., and also have many other books on different subjects such as Physics, Chemistry, Biology, Mathematics, etc. While keeping them in a bookshelf, you will naturally try to keep them in an organized way so that you can get any book down from the shelf right away, whenever needed. The same idea is used when you have to keep many elements in the computer memory. You have to arrange them in an organized way in the memory so that any of them can immediately be accessed when the code is running. For this arrangement, the concept of data structure comes into use. One such data structure is **array**—it is just analogous to your bookshelf!

### 5.1 What is array?

A **data structure**, often referred to as a **data type** in programming, is a collection of distinct or non-distinct items arranged in a specific order. Common examples of simple data structures include lists, queues, and stacks. Among these, the list is the simplest and can be implemented using an **array** or a **linked list**.<sup>1</sup>

An array is a data structure used to represent and store items of the same data type. This data type can be a basic one such as **char**, **int**, or **float**, or it can be a user-defined data type or structure (e.g., **struct**, which will be discussed later). In fact, the items stored in an array can themselves be arrays!<sup>2</sup>

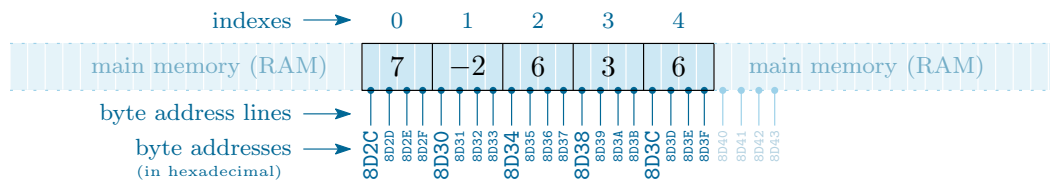
An array is similar to a set. Just as a set is a collection of items of the same type, so too is an array. However, unlike a set, which always contains distinct items, an array does not have this restriction. When a set  $A$  stores  $n$  items, they are typically denoted by  $a_1, a_2, \dots, a_n$ . Similarly, when a one-dimensional array **a[]** contains **n** items, they are represented by **a[0]**, **a[1]**, ..., **a[n-1]**. For any **i** ranging from **0** to **n-1**, **i** is called the **index** of the element **a[i]** in the array **a[]**. That is, the elements **a[0]**, **a[1]**, ..., **a[n-1]** have indexes **0**, **1**, ..., **n-1**, respectively. Figure 5.1 shows an example of an integer array with 5 elements. Note that in this array, the elements at index 2 and index 4 are identical.

### 5.2 Why array?

**First**, the concept of array is useful to store in memory a large number of items of similar data type, using a single declaration. For example, if we need 1000 integers, then we can simply declare **int a[1000]**; the elements here are indexed from 0 to 999 and denoted by **a[0]**, **a[1]**, ..., **a[999]** in the language of C. Had the concept of array not been there, we could have no other way than to denote the 1000 integers by 1000 variables with 1000 different names—something as **int a000**, **a001**, ..., **a999**—a nightmare!

<sup>1</sup>A linked list is implemented using **pointers**. We will explore this concept later in the course.

<sup>2</sup>For example, a **two-dimensional array** is simply a one-dimensional array of one-dimensional arrays. We will study two-dimensional arrays later. In this chapter, the term '**array**' refers specifically to a **one-dimensional** or **linear array**.



**Figure 5.1:** An array with 5 elements having values 7,  $-2$ , 6, 3, 6. Each element is a 4-byte integer, so the total space allocated in the memory for this array is  $4 \times 5 = 20$  bytes. Its first and last byte have the addresses `8D2C` and `8D3F`, respectively. The address of an element is basically the address of its first byte (expressed as a hexadecimal number, a typical convention). So, the respective addresses of the five elements here are `8D2C`, `8D30`, `8D34`, `8D38`, and `8D3C`.

**Second**, the elements of an array occupy contiguous locations in the memory when the executable file is executed. As a result, any element of the array can be accessed directly, just by specifying its index. In particular, `a[i]` gives the element with the index `i` in `a[]`. The direct accessing is possible because the address of the element with index `i` easily follows from the address of the element with index `0`. The simple reason is that all elements have the same data type and hence they all take equal amount of space (measured in bytes) in the memory. Further, every byte in the memory has a unique address, and two contiguous bytes have consecutive addresses. To see this, suppose that the amount of space for each element is `k` bytes. As `&a[0]` is the address of the element with index `0`, the indexes of the subsequent elements are `&a[0]+k` for index `1`, `&a[0]+2*k` for index `2`, `&a[0]+3*k` for index `3`, and so on. That is, the address of `a[i]` will be

$$\&a[i] = \&a[0] + i*k.$$

As an example, see Figure 5.1. Here `a[]` is an integer array and its 1st element `a[0]` has the address `8D2C`; so, its 2nd element `a[1]` has the address `8D2C + 4 = 8D30`, 3rd element `a[2]` will have the address `8D2C + 2*4 = 8D34`, and so on. If `b[]` is an array of characters, then `&b[i] = &b[0] + i`. If the 1st character of `b[]` (i.e., `b[0]`) has the address `AC8F`, then its 2nd character `b[1]` will have the address `AC8F + 1 = AC90`, its 3rd character `b[2]` will have the address `AC8F + 2 = AC91`, and so on.

**Third**, the index need not be just a single variable but can also be an expression whose value is an integer in the interval `[0, n-1]`, where `n` is the number of elements in the array. For example, in an array `a[]` with 100 elements, `a[i+2*j]` is a valid element with `i` and `j` as two integers, as long as the value of the expression `i+2*j` lies in `[0, 99]`.

Another example could be `a[b[i]]`, where the element `b[i]` of an array `b[]` acts as the index of an element in the array `a[]`. Such kinds of indexing are sometimes required while writing codes for tricky problems. For example, see the following code. It takes as input some integers in `[0, 9]`, stores them in an array `a[]`, and then prints them in increasing order, avoiding repetitions. It uses an additional array `b[]`, initialized with all zeros, to count the frequency of each distinct element stored in `a[]`. Since there will be at most 10 distinct elements in `a[]`, the size of `b[]` is set to 10. The trick is that `b[a[i]]` means how many times an element `a[i]` appears in `a[]`.

```

1  #include <stdio.h>
2
3  int main() {
4      int n, i, j;
5      printf("Enter the number of elements: ");
6      scanf("%d", &n);
7      int a[n], b[10];
8      for (i = 0; i < 10; i++)
9          b[i] = 0;
10
11     printf("Enter the elements (0 to 9):\n");

```

```

12     for (i = 0; i < n; i++)
13         scanf("%d", &a[i]);
14
15     for (i = 0; i < n; i++)
16         b[a[i]]++; // the trick
17
18     printf("Sorted array: ");
19     for (i = 0; i < 10; i++) {
20         if (b[i] > 0) // i appears in a[]
21             printf("%d ", i);
22     }
23     printf("\n");
24
25     return 0;
26 }
27 /*
28 Enter the number of elements: 13
29 Enter the elements (0 to 9): 5 4 7 8 4 0 4 4 0 5 7 8 7
30 Sorted array: 0 4 5 7 8
31 */

```

### 5.3 Declaring arrays

Like other variables, an array should be declared first before using it. The general syntax is

```
type array_name[size];
```

where `type` specifies the type of elements (`char`, `int`, `float`, etc.) that will be stored in the array, `array_name` denotes the name of the array, and `size` is the maximum number of elements that can be stored in the array. Here are some examples:

```

int roll_num[100]; // array name is roll_num, can store up to 100 integers
char gender[100]; // array name is gender, can store up to 100 characters
float marks[100]; // array name is marks, can store up to 100 floating-point numbers

```

### 5.4 Initializing arrays

The general form is:

```
type array_name[size] = {comma-separated values};
```

And here are some examples:

```

int roll_num[5] = {31, 32, 33, 34, 35};
char gender[5] = {'F', 'M', 'M', 'F', 'M'};
float marks[10] = {72.5, 83.25, 65.5, 80, 76.25};

```

In the above examples, all three arrays are declared to contain 5 elements each, and their values are initialized. We can also declare them to contain more elements, say 10 each, but only the first five are initialized. Here is how:

```
int roll_num[10] = {31, 32, 33, 34, 35};
char gender[10] = {'F', 'M', 'M', 'F', 'M'};
float marks[10] = {72.5, 83.25, 65.5, 80, 76.25};
```

When you partially initialize an array in C, the uninitialized elements are automatically set to 0 for numeric arrays and '\0' (null character) for character arrays. The non-initialized as well as the initialized values can be changed later if needed. Here's what happens in the above cases:

```
roll_num = {31, 32, 33, 34, 35, 0, 0, 0, 0, 0};
gender = {'F', 'M', 'M', 'F', 'M', '\0', '\0', '\0', '\0', '\0'};
marks = {72.5, 83.25, 65.5, 80.0, 76.25, 0.0, 0.0, 0.0, 0.0, 0.0};
```

## 5.5 Accessing and working with arrays

In order to populate or write to an array, we must use a loop, as follows.

```
int a[5], i;
for (i=0; i<5; i++){ // writing to a[]
    scanf("%d", &a[i]);
}
```

Similarly, to read from the array `a[]`, here is how:

```
int p;
for (i=0; i<5; i++){ // reading from a[]
    p = a[i];
}
```

We cannot use the operator `=` to assign one array to another; i.e., even if `a[]` and `b[]` are two arrays of same type and same size, then `a = b`; or `a[] = b[]`; cannot copy the elements of `b[]` to `a[]` (it will give compilation error). For copying the elements of `b[]` to `a[]`, we have to use a loop as follows:

```
int i;
for (i=0; i<n; i++){ // n = number of elements in a[] and in b[]
    a[i] = b[i];
}
```

As a specific example, suppose that there are at most 100 students in a class. Now, consider the problem of getting their marks and finding how many of them have marks above the average. Clearly, following are there main tasks:

1. Read the marks as input and write them in an array.
2. Compute the total marks.
3. Count how many students meet the given criterion.

Clearly, Task 1 and Task 2 can be done in a single loop. But Task 3 needs an additional loop after that because it can be done only after Task 2 is over. That is, it is impossible to write a code for this problem using a single loop only. (Understand the importance of this observation!) The following code is written based on this observation. Notice that for Task 1, we are *writing into* the array using the indexes, while for Task 2 and Task 3, we are *reading from* the array using the indexes again. In either case, the integer `i` is used as the index variable.

```

1  #include <stdio.h>
2  int main(){
3      int n, i, k;
4      float marks[100], total, avg;
5      printf("Enter the number of students: ");
6      scanf("%d", &n);
7      printf("Enter their marks: ");
8      for (i=0, total=0; i<n; i++){ // Task 1 and Task 2
9          scanf("%f", &marks[i]);
10         total = total + marks[i];
11     }
12     avg = total/n;
13     for (i=0, k=0; i<n; i++){ // Task 3
14         k += (marks[i] > avg) ? 1 : 0;
15     }
16     printf("Number of students above the average = %d\n", k);
17 }

```

## 5.6 Solved problems

Note the following regarding the problems stated in this section.

- i. All arrays here are one-dimensional. The number of elements that an array can store, is referred to as the **size of array**. However, the full array may not be used. For example, the size of an array `a[]` may be 10, but we store and work with only 7 elements from `a[0]` to `a[6]`, whereby `a[7]` to `a[9]` are of no use.
  - ii. Unless mentioned, for an array, assume that its size is at most 1000.
  - iii. Unless mentioned, assume that the input elements need not be distinct.
1. [**max**] Given a positive integer  $n$  as the first input, take in as the next input  $n$  integer elements from the user, store them in an array, and find the largest among them. The total number of comparisons should be at most  $n - 1$ .

```

1  // Find the largest element in an array
2
3  #include <stdio.h>
4  #define SIZE 1000
5
6  int main(){
7      int a[SIZE], n, i, max;
8
9      printf("Enter n: ");
10     scanf("%d", &n);
11     printf("Enter the elements: ");
12
13     for(i=0; i<n; i++)
14         scanf("%d", &a[i]);
15
16     max = a[0];
17
18     for(i=1; i<n; i++)

```



```

19     if (max<a[i])
20         max = a[i];
21
22     printf("Max = %d\n", max);
23     return 0;
24 }

```

2. **[Check distinctness]** Given a positive integer  $n$  as the first input, take in as the next input  $n$  integer elements from the user, store them in an array, and check whether all elements in that array are distinct.

```

1 // Check whether all elements are distinct
2
3 #include <stdio.h>
4 #define SIZE 1000
5
6 int main(){
7     int a[SIZE], n, i, j, flag = 1;
8
9     printf("Enter n: ");
10    scanf("%d", &n);
11    printf("Enter the elements: ");
12
13    for(i=0; i<n; i++)
14        scanf("%d", &a[i]);
15
16    for(i=1; i<n && flag; i++){
17        for(j=0; j<i && flag; j++){
18            if(a[i]==a[j]){
19                flag = 0;
20                printf("Not distinct: a[%d] = a[%d] = %d.\n", j, i, a[i]);
21            }
22        }
23    }
24
25    if(flag)
26        printf("Distinct.\n");
27
28    return 0;
29 }

```

3. **[Fibonacci sequence]** The Fibonacci sequence  $\{f(i) : i = 0, 1, 2, \dots\}$  is defined as

$$f(0) = 0, f(1) = 1, \text{ and } f(i) = f(i-1) + f(i-2) \text{ if } i \geq 2. \quad (5.1)$$

Given a non-negative integer  $n$ , compute  $f(i)$  for  $i = 0, 1, 2, \dots, n$ , store them in an array of  $n + 1$  integers, and print the elements of this array.

```

1 // Fibonacci sequence
2
3 #include <stdio.h>
4 #define SIZE 1000
5
6 int main(){
7     int a[SIZE], n, i;
8

```

```

9   printf("Enter n: ");
10  scanf("%d", &n);
11  a[0] = 0, a[1] = 1;
12
13  for(i=2; i<=n; i++)
14      a[i] = a[i-1] + a[i-2];
15
16  printf("Fibonacci elements: ");
17  for(i=0; i<n; i++)
18      printf("%d ", a[i]);
19  printf("\n");
20
21  return 0;
22 }

```

4. **[Selection Sort: positive elements, extra array]** Given a positive integer  $n$  as the first input, take in as the next input  $n$  positive integers from the user and store them in an array  $A$ . Take another array  $B$  of the same size and put the elements of  $A$  into  $B$  in non-decreasing order. For example, if  $A = [4, 2, 3, 2, 6, 4]$ , then  $B = [2, 2, 3, 4, 4, 6]$ .

```

1 // Selection Sort: positive elements, extra array
2
3 #include <stdio.h>
4 #define SIZE 1000
5
6 int main(){
7     int a[SIZE], b[SIZE], n, i, j, max;
8
9     printf("Enter n: ");
10    scanf("%d", &n);
11    printf("Enter the elements (all positive): ");
12
13    for(i=0; i<n; i++)
14        scanf("%d", &a[i]);
15
16    // max = index of the largest element in a[]
17
18    for(i=max=0; i<n; i++){
19        for(j=0; j<n; j++){
20            if(a[j] > a[max]) // the 1st trick!
21                max = j;
22        }
23        b[n-1-i] = a[max];
24        a[max] = 0; // the 2nd trick!
25    }
26
27    printf("Elements in b[] after sorting a[]: ");
28    for(i=0; i<n; i++)
29        printf("%d ", b[i]);
30    printf("\n");
31
32    return 0;
33 }

```

5. **[Array union]** Given two arrays  $A$  and  $B$  containing  $m$  and  $n$  integer elements respectively, find the

elements in  $A \cup B$ , store them in another array  $C$ , and print  $C$ . Assume that all elements of  $A$  are distinct, and so also for  $B$ , although an element of  $A$  may be present in  $B$ .

```

1 // Array union
2
3 #include <stdio.h>
4 #define SIZE 1000
5
6 int main(){
7     int a[SIZE], b[SIZE], c[2*SIZE], m, n, i, j, k, flag;
8
9     printf("Enter m: ");
10    scanf("%d", &m);
11    printf("Enter the elements of a[]: ");
12    for(i=0; i<m; i++)
13        scanf("%d", &a[i]);
14
15    printf("Enter n: ");
16    scanf("%d", &n);
17    printf("Enter the elements of b[]: ");
18    for(i=0; i<n; i++)
19        scanf("%d", &b[i]);
20
21    // copy a[] to c[]
22    for(i=k=0; i<m; i++, k++)
23        c[k] = a[i];
24
25    // Now copy each element of b[] to c[] if it's not in a[]
26    for(i=0; i<n; i++){
27        flag = 0; // assume that b[i] is not in a[]
28        for(j=0; j<m; j++){
29            if(b[i]==a[j]){ // b[i] is in a[]
30                flag = 1;
31                break;
32            }
33        }
34        if(!flag) // b[i] is not in a[], so copy it to c[]
35            c[k++] = b[i];
36    }
37
38    printf("c[] = ");
39    for(i=0; i<k; i++)
40        printf("%d ", c[i]);
41    printf("\n");
42
43    return 0;
44 }
```

6. [Closest pair in 2D] Given an integer  $n \geq 2$ , followed by a sequence of  $n$  distinct points with real coordinates, determine a pair of closest points. You can store the  $x$ -coordinates and the  $y$ -coordinates in two arrays.

```

1 #include<stdio.h>
2 #include <float.h>
```

```

3
4 int main(){
5     float x[100], y[100]; // assuming that there are at most 100 points
6     int n, i, j, p, q;
7     float dcur, dmin; // we consider the squares of distances; need not use math library
8     dmin = FLT_MAX; // the maximum value of a float
9
10    printf("Enter #points: ");
11    scanf("%d", &n);
12    printf("Enter the coordinates:\n");
13    for(i=0; i<n; i++){
14        printf("%2d: ", i+1);
15        scanf("%f%f", &x[i], &y[i]);
16    }
17
18    for(i=1; i<n; i++){
19        for(j=0; j<i; j++){
20            dcur = (x[i]-x[j])*(x[i]-x[j]) + (y[i]-y[j])*(y[i]-y[j]);
21            if (dcur < dmin)
22                p = i+1, q = j+1, dmin = dcur;
23        }
24    }
25
26    printf("Closest pair = (%d, %d), distance = %0.3f.\n", p, q, dmin);
27    return 0;
28 }

```

## 5.7 Exercise problems

Note the following regarding the problems stated in this section.

- i. All arrays here are one-dimensional. The number of elements that an array can store, is referred to as the **size of array**. However, the full array may not be used. For example, the size of an array  $A[ ]$  may be 10, but we store and work with only 7 elements from  $A[0]$  to  $A[6]$ , whereby  $A[7]$  to  $A[9]$  are of no use.
  - ii. Unless mentioned, for an array, assume that its size is at most 1000.
  - iii. Unless mentioned, assume that the input elements need not be distinct.
1. (**Check if sorted**) Given a positive integer  $n$  as the first input, take in as the next input  $n$  integer elements from the user, store them in an array, and check if they are sorted in non-decreasing order. The total number of comparisons should be at most  $n - 1$ .
  2. (**Largest-sum pair**) Given a positive integer  $n$  as the first input, take in as the next input  $n$  integer elements from the user, store them in an array, and find a pair of elements that add up to the largest value over all pairs. For example, for  $[-2, 5, -1, 3, -1, 2, 3]$ , the largest sum is  $5 + 3 = 8$ . (Hint: It's the pair of max and 2nd max.)
  3. (**Pair sum**) Given a positive integer  $n$  and an integer  $k$ , followed by  $n$  integer elements, store them in an array and determine whether there are two elements with distinct indices in the array such that their sum is  $k$ .
  4. (**Binomial coefficients**) From the following recurrence of binomial coefficients

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \binom{n-1}{k-1} + \binom{n-1}{k}, \quad (5.2)$$

compute the values of  $\binom{n}{k}$  for  $k = 0, 1, \dots, n$ , using the value of  $n$  as input. You should do it with a single array.

5. **(Selection Sort: positive and distinct elements, extra array)** Given a positive integer  $n$  as the first input, take in as the next input  $n$  positive integers from the user and store them in an array  $A$ . Take another array  $B$  of the same size and put the elements of  $A$  into  $B$  so that they are in increasing order and all the elements of  $B$  are distinct. For example, if  $A = [4, 2, 3, 2, 6, 4]$ , then  $B = [2, 3, 4, 6]$ . Since the size of  $B$  is same as that of  $A$ , you have to keep track of the number of elements in  $B$ .
6. **(Selection Sort)** Given a positive integer  $n$  as the first input, take in as the next input  $n$  integers from the user and store them in an array  $A$ . Without using any extra array, rearrange the elements in non-decreasing order. For example, if input is  $A = [4, -2, 3, 0, -2, 6, 4]$ , then the output will be  $A = [-2, -2, 0, 3, 4, 4, 6]$ .
7. **(Array intersection)** Given two arrays  $A$  and  $B$  containing  $m$  and  $n$  integer elements respectively, find the elements in  $A \cap B$ , store them in another array  $C$ , and print  $C$ . Assume that all elements of  $A$  are distinct, and so also for  $B$ .
8. **(Maximum collinearity)** Given an integer  $n \geq 2$ , followed by a sequence of  $n$  distinct points with integer coordinates, determine the maximum number of collinear points. You can store the  $x$ -coordinates and the  $y$ -coordinates in two arrays.



# 6 | Functions

## 6.1 What is function?

A **function** (also known as ‘routine’ or ‘procedure’) is a block of code written to perform a specific computation, such as calculating a mathematical function like  $n!$ ,  $\binom{n}{r}$ ,  $\text{GCD}(a,b)$ ,  $x^y$ , or  $\sin x$ . See, for example, Code 6.1 to compute  $n!$ . In addition to mathematical functions, functions are crucial for solving various computational problems and writing efficient code. For instance, they can be used to enumerate all primes less than a given natural number (see the solution in §6.14).

Here are some key points about functions:

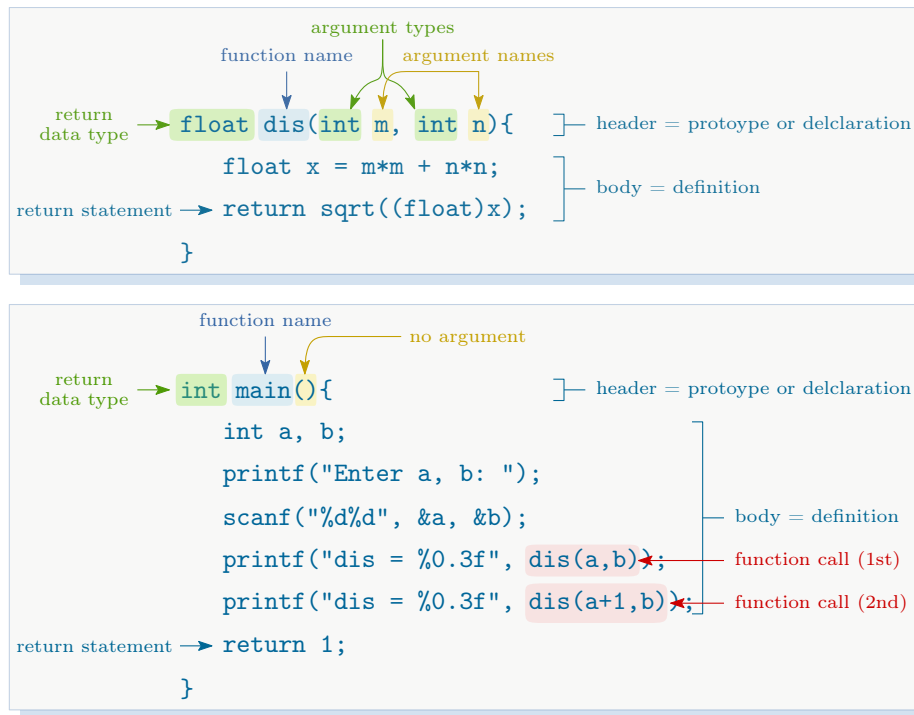
1. Any function other than `main()` runs only when it is called by a function such as `main()` or some other function, including itself. If it calls itself, it is said to be **recursive** (see §6.9).
2. Functions are of two categories: i) **library functions** and ii) **user-defined functions**.
3. A **library** is a collection of predefined functions called **library functions**. C provides a wide range of libraries, including `stdio.h`, `stdlib.h`, `math.h`, and `string.h`, among others. The functions `scanf`, `printf`, `getc`, and `putc` are defined in `stdio.h`. The functions `abs`, `atoi`, `calloc`, and `malloc` are some commonly used functions defined in `stdlib.h`. In the math library `math.h`, we have many predefined mathematical functions such as `sqrt`, `pow`, `sin`, `cos`, `asin`, `abs`, etc. While processing strings, we need to include the library `string.h` to use functions like `strlen`, `strcpy`, `strcmp`, etc.

A detailed list of C library functions can be seen at  
[https://en.wikipedia.org/wiki/C\\_standard\\_library](https://en.wikipedia.org/wiki/C_standard_library) or  
<https://www.ibm.com/docs/en/i/7.3?topic=extensions-standard-c-library-functions-table-by-name>

Code 6.1: Computing  $n!$  using a function named `fact`.

`factorial_iter.c`

```
1 #include <stdio.h>
2
3 int fact(int n) {
4     int result = 1;
5     for (int i = 2; i <= n; i++)
6         result *= i;
7     return result;
8 }
9
10 int main(){
11     int n;
12     printf("Enter n: ");
13     scanf("%d", &n);
14     printf("fact(%d) = %d\n", n, fact(n));
15 }
```



**Figure 6.1:** **Top:** Components of a user-defined function named `dis`. Notice that `dis` calls another function, `sqrt`, which is defined in the math library.

**Bottom:** Calling the function `dis` from `main()`. Here, `dis` is the **called function**, and `main()` the **caller**.

4. A function written by the programmer is called **user-defined function**. It needs to be written when it is not predefined in any standard library. One such example is to compute the number of permutations of  $r$  elements chosen from  $n$  distinct elements, given by the formula  $P(n, r) = \frac{n!}{(n-r)!}$ . We will discuss in §6.4 about its two possible implementations, given in Code 6.2 and Code 6.3.
5. A function may take some input arguments or parameters and may return some value. It may be called multiple times, with same or different arguments. For example, in Figure 6.1, the function `dis(...)` is called twice from the function `main()`. Thus, functions are particularly useful when they need to be called repeatedly. It makes a smart programming—write once, call again and again. They come handy while developing a software in a modular fashion. Large codes become easy to read and easy to debug or revise.

## 6.2 Defining a function

A function **definition** has two parts: **header** and **body** (Figure 6.1). The header carries the function **prototype** or **declaration**, which consists of:

1. return data type, i.e., data type of the variable whose value is returned by the function;
2. name of the function;
3. names and data types of the variables taken as input arguments or parameters.

When no value needs to be returned, the return data type is `void`. When no input argument is required, there is nothing between the opening and the closing brackets; one such example is the function `int main()`, shown in Figure 6.1. Observe that the `main()` in this case calls the user-defined function `float dis(int m, int n)` with two integer variables as arguments.

The header of a function in a code is analogous to the domain of definition of a mathematical function; e.g., for  $n!$ , the domain is the set of positive integers. The body, on the other hand, contains the definition of the function and its returned value, if any; e.g., for  $n!$ , the definition of the function `fact` in Code 6.1 is  $1 \cdot 2 \cdot 3 \cdots n$  and its returned value is an integer. We shall study this again in §6.4.

### 6.3 Execution of a Function

Let's now see what happens when a function is called. Consider the previous example (Figure 6.1). In the first call of the function `dis` from `main()`, the arguments are `a` and `b`, while in the second, they are `a+1` and `b`.

Let's see what happens inside the computer when `a.out` is executed. After compiling the C program to get the binary executable `a.out` and running it, the operating system initiates the execution of `a.out`. This running instance of `a.out`, involving both the CPU and the RAM, is referred to as a **process**.

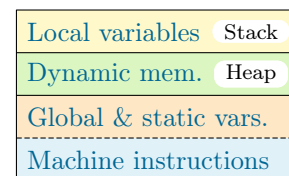
In many cases, a process involves both RAM and the hard disk. While running, its code and data may not always be in RAM but might be stored on the hard disk. Additional elements, such as library files and data files, may also reside on the hard disk. These are loaded into RAM from the hard disk as needed to run the process.

The following steps occur in the aforesaid process:

#### 1. Loading the executable:

- (i) The operating system loads the executable file `a.out` into main memory (RAM).
- (ii) The part of RAM allocated to `a.out` is divided into segments, such as the code segment, data segment, stack, and heap.
- (iii) The code segment contains the machine instructions for the `main()` and `dis()` functions.
- (iv) The data segment holds global and static variables (none in this example).

Memory space for `a.out`



RAM

#### 2. Allocating memory:

- (i) Memory is allocated for the stack and heap.
- (ii) The stack manages function calls, local variables, and return addresses. Here, it includes the return address where the result of `sqrt((float)x)` will be stored.
- (iii) The heap manages function calls, local variables, and return addresses. In this example, it holds the return address to resume execution in `main()` after the `dis()` function completes. The result of `sqrt((float)x)` is initially stored in a CPU register. When the function returns, this value is transferred to a location in the stack frame of `main()`.

**Registers** are located within the CPU chip. A modern CPU typically has 16 to 32 general-purpose registers, along with specialized ones. Unlike RAM, which is slower and larger, registers provide much faster access for the CPU. Hence, they are used to store frequently used values and intermediate results during the execution of a process.

- (iv) The heap is reserved for dynamic memory allocation (not utilized in this example).

#### 3. Initializing execution:

- (i) The entry point of the program, i.e., the first instruction of `main()`, is determined.
- (ii) The CPU begins executing the code from `main()`.

#### 4. Executing `main()`:



- (i) `main()` outputs the prompt "Enter a, b: " to the terminal.
- (ii) It then waits for user input, which is read by `scanf()` and stored in `a` and `b`.
- (iii) The values of `a` and `b` are passed to the `dis()` function, transferring them to the memory locations of `m` and `n`.

#### 5. Calling `dis()`:

- (i) The function `dis()` receives the values of its arguments `m` and `n`.
- (ii) It calculates the distance using the formula  $\sqrt{m^2 + n^2}$ .
- (iii) The result is sent back to `main()`.

#### 6. Printing the result:

- (i) `main()` prints the result of `dis(a, b)` formatted to three decimal places.
- (ii) `main()` then calls `dis()` again with `a+1` and `b`.
- (iii) The result is printed in the same format.

#### 7. Program termination:

- (i) After `main()` has completed all instructions, the program returns `1` (indicating the termination status), and the process concludes.
- (ii) The operating system reclaims the memory allocated to the process and may allocate it to other processes.

## 6.4 Prototype versus Definition

We should carefully distinguish between the terms ‘prototype’ and ‘definition’, as together they constitute a complete function. This distinction is illustrated in Figure 6.1 and elaborated upon in the corresponding discussion. Here, we focus on writing a prototype, which can follow two distinct approaches depending on the interdependencies of the user-defined functions within the code.

Typically, each function is declared and defined before any of its calling functions. In other words, both its header (i.e., prototype) and body (i.e., definition) are written together before any function that calls it. This approach is illustrated earlier in Figure 6.1. See also Code 6.3 for an example of this approach. Consequently, the `main()` function usually appears at the end of the program. This method allows the compiler to identify all function definitions in a single pass through the program file.

However, in cases where functions call each other in an arbitrary manner, it may not be feasible to write all function definitions before `main()`. For instance, consider two functions, `f()` and `g()`, where `main()` calls `f()`, `f()` calls `g()`, and `g()` in turn calls `f()`. This scenario creates a paradox: `g()` must be defined before `f()`, and `f()` must be defined before `g()`. In such scenarios, functions cannot be written in full before `main()`. So, it is effective to write just the prototypes of all functions before `main()`, regardless of the order of calls, and then define the functions after `main()`. This strategy ensures that the compiler is aware of all functions in advance and can efficiently organize the executable file `a.out`. This is referred to as **mutual recursion** or **cyclic recursion**, which is related to **indirect recursion** and discussed in more detail in §6.9.4.

As a specific example, consider the problem of computing the number of permutations of  $r$  elements chosen from  $n$  distinct elements. It is given by the formula  $P(n, r) = n!/(n - r)!$ . Our task is to compute the values of  $P(n, r)$  for  $r \in [1, n]$ , where the value of  $n$  is provided by the user. We discuss two versions of the code to perform this task. In the first version (Code 6.2), the function prototypes are declared separately before `main()`, with their definitions appearing later. In the second version (Code 6.3), the prototypes are not written separately, requiring the functions to be written in full before `main()`. Further, the `fact` function is written before the `npr` function because `npr` calls `fact`. If `fact` were written after `npr`, a compilation error would occur, as a function must be defined before it is called if prototypes aren’t separately provided.

**Code 6.2:** Computing  $P(n, r)$ , with separate declaration of prototypes.

`n_permute_r_ver1.c`

```

1 #include <stdio.h>
2
3 int npr (int n, int r); // prototype of the function npr
4 int fact (int n); // prototype of the function fact
5
6 int main(){
7     int i, n;
8     printf("Enter n: ");
9     scanf ("%d", &n);
10
11     for (i=1; i<=n; i+=1){
12         printf ("%d choose %d = %d \n", i, n, npr(n, i));
13     }
14 }
15
16 int npr(int n, int r){ // definition of the function npr
17     return fact(n)/fact(n-r);
18 }
19
20 int fact(int n){ // definition of the function fact
21     int i, f=1;
22     for (i=1; i<=n; i++){
23         f *= i;
24     }
25     return f;
26 }

```

**Code 6.3:** Computing  $P(n, r)$ , without separate declaration of prototypes.

`n_permute_r_ver2.c`

```

1 #include <stdio.h>
2
3 int fact(int n){ // definition of the function fact
4     int i, f=1;
5     for (i=1; i<=n; i++){
6         f *= i;
7     }
8     return f;
9 }
10
11 int npr(int n, int r){ // definition of the function npr
12     return fact(n)/fact(n-r);
13 }
14
15 int main(){
16     int i, n;
17     printf("Enter n: ");
18     scanf ("%d", &n);
19     for (i=1; i<=n; i+=1){
20         printf ("%d choose %d = %d \n", i, n, npr(n, i));
21     }
22 }

```

## 6.5 The `return` statement

The `return` statement plays a critical role in function execution by managing control flow within a program. To understand this, let  $f$  be a function that calls a function  $g$ . Then,  $f$  is said to be the **caller function**, and the function  $g$ , being invoked by  $f$ , is the **called function**.

In case  $f$  is recursive,  $g$  may be  $f$  also—even then, the discussion here applies. In that case, the first instance of  $f$  refers to “the caller”, while the second refers to “the called”.

When the function  $g$  is called from  $f$ , the execution transfers from  $f$  to  $g$ . Suppose  $g$  has one or more `return` statements. Upon reaching a `return` statement in  $g$ , the process terminates  $g$  and passes the control back to  $f$ , the caller. The program then resumes execution from the point in  $f$  where the function  $g$  was originally invoked. Additionally, the `return` statement in  $g$  can return a value to  $f$ , which is used in subsequent operations in  $f$ .

The `return` statement also frees up memory allocated for the local variables and the stack frame of  $g$  (discussed in §6.9.1). Thus, the `return` statement not only marks the end of  $g$ 's execution but also facilitates control transfer, value return, and memory cleanup, ensuring orderly and efficient program flow.

To summarize, here are the key points:

1. A function with a return type other than `void` always returns a single value that matches the return type.
2. If the return type is `void`, the `return` statement is optional. In Code 6.4, the function `printLine` includes a `return` statement.
3. It is also logically correct to omit the `return` statement in a `void` function, as written in Code 6.5.
4. The body of a function may contain multiple `return` statements, which can terminate the function's execution before it reaches the end of its body. For example, Code 6.6 uses `return` three times to determine the largest of three integers.
5. In a value-returning function, `return` performs two distinct actions:
  - (i) Specifies the value to be returned once it is computed.
  - (ii) Terminates the function execution and transfers control back to the caller.

---

**Code 6.4:** Example of a function with return type `void`. Note that the function includes a `return` statement at the end, although it is not required for functions with a `void` return type.

`returnStatement_void.c`

```
1  #include <stdio.h>
2
3  void printLine(char c, int n){
4      for(int i=0; i<n; i++)
5          printf("%c", c);
6      return;
7  }
8
9  int main() {
10     char c;
11     int n;
12     printf("Enter the symbol and the length of the line: ");
13     scanf("%c%d", &c, &n);
14     printLine(c, n);
15     return 1;
16 }
```

---

**Code 6.5:** Example of the same `printLine` function used in Code 6.4. The only change here is exclusion of the `return` statement, which is also logically and syntactically correct.

`returnStatement_void2.c`

```

1 void printLine(char c, int n){
2     for(int i=0; i<n; i++)
3         printf("%c", c);
4 }
```

**Code 6.6:** Example of a function with multiple `return` statements.

`returnStatement_many.c`

```

1 int max(int a, int b, int c){
2     if (a >= b && a >= c)
3         return a;
4     if (b >= a && b >= c)
5         return b;
6     return c;
7 }
```

## 6.6 Local and global variables

The variables declared inside a function are said to be its **local variables**. Called so because there is a concept of **global variables** on the contrary, which are declared outside all functions, i.e., just after including all libraries (`#include ...`). The local variables can be accessed only within the function in which they are declared. Local variables cease to exist when the function terminates. Each execution of the function uses a new set of local variables, and input arguments or parameters are also local in the same sense.

For example, in Figure 6.1, `x` is a local variable of `dis`, whereas `a` and `b` are its two input parameters. Know that the input parameters `a` and `b` of `dis` are physically different from the local variables `a` and `b` in `main()`. Yes, although their names in `dis` are same with those in `main()`, they are physically different. This is because they have different addresses in the memory. The same holds for the variables `c` and `n` in Code 6.4. It is analogous to our very common experience that two persons from two different families may have the same name!

## 6.7 Scope of a variable

The **scope** of a variable means the part of the program that can use the value of the variable. It is basically the block in which it is defined. Recall that a **block** means the sequence of statements enclosed within two matching curly brackets.

For a local variable, the scope the entire function in which it is defined, provided it is declared in the very beginning of the body of the function. Two local variables of two functions can have the same name, but they are actually different variables because the variable name works as its first name whereas its surname is the function name.

Any global variable, since being declared outside all functions including `main()`, has its scope all over the program by default. However, its scope will be hidden in a block if a local variable of the same name is defined in that block. Use of global variables, unless there is a compelling reason, should be avoided.

Here is an example showing how a global variable `A` is declared and used by two different functions, namely `main()` and `myProc()`. The value of `A` is printed twice: first from `myProc()` and then from `main()`. Since `myProc()` changes the value of `A` to 2 and `main()` reads that value later, in both cases, the printed value will be 2.

```
1 | #include <stdio.h>
2 | int A; // This A is a global variable
3 | void main(){
4 |     A = 1;
5 |     myProc();
6 |     printf("A = %d\n", A);
7 | }
8 |
9 | void myProc(){
10 |     A = 2;
11 |     printf("A = %d\n", A);
12 | }
```

In the following code, note that the declaration `int A = 2;` in `myProc()` creates an integer variable with the same name, `A`. This `A` is local to `myProc()`, and when it is printed, the output is 2. This does not affect the value of the global variable `A`. Consequently, when `A` is printed from `main()`, the output is 1, which is the value of the global variable `A`.

```
1 | #include <stdio.h>
2 | int A; // This A is a global variable
3 | void main(){
4 |     A = 1;
5 |     myProc();
6 |     printf("A = %d\n", A);
7 | }
8 |
9 | void myProc(){
10 |     A = 2;
11 |     printf("A = %d\n", A);
12 | }
```

This practice of declaring local and global variables with the same name should be avoided when writing code.

## 6.8 Parameter passing

Parameter or argument passing is required while invoking functions, i.e., when a function calls another function or calls itself, the former being referred to as the **calling** or **caller function**, whereas the latter as **called function**. The parameter passing is done to the called function from the caller function. For example, in Figure 6.1, the values of the parameters `a` and `b` are passed to `dis` from `main()`.

### 6.8.1 Passing by value

It is also referred to as **call by value**. The called function gets a ‘copy’ of the value of each parameter, as it is passed to it by the caller function. In simpler words, the copy is treated as the value of a **new variable** associated with the called function only. Thus, execution of the called function has no effect on the values of the parameters of the caller. The copy does not exist when the called function ends and the program control returns to the caller function. A parameter passed from the caller may also be an expression, e.g., `n-r` is the argument passed from `main()` when it invokes `fact(n-r)`.

### 6.8.2 Passing by reference

It is also termed as **call by reference**. It is not directly supported in `C`, but supported in some other languages like `C++`. In `C`, we can instead pass copies of addresses to get the desired effect, as follows. The caller passes a copy of the address of a variable to the called function. Since the address is passed, execution of the called function may affect the value of the parameter in the caller function. A very common example is the function `swap` that exchanges the values of two variables declared and defined in `main()`. We shall see this later when working with pointers.

## 6.9 Recursive function

A **recursion** or **recursive function** is one that calls itself with **smaller values of its arguments**. For example,  $f(n) = n \cdot f(n - 1)$  is a recursion, while  $f(n) = n \cdot f(n + 1)$  is not. A mathematical recurrence, in general, can be quickly coded if we write it as a recursive function, provided the base cases are properly handled. Take, for example, computing the value of factorial  $n!$  of a positive integer  $n$ . Its recurrence is:

$$n! = \begin{cases} 1 & \text{if } n = 1 \\ n \cdot (n - 1)! & \text{otherwise.} \end{cases} \quad (6.1)$$

Its recursive implementation is straightforward and shown in Code 6.7. Note that the base case is specified before the recursive call, making the recursive call appear at the end or ‘tail’ of the function. This is known as **tail recursion**. An equivalent implementation using **non-tail recursion**, where the recursive call precedes the base condition, is provided in Code 6.8.

Although Code 6.8 is correct, it is not preferred due to its use of non-tail recursion. Non-tail recursion generally consumes more memory than tail recursion because each recursive call generates a new stack frame. The concept of stack frames is further discussed in §6.9.1. Therefore, whenever possible, recursive functions should be written using tail recursion. However, some computational problems, such as the Tower of Hanoi, cannot be solved with tail recursion, which we will examine in §6.9.2.

**Tail recursion** means that all base conditions and other non-recursive operations appear before all recursive calls, and the function returns immediately after the last recursive call. If not, we call it a **non-tail recursion**. Tail recursion can often be optimized to use a constant amount of memory space. See, for example, the iterative function to compute  $n!$ , shown in Code 6.9. It is obtained by removing the tail recursion in Code 6.7.

**Code 6.7:** Computing  $n!$  using **tail recursion**.

factorialFun1.c

```

1 | int fact(int n){
2 |     if (n==1) // base condition
3 |         return 1;
4 |     else
5 |         return n * fact(n-1); // recursion
6 | }
```

**Code 6.8:** Computing  $n!$  using **non-tail recursion**.

factorialFun2.c

```

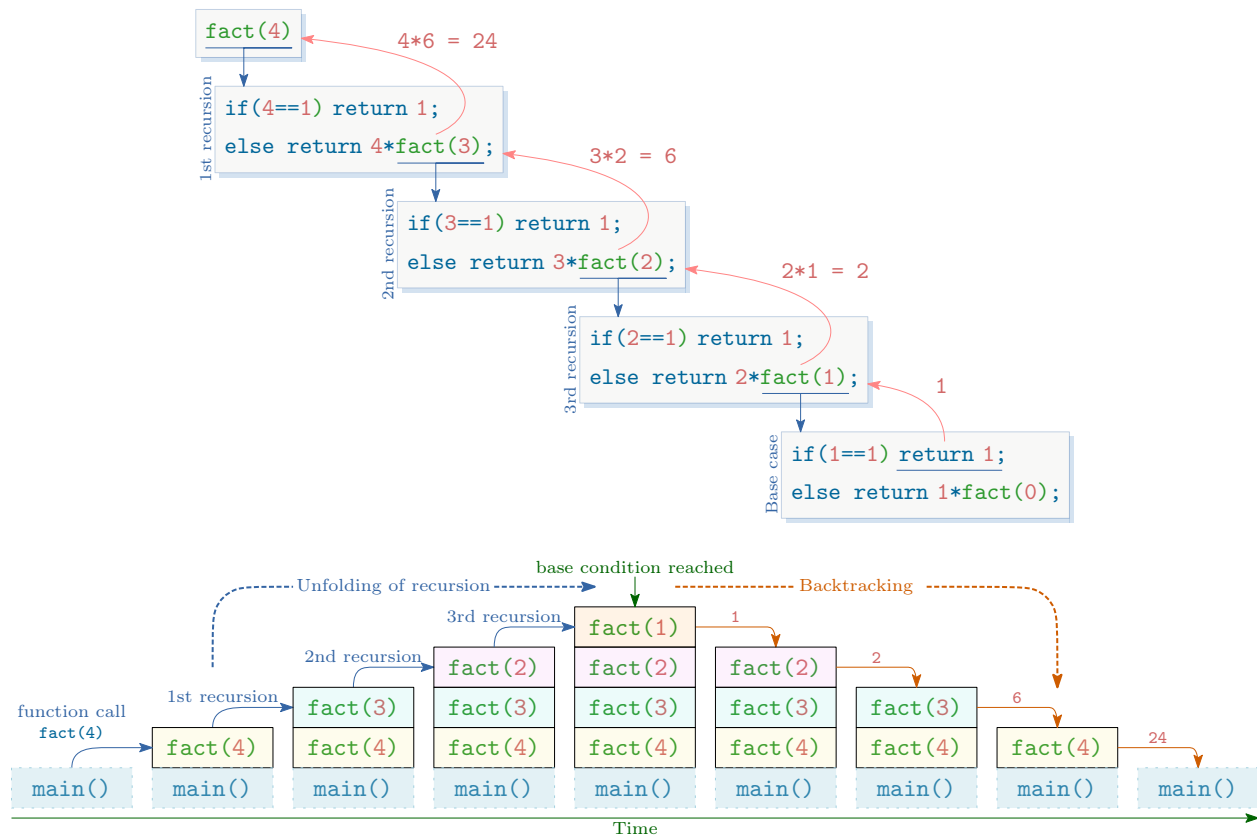
1 | int fact(int n){
2 |     if (n>1)
3 |         return n * fact(n-1); // recursion
4 |     else // base condition
5 |         return 1;
6 | }
```

**Code 6.9:** Computing  $n!$  by an **iterative** method, designed by removing the tail recursion in Code 6.7.

factorialIterative.c

```

1 | int fact(int n) {
2 |     int result = 1;
3 |     for (int i = 2; i <= n; i++)
4 |         result *= i;
5 |     return result;
6 | }
```



**Figure 6.2:** Effect of calling the function `fact(4)` from `main()`. **Top:** Unfolding and backtracking of recursive calls. **Bottom:** Expansion and contraction of the recursion stack during the unfolding and folding phases of recursion.

An important property of recursion is that when the recursive function is called next time, it acts on a smaller input, and so eventually the recursion reaches a/the base case. For example, consider the code of Fibonacci sequence written after Eq. 6.1. As demonstrated in Figure 6.2, the base case is for 1, and from that point the recursion wraps up, to get the values of `fact(2)`, `fact(3)`, and `fact(4)`, in that order. This ‘wrapping up’ is referred to as **backtracking**. Observe how the recursion stack grows when the function unfolds by recursive calls and diminishes when they terminate during backtracking. The structure and working principle of the stack are discussed in detail in §6.9.1.

### 6.9.1 Activation record and recursion stack

**Activation record** is a data structure that stores information about a single execution of a function. It includes all local variables and auxiliary data\* related to the called function.

\*“Auxiliary data” means all these:

- the return address (where to resume after the call, in order to execute the pending instructions of the caller function)
- local variables (variables declared within the function)
- parameters (values passed to the function from the caller)
- saved registers (CPU state preservation)
- reference to the caller’s activation record

**Recursion stack** (or “stack frame”) refers to the entire stack of activation records for a recursive function. It



is dynamic because of growth and diminish of active functions over time. Each time a recursive function calls itself, a new activation record is pushed onto the stack. The recursion stack grows as more recursive calls are made and shrinks when the calls return.

The **stack** is a data structure that holds all the necessary information and can be implemented by array. It resides in RAM. Data are stored in **LIFO** order (**L**ast **I**n, **F**irst **O**ut), hence the name "stack". If  $A_1$  is pushed onto the stack first, followed by  $A_2$ , then  $A_2$  will be popped before  $A_1$ . ('Push' and 'pop' are two stack operations; 'push' refers to insertion, while 'pop' denotes deletion.)

### Example: factorial of $n$

To understand the stack, let's go back the computation of  $n!$  using tail recursion (Code 6.7). As evident from Figure 6.2, upon reaching the base case, a recursive call ends, and the program control backtracks to the previous call. While a recursion unfolds, the stack grows up, and when backtracking starts, it shrinks down, as shown in Figure 6.2.

The stack maintains the activation records of all the unfinished recursive calls in an orderly manner. The last record is of the last recursive call, and while backtracking, it is the last record which is first processed. For example, the record of `fact(2)` is first processed while backtracking after reaching the base condition (corresponding to `fact(1)`).

### Example: Fibonacci sequence

Let's now consider a very well-known sequence called **Fibonacci sequence**, namely  $\{f(n) : n = 0, 1, 2, \dots\}$ , defined by the following recurrence:

$$f(0) = 0, f(1) = 1, \text{ and } f(n) = f(n-1) + f(n-2) \text{ if } n \geq 2. \quad (6.2)$$

---

**Code 6.10:** Computation of Fibonacci sequence by a **recursive** function.

`fibonacciRec.c`

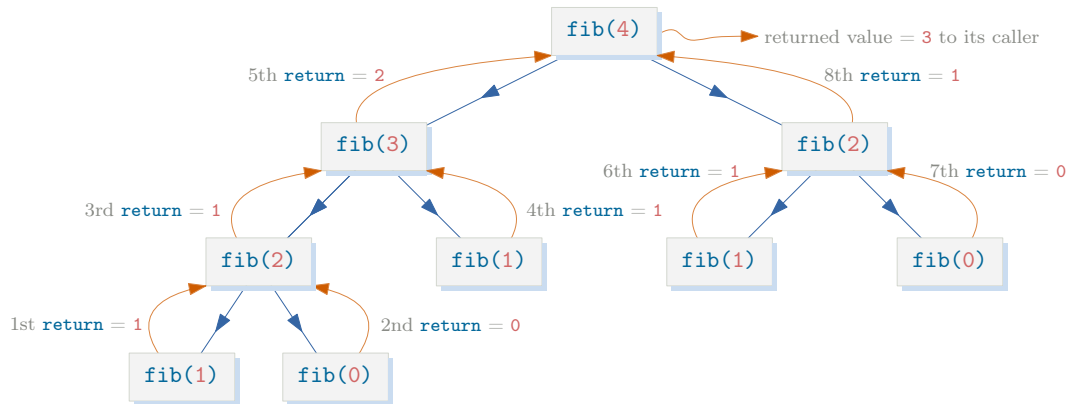
```

1 | int fib(int n) {
2 |     if (n <= 1) // Base condition: holds for n = 0 and 1
3 |         return n;
4 |     return fib(n-1) + fib(n-2); // Two recursive calls
5 | }
```

---

A recursive function to compute  $f(n)$  for  $n \geq 0$  is presented in Code 6.10. Consider the two recursive calls in the 4th line of Code 6.10: `return fib(n-1) + fib(n-2);`. For the first call (i.e., `fib(n-1)`), a new activation record is pushed onto the recursion stack, which holds the value of `n-1` and all auxiliary data. The same occurs for the second call (i.e., `fib(n-2)`). These calls, in turn, may invoke further recursive calls, with similar actions performed each time. In essence, every time a recursive function is called, a new activation record is created on the recursion stack, storing the argument value and auxiliary data. Once the function completes, its activation record is popped from the stack. This stack-based mechanism enables the program to manage multiple levels of recursion effectively.

A demonstration of Code 6.10 for  $n = 4$  is shown through its recursion tree in Figure 6.3. Each node in the tree represents a function call, and the directed edges illustrate the recursive calls between functions. The tree starts with the initial call for  $n = 4$  at the root and branches out to show all subsequent calls made by that function. For a function that calls itself recursively, the recursion tree reveals multiple levels of the function, demonstrating how each call generates further calls. This tree helps visualize how the problem is divided into sub-problems and how many times each sub-problem is solved.



**Figure 6.3:** The recursion tree corresponding to the function `fib(4)`.

The execution time and memory space required for running the recursive code are critical for understanding the efficiency of Code 6.10. Observe that the size of the recursion tree is exponential in  $n$ , implying that the time complexity is also exponential. This is because the function `fib( $n$ )` obtains its value only after exploring all nodes of the tree. However, the actual space used in the RAM at any point is at most linear in  $n$ , as each call to `fib( $n$ )` occupies an activation record in the recursion stack, and the maximum depth of the tree is  $n$ .

The function recomputes the values of `fib( $n - k$ )` multiple times for  $k \geq 1$  due to overlapping sub-problems. For example, `fib( $n - 2$ )` is computed twice, `fib( $n - 3$ )` is computed four times, and so on. This redundant computation contributes to the exponential runtime. This redundancy can be completely eliminated by using an iterative approach, which results in a linear runtime in  $n$ .

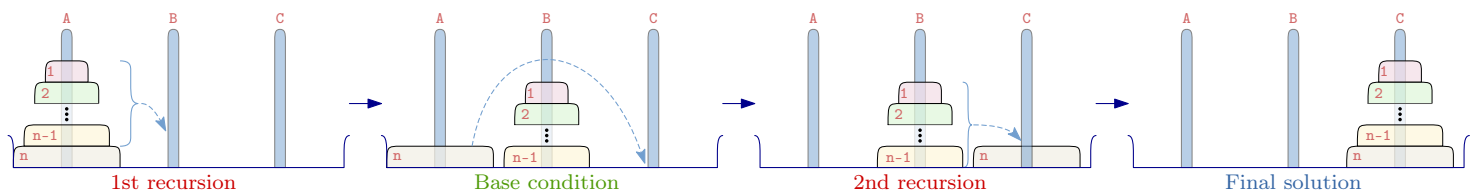
### 6.9.2 Tower of Hanoi

The **Tower of Hanoi** problem was invented by the French mathematician Édouard Lucas in 1883. He was inspired by a legend about a temple in India where monks were tasked with moving **64 golden disks** from one peg to another. According to the legend, once all the disks were moved, the world would come to an end. The problem became famous for illustrating recursion and algorithmic problem solving.

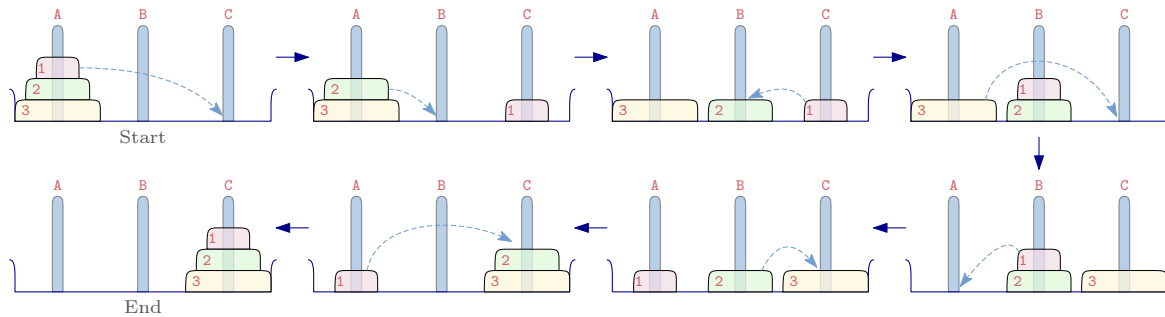
The computational problem goes as follows: Given three pegs  $A$ ,  $B$ ,  $C$ , and  $n$  disks of different sizes stacked on peg  $A$  in decreasing order of their sizes from bottom to top, the task is to move all the disks from peg  $A$  (source peg) to peg  $C$  (destination peg), using peg  $B$  (auxiliary peg). The constraints are:

1. Only one disk can be moved at a time.
2. A larger disk cannot be placed on top of a smaller disk.

The Tower of Hanoi problem **does not admit tail recursion**, as mentioned earlier in §6.9. This is because, after moving the top  $n - 1$  disks from peg  $A$  to peg  $B$ , you must move the largest disk (currently the sole disk in  $A$ ) to peg  $C$ . After that, the entire set of  $n - 1$  disks on peg  $B$  needs to be moved to peg  $C$ . This process



**Figure 6.4:** Tower of Hanoi: Three basic steps.



**Figure 6.5:** Tower of Hanoi: Sequence of moves while running Code 6.11 with  $n = 3$ .

requires additional work after each recursive call, which prevents the function from being tail recursive. The fundamental steps for solving the **Tower of Hanoi** problem, illustrated in Figure 6.4, are as follows:

1. **1st recursion:** Move the top  $n - 1$  disks from peg  $A$  to peg  $B$ , using peg  $C$  as the auxiliary peg.
2. **Base condition:** Move Disk  $n$  from peg  $A$  to peg  $C$ .
3. **2nd recursion:** Move all  $n - 1$  disks from peg  $B$  to peg  $C$ , using peg  $A$  as the auxiliary peg.

The corresponding code is given in Code 6.11, and here is the output for  $n = 3$ :

```

Enter #disks: 1      Enter #disks: 2      Enter #disks: 3
Disk 1: A → C      Disk 1: A → B      Disk 1: A → C
                   Disk 2: A → C      Disk 2: A → B
                   Disk 1: B → C      Disk 1: C → B
                                           Disk 3: A → C
                                           Disk 1: B → A
                                           Disk 2: B → C
                                           Disk 1: A → C

```

**Code 6.11:** Tower of Hanoi: Solution using non-tail recursion.

towerHanoi.c

```

1 | #include <stdio.h>
2 | #include <wchar.h>
3 |
4 | void hanoi(int n, char A, char C, char B) {
5 |     if (n == 1) { // Base case
6 |         printf("Disk 1: %c \u2192 %c\n", A, C); // \u2192 = Unicode character of right arrow
7 |         return;
8 |     }
9 |     hanoi(n - 1, A, B, C); // 1st recursive call
10 |    printf("Disk %d: %c \u2192 %c\n", n, A, C);
11 |    hanoi(n - 1, B, C, A); // 2nd Recursive call
12 | }
13 |
14 | int main() {
15 |     int n;
16 |     printf("Enter #disks: ");
17 |     scanf("%d", &n);
18 |     hanoi(n, 'A', 'C', 'B');
19 |     return 0;
20 | }

```

### 6.9.3 Direct and indirect recursion

As mentioned in §6.9, a recursive function repeatedly calls itself. This can occur either directly or indirectly in a cyclic chain. A **direct recursion** means the function calls only itself, while an **indirect recursion** means it calls another recursive function. Below are three examples defined for all positive integers  $n$ .

$$\text{Direct recursion: } f(n) = \begin{cases} 1 & \text{if } n = 1 \\ n \cdot f(n-1) & \text{otherwise.} \end{cases}$$

$$\text{Indirect recursion: } g(n) = \begin{cases} 1 & \text{if } n = 1 \\ n \cdot h(n-1) & \text{otherwise.} \end{cases}$$

$$\text{Indirect recursion: } h(n) = \begin{cases} 1 & \text{if } n = 1 \\ 1 + g(n-1) & \text{otherwise.} \end{cases}$$

The function  $f(n)$  is an example of direct recursion because it only calls itself. On the contrary, the function  $g(n)$  does not call itself but instead calls  $h(n)$ , which, in turn, calls back  $g(n)$ . Thus, for both  $g$  and  $h$ , it is an example of **indirect recursion**. Furthermore, since  $g$  and  $h$  call each other, they form what is known as **mutual recursion** or **cyclic recursion**.

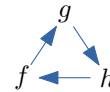
**Call graph:** In a recursive program, it represents the calling relationships between functions. This is specially important when there are many recursive functions. For example, if  $f$  calls  $f$  and  $g$ ,  $g$  calls  $f$ ,  $g$ , and  $h$ , and  $h$  calls only itself, the call graph will look as shown aside.



### 6.9.4 Mutual recursion

Mutual or cyclic recursion is one in which two or more functions call each other in a cycle. For example, when  $f$  calls  $g$ ,  $g$  calls  $h$ , and  $h$  calls  $f$ , as shown in the inset figure.

As a specific example, consider this problem: compute the  $n$ -th number in a **Fibonacci-like sequence**, defined by **two mutually recursive functions**, as follows.



$$f_{\text{even}}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 2 \\ f_{\text{even}}(n-2) + f_{\text{odd}}(n-1) & \text{otherwise.} \end{cases} \quad (6.3)$$

$$f_{\text{odd}}(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2 & \text{if } n = 3 \\ f_{\text{odd}}(n-2) + f_{\text{even}}(n-1) & \text{otherwise.} \end{cases} \quad (6.4)$$

A solution using tail recursion is shown in Code 6.12. This code is revised to make it free from recursions. The revised code uses only iterations (loops), which is shown in Code 6.13. For index  $n$  ranging from 1 to 3, the respective values obtained by running either of these codes are as follows:

$n$	1	2	3	4	5	6	7	8	9	10
$f$	1	1	2	3	5	8	13	21	34	55

**Code 6.12:** Computing the  $n$ -th number in a Fibonacci-like sequence using tail recursion.

```

1  #include <stdio.h>
2
3  int f_even(int n);
4  int f_odd(int n);
5
6  int main() {
7      int n;
8      printf("Enter the index (n): ");
9      scanf("%d", &n);
10     if (n % 2 == 0)
11         printf("f_even(%d) = %d\n", n, f_even(n));
12     else
13         printf("f_odd(%d) = %d\n", n, f_odd(n));
14     return 0;
15 }
16
17 int f_even(int n) {
18     if (n == 0)
19         return 0; // Base case
20     else if (n == 2)
21         return 1; // Base case
22     else
23         return f_even(n - 2) + f_odd(n - 1); // Recursive case
24 }
25
26 int f_odd(int n) {
27     if (n == 1)
28         return 1; // Base case
29     else if (n == 3)
30         return 2; // Base case
31     else
32         return f_odd(n - 2) + f_even(n - 1); // Recursive case
33 }

```

**History:** The Fibonacci sequence has its roots in the work of the Italian mathematician Leonardo of Pisa, known as Fibonacci, who introduced the sequence to Western mathematics in his book *Liber Abaci* (1202). Although the sequence had appeared earlier in Indian mathematics, Fibonacci used it to model the growth of a rabbit population. The sequence starts with 0 and 1, and each subsequent number is the sum of the two preceding ones, forming the series: 0, 1, 1, 2, 3, 5, 8, 13, ... . Fibonacci's work with this sequence has since found applications in various fields of science and nature.

**Applications:** In computer science, Fibonacci numbers are used in algorithms like Fibonacci search and for optimizing dynamic programming solutions. They also appear in data structures such as Fibonacci heaps. In biology, the sequence describes phenomena such as the arrangement of leaves on a stem, the branching of trees, or the reproduction patterns of bees. In finance, Fibonacci retracement levels are used in technical analysis to predict stock market movements. Moreover, the sequence appears in architecture and art, where it is often associated with the golden ratio, as the ratio of consecutive Fibonacci numbers approaches the golden ratio  $\varphi \approx 1.618$ .

**Similar sequences:** Several other sequences exhibit similar recursive patterns. The Lucas sequence, for example, starts with 2 and 1 but follows the same recursive relation. The Tribonacci sequence generalizes the Fibonacci rule by summing the three preceding numbers instead of two. Similarly, the Padovan and Perrin sequences follow related patterns, highlighting the broad applicability and fascination with such recursive relationships in mathematics and natural phenomena.

**Code 6.13:** Computing the  $n$ -th number in a Fibonacci-like sequence using an iterative method, designed by removing the tail recursions in Code 6.12. `fibonacci-like-sequence-1-iter.c`

```

1  #include <stdio.h>
2
3  int f_even(int n);
4  int f_odd(int n);
5
6  int main() {
7      int n;
8      printf("Enter the index (n): ");
9      scanf("%d", &n);
10     if (n % 2 == 0)
11         printf("f_even(%d) = %d\n", n, f_even(n));
12     else
13         printf("f_odd(%d) = %d\n", n, f_odd(n));
14     return 0;
15 }
16
17 int f_even(int n) {
18     int prev_even = 0, prev_odd = 1, current_even = 0, current_odd = 1;
19
20     for (int i = 2; i <= n; i++) {
21         if (i % 2 == 0) {
22             current_even = prev_even + prev_odd;
23             prev_even = current_even;
24         } else {
25             current_odd = prev_even + prev_odd;
26             prev_odd = current_odd;
27         }
28     }
29
30     return current_even;
31 }
32
33 int f_odd(int n) {
34     int prev_even = 0, prev_odd = 1, current_even = 0, current_odd = 1;
35
36     for (int i = 2; i <= n; i++) {
37         if (i % 2 == 0) {
38             current_even = prev_even + prev_odd;
39             prev_even = current_even;
40         } else {
41             current_odd = prev_even + prev_odd;
42             prev_odd = current_odd;
43         }
44     }
45
46     return current_odd;
47 }

```

1. **Fibonacci sequence:**  $F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$  for  $n \geq 2$ .
2. **Lucas sequence:**  $L_0 = 2, L_1 = 1, L_n = L_{n-1} + L_{n-2}$  for  $n \geq 2$ .
3. **Tribonacci sequence:**  $T_0 = 0, T_1 = 1, T_2 = 1, T_n = T_{n-1} + T_{n-2} + T_{n-3}$  for  $n \geq 3$ .
4. **Padovan sequence:**  $P_0 = P_1 = P_2 = 1, P_n = P_{n-2} + P_{n-3}$  for  $n \geq 3$ .
5. **Perrin sequence:**  $P_0 = 3, P_1 = 0, P_2 = 2, P_n = P_{n-2} + P_{n-3}$  for  $n \geq 3$ .

## 6.10 Passing an array to a function

While passing an array to a function, the name of the array is used as an argument. Now, beware of the catch here—since the name of the array is the argument, the values of the array elements are not passed to the called function. Rather, the array name is interpreted as the address of the first element of the array, and so what is passed is basically the first element's address. In this regard, the way it is passed differs from that for ordinary variables.

Repeating again, the argument carries the address of the first element of the array. When any element of that array has to be accessed inside the called function, its address is calculated by the compiler using the technique discussed in §5.2. Clearly, this address is address is same to both the caller and the called for any element of the array, and it is also accessible to both of them. Hence, any change made to any element inside the called function is eventually reflected in the calling function, once the called function ends. In Code 6.14, for example, the elements of a 5-element array are squared in the called function, and when printed from the caller, the squared elements will be printed as desired.

Code 6.14: Example of passing an array to a function.

passArray2Fun.c

```
1 void doSqr(int a[], int n){
2     for(int i=0; i<n; i++)
3         a[i] *= a[i];
4 }
5
6 int main(){
7     int a[] = {3, -2, 1, 2, 1};
8     doSqr(A, 5);
9     for(int i=0; i<5; i++)
10        printf("%d ", a[i]); // will print 9 4 1 4 1
11    return 1;
12 }
```

## 6.11 Macros (#define)

**Macros** are a powerful feature in C language that allows you to define constants or expressions that can be reused throughout the program. A macro serves as a placeholder for the specified value or expression. It is created by the `#define` directive. During preprocessing, which occurs before the compilation stage, the preprocessor scans through the code and substitutes every occurrence of the macro with the value or expression it represents.

For example, consider the macro definition: `#define PI 3.14159`. Whenever the macro `PI` is used in the code, the preprocessor will automatically replace it with `3.14159`. This substitution happens at every instance where `PI` appears, excepting the format string in `printf`. This is done by the programmer to ensure consistency and to reduce the risk of errors associated with manual input of constant values.

Code 6.15 illustrates shows how you can define and use a macro for  $\pi$ . In this code, we compute the perimeter and the area of a circle, given by  $2\pi r$  and  $\pi r^2$ , respectively, where  $r$  is the radius of the circle.

The `#define` directive in C not only allows you to define simple constants but also enables the creation of macros with arguments, which are similar to functions. These macros can take one or more arguments, and during preprocessing, the arguments are replaced with the values passed when the macro is invoked. This capability allows you to create flexible and reusable code snippets.

**Code 6.15:** Example of using a macro.

macro\_PI.c

```

1  #include <stdio.h>
2  #define PI 3.14159
3
4  int main() {
5      float radius, perimeter, area;
6
7      printf("Enter the radius of the circle: ");
8      scanf("%f", &radius);
9
10     perimeter = 2.0 * PI * radius;
11     area = PI * radius * radius;
12     printf("Perimeter = %0.3f, area = %0.3f\n", perimeter, area);
13
14     return 0;
15 }
```

## 6.12 #define with arguments

A macro with arguments is defined using the same `#define` directive but includes a list of parameters in parentheses after the macro name. When the macro is used in the code, the preprocessor replaces the macro call with the macro definition, substituting the arguments with the corresponding values passed in the call. For example, this is a macro that calculates the square of a number:

```
#define SQR(x) ((x)*(x))
```

In this example, the macro `sqr` takes one argument, `x`. When you use `sqr` in your code, the preprocessor replaces it with the expression `((x)*(x))`. Here is how you can use it:

```

#define SQR(x) ((x)*(x))
int main() {
    int n = 5;
    printf("The square of %d is %d\n", n, SQR(n));
    return 0;
}
```

When the preprocessor processes this code, the line:

```
printf("The square of %d is %d\n", n, SQR(n));
```

is replaced with:

```
printf("The square of %d is %d\n", n, ((x)*(x)));
```

This substitution ensures that the macro works correctly, even if the argument `n` is an expression rather than a simple variable. For example, if you write `SQR(a+b)`, the macro expands to `((a+b)*(a + b))`, ensuring that the entire expression is squared.

You have to be careful while defining a macro. For example, if you write `#define SQR(x) x*x`, then `SQR(a+b)` will result in `a+b*a+b`, which is incorrect.

As you understand now, macros with arguments have definite advantages in coding. First, they simplify complex expressions by encapsulating them in a single, easy-to-use macro. Second, they are efficient because



**Table 6.1:** Some typical examples of macros.

---

```

#define SQR(x) ((x) * (x)) // Square of a number x
#define CUBE(x) ((x) * (x) * (x)) // Cube of a number x
#define AVG(a, b) ((a) + (b)) / 2.0 // Average of two numbers a and b
#define ABS(x) ((x) < 0) ? -(x) : (x) // Absolute value of a number x
#define IS_EVEN(x) (((x) % 2) == 0) // Check if a number x is even
#define IS_ODD(x) (((x) % 2) != 0) // Check if a number x is odd
#define MIN(a, b) (((a) < (b)) ? (a) : (b)) // Minimum of two numbers a and b

// Check if a number x is within a range [low, high]
#define IN_RANGE(x, low, high) (((x) >= (low)) && ((x) <= (high)))

// Determine the maximum of three numbers a, b, and c
#define MAX3(a, b, c) (((a) > (b)) ? ((a) > (c) ? (a) : (c)) : ((b) > (c) ? (b) : (c)))

// Calculate the distance between two points (x1, y1) and (x2, y2)
#define DIS(x1, y1, x2, y2) sqrt(((x2)-(x1)) * ((x2)-(x1)) + ((y2)-(y1)) * ((y2)-(y1)))


#define PI (2.0 * asin(1)) // Define the best-possible value of pi using math.h
#define DEG_TO_RAD(x) ((x) * (PI) / 180.0) // Convert an angle x from degrees to radians
#define RAD_TO_DEG(x) ((x) * 180.0 / (PI)) // Convert an angle x from radians to degrees
#define CIRC(r) (2 * PI * (r)) // Calculate the perimeter of a circle with radius r
#define CIRA(r) (PI * (r) * (r)) // Calculate the area of a circle with radius r

// Swap two variables a and b
#define SWAP(a, b, type) { type temp = (a); (a) = (b); (b) = temp; }

```

---

they are expanded inline, avoiding the overhead of function calls. Third, they allow you to create reusable code snippets that can be adapted to different inputs. Some typical use case for macros are given in Table 6.1. Following is a program where the macro `SWAP(a, b, type)` is used.



```

1 #include <stdio.h>
2
3 // Macro to swap two variables of a given type
4 #define SWAP(a, b, type) { type temp = (a); (a) = (b); (b) = temp; }
5
6 int main() {
7     // Declare and initialize variables
8     int x = 10, y = 20;
9     double p = 3.14, q = 2.71;
10    char c1 = 'A', c2 = 'B';
11
12    // Swap integers
13    printf("Before swapping: x = %d, y = %d\n", x, y);
14    SWAP(x, y, int);
15    printf("After swapping: x = %d, y = %d\n\n", x, y);
16
17    // Swap doubles
18    printf("Before swapping: p = %.2f, q = %.2f\n", p, q);
19    SWAP(p, q, double);

```

```

20     printf("After swapping: p = %.2f, q = %.2f\n\n", p, q);
21
22     // Swap characters
23     printf("Before swapping: c1 = %c, c2 = %c\n", c1, c2);
24     SWAP(c1, c2, char);
25     printf("After swapping: c1 = %c, c2 = %c\n", c1, c2);
26
27     return 0;
28 }

```

## 6.13 Extra topics

These topics may not be in your syllabus. Please get it confirmed from your teacher.

### 6.13.1 Generating random input using `rand`

Random input is crucial for testing and analyzing systems with large data sizes because it helps in simulating a wide range of possible scenarios and corner cases. When dealing with substantial datasets, manually crafting diverse test-cases becomes impractical. Randomly generated data ensures that the system is exposed to various combinations and distributions of input, which can reveal potential issues such as performance bottlenecks, boundary conditions, or unexpected behaviors. This approach is essential for validating the robustness and reliability of algorithms and applications, ensuring they perform well under realistic and unpredictable conditions.

The next C program demonstrates how to use the `rand` function with `srand` for seed generation to fill an array of 100 integers with values in a user-defined interval `[a,b]`. The user inputs the interval bounds `a` and `b`. The `srand` function is used to seed the pseudo-random number generator with the current time, ensuring different random sequences on each execution. The `rand` function generates random numbers, which are then scaled to fall within the interval `[a,b]`. The program finally prints the filled array.

The macro `#define RANDOM_IN_RANGE(a, b)` generates a random integer in `[a,b]`, by using the `rand()` function and adjusting the range accordingly. The library `stdlib.h` provides access to the `rand()` function for generating random numbers and `srand()` for setting the seed, while the library `time.h` is used to obtain the current time with `time(NULL)`, which initializes the seed for `rand()` to ensure that the sequence of random numbers varies with each execution. This combination is essential for producing diverse and unpredictable random values.



```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  #define SIZE 100
6
7  // Macro to generate a random integer between a and b
8  #define RANDOM_IN_RANGE(a, b) ((a) + rand() % ((b) - (a) + 1))
9
10 int main() {
11     int a, b;
12     int array[SIZE];
13
14     // Get the interval bounds from the user
15     printf("Enter the lower bound (a): ");
16     scanf("%d", &a);

```

```

17 | printf("Enter the upper bound (b): ");
18 | scanf("%d", &b);
19 |
20 | // Seed the random number generator with the current time
21 | srand(time(NULL));
22 |
23 | // Fill the array with random integers in the interval [a, b]
24 | for (int i = 0; i < SIZE; i++) {
25 |     array[i] = RANDOM_IN_RANGE(a, b);
26 | }
27 |
28 | // Print the filled array
29 | printf("Array filled with random integers between %d and %d:\n", a, b);
30 | for (int i = 0; i < SIZE; i++) {
31 |     printf("%d ", array[i]);
32 | }
33 | printf("\n");
34 |
35 | return 0;
36 | }

```

Below is a modified version of the above code in which the seed is an integer `r` taken in from the user, rather than using a time-dependent seed. Note that the number of elements is also taken from the user.



```

1 | #include <stdio.h>
2 | #include <stdlib.h>
3 |
4 | #define RANDOM_IN_RANGE(a, b) ((a) + rand() % ((b) - (a) + 1))
5 |
6 | int main() {
7 |     int a, b, r, n;
8 |
9 |     // Prompt user for the range [a, b], the seed value, and the number of elements
10 |    printf("Enter the lower bound (a): ");
11 |    scanf("%d", &a);
12 |    printf("Enter the upper bound (b): ");
13 |    scanf("%d", &b);
14 |    printf("Enter the seed value (r): ");
15 |    scanf("%d", &r);
16 |    printf("Enter the number of elements (n): ");
17 |    scanf("%d", &n);
18 |
19 |    // Initialize the random number generator with the user-provided seed
20 |    srand(r);
21 |
22 |    // Fixed-size array
23 |    int arr[100];
24 |
25 |    // Ensure n does not exceed the array size
26 |    if (n > 100) {
27 |        printf("The number of elements cannot exceed 100.\n");
28 |        return 1;
29 |    }
30 | }

```

```

31 | // Fill the array with random integers in the range [a, b]
32 | for (int i = 0; i < n; i++) {
33 |     arr[i] = RANDOM_IN_RANGE(a, b);
34 | }
35 |
36 | // Display the generated array
37 | printf("Generated array:\n");
38 | for (int i = 0; i < n; i++) {
39 |     printf("%d ", arr[i]);
40 | }
41 | printf("\n");
42 |
43 | return 0;
44 | }

```

Here are the outputs for two different seeds provided by the user, while the interval `[a,b]` remains unchanged. Notice how the outputs differ due to the variation in seeds.

```

Enter the lower bound (a): 10
Enter the upper bound (b): 13
Enter the seed value (r): 1
Enter the number of elements (n): 5
Generated array:
13 12 11 13 11

```

```

Enter the lower bound (a): 10
Enter the upper bound (b): 13
Enter the seed value (r): 2
Enter the number of elements (n): 5
Generated array:
12 13 10 13 11

```

### 6.13.2 `main()` with arguments

The `main()` function serves as the entry point of a C program, where the execution begins. When `main()` is defined without arguments, it takes the form `int main(void)` or simply `int main()`. In this case, the function does not accept any input from the command line, and the program runs independently of any external input.

Alternatively, `main()` can be defined with arguments, typically as `int main(int argc, char *argv[])`. Here, `argc` (argument count) represents the number of command-line arguments passed to the program, and `argv` (argument vector) is an array of strings holding the actual arguments. This form of `main()` is crucial when a program needs to process input provided by the user at runtime, such as file names, options, or other data.

To run a program with arguments, you execute the compiled program in the command line followed by the desired arguments. For instance, if your executable is named `a.out`, and you want to pass two arguments `input.txt` and `output.txt`, you would run the command:

```
./a.out input.txt output.txt
```

This allows the program to dynamically adjust its behavior based on the provided inputs, making it more versatile and interactive.

Here is a C program that takes numbers as command-line arguments and prints their square roots to the terminal:



```
1 #include <stdio.h>
2 #include <stdlib.h> // Include stdlib.h for atof()
3 #include <math.h>   // Include math.h for sqrt()
4
5 int main(int argc, char *argv[]) {
6     // Check if at least one number is provided as argument
7     if (argc < 2) {
8         printf("Usage: %s <number1> <number2> ... <numberN>\n", argv[0]);
9         return 1;
10    }
11
12    // Loop through each argument provided
13    for (int i = 1; i < argc; i++) {
14        // Convert the argument from string to double
15        double num = atof(argv[i]);
16
17        // Calculate the square root
18        double squareRoot = sqrt(num);
19
20        // Print the result to the terminal
21        printf("Square root of %.2lf is %.4lf\n", num, squareRoot);
22    }
23
24    return 0;
25 }
```

Here is a snapshot of its compilation and execution:

```
gcc mainWithArgs_SqRootsTerminal.c -lm
./a.out 4 6 7 9
Square root of 4.00 is 2.0000
Square root of 6.00 is 2.4495
Square root of 7.00 is 2.6458
Square root of 9.00 is 3.0000
```

Here is another C program where the input file contains some numbers, and the output file will store their square roots:



```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 int main(int argc, char *argv[]) {
6     FILE *inputFile, *outputFile;
7     double num;
8
9     // Open input file in read mode
10    inputFile = fopen("input.txt", "r");
11    if (inputFile == NULL) {
12        printf("Error opening input file.\n");
13        return 1;
14    }
15 }
```

```

16 // Open output file in write mode
17 outputFile = fopen("output.txt", "w");
18 if (outputFile == NULL) {
19     printf("Error opening output file.\n");
20     fclose(inputFile);
21     return 1;
22 }
23
24 // Read each number, calculate its square root, and write to the output file
25 while (fscanf(inputFile, "%lf", &num) != EOF) {
26     fprintf(outputFile, "Square root of %.2lf is %.4lf\n", num, sqrt(num));
27 }
28
29 // Close the files
30 fclose(inputFile);
31 fclose(outputFile);
32
33 return 0;
34 }

```

## 6.14 Solved problems

1. [Printing primes up to a given number] Write a function of the prototype `int checkPrime(int x)`. It takes as input an integer `x`, checks whether it's prime, and returns `1` if so, and `0` otherwise. Scan a positive integer `n` from `main()` and print all primes in  $[1, n]$  by calling the above function in a loop.

```

1 #include <stdio.h>
2 #include <math.h>
3
4 int checkPrime(int x){
5     int isPrime = 1;
6     for(int i = 2; (i <= sqrt(x)) && isPrime; i++){
7         if(x%i==0)
8             isPrime = 0;
9         i++;
10    }
11    return isPrime;
12 }
13
14 int main(){
15     int n;
16     printf("Enter a positive integer: ");
17     scanf("%d", &n);
18     for(int j=3; j<=n; j++)
19         if(checkPrime(j))
20             printf("%d is prime\n", j);
21     return 0;
22 }

```

2.  $[n \text{ choose } r: \binom{n}{r}]$  Write two functions of the following prototypes:

```

int ncr (int n, int r);
int fact (int n);

```

The first function will compute the value of  $\binom{n}{r} = \frac{n!}{r!(n-r)!}$ , and to do so, it will call the second one to get the values of  $n!$ ,  $r!$ , and  $(n-r)!$ . Scan a positive integer  $n$  from `main()` and print the values of  $\binom{n}{r}$  for  $r \in [0, n]$ , by calling the function `ncr` in a loop. For example, for  $n = 5$ , it will print 1, 5, 10, 10, 5, 1.

```

1 #include <stdio.h>
2
3 int ncr (int n, int r); // prototype of the function ncr
4 int fact (int n); // prototype of the function fact
5
6 int main(){
7     int i, n;
8     printf("Enter n: ");
9     scanf ("%d", &n);
10    for (i=0; i<=n; i+=1)
11        printf ("%d choose %d = %d \n", i, n, ncr(n, i));
12 }
13
14 int ncr(int n, int r){ // definition of the function ncr
15     return fact(n)/fact(r)/fact(n-r);
16 }
17
18 int fact(int n){ // definition of the function fact
19     int i, f=1;
20     for (i=1; i<=n; i++)
21         f *= i;
22     return f;
23 }

```

3. [Value of a quadratic function] Write a function to compute the value of  $ax^2 + bx + c$  with  $a, b, c, x$  as real values and taken as arguments in that order. Scan their respective values from `main()`, call that function from `main()`, and print its returned value from `main()`.

```

1 // Value of a quadratic function
2
3 #include <stdio.h>
4
5 float f(float a, float b, float c, float x){
6     return a*x*x + b*x + c;
7 }
8
9 int main(){
10    float a, b, c, x;
11    printf("\nEnter a, b, c, x: ");
12    scanf("%f%f%f%f", &a, &b, &c, &x);
13    printf("f = %f\n", f(a,b,c,x));
14    return 0;
15 }

```

4. [Arithmetic and geometric means] Write a function to compute and print the arithmetic mean and the geometric mean of  $n$  real numbers, with  $n$  and an  $n$ -element array as arguments. The value of  $n$  and the array elements will be supplied as input from `main()`. The `math` library can be used, but only for the `pow()` function.

```

1 // Arithmetic and geometric means
2
3 #include <stdio.h>
4 #include <math.h>
5 #define SIZE 1000
6
7 void computeAMGM(int n, float a[]){
8     int i;
9     float sum=0, prod=1;
10
11     for(i=0; i<n; i++){
12         sum += a[i];
13         prod *= a[i];
14     }
15     printf("AM = %f, GM = %f.\n", sum/n, pow(prod, 1.0/(float)n));
16 }
17
18 int main(){
19     int n, i;
20     float a[SIZE];
21
22     printf("Enter n: ");
23     scanf("%d", &n);
24     printf("Enter the elements: ");
25
26     for(i=0; i<n; i++)
27         scanf("%f", &a[i]);
28
29     computeAMGM(n, a);
30
31     return 0;
32 }

```

5. **[Median]** Write a function to compute and return the median of  $n$  integers, with  $n$  and an  $n$ -element array as arguments. Scanning the value of  $n$  along with the array elements, and printing the median, should all be done from `main()`. Assume that  $n$  is odd and all elements are distinct. Then, exactly  $\frac{n-1}{2}$  elements of the array will be smaller than the median—a fact that can be used to write the function.

```

1 // Find the median element in an array with odd number of elements
2
3 #include <stdio.h>
4 #define SIZE 1000
5
6 int readInput(int a[]){
7     int n, i;
8     printf("Enter n: ");
9     scanf("%d", &n);
10    printf("Enter the elements: ");
11    for(i=0; i<n; i++)
12        scanf("%d", &a[i]);
13    return n;
14 }
15
16 int findMedian(int a[], int n){
17     int i, j, k, m;

```



```

18 |   for(i=0; i<n; i++){
19 |       for(j=0, k=0; j<n; j++){
20 |           if(a[j]<a[i])
21 |               k++;
22 |       }
23 |       if (k==n/2){
24 |           m = a[i];
25 |           break;
26 |       }
27 |   }
28 |   return m;
29 | }
30 |
31 |
32 | int main(){
33 |     int a[SIZE], n, m;
34 |
35 |     n = readInput(a);
36 |     m = findMedian(a, n);
37 |     printf("Median = %d\n", m);
38 |
39 |     return 0;
40 | }

```

6. [GCD function, recursive] Write a recursive function of the prototype `int gcd(int, int)` to compute the GCD of two numbers, using Eq. 4.1. Take in two positive integers  $m, n$  from `main()`, call `gcd(m,n)` from `main()`, and print its returned value from `main()`.

```

1 //GCD function: recursive
2
3 #include <stdio.h>
4
5 int gcd(int a, int b){
6     if(a==0) return b;
7     else return gcd(b%a, a);
8 }
9
10 int main(){
11     int m, n;
12     printf("\nEnter two positive integers: ");
13     scanf("%d%d", &m, &n);
14     printf("GCD = %d\n", gcd(m,n));
15     return 0;
16 }
17

```

7. [GCD function, iterative] Write an iterative version of the GCD function using Eq. 4.1, keeping the `main()` same as the recursive version.

```

1 //GCD function
2
3 #include <stdio.h>
4
5 int gcd(int a, int b){
6     int c;

```

```

7   while(a!=0){
8       c = a;
9       a = b%a;
10      b = c;
11  }
12  return b;
13 }
14
15 int main(){
16     int m, n;
17     printf("\nEnter two positive integers: ");
18     scanf("%d%d", &m, &n);
19     printf("GCD = %d\n", gcd(m,n));
20     return 0;
21 }
22

```

8. **[Generate all possible permutations of an array]** Write a recursive function to generate all possible permutations of a random array. The seed, the number of elements, and the range of elements are taken in as input from the user. The function should generate permutations by swapping each element with the first element and then recursively permuting the remaining sub-array. This problem inherently relies on recursion to explore all possible orderings of the array elements.



```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  // Macro to generate a random integer in the range [a, b]
5  #define RANDOM_IN_RANGE(a, b) ((a) + rand() % ((b) - (a) + 1))
6
7  // Recursive function to generate all permutations
8  void generatePermutations(int arr[], int start, int end) {
9      if (start == end) {
10         // Print the current permutation
11         for (int i = 0; i <= end; i++) {
12             printf("%d ", arr[i]);
13         }
14         printf("\n");
15     }
16     else {
17         for (int i = start; i <= end; i++) {
18             // Swap
19             int temp = arr[start];
20             arr[start] = arr[i];
21             arr[i] = temp;
22
23             // Recursively generate permutations for the sub-array
24             generatePermutations(arr, start + 1, end);
25
26             // Backtrack by swapping back the elements
27             temp = arr[start];
28             arr[start] = arr[i];
29             arr[i] = temp;
30         }
31     }
32 }

```

```

33
34 int main() {
35     int n, seed, a, b;
36
37     // Take user input for the seed, number of elements, and the range [a, b]
38     printf("Enter the seed: ");
39     scanf("%d", &seed);
40     printf("Enter the number of elements: ");
41     scanf("%d", &n);
42     printf("Enter the range [a, b]: ");
43     scanf("%d %d", &a, &b);
44
45     // Set the seed for random number generation
46     srand(seed);
47
48     // Generate a random array within the range [a, b]
49     int arr[n];
50     for (int i = 0; i < n; i++) {
51         arr[i] = RANDOM_IN_RANGE(a, b);
52     }
53
54     // Print the generated array
55     printf("Generated array: ");
56     for (int i = 0; i < n; i++) {
57         printf("%d ", arr[i]);
58     }
59     printf("\n\n");
60
61     // Generate and print all permutations of the array
62     printf("Permutations:\n");
63     generatePermutations(arr, 0, n - 1);
64
65     return 0;
66 }

```

Here is an output of the above code:

```

Enter the seed: 1
Enter the number of elements: 3
Enter the range [a, b]: 1 9
Generated array: 2 8 1

```

Permutations:

```

2 8 1
2 1 8
8 2 1
8 1 2
1 8 2
1 2 8

```

## 6.15 Exercise problems

1. [Values of a quadratic sequence] Write a function to compute the value of  $x^2 + bx + c$  with  $b$  and  $c$  as positive integers and  $x$  as real, taken as arguments in that order. Scan their respective values from

`main()`, call that function from `main()` for every integer value of  $x$  ranging from  $-10$  to  $10$ , and print every returned value from `main()`. That is, the function will be called 21 times from `main()`.

2. **[Fibonacci function, recursive]** Write a recursive function of the prototype `int f(int)` to compute the  $n$ th Fibonacci term,  $f(n)$ , using Eq. 5.1. Take in a positive integer  $n$  from `main()`, call `f(n)` from `main()`, and print its returned value from `main()`.
3. **[Fibonacci function, iterative]** Write an iterative version of the Fibonacci function using Eq. 5.1, keeping the `main()` same as the recursive version.
4. **[Two-element sum as an element in another]** Read in two integers  $m$  and  $n$ . Assume that both  $m$  and  $n$  are in  $[2, 100]$ . Take in  $m$  integers to an array `a[]` and  $n$  integers to an array `b[]`. Write a function that can be called from `main` to check whether there are two elements in `a[]` that add up to a single element in `b[]`. The arguments of that function should be `a[]`, `b[]`,  $m$ , and  $n$ .

# 7 | Strings

## 7.1 Characters and strings

A **character** in C programming is an unsigned 8-bit number representing its ASCII (American Standard Code for Information Interchange) value, ranging from 0 to 255. Some of these characters are keyboard characters, e.g., the English letters, digits, punctuation marks, brackets, space, mathematical symbols like +, -, etc., and symbols such as ~, @, etc. Additionally, some characters serve special functions, such as the newline character denoted by '\n'. The **null character** or **null terminator**, represented by '\0', is another significant special character. Other special characters, which are less commonly used, are listed in Table 7.1 for completeness.

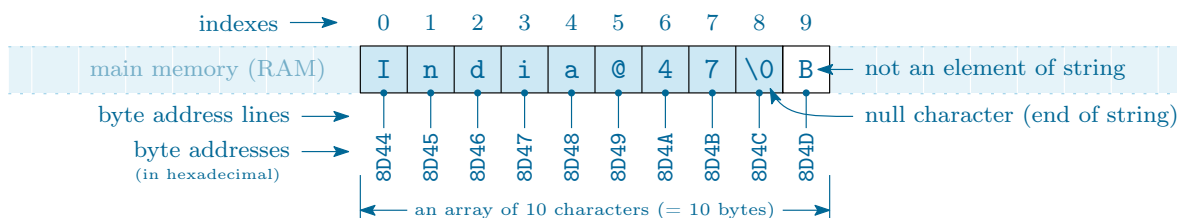
A **string** is an array of characters ending with a null character, i.e., '\0'. To be specific, this is called a **null-terminated string**. Unless stated otherwise, a string refers to a null-terminated string. In this sense, an array of characters without '\0' is a **character array** but **not a string**. In other words, the null character, also known as the **sentinel character**, is the last character of a string. For example, `India@1947` is a character array but not a string, while `India@1947\0` is both a character array and a string.

Another example is shown in Figure 7.1. The array contains 10 characters: `India@47\0B`, with the 9th character being '\0'. This implies that the string consists of the first 9 characters, i.e., `India@47\0`, and the 10th byte ('B') does not belong to the string. Note that the **length** of the string excludes the null character; therefore, in this example, the length is 8, not 9.

In short, if the array contains additional characters after the first null character, those characters are not part of the string. For example, if the array has 10 bytes and its content is `India\0@47\0`, the string starting from `I` is `India\0`, which consists of 6 characters, including `\0`. In fact, this array contains two strings, the second being `@47\0`.

**Table 7.1:** Special characters starting with a backslash in C.

Sl.no.	Character	Meaning	Description
1	\n	Newline	Moves the cursor to the beginning of the next line.
2	\t	Horizontal Tab	Inserts a horizontal tab.
3	\r	Carriage Return	Moves the cursor to the beginning of the current line without advancing to the next line.
4	\v	Vertical Tab	Moves the cursor down to the next vertical tab stop.
5	\b	Backspace	Moves the cursor one position back.
6	\f	Form Feed	Advances the paper feed in a printer to the start of the next page.
7	\0	Null Character	Represents a null character (ASCII value 0), often used to terminate strings in C.
8	\\	Backslash	Inserts a backslash (\) in the text.
9	\'	Single Quote	Inserts a single quotation mark in the text.
10	\"	Double Quote	Inserts a double quotation mark in the text.



**Figure 7.1:** An array of 10 characters, requiring 10 bytes in the memory. Its first and last bytes have the addresses `8D44` and `8D4D`, respectively, in hexadecimal number system. Since the 9th (i.e., of index 8) character is `'\0'`, the first 9 characters comprise the string, and the 10th byte, although belongs to the array, is not a part of the string.

## 7.2 Declaring a string

A string is written as the sequence of characters between double quotes. For example, `""` represents the empty string of length 0, with 1 byte storage to store `'\0'`. Here is the declaration:

```
char s[1] = "";
```

To continue with other examples, `"IIT"` represents a 4-byte string of length 3, and `"hello"` is a 6-byte string of length 5. Each string implicitly ends with the null character `'\0'`, which is included in the byte count but not in the string length, as mentioned earlier. Their respective declarations with optimal storage are as follows:

```
char t[4] = "IIT";
char u[6] = "hello";
```

To declare a string of characters without initializing it, you have several options in C. Here are the possible ways:

1. Fixed-size array declaration for 1000 characters:

```
char s[1000];
```

2. Using `malloc`, you can allocate memory dynamically for `n` characters:

```
char *s = (char *)malloc(n * sizeof(char));
```

3. Using `calloc`:

```
char *s = (char *)calloc(n, sizeof(char));
```

The notation `*` used in the last two options denotes pointers. Understanding their implementation requires knowledge of memory management, which will be discussed later.

## 7.3 Initializing a string

Don't confuse single quotes with double quotes. Single quotes are used to denote characters, while double quotes are used for strings. For example, `'a'` denotes a single character, stored as the ASCII value for the letter `a` in 1 byte. In contrast, `"a"` is an array consisting of two characters: the first is `'a'`, and the second is the null character `'\0'`. These two characters are stored as their ASCII values in two contiguous bytes in the memory.

Consider the string `"India@47"`. It has nine characters in total including the implicit null character after the character `'7'`. An array that will contain just this string and no more characters, can be declared and initialized as follows.

```
char s[9] = {'I', 'n', 'd', 'i', 'a', '@', '4', '7', '\0'};
```

By the above declaration, `s[0]` gets the character `I`, `s[1]` gets `'n'`, and so on, and the last cell of the array, i.e., `s[8]`, stores the null character.

Two other correct declarations are:

```
char s[] = "India@47\0"; // you must put the string within double quotes
```

and

```
char s[] = "India@47"; // you must put the string within double quotes
```

Note that when you declare a string using the last syntax, the compiler automatically adds the null terminator to the end of the string. So the array `s` takes 9 bytes to store the string `India@47` along with the null terminator.

However, if you write `char s[9] = {'I', 'n', 'd', 'i', 'a', '@', '4', '7'};`, then the array `s` will contain exactly the characters you specified, without an explicit null terminator. In this case, the array `s` will have a size of 9 bytes, where the first 8 bytes are used for the characters you specified, and the 9th byte is uninitialized and contains some 'garbage value'. Since you did not include the null terminator explicitly, the array `s` will not be a valid string if the garbage value at the last byte is not the null character.

## 7.4 Reading strings with `%s`

Let `str` be a string. It is important to know how it will be read or scanned. For this, you should have a basic idea about pointers. For the time being, you just know that a **pointer** is essentially a variable that stores the memory address of another variable or datatype.

You will study pointers in more detail later. If you are curious at this early stage, here are some additional details:

- `&c` denotes the **address** of a variable `c`.
- `&c` also represents a **pointer** to `c` because it holds the address of `c`.

The distinction is that **address** refers to a specific location in memory (RAM), while a **pointer** is a variable that stores such an address. The type of a pointer depends on the type of the variable it points to. For example, if the type of `c` is `char`, then `&c` is of type `char *`. Similarly, if the type of `n` is `int`, then `&n` is of type `int *`. Yes, `char *` and `int *` are data types in C. They are pointer types, meaning they hold addresses of variables of type `char` and `int`, respectively.

Now, as you already know, in the function `scanf`, for any variable such as `char` or `int`, we use the symbol `&` as the **address-of operator** to pass the memory address of the variable. For example, to scan a character variable `c` and an integer variable `n`, and to store them at their respective memory addresses, denoted by `&c` and `&n`, we write:

```
scanf("%c%d", &c, &n);
```

That is, the address-of operator `&` is used by `scanf` to store a variable's value directly in its memory location. However, for a string `str` (which is basically a null-terminated array of characters), its name itself acts as a pointer to the first element of the array. That is, `str` is inherently a pointer pointing to the beginning of the string, or, equivalently, `str` contains the address of the first character of the array it represents.

Since `scanf` needs a pointer to store the input string, using `str` directly gives the correct memory address. For example, to read a string up to 24 characters, you can use the following code:

```
char str[25];
scanf("%s", str);
```

However, `scanf("%s", &str)` is incorrect. The reason is that `&str` has the type `char (*) [25]`, which is a pointer to an array of 25 characters. But `scanf` expects a `char *`, which is a pointer to a single character. This mismatch in types leads to a compilation error.

As a specific example, consider the following program. It prompts the user to enter a string, reads it into a character array `str`, and prints the string to the terminal. The array can hold up to 24 characters plus the null terminator, meaning the input should be no more than 24 characters long to avoid overflow.

```
int main(){
    char str[25];
    printf("Enter the name (at most 24 characters): ");
    scanf("%s", str);
    printf("Name = %s\n", str);
    return 0;
}
```

The code begins by declaring a character array named `str` with a size of 25. This array is intended to store a string (a sequence of characters terminated by the null character). The function call `scanf("%s", str)` is used to read a string from the user. The `%s` format specifier tells `scanf` to read a string until it encounters white space (i.e., space, tab, or newline). The function call `printf("Name = %s \n", str)` is used to display the string that was input by the user.

## 7.5 Reading strings with white spaces

**White space** means space, tab, or newline. In many applications, we need to read an entire line of text, including spaces. For example, we may want to read someone's name, middle name, and surname, separated by spaces, as a single string. However, `scanf` with `%s` cannot read a string completely if there are spaces within it. For example, the string `IIT KGP` cannot be read as a single string due to the space between `IIT` and `KGP` (`scanf` just reads `IIT` as the string). One way to handle this is to use `getchar()`, as shown below.

### Using `getchar`

```
int main(){
    char line[101], c;
    int k = 0;
    do{
        c = getchar();
        line[k++] = c;
    }
    while (c != '\n');
    k = k-1;
    line[k] = '\0';
    return 0;
}
```

To read an entire line of text including spaces, we use `getchar()` to read each character until a newline character is encountered. The characters are stored in the `line` array, and the loop terminates when `getchar()`



reads the newline character. The last character of the array is then set to `'\0'` to properly terminate the string.

## Without using `getchar`

Here is an alternative code that reads a line of text including spaces using `scanf` only:

```
int main() {
    char line[101];
    scanf("%[^\n]", line);
    return 0;
}
```

This code uses the format specifier `"%[^\n]"` in `scanf` to read a string until a newline character is encountered, allowing spaces within the string.

Here is another code that reads all characters including newlines and terminates when it encounters the character `#`:

```
int main() {
    char line[101];
    scanf("%[^#]", line);
    return 0;
}
```

In this code:

- The format specifier `%[^#]` is used with `scanf` to read characters until the character `#` is encountered.
- The `#` character itself is not included in the `line` array.
- The null terminator `\0` is automatically appended at the end of the string.
- The symbol `^` in the specifier denotes logical not, so all characters except `#` are accepted.

For example, if the user enters `IIT_KGP PDS@2024#Autumn`, the `scanf` will store just this:

```
IIT_KGP PDS@2024\0.
```

So, if you write `printf("%s", line)` after the `scanf`, it will display `IIT_KGP PDS@2024` on the terminal.

The character `\0` is not displayed.

## 7.6 String library

The string library `<string.h>` is essential for managing and manipulating character arrays. It provides functions for copying, concatenating, comparing, and searching strings, as well as calculating their lengths. These functions help efficiently handle tasks like parsing user input and processing text data. The list of important string functions available in `<string.h>` is given below. Among these, the most commonly used are `strlen`, `strcpy`, `strncpy`, `strcat`, `strncat`, `strcmp`, and `strncmp`—you should know their use. The rest you can learn later.

### 1. String Length Function

- (i) `strlen`: Returns the length of a string.  
`size_t strlen(const char *str);`

### 2. String Manipulation Functions

- (i) `strcpy`: Copies a string to another.  
`char *strcpy(char *dest, const char *src);`

- (ii) `strncpy`: Copies up to `n` characters of a string to another.  
`char *strncpy(char *dest, const char *src, size_t n);`
- (iii) `strcat`: Concatenates two strings.  
`char *strcat(char *dest, const char *src);`
- (iv) `strncat`: Concatenates up to `n` characters of one string to another.  
`char *strncat(char *dest, const char *src, size_t n);`

### 3. String Comparison Functions

- (i) `strcmp`: Compares two strings.  
`int strcmp(const char *str1, const char *str2);`
- (ii) `strncmp`: Compares up to `n` characters of two strings.  
`int strncmp(const char *str1, const char *str2, size_t n);`

### 4. String Search Functions

- (i) `strchr`: Finds the first occurrence of a character in a string.  
`char *strchr(const char *str, int c);`
- (ii) `strrchr`: Finds the last occurrence of a character in a string.  
`char *strrchr(const char *str, int c);`
- (iii) `strstr`: Finds the first occurrence of a substring in a string.  
`char *strstr(const char *haystack, const char *needle);`

### 5. String Tokenization

- (i) `strtok`: Splits a string into tokens based on a delimiter.  
`char *strtok(char *str, const char *delim);`

### 6. Memory Manipulation Functions

- (i) `memcpy`: Copies a block of memory.  
`void *memcpy(void *dest, const void *src, size_t n);`
- (ii) `memmove`: Moves a block of memory.  
`void *memmove(void *dest, const void *src, size_t n);`
- (iii) `memcmp`: Compares two blocks of memory.  
`int memcmp(const void *str1, const void *str2, size_t n);`
- (iv) `memset`: Fills a block of memory with a specific value.  
`void *memset(void *str, int c, size_t n);`

### 7. String Duplication

- (i) `strdup`: Duplicates a string (POSIX, not part of C standard).  
`char *strdup(const char *str);`

### 8. String Error Messages

- (i) `strerror`: Returns a pointer to the string that describes the error code passed in the argument.  
`char *strerror(int errnum);`

Let's explore some examples to see how the functions from `string.h` make it easier to write concise and efficient code. Here, we'll focus on three key functions: `strlen`, `strcpy`, and `strcat`.

#### 7.6.1 `strlen`

Consider finding the length of a string. As mentioned earlier in §7.6, the function `strlen` measures the length of a string and returns that value. Recall from §7.1 that the **length** of a string is defined as the number of its constituent characters, excluding the null character. The first code snippet demonstrates how

to achieve this without using `string.h`. The second code snippet shows how to use the `strlen` function from `string.h` to perform the same task.

```
1 // without using string.h
2
3 #include <stdio.h>
4
5 int main() {
6     char s[10];
7     printf("Enter a string (up to 9 characters): ");
8     scanf("%s", s);
9
10    int len = 0;
11    while (s[len] != '\0')
12        len++;
13    printf("Length of the input string: %d\n", len);
14    return 0;
15 }
```

```
1 // using the function strlen of string.h
2
3 #include <stdio.h>
4 #include <string.h>
5
6 int main() {
7     char s[10];
8     printf("Enter a string (up to 9 characters): ");
9     scanf("%s", s);
10    int len = strlen(s);
11    printf("Length of the input string: %d\n", len);
12    return 0;
13 }
```

### 7.6.2 strcpy

Consider the problem of copying a string `s` to `t`. The first code snippet demonstrates how to do this without using `strcpy`. The second code snippet shows how to use the `strcpy` function from `string.h` to accomplish the same task.


```
1 // without using strcpy
2
3 #include <stdio.h>
4 #include <string.h>
5
6 int main() {
7     char s[10], t[10];
8     int len, i;
9     printf("Enter a string (max 9 characters): ");
10    scanf("%9s", s); // for safety against buffer overflow
11
12    len = strlen(s);
```

```

13 |   for (i = 0; i < len; i++)
14 |       t[i] = s[i];
15 |   t[len] = '\0';
16 |   printf("Copied string: %s\n", t);
17 |   return 0;
18 | }

```

```


 1 | // using strcpy
2 |
3 | #include <stdio.h>
4 | #include <string.h>
5 |
6 | int main() {
7 |     char s[10], t[10];
8 |     printf("Enter a string (max 9 characters): ");
9 |     scanf("%9s", s); // for safety against buffer overflow
10 |     strcpy(t, s);
11 |     printf("Copied string: %s\n", t);
12 |     return 0;
13 | }

```

### 7.6.3 strcat


*Concatenation* means appending one string to another. For example, concatenating @47 to India results in India@47. Consider the problem of concatenating a string `t` to a string `s`. The first code snippet demonstrates how to do this without using `strcat`. The second code snippet shows how to use the `strcat` function from `string.h` to achieve the same result.

```

 1 | #include <stdio.h>
2 | #include <string.h>
3 |
4 | int main() {
5 |     char s[10], t[10];
6 |     strcpy(s, "India");
7 |     strcpy(t, "@47");
8 |
9 |     int i = strlen(s);
10 |    for (int j = 0; t[j] != '\0'; j++) {
11 |        s[i] = t[j];
12 |        i++;
13 |    }
14 |    s[i] = '\0';
15 |
16 |    printf("%s\n", s);
17 |    return 0;
18 | }

```

```

 1 | #include <stdio.h>
2 | #include <string.h>
3 |
4 | int main() {

```

```

5 | char s[10], char t[10];
6 | strcpy(s, "India");
7 | strcpy(t, "@47");
8 | strcat(s, t);
9 | printf("%s\n", s);
10 | return 0;
11 | }

```

## 7.7 Solved problems

Note the following regarding the problems stated in this section.

1. A string should be treated as an array of characters, each of one byte, ending with the *null character*, i.e., `'\0'`, which is mandatory. An array without any `'\0'` is not a string.
2. Unless mentioned, assume that a string will contain at most 1000 characters including the null character.
3. Unless mentioned, assume that a string contains at least one non-null character.
4. A newline means the character `'\n'`, which appears as the key [Enter](#) on the keyboard.
5. By “any character” or “an arbitrary character”, we mean a character that can be typed in from the keyboard; for example, all lowercase and uppercase letters, digits, space, newline, punctuation marks, and the following symbols:

~ ! @ # ^ % \$ & \* ( ) \_ + - / { } [ ] | \ < >

1. **[Scan any character]** Scan as input any character other than newline; print it as character and also print its ASCII value. The user will type in that character followed by a newline.

```

1 | #include <stdio.h>
2 |
3 | int main(){
4 |     char a;
5 |     printf("Type any character and press <Enter>: ");
6 |     scanf("%[^\\n]c", &a); // accepts any character followed by newline
7 |     printf("Scanned = '%c' (ASCII value = %d).\n", a, a);
8 |     return 0;
9 | }

```

2. **[Scan any two characters]** Scan as input two arbitrary characters, one at a time, using two `scanf` calls, and finally print them by a single `printf` at the end. The user will type in each character followed by a newline.

```

1 | #include <stdio.h>
2 |
3 | int main(){
4 |     char a, b;
5 |     printf("Type in the 1st character and press <Enter>: ");
6 |     scanf("%[^\\n]c", &a); // accepts any character followed by newline
7 |     printf("Scanned = '%c'.\n", a);
8 |     printf("Type in the 2nd character and press <Enter>: ");
9 |     scanf("\\n%[^\\n]c", &b); // The first \\n means the newline entered earlier is skipped
10 |    printf("Scanned = '%c'.\n", b);
11 |    return 0;
12 | }

```

3. **[Scan integer and character]** Scan first as input an integer and then a non-white-space character, using two `scanf` calls, and finally print them by a single `printf` at the end.

```

1 #include <stdio.h>
2
3 int main(){
4     int n;
5     char a;
6     printf("Type in any integer and press <Enter>: ");
7     scanf("%d", &n);
8     printf("Type in any character and press <Enter>: ");
9     scanf("\n%c", &a); // The first \n means the previous <Enter> is skipped
10    printf("Scanned = %d and '%c'.\n", n, a);
11    return 0;
12 }

```

**Note:** The `\n` in `scanf("\n%c", &a)` is used to consume any leftover newline character that may be in the input buffer from the previous input against `scanf("%d", &n)`. When you input an integer and press `<Enter>`, the newline character generated by the `<Enter>` key remains in the buffer. If you don't include `\n` before `%c`, the `scanf` function would read this newline character instead of waiting for a new character input. The `\n` effectively skips over any newline characters, ensuring that `scanf("%c", &a)` correctly reads the next character entered by the user.

4. **[Bit flip]** Given as input any arbitrary character other than a newline, print the new character formed by flipping each bit excepting the leftmost one. Do it by two methods—once using loop, and once without loop.

```

1 #include <stdio.h>
2
3 int main(){
4     char a, b = 0;
5     int i;
6
7     printf("Type any character and press <Enter>: ");
8     scanf("%[^\n]c", &a); // accepts any character followed by newline
9     printf("Input = '%c' (ASCII value = %d)\n", a, a);
10
11    printf("Method 1: ");
12    for(b=0, i=1; i<=64; i=i<<1)
13        b += ((a & i) == 0)? i : 0; // flips the i-th bit
14    printf("Output = '%c' (ASCII value = %d)\n", b, b);
15
16    printf("Method 2: ");
17    b = 255 - a; // flips all 8 bits
18    b &= (a<128)? 127 : 255; // retains the leftmost bit of a
19    printf("Output = '%c' (ASCII value = %d)\n", b, b);
20
21    return 0;
22 }

```

5. **[String reversal]** Write a function that takes an alphanumeric string as argument and reverses it. For example, if it is `IIT`, then the reverse string created by the function will be `TI`. Scanning the input string and printing the reverse one must be done from `main()`. String library can't be used.

```

1 #include <stdio.h>
2 #define MAXLEN 1000
3
4 void stringReversal(char *s, char *t){
5     int n = 0, i = 0; // n = length of s
6
7     do
8         n++;
9     while(s[++i]!='\0');
10    t[n] = '\0';
11
12    for(i=0; i<n; i++)
13        t[n-1-i] = s[i];
14 }
15
16 int main(){
17     char s[MAXLEN], t[MAXLEN];
18     printf("Enter an alphanumeric string: ");
19     scanf("%s", s);
20     stringReversal(s, t);
21     printf("Reverse = %s.\n", t);
22     return 0;
23 }

```

## 7.8 Exercise problems

1. **[Alphanumeric character]** Scan any character input (excluding newline) and check whether it is alphanumeric without using ASCII values or the string library. The user will type the character followed by a newline.
2. **[Scan an arbitrary string]** Scan a string of arbitrary characters (including alphanumeric characters, spaces, symbols, etc.) and print it. For example, it should scan the following as a single string:

```
iit kgp $1951, Autumn; ... '&' "www.ac.in" +-*/~pb*& (@CSE) #_~PDS
```

The user will input the characters and terminate with a newline, which should not be treated as part of the string.

3. **[Odd characters]** Without using the string library, print the characters of an alphanumeric string occurring in odd positions. Assume that the string has at least one alphanumeric character.
4. **[Palindrome]** Write a function that takes an alphanumeric string as argument and returns 1 if it is a palindrome, and 0 otherwise. For example, **IIT** is not a palindrome but **ITI** is. After a string is taken in from `main()`, the function should be called from `main()` with that string as input, and the value returned by the function should be printed from `main()`.
- ♣ 5. **[Byte reversal]** Given a byte as input in the form of a character, print the new character formed by reversing the 7-bit string created from all but the leftmost bit. For example, **a** becomes **C**, **C** becomes **a**, **b** becomes **#**, **#** becomes **b**, etc.

**Hint:** Let's walk through the process step by step for the input character **'a'**. The ASCII value of **'a'** is 97, which is **01100001** in binary. You can mask this value using **0x7F** (= **01111111** in binary), resulting in **01100001**. (Masking here means bit-wise AND using the operator **&**.) Next, reversing the 7 bits of this binary sequence (excluding the leftmost one), producing **1000011**. This corresponds to the ASCII value 67, which is the character **'C'**.

# 8 | Pointers



When Vishwanathan Anand moves a piece across the chessboard, it feels as though he is orchestrating a symphony of pointers in a program of strategy. Each square, like a memory location, holds the potential of a piece — a knight, rook, or queen — waiting to be redefined. The piece itself, much like a pointer, remains unchanged in its essence but shifts its position, leaping from one address to another, dereferencing power with precision. Just as pointers in C unlock new data or modify the state of a variable, Anand’s skillful moves reveal the latent power of each and every chess piece. With every calculated step, he navigates an unseen realm (isn’t it amazing?!), referencing possibilities and dereferencing threats, weaving an intricate game of logic and foresight.

Yet, unlike a chessboard limited to 64 squares only, the challenges of computer programming extend far beyond this compact domain. As the size of the board expands higher and higher, the complexity of a solution and managing such an abstract space becomes unmanageable even by the most-talented human being. Here is where programming steps in, transforming vast memory spaces into ordered systems. With the help of pointers, arrays, and functions, programmers can solve problems that would overwhelm even the sharpest minds. In this larger game, practically speaking, memory is infinite, moves are countless, and mastering “position” isn’t just a victory but the key to unraveling a universe of possibilities.

## 8.1 What is a pointer?

A pointer is a variable that stores the memory address of another variable. It is denoted using an asterisk (\*) before its name.

Essentially, it stores the address of a particular byte in RAM; and this byte is the first one out of all the bytes required to store the variable.

### Quick examples

- **Declaration:** `int *p;`  
This declares `p` as a pointer to an integer.
- **Initialization:** `int n = 5; p = &n;`  
The pointer `p` is initialized to point to `n`, i.e., `p` will contain the address of `n` during execution.
- **Modification:** `*p = 10;`  
The value at the memory address stored by `p` is modified to 10.
- **Usage:** `printf("%d", *p);`  
The value stored at the address pointed to by `p` is printed.



Consider the following piece of code:

```
char a = 'w'; // declaration and initialization
char *p;      // declaration
p = &a;       // initialization
```

We can express the above three statements using two statements as follows:

```
char a = 'w'; // declaration and initialization
char *p = &a; // declaration and initialization
```

The variable `p` here is a pointer to the character `a`, termed as **character pointer**. Since the address of `a` is denoted by `&a`, we write `char *p = &a`; which means:

`p` points to `a`

With a bit more explanation, it means:

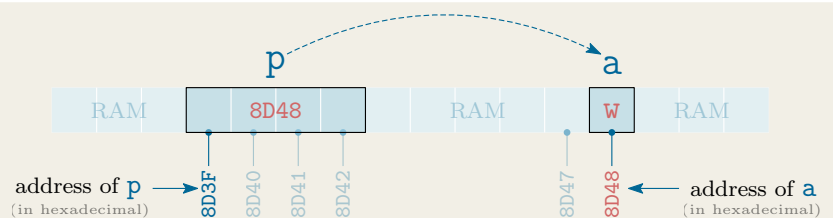
`p` stores the address of `a`

With full explanation, it means:

`p` stores the address of the byte in RAM that holds the value of `a`

The figure below illustrates the details.

`p` has 4 bytes (usual size for a pointer variable) and it stores the address (`8D48`) of `a`. The address of `p` is simply the address of its first byte. The addresses are shown here as hexadecimal numbers for convenience, but actually they are binary numbers, just like any data in computer.



## 8.2 Types of pointer

Pointers are categorized based on the data types they point to, as listed and discussed below.

1. **int pointer** (`int *`): stores the memory address of an integer variable.

**Example:** See Code 8.16, and consider its statement:

```
int *p;
```

This statement means `p` is a pointer to an integer variable, i.e., `p` will store the address of an integer variable when the program runs. The next statement `p = &n`; assigns the address of `n` to `p`, which means during the execution of the code, `p` will be assigned the address of the **first byte** of `n` (out of the 4 consecutive bytes allocated in RAM for the integer `n`).

2. **char pointer** (`char *`): stores the memory address of a character variable.

**Example:**

```
char *cp;
```

This means `cp` is a pointer to a character, i.e., `cp` will store the address of a character variable at runtime. For example: `char c = 'A'`; `cp = &c`; assigns the address of `c` to `cp`. Now, `*cp` can be used to access or modify the value of `c`. For instance, `*cp = 'B'`; followed by `*cp += 1`; will change the value of `c` to `'C'`. This is because `*cp = 'B'`; sets the value of `c` to `'B'`. The next statement `*cp += 1`; increments the value of `c` by 1, making it `'C'`, since `'C'` is the next character after `'B'` in the ASCII table.

Code 8.16: Example of working with a pointer to integer.

pointer\_int\_ex1.c

```

1  #include <stdio.h>
2
3  int main() {
4      int n = 10;
5      int *p; // p is a pointer to an integer; it can store the address of any integer
6      p = &n; // p stores the the address of n
7
8      printf("Original value of n: %d\n", n);           // It will print 10
9      *p = 15;                                         // n becomes 15
10     printf("Modified value of n using *p: %d\n", n); // It will print 15
11     printf("Result of *p + 5: %d\n", *p + 5);        // It will print 20
12     printf("Result of *p * 5: %d\n", *p * 5);        // It will print 75
13     printf("Value of n after arithmetic operation: %d\n", n); // It will print 15
14     return 0;
15 }

```

3. **float pointer** (`float *`): stores the memory address of a floating-point variable.

**Example:**

```
float *fp; float f = 5.75; fp = &f;
```

This means `fp` is a pointer to a floating-point variable. `fp = &f`; assigns the address of `f` to `fp`, i.e., `fp` will contain the address of a `f` during execution. `*fp` can be used to access or modify the value of `f`.

4. **void pointer** (`void *`): it is a generic pointer that can hold the address of any data type.

**Example:**

```
void *vp;
```

This means `vp` is a pointer that can point to any data type.

For example: `int n = 20; vp = &n`; assigns the address of `n` to `vp`. To dereference it, you need to cast it to the correct type: `printf("%d", *(int *)vp)`; prints the value of `n`.

5. **Array pointer**: stores the memory address of the first element of an array, i.e., the address of the first byte of the first element of the array.

**Example:**

```
int a[4] = {2, 3, 5, 7};
```

In this case, `a` is a pointer to the first element of the array, i.e., `a` holds the address of `a[0]`.

The array elements can be accessed using pointer arithmetic. For example: `*(a + 1)` accesses the second element of the array `a`, which is 3. `a[1]` also accesses the second element of the array `a`. Arrays and pointers in C are closely related, as array names decay into pointers when passed to functions. We'll see more in §8.5.

## 8.3 Use of pointer

Pointers allow for direct access and manipulation of data stored in memory. They are used for various purposes in C programming, some being mentioned below.

1. **Passing references (i.e., pointers) as arguments to functions**

Pointers allow you to pass the addresses of variables to functions, enabling the called function to access and modify the actual values in the caller's context. If only the value is passed without the address, the original variable remains unchanged.

**Example:**

`void update(int *p) { *p = 10; }` can modify the value of a variable passed by reference, such as `int x = 5; update(&x);`, which updates `x` to 10. On the other hand, if you define the function as `void update(int x) { x = 10; }`, and pass `x` directly, as in `update(x);`, the value of `x` will not be changed in the caller function. This is because the `x` in the caller and the `x` in the `update` function are different variables, as explained in §6.7.

**2. Accessing arrays and strings directly via memory addresses**

Array elements and string characters can be accessed through pointers, allowing efficient navigation through memory.

**Example:**

`char *p = "Hello"; printf("%c-%c-%c", *p, *(p + 1), *(p + 2));` prints `H-e-l`.

The first statement assigns the base address of the string "Hello" to `p`; i.e. `p` stores the address of 'H'. Thus, the value of `*p` is 'H'. The expression `p + 1` calculates the address of the second character, so `*(p + 1)` equals ('e'), which is printed. Similarly, `*(p + 2)` accesses the third character, 'l', which is also printed.

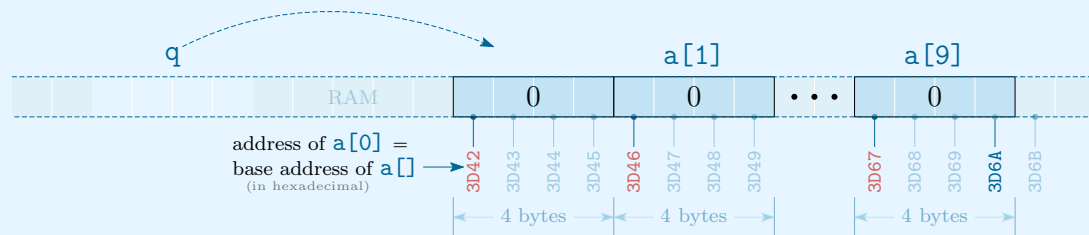
**3. Dynamic memory allocation**

Pointers are essential for allocating memory dynamically at runtime using functions like `malloc`, `calloc`, and `realloc` from `stdlib.h`. Each of them returns a pointer of type `void *`, which is a **generic pointer** not associated with any specific data type. Therefore, typecasting is required when assigning the returned address to a pointer of a specific type.

**Example:**

`int *p = (int *)malloc(10 * sizeof(int));` allocates memory for an array of 10 integers but they are uninitialized, i.e., contain **garbage values**. `p` is a pointer that stores the base address of the allocated memory.

`int *q = (int *)calloc(10, sizeof(int));` also allocates an array for 10 integers, with all 10 values initialized to zero, as illustrated below.

**4. Passing large structures to functions efficiently**

Instead of passing large structures by value, passing a pointer to the structure avoids copying the entire data. (We'll study it later.)

**Example:**

`void fun(struct MyStruct *s);`

Here, `fun` takes a pointer to a structure named `MyStruct`. This allows `fun` to modify the structure's data directly by accessing its memory location fixed for the caller function, eliminating the overhead of copying the structure.

**5. Supporting data structures like linked lists, trees, etc.**

Pointers are used to build dynamic data structures such as linked lists and trees, where each node contains a pointer to the next node or child. (We'll study linked lists later; trees are not in this course.)

**Example:**

`struct Node *next;`

In a linked list, each node points to the next node using a pointer, allowing traversal through the list dynamically.

## 8.4 Operations with pointers and dereferenced pointers

The **reference operator** `&` operates on a single variable and returns the address of that variable. The **dereference operator** `*` operates on an address and returns the value stored at that address. Let's see what happens by the execution of the statements in Code 8.16. For convenience, the declarations and assignments from that code are given below.

```
int n = 10;
int *p;
p = &n;
*p = 15;
```

### 1. Declaration and initialization:

- `int n = 10;` declares and initializes an integer variable `n` with the value `10`.
- `int *p;` declares a pointer `p` that can point to (**only and any**) integer variable.

### 2. Pointer assignment:

- `p = &n;` assigns the address of `n` to the pointer `p`. Here, the reference operator `&n` operates on `n` and returns the address of `n` to `p`.

### 3. Modification via pointer:

- `*p = 15;` modifies the value at the address `p` points to. Thus, `n` becomes `15`. The statement `*p = 15` involves **dereferencing**, which is discussed in §8.4.1.

`int *p;` means `p` is a pointer to an integer, which means `*p` is an integer. Hence, assigning the number `15` to `*p` is a valid operation. In fact, `*p = n;` is also valid for the same reason.

### 8.4.1 Dereferencing

**Dereferencing** a pointer `p` is written as `*p`. It means using the address stored in `p` to access the value at that address. For example, the statement `*p = 15;` involves dereferencing `p`, as explained below.

1. `p = &n;` makes `p` store the memory address of the variable `n`.
2. Dereferencing `p` enables using the address of `n` (stored in `p`) to access and change the value of `n`.
3. `*p = 15;` changes the value of `n` to `15` through dereferencing.

To understand more about dereferencing, consider the following piece of code. In this code, the last two lines are added to the previous code we just discussed about.

```
int n = 10, k;
int *p;
p = &n;
*p = 15;
*p = *p + 5;
k = *p * 5;
```

Below is an explanation about the last two lines in the above code.

1. `*p = *p + 5;`
  - This line first dereferences the pointer `p`, which points to the variable `n`.
  - The expression `*p` accesses the current value of `n`. Due to the previous line, `n` was set to `15`.

- The statement `*p = *p + 5;` means we take the current value of `n` (which is 15), add 5 to it, and store the result back into `n`. Thus, the new value of `n` becomes  $15 + 5 = 20$ .
2. `k = *p * 5;`
- This line also dereferences the pointer `p`, which still points to `n`.
  - After the previous operation, the value of `n` is now 20.
  - The expression `*p * 5` calculates  $20 \times 5 = 100$ , which is assigned to the variable `k`.

## 8.5 Pointer to 1D array

### 8.5.1 Usefulness

A pointer to a 1D array is useful for several reasons:

1. Instead of handling individual elements or copying entire arrays, a pointer allows referencing the array's memory address, enabling efficient access to elements in constant time.
2. It is particularly helpful when passing arrays to functions, as passing a pointer avoids creating a copy of the entire array, reducing memory overhead and computational cost.
3. Pointers facilitate dynamic memory allocation, allowing flexibility in handling arrays of variable sizes and simplifying complex operations like iterating over or modifying array contents.

### 8.5.2 Indexing with pointer

Array elements are indexed starting from 0, and indexing is crucial for accessing elements in a 1D array. Below are some important points regarding array indexes and pointers.

1. Each element of an array is stored sequentially in memory, with the index serving as an offset from the array's starting memory address.
2. By specifying an index, we can directly access any array element based on its position. For instance, in the array `arr`, the element at index `i` can be retrieved using `arr[i]`.
3. The expression `arr[i]` is equivalent to `*(arr + i)`. Both notations access the element at index `i`. In `arr[i]`, the index is used to directly retrieve the element. In `*(arr + i)`, pointer arithmetic is used, where `arr` is treated as a pointer, and `i` is added to the base address of the array to access the element stored at that location.
4. This indexing mechanism provides constant-time access to elements and is essential for iterating over arrays, performing operations like searching, sorting, or modifying specific elements.

Code 8.17 demonstrates the equivalence between `a[1]` and `*(a + 1)`. The expression `a[1]` accesses the second element of the array, which is 3. Similarly, `*(a + 1)` accesses the same element, as `a + 1` gives the address of the second element of the array, and the `*` operator dereferences that address to retrieve the value.

However, `*(a + 1)` and `*a[1]` are not the same. `*(a + 1)` accesses the second element of the array `a` using pointer arithmetic to move the pointer to the second element and dereferencing it to get the value stored at that location. On the other hand, `*a[1]` attempts to dereference the value stored at `a[1]`. Since `a[1]` is an integer but not a pointer, this would lead to a compilation error or undefined behavior because dereferencing a non-pointer value is not valid.

Code 8.18 shows how to print addresses using two different ways for the elements in array. The format `"%p"` specifies the addresses to be printed as hexadecimal numbers, the first two characters (`0x`) implying that they are so.

**Code 8.17:** Example of working with a pointer to array.

`pointer1dArrayEx1.c`

```

1 | #include <stdio.h>
2 | int main() {
3 |     int a[] = {2, 3, 5, 7};
4 |     printf("a[1] = %d\n", a[1]);           // this will print 3
5 |     printf("*(a + 1) = %d\n", *(a + 1)); // this will also print 3
6 |     return 0;
7 | }
```

**Code 8.18:** Printing addresses and values of elements in array.

`pointer1dArrayEx2.c`

```

1 | #include <stdio.h>
2 | int main() {
3 |     int a[] = {2, 3, 5, 7};
4 |     for (int i = 0; i < 4; i++)
5 |         printf("i = %d: a+i = %p, &a[i] = %p, a[i] = %d\n", i, a+i, &a[i], a[i]);
6 |     return 0;
7 | }
```

The addresses are fixed during execution of the code, so they change from one execution to the other, as you can see below. Since `a` is an array of integers and each integer takes 4 bytes, the addresses of two consecutive elements differ by 4.

```

$ ./a.out
i = 0: a+i = 0x7fffc2527520, &a[i] = 0x7fffc2527520, a[i] = 2
i = 1: a+i = 0x7fffc2527524, &a[i] = 0x7fffc2527524, a[i] = 3
i = 2: a+i = 0x7fffc2527528, &a[i] = 0x7fffc2527528, a[i] = 5
i = 3: a+i = 0x7fffc252752c, &a[i] = 0x7fffc252752c, a[i] = 7
$ ./a.out
i = 0: a+i = 0x7fff2f412470, &a[i] = 0x7fff2f412470, a[i] = 2
i = 1: a+i = 0x7fff2f412474, &a[i] = 0x7fff2f412474, a[i] = 3
i = 2: a+i = 0x7fff2f412478, &a[i] = 0x7fff2f412478, a[i] = 5
i = 3: a+i = 0x7fff2f41247c, &a[i] = 0x7fff2f41247c, a[i] = 7
```

When the addresses are written as hexadecimal numbers, you have to be careful while adding a number with an address. For example, the hexadecimal number `528` means  $5 \times 16^2 + 2 \times 16^1 + 8 \times 16^0 = 1320$  in the decimal number system. So, adding 4 with `528` gives the decimal number 1324, which maps to the hexadecimal number `52c`.

## 8.6 Pointer arithmetic

1. Like other variables, **contents** of pointer variables can be used in **expressions**.

```

int *p1, *p2, sum, prod; float x;
sum = *p1 + *p2;
prod = *p1 * *p2; prod = (*p1) * (*p2);
*p1 = *p1 + 2; x = *p1 / *p2 + 5;
```

2. The following are allowed in C.

- (i) Add an integer to a pointer.

```
int i, a[5];
for (i=0; i<5; i++)
    printf("%u ", (unsigned)(a+i));
```

Output: 3214950148 3214950152 3214950156 3214950160 3214950164.

Here addresses are printed as unsigned integers using "%u" format. In Code 8.18, they have been printed as hexadecimal numbers. Either one is fine, as far as they are properly understood and used.

- (ii) Subtract an integer from a pointer.  
 (iii) Subtract one pointer from another of the same type.  
 Note: If `p1` and `p2` are both pointers to the same array, then `p2 - p1 + 1` gives the number of elements starting from `p1` and ending at `p2`.

3. The following are **not allowed** in C.

- (i) Add two pointers (although subtraction is allowed, as mentioned before).  
`p1 = p1 + p2;` — not allowed  
 (ii) Multiply or divide a pointer in an expression.  
`p1 = p2 / 5;` — not allowed  
`p1 = p1 - p2 * 10;` — not allowed

## 8.7 Scale factor: `sizeof()`

When an integer `i` is added to or subtracted from an integer-pointer `p`, the actual value added to or subtracted from `p` is not simply `i`, but rather `i` multiplied by the scale factor `sizeof(int)`. This is because pointer arithmetic accounts for the size of the data type being pointed to, ensuring that the pointer moves correctly across memory locations based on the size of the type.

The `sizeof()` is actually an operator in C, not a function. It is a compile-time operator that determines the size, in bytes, of a variable or data type. It does not perform any runtime computation and is evaluated at compile-time only.

See Code 8.19 for an example of how the `sizeof()` function impacts pointer arithmetic, and also see its output therein. The statement `pi = pi+10;` implies that the address stored in `pi` increases by

**Code 8.19:** Example of **scale factor**: `sizeof()`.

Here is the output:

```
pi = 3214711672, pc = 3214711679
pi = 3214711712, pc = 3214711689
```

`scaleFactor_sizeof.c`

```
1 #include <stdio.h>
2 int main(){
3     int i=1, *pi;
4     char c='A', *pc;
5     pi = &i, pc = &c;
6     printf("pi = %u, pc = %u \n", (unsigned)pi, (unsigned)pc);
7
8     pi = pi+10; pc = pc+10;
9     printf("pi = %u, pc = %u \n", (unsigned)pi, (unsigned)pc);
10    return 0;
11 }
```

`sizeof(int) × 10`. Given that an integer takes 4 bytes, this results in  $4 \times 10 = 40$  bytes being added to `pi`, effectively shifting the pointer forward by 10 integers in memory. Similarly, the statement `pc = pc + 10;` advances the character pointer `pc` by 10 characters or 10 bytes, since each character occupies 1 byte of memory.

## 8.8 Passing pointers to a function

Passing pointers to a function allows direct manipulation of the original variables of the **caller function**, making it possible for the **called function** to access and modify those values stored at specific memory addresses. The key points are:

1. When a pointer to a variable `var` (e.g., integer, character, array, or structure) defined in the caller function is passed to the **called function**, the called function receives the memory address of `var`, rather than a copy of its value.
2. This enables the called function to access the variable of the caller function at that address, which is useful for tasks like updating multiple values (e.g., in array or structure) or working with dynamically allocated memory.
3. Syntax example: `void f(int *pi);` is meant to pass an integer pointer while calling the function `f`. Inside the function, dereferencing the pointer (e.g., `*pi`) allows access to and modification of the value it points to.

**Code 8.20: Incorrect** passing of arguments.

`pointerArgFun_incorrect.c`

```

1 | #include <stdio.h>
2 |
3 | void f(int i){ i = 10*i; }
4 |
5 | int main(){
6 |     int i = 1;
7 |     f(i);
8 |     printf("i = %d\n", i); // prints 1
9 |     return 0;
10| }
```

**Code 8.21: Correct** passing of arguments.

`pointerArgFun_correct.c`

```

1 | #include <stdio.h>
2 |
3 | void f(int *pi){ *pi = 10 * (*pi); }
4 |
5 | int main(){
6 |     int i = 1;
7 |     f(&i);
8 |     printf("i = %d\n", i); // prints 10
9 |     return 0;
10| }
```

As shown in Code 8.20, passing the value of the variable `i` to the function `f` does not change the value of `i` in `main()`. This is because the `i` in `f` is local to the function `f`, and the `i` in `main()` is a separate variable, local to `main()`. However, passing a pointer to `i` to the function `f`, as shown in Code 8.21, does modify the value of `i` in `main()`. This works because the pointer contains the address of `i` in `main()`, allowing `f` to access and modify the original variable.



To swap the values of two variables, `a` and `b`, using the function `swap`, the correct approach is shown in Code 8.22. This method uses pointers to the variables. If the values of the variables are passed directly as arguments, the swap will fail, as demonstrated in Code 8.23.

---

**Code 8.22: Correct** code for swapping (arguments passed by pointers).

`swapFun_correct.c`

```
1 | #include <stdio.h>
2 |
3 | void swap(int *x, int *y){
4 |     int t;
5 |     t = *x; *x = *y; *y = t;
6 | }
7 |
8 | int main(){
9 |     int a = 5, b = 10;
10 |    swap (&a, &b);
11 |    printf ("a = %d, b = %d\n", a, b); // prints a = 10, b = 5
12 |    return 0;
13 | }
```

---

**Code 8.23: Incorrect** code for swapping (arguments passed by values).

`swapFun_incorrect.c`

```
1 | #include <stdio.h>
2 |
3 | void swap (int x, int y){
4 |     int t;
5 |     t = x; x = y; y = t;
6 | }
7 |
8 | int main(){
9 |     int a = 5, b = 10;
10 |    swap(a, b);
11 |    printf ("a = %d, b = %d\n", a, b); // prints a = 5, b = 10
12 |    return 0;
13 | }
```

---

`scanf()` versus `printf()`

```
int x, y;
scanf("%d %d", &x, &y);
printf("%d %d %d", x, y, x + y);
```

Why use `&` in `scanf()` but not in `printf()`?

The function `printf()` requires only the value to display it, whereas `scanf()` needs the address to store a value. Thus, `&` is used to pass the variable's address to `scanf()` via a pointer.

## 8.9 Dynamic memory allocation

Many times, we encounter situations where the amount of data is dynamic due to the following reasons:

1. The amount of data cannot be predicted beforehand.
2. The number of data items keeps changing during program execution.

Such scenarios are more effectively managed using **dynamic memory management techniques**.

**Why not use static arrays?** In C, the number of elements in an array must be specified during **compilation**. This often leads to issues like:

1. **Over-allocation**, which wastes memory space.
2. **Under-allocation**, leading to program failure.

Through **dynamic memory allocation**, the required memory space can be allocated during **execution time**, allowing flexibility as the data size changes. C supports this with a set of library functions for dynamic memory management. **Dynamic memory** is allocated from the free memory pool, also known as the **heap**, during run time.

### 8.9.1 Memory Allocation Functions

1. **malloc**: Allocates the requested number of bytes and returns a pointer to the first byte of the allocated space.

```
int *p;
p = (int *)malloc(100 * sizeof(int));
```

This allocates memory for 100 integers.

2. **calloc**: Allocates space for an array of elements, initializes them to zero, and returns a pointer to the memory.

```
int *q;
q = (int *)calloc(100, sizeof(int));
```

This allocates memory for 100 integers and initializes all elements to zero.

3. **free**: Frees previously allocated space, making it available for future use.

```
free(p);
free(q);
```

This releases the memory pointed to by `p` and `q`.

4. **realloc**: Modifies the size of previously allocated space to accommodate changing needs.

```
p = (int *)realloc(p, 200 * sizeof(int));
```

This resizes the memory block originally allocated to hold 100 integers to now hold 200 integers. The previous data, if any, are preserved. For example, if there were initially 100 integers stored in the array pointed by `p`, these 100 integers remain intact in the resized array.

These functions are defined in `stdlib.h`. Use the command `man` on your Linux/Ubuntu terminal to view more details. For example, on writing the following command in my Ubuntu computer:

```
man malloc
```

what I can see is partially shown below.

```
NAME
    malloc, free, calloc, realloc - allocate and free dynamic memory

SYNOPSIS
    #include <stdlib.h>

    void *malloc(size_t size);
    void free(void *ptr);
```

```
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
void *reallocarray(void *ptr, size_t nmemb, size_t size);
```

#### DESCRIPTION

The `malloc()` function allocates `size` bytes and returns a pointer to the allocated memory. The memory is not initialized. If `size` is 0, then `malloc()` returns either `NULL`, or a unique pointer value that can later be successfully passed to `free()`.

.....

### 8.9.2 Dynamic memory allocation and error handling

The `malloc()` function always allocates a block of contiguous bytes. The allocation can fail if sufficient contiguous memory space is not available. If it fails, `malloc` returns `NULL`. So whenever a program calls `malloc`, it should check whether the allocation has been done. If not, the program should be exited to prevent error. This is how the lines of code should be written:

```
int *p;
p = (int *)malloc(100 * sizeof(int));
if (p == NULL) {
    printf("Memory cannot be allocated");
    exit(1);
}
```

Alternatively, it can be written as follows:

```
int *p;
if ((p = (int *)malloc(100 * sizeof(int))) == NULL) {
    printf("Memory cannot be allocated");
    exit(1);
}
```

The same `NULL` check must be done for `calloc` and `realloc`, since they also return `NULL` if memory allocation fails.

`exit()` is a function that immediately terminates the program execution. Unlike `return`, which is a statement that simply returns control to the calling function, `exit` completely exits the program and ends all its activities. You must `#include <stdlib.h>` to use `exit()`.

While `return` is used to return a value from a function or terminate a function's execution, `exit()` is used when the program needs to be stopped immediately, often due to an error or a critical condition.

## 8.10 Solved problems

1. **[First upper-case letter in a string]** In this problem, we will see a function that returns a pointer. Here is the problem statement: Given a string as input, find its first upper-case letter, if any. To do this, we use a user-defined function `firstUpper` that returns a pointer to that letter if it exists, and returns `NULL` otherwise.

A function should not return a pointer to its local variable, because after the function returns, the local variable no longer exists, and thus the address stored in that pointer is invalid.

```

1 #include <stdio.h>
2
3 char *firstUpper(char s[]){ // We can also write char *s as argument
4     while(*s){
5         if ((*s >= 'A') && (*s <= 'Z'))
6             return s; // pointer to the 1st uppercase letter
7         else ++s;
8     }
9     return NULL; // no uppercase letter exists
10 }
11
12 int main (){
13     char *p, s[100]; // assuming that the s consists of at most 100 characters
14     scanf("%s", s);
15
16     p = firstUpper(s);
17
18     if (p) // p is not NULL
19         printf("%c found\n", *p);
20     else // p is NULL
21         printf("No upper-case letter found\n");
22
23     return 0;
24 }

```

2. **[Dynamic array management]** Write a program that dynamically manages an array of integers. The user will input a number of elements, and the program will allocate memory accordingly. The program should also allow resizing the array if the user decides to add more elements than initially specified.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5     int *arr, size, newSize, i;
6
7     printf("Enter initial size of the array: ");
8     scanf("%d", &size);
9
10    // Allocate memory for the array
11    arr = (int *)malloc(size * sizeof(int));
12    if (arr == NULL) {
13        printf("Memory allocation failed\n");
14        return 1;
15    }
16
17    // Input elements
18    for (i = 0; i < size; i++) {
19        printf("Enter element %d: ", i);
20        scanf("%d", &arr[i]);
21    }

```

```

22
23 // Print elements
24 printf("Array elements: ");
25 for (i = 0; i < size; i++) {
26     printf("%d ", arr[i]);
27 }
28 printf("\n");
29
30 // Ask if the user wants to resize the array
31 printf("Enter new size for the array: ");
32 scanf("%d", &newSize);
33
34 // Resize the array
35 arr = (int *)realloc(arr, newSize * sizeof(int));
36 if (arr == NULL) {
37     printf("Memory reallocation failed\n");
38     return 1;
39 }
40
41 // Input new elements
42 for (i = size; i < newSize; i++) {
43     printf("Enter element %d: ", i);
44     scanf("%d", &arr[i]);
45 }
46
47 // Print all elements
48 printf("Updated array elements: ");
49 for (i = 0; i < newSize; i++) {
50     printf("%d ", arr[i]);
51 }
52 printf("\n");
53
54 // Free allocated memory
55 free(arr);
56
57 return 0;
58 }
59
60 /* Input and output:
61
62 Enter initial size of the array: 3
63 Enter element 0: 2
64 Enter element 1: 3
65 Enter element 2: 5
66 Array elements: 2 3 5
67 Enter new size for the array: 4
68 */

```

3. [Dynamic array management with user-defined functions] Rewrite the last program with user-defined functions.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 // Function prototypes

```

```
5 int** allocateMatrix(int rows, int cols);
6 void readMatrix(int **matrix, int rows, int cols);
7 void reallocateMatrix(int ***matrix, int oldRows, int newRows, int newCols);
8 void printMatrix(int **matrix, int rows, int cols);
9
10 int main() {
11     int **matrix;
12     int rows, cols, newRows, newCols;
13
14     printf("Enter number of rows and columns: ");
15     scanf("%d %d", &rows, &cols);
16
17     // Allocate memory for the matrix
18     matrix = allocateMatrix(rows, cols);
19
20     // Input matrix elements
21     readMatrix(matrix, rows, cols);
22
23     // Print matrix
24     printMatrix(matrix, rows, cols);
25
26     // Ask if the user wants to resize the matrix
27     printf("Enter new number of rows and columns: ");
28     scanf("%d %d", &newRows, &newCols);
29
30     // Reallocate the matrix
31     reallocateMatrix(&matrix, rows, newRows, newCols);
32
33     // Input new matrix elements
34     readMatrix(matrix, newRows, newCols);
35
36     // Print updated matrix
37     printMatrix(matrix, newRows, newCols);
38
39     // Free allocated memory
40     for (int i = 0; i < newRows; i++) {
41         free(matrix[i]);
42     }
43     free(matrix);
44
45     return 0;
46 }
47
48 // Function definitions
49 int** allocateMatrix(int rows, int cols) {
50     int **matrix = (int **)malloc(rows * sizeof(int *));
51     if (matrix == NULL) {
52         printf("Memory allocation failed\n");
53         exit(1);
54     }
55     for (int i = 0; i < rows; i++) {
56         matrix[i] = (int *)malloc(cols * sizeof(int));
57         if (matrix[i] == NULL) {
58             printf("Memory allocation failed\n");
```

```
59     exit(1);
60     }
61 }
62 return matrix;
63 }
64
65 void readMatrix(int **matrix, int rows, int cols) {
66     for (int i = 0; i < rows; i++) {
67         for (int j = 0; j < cols; j++) {
68             printf("Enter element [%d][%d]: ", i, j);
69             scanf("%d", &matrix[i][j]);
70         }
71     }
72 }
73
74 void reallocateMatrix(int ***matrix, int oldRows, int newRows, int newCols) {
75     *matrix = (int **)realloc(*matrix, newRows * sizeof(int *));
76     if (*matrix == NULL) {
77         printf("Memory reallocation failed\n");
78         exit(1);
79     }
80     for (int i = oldRows; i < newRows; i++) {
81         (*matrix)[i] = (int *)malloc(newCols * sizeof(int));
82         if ((*matrix)[i] == NULL) {
83             printf("Memory allocation failed\n");
84             exit(1);
85         }
86     }
87     for (int i = 0; i < newRows; i++) {
88         (*matrix)[i] = (int *)realloc((*matrix)[i], newCols * sizeof(int));
89         if ((*matrix)[i] == NULL) {
90             printf("Memory reallocation failed\n");
91             exit(1);
92         }
93     }
94 }
95
96 void printMatrix(int **matrix, int rows, int cols) {
97     printf("Matrix:\n");
98     for (int i = 0; i < rows; i++) {
99         for (int j = 0; j < cols; j++) {
100             printf("%d ", matrix[i][j]);
101         }
102         printf("\n");
103     }
104 }
```

## 8.11 Exercise problems

1. **[Dynamic Array Initialization and Access]** Write a program that dynamically allocates memory for an array of integers based on user input. The user specifies the number of elements, and the program initializes the array with Fibonacci numbers up to that length. Implement a function to print the array. Use `malloc` to allocate memory and `free` to deallocate it.

2. **[Sum and Maximum Value Calculation]** Create a program that allocates memory for an array of integers based on user input. After filling the array with values, compute and display the sum and maximum value of the elements. Implement functions for allocation, input, sum and maximum value computation, and printing. Use `calloc` for memory allocation and `free` to deallocate.
3. **[Dynamic Array Sorting]** Implement a program that dynamically allocates memory for an array of integers. The user inputs the number of elements and their values. After sorting the array in ascending order, print the sorted array. Implement functions for allocation, input, sorting, and printing. Use `malloc` to allocate memory and `free` to deallocate. After sorting is discussed in the class, you can implement this.
4. **[Array Element Replacement]** Write a program that allocates memory for an array of integers. After the user inputs values, prompt the user to specify an index and a new value. Replace the value at the specified index with the new value and print the updated array. Implement functions for allocation, input, replacement, and printing. Use `malloc` to allocate memory and `free` to deallocate.
5. **[Frequency Counter of Elements]** Create a program that dynamically allocates memory for an array of integers in the interval  $[a, b]$ , where  $a$  and  $b$  are input. After filling the array with values, compute the frequency of each unique element in the array and display the results. Implement functions for allocation, input, frequency counting, and printing. Use `malloc` for memory allocation and `free` to deallocate. You can use an additional dynamic array.
- ♣ 6. **[Maximum Subarray Sum]** Given an array of integers, the task is to find a/the contiguous subarray with the maximum sum. Implement a program that uses `malloc` to allocate memory for the array depending on the number of elements (given as input). Then, populate the array using user's input and find the maximum subarray sum. You shouldn't use more than one loop. Ensure that the program frees the allocated memory using `free`.

A **contiguous subarray** of an array consists of consecutive elements of the array. Clearly, if all elements are positive, then the maximum subarray sum equals the sum of all elements. Otherwise, it will be a proper subarray. For example, in the array `[-18, -9, 16, -11, 7, 15, -23, 20, -21, 17]`, the maximum subarray sum is 27, and it corresponds to the subarray `[16, -11, 7, 15]`.

- ♣ 7. **[Longest Increasing Subsequence]** Given an array of distinct integers, determine the length of the longest increasing subsequence. If using nested loops, ensure that there are exactly two loops: one inside the other. Your program should dynamically allocate memory for the array using `malloc`, based on the number of elements provided by the user, and deallocate the memory using `free`.

A **subsequence** is a sequence derived from another sequence by deleting some or no elements without changing the order of the remaining elements.

For example, the array `[18, 9, 16, 11, 7, 15, 23, 20, 21, 17]` has a unique longest increasing subsequence: `[9, 11, 15, 20, 21]`, which has a length of 5. Any other increasing subsequence has length less than 5.

- ♣ 8. **[Equal-Sum Partition]** Given an array of single-digit integers, determine if it can be partitioned into two subsets with equal sum. Your program should print **Yes** if it is possible, and **No** otherwise. Dynamically allocate memory for the array using `malloc`, based on the number of elements provided by the user, and deallocate the memory using `free`.

A **partition** of an array refers to dividing it into two subsets such that they are disjoint and their sums are equal. For example, the array `[-3, 5, -1, 0, -3]` can be partitioned into `[-3, 5, -3]` and `[-1, 0]` where both subsets have a sum of  $-1$ . But `[-3, 5, -1, 0]` doesn't admit any equal-sum partition.



## 9 | Two-dimensional arrays

In Chapter 5, we discussed how a 1D array essentially stores a **list** of elements. Many applications, however, require storing data in the form of a **table**. For this, we need a two-dimensional array. For example, to store a table containing the marks of  $n$  students in  $k$  subjects, we use a 2D array where each row represents a student and each column represents a subject. The table below illustrates an example with  $n = 4$  students and  $k = 5$  subjects. In this array, the rows correspond to students, and the columns represent the subjects.

71	82	90	63	76
68	75	80	70	72
88	74	85	76	80
50	65	68	40	70

(9.1)

### 9.1 Examples

Here are some common examples of tables that can be stored in 2D arrays:

- 1. Multiplication Table:** A table showing the products of pairs of numbers (e.g., a 10-by-10 table showing products from  $1 \times 1$  to  $10 \times 10$ ).
- 2. Logarithm Table:** A table used to find the logarithms of numbers to a certain base (e.g., base 10). It typically contains logarithmic values for various numbers.
- 3. Periodic Table of Elements:** The periodic table can be stored in a 2D array, with rows representing periods and columns representing groups.
- 4. Gray-scale Image:** A 2D array can represent a gray-scale image, where each element of the array holds the pixel intensity value.
- 5. Seating Arrangement:** Seating plans for events, classrooms, trains, or airplanes can be represented by a 2D array, where each element denotes a seat.
- 6. Chess Board:** A chessboard layout can be represented as an  $8 \times 8$  matrix, where each element denotes a piece or an empty space.
- 7. Sudoku Puzzle:** A  $9 \times 9$  grid can represent a Sudoku board, where each cell contains numbers or placeholders for solving the puzzle.
- 8. Matrix (Math):** Any mathematical matrix, which is essentially a 2D array of numbers, such as for linear algebra problems.
- 9. Game Board (e.g., Tic-Tac-Toe):** A 3-by-3 array can represent a Tic-Tac-Toe game board, where each cell holds a player's mark or remains empty.
- 10. Distance Table:** A 2D array can represent all inter-node distances (e.g., all inter-airport distances in India) in a 'graph'.

## 9.2 Declaration (static)

A 2D array is defined using two indexes: the first for row and the second for column. The syntax for declaring is as follows:

```
type arrayName[rows][columns];
```

If its name is `arr`, it has `m` rows and `n` columns, and it stores integers, then it should be declared as follows:

```
int arr[m][n];
```

Let's consider the table given in (9.1). To store it in a 2D array named `marks`, we write the following declaration:

```
int marks[4][5];
```

Each element is denoted by `marks[i][j]` in which the first index `i` denotes the row (student), and the second index `j` denotes the column (subject). They are referred to as **row index** and **column index**, respectively.

Similar to a 1D array, indexing for each dimension starts from 0. Since `marks` has 4 rows and 5 columns, the row indexes range from 0 to 3, and the column indexes range from 0 to 4. Thus, the marks of the 1st student in the 1st subject is `marks[0][0] = 71`. Similarly, the marks of the 2nd student in the 3rd subject is `marks[1][2] = 80`.

Here are some more examples:

```
int marks[2000][5];
char Sudoku[9][9];
float sales[12][25];
double matrix[100][100];
```

(9.2)

There are several other ways to declare 2D arrays, which will be discussed in §9.8. Also note that, instead of simple elements such as numbers or characters, a 2D array can also store more complex data types, such as structures, which we will study later (in the chapter on structures).

**Q1** Write the amount of memory space consumed for each array declared in (9.2).

**Q2** You have 1000 two-dimensional points with integer coordinates. Write a declaration for a 2D array that will contain their  $(x, y)$  coordinates.

**Q3** You have 1000 three-dimensional points with integer coordinates. Write a declaration for a 2D array that will contain their  $(x, y, z)$  coordinates.

**Q4** Consider a special version of Question 2 in which for all 1000 points, all the coordinates are integers in  $[0, 127]$ . Can you store them in a 2D array that will be smaller than the one used for Question 2? Justify.

**Q5** You have  $n$  circles with integer centers and integer radii. Write a declaration for a 2D array that will contain their centers and radii. How many bytes are needed for this array?

**Q6** Write five practical applications (each within 25 words) where 2D arrays can be used. Write for each of them how the arrays should be declared.

## 9.3 Initialization

After declaring a 2D array, we can initialize it later by filling it with elements taken as input during execution. Alternatively, we can initialize a 2D array at the time of declaration. The syntax for this is shown below for the array `marks` from (9.1).

```
int marks[4][5] = {
    {71, 82, 90, 63, 76},
    {68, 75, 80, 70, 72},
    {88, 74, 85, 76, 80},
    {50, 65, 68, 40, 70}
};
```

Alternatively, you can write it all in a single line:

```
int marks[4][5] = {{71, 82, 90, 63, 76}, {68, 75, 80, 70, 72}, {88, 74, 85, 76, 80},
{50, 65, 68, 40, 70}};
```

The convention here is easy to remember: A 2D array can be viewed as a 1D array of 1D arrays. Each 1D array is enclosed in curly braces, with its elements separated by commas.

Code 9.24 provides an example where a 4-by-5 2D array is declared, initialized using user input, and used to compute the total marks of 4 students.

**Code 9.24:** Example of working with a 2D array named `marks`.

`2dArrayStudentMarks.c`

```
1 | #include <stdio.h>
2 |
3 | #define ROWS 4
4 | #define COLS 5
5 |
6 | int main(){
7 |     int marks[ROWS][COLS], row, col, total;
8 |
9 |     for (row = 0; row < ROWS; row++){
10 |         for (col = 0; col < COLS; col++){
11 |             printf("Enter marks for student %d, course %d: ", row + 1, col + 1);
12 |             scanf("%d", &marks[row][col]);
13 |         }
14 |     }
15 |
16 |     for (row = 0; row < ROWS; row++){
17 |         for (total = 0, col = 0; col < COLS; col++){
18 |             total += marks[row][col];
19 |             printf("Total marks for student %d: %d\n", row + 1, total);
20 |         }
21 |     }
22 |     return 0;
23 | }
```

**Q7** Revise Code 9.24 so that it prints the average marks of each student and the average marks for each subject.

## 9.4 Operations

Each element of the 2D array can be treated as a usual variable and operated on using the usual operands for its corresponding datatype. For example, these are all valid for the array `marks`:

```
marks[0][0] = 50;           // Assign 50 to the first element
marks[3][4] *= 2;         // Multiply the value of marks[3][4] by 2
marks[2][1] /= 3;        // Divide the value of marks[2][1] by 3
marks[1][0] %= 10;       // Calculate the remainder of marks[1][0] divided by 10
marks[0][2] += 15;       // Add 15 to the value of marks[0][2]
marks[3][3]--;          // Decrement the value of marks[3][3] by 1
marks[1][4]++;          // Increment the value of marks[1][4] by 1
marks[1][2] = marks[0][1] + 10; // Assign the sum of marks[0][1] and 10 to marks[1][2]
int x = marks[2][3] - 5; // Subtract 5 from marks[2][3] and store the result in x
```

Code 9.25 provides an example where a 2D array named `marks` is first declared and initialized, and then its elements are modified depending on the user's choice.

**Code 9.25:** Revising the values in a 2D array named `marks`.

`2dArrayStudentMarksRevise.c`

```
1 | #include <stdio.h>
2 |
3 | int main() {
4 |     int marks[4][5] = {
5 |         {71, 82, 90, 65, 76}, {68, 75, 80, 70, 72},
6 |         {88, 74, 85, 76, 80}, {50, 65, 68, 40, 70}
7 |     };
8 |
9 |     int row, col, change;
10 |    char choice;
11 |
12 |    do {
13 |        printf("Enter row (0-3) and column (0-4) of the element to view: ");
14 |        scanf("%d %d", &row, &col);
15 |
16 |        if (row < 0 || row > 3 || col < 0 || col > 4) {
17 |            printf("Invalid position!\n"); continue; }
18 |        printf("Current value at marks[%d][%d] is %d\n", row, col, marks[row][col]);
19 |
20 |        printf("Would you like to revise it? (y/n): ");
21 |        scanf(" %c", &choice);
22 |
23 |        if (choice == 'y' || choice == 'Y') {
24 |            printf("Enter the amount to increase (+) or decrease (-): ");
25 |            scanf("%d", &change);
26 |            marks[row][col] += change;
27 |            printf("Updated value at marks[%d][%d] is %d\n", row, col, marks[row][col]);
28 |        }
29 |
30 |        printf("Do you want to revise another element? (y/n): ");
31 |        scanf(" %c", &choice);
32 |
33 |    } while (choice == 'y' || choice == 'Y');
34 |    return 0;
35 | }
```

## 9.5 2D array storage

The **base address** or **starting address** of a 2D array refers to the memory address of its first element. Starting from the base address, all elements of the array are stored sequentially in memory, row by row. For each row, the elements are stored from left to right.

To illustrate this, let us consider the 4-by-5 array `marks` from (9.1). Let `x` represent its base address, i.e., `x = &marks[0][0]`. Use the convention that each integer occupies 4 bytes. Then, the memory addresses of the elements in `marks` will be as follows:

<code>x</code>	<code>x+4</code>	<code>x+8</code>	<code>x+12</code>	<code>x+16</code>
<code>x+20</code>	<code>x+24</code>	<code>x+28</code>	<code>x+32</code>	<code>x+36</code>
<code>x+40</code>	<code>x+44</code>	<code>x+48</code>	<code>x+52</code>	<code>x+56</code>
<code>x+60</code>	<code>x+64</code>	<code>x+68</code>	<code>x+72</code>	<code>x+76</code>

(9.3)

Eventually, this enables the 2D array to be treated as a 1D array. (So cute! Isn't it?) As an example, let's rewrite the array `marks` given in (9.1):

	<code>j=0</code>	<code>1</code>	<code>2</code>	<code>3</code>	<code>4</code>
<code>i=0</code>	71	82	90	63	76
<code>1</code>	68	75	80	70	72
<code>2</code>	88	74	85	76	80
<code>3</code>	50	65	68	40	70

The above 2D array is simply **equivalent** to the following 1D array with 20 elements.

index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
	71	82	90	63	76	68	75	80	70	72	88	74	85	76	80	50	65	68	40	70

The reason is that the addresses of the elements in this 1D array adhere to the same rule. For the 2D array, we have

$$\text{address of } \text{marks}[i][j] = \&\text{marks}[i][j] = \&\text{marks}[0][0] + (i * 5 + j) * 4$$

which is also the address of the element with index `i*5+j` in the 1D array. For example, the element in the 2D array with row index `i=3` and column index `j=1` is 65. For this element, we have `i*5+j=16`, and that's the index of the same element in the 1D array.

Now, let us consider the general scenario in which `k` is the size (measured in terms of bytes) of each element of **any 2D array**. It is given by the operator `sizeof()` discussed in Chapter 8. For example, for an integer variable, it is given by `sizeof(int)`, which is basically 4 bytes in standard computers. So, if an integer array has `r` rows and `c` columns, then the total space allocated for it will be  $4 \times r \times c$  bytes. For example, the space allocated for the 4-by-5 array `marks` will be  $4 \times 4 \times 5 = 80$  bytes. For a character array, the space required will be  $r \times c$  bytes, as each character takes just one byte.

Now, let us see how to find the address of an arbitrary element of a 2D array from its base address. Suppose `A` is a 2D array with `x` as its base address, and `c` as the number of its columns. Then the address of `A[i][j]` can be calculated as `x + (i * c + j) * k`, or,

$$\&A[i][j] = \&A[0][0] + (i * c + j) * k. \quad (9.4)$$

As an example, the actual addresses of the elements in `marks` are displayed below. These addresses are obtained by running Code 9.26. These addresses usually vary with each execution, as reflected here.

Addresses are very likely to change with each execution because memory is actually allocated only at runtime. The allocation is done by the operating system as per the available free space in the memory during that particular execution.

1st execution of Code 9.26

0x7ffe086ff620	0x7ffe086ff624	0x7ffe086ff628	0x7ffe086ff62c	0x7ffe086ff630
0x7ffe086ff634	0x7ffe086ff638	0x7ffe086ff63c	0x7ffe086ff640	0x7ffe086ff644
0x7ffe086ff648	0x7ffe086ff64c	0x7ffe086ff650	0x7ffe086ff654	0x7ffe086ff658
0x7ffe086ff65c	0x7ffe086ff660	0x7ffe086ff664	0x7ffe086ff668	0x7ffe086ff66c

2nd execution of Code 9.26

0x7ffe9aa633b0	0x7ffe9aa633b4	0x7ffe9aa633b8	0x7ffe9aa633bc	0x7ffe9aa633c0
0x7ffe9aa633c4	0x7ffe9aa633c8	0x7ffe9aa633cc	0x7ffe9aa633d0	0x7ffe9aa633d4
0x7ffe9aa633d8	0x7ffe9aa633dc	0x7ffe9aa633e0	0x7ffe9aa633e4	0x7ffe9aa633e8
0x7ffe9aa633ec	0x7ffe9aa633f0	0x7ffe9aa633f4	0x7ffe9aa633f8	0x7ffe9aa633fc

3rd execution of Code 9.26

0x7ffdd206d890	0x7ffdd206d894	0x7ffdd206d898	0x7ffdd206d89c	0x7ffdd206d8a0
0x7ffdd206d8a4	0x7ffdd206d8a8	0x7ffdd206d8ac	0x7ffdd206d8b0	0x7ffdd206d8b4
0x7ffdd206d8b8	0x7ffdd206d8bc	0x7ffdd206d8c0	0x7ffdd206d8c4	0x7ffdd206d8c8
0x7ffdd206d8cc	0x7ffdd206d8d0	0x7ffdd206d8d4	0x7ffdd206d8d8	0x7ffdd206d8dc

Code 9.26: Printing the addresses of the elements of `marks`.

`2dArrayStudentMarksAddress.c`

```

1 | #include <stdio.h>
2 |
3 | #define ROWS 4
4 | #define COLS 5
5 |
6 | int main(){
7 |     int marks[ROWS][COLS], row, col, total;
8 |
9 |     for (row = 0; row < ROWS; row++, printf("\n"))
10 |         for (total = 0, col = 0; col < COLS; col++)
11 |             printf("%p ", &marks[row][col]);
12 |
13 |     return 0;
14 | }
```

**Q8** You have 1000 two-dimensional points with integer coordinates. Can you store them in a 1D array? If possible, write a declaration for that 1D array. Is this a better scheme compared to the 2D array (Question 2)? Justify.

**Q9** Consider `int a[4][5]`;  
 What are meant by `a`, `&a`, `a+1`, `&a+1`, `&(a + 1)`, `(&a) + 1`, `a[0][0]`, and `&a[0][0]`?  
 Write their values if the first element of `a` is 1 and its hexadecimal address is 84E9.

## 9.6 Passing 2D arrays to functions

Passing 2D arrays as function arguments follows a similar approach to passing 1D arrays. Suppose `f1` defines a 2D array `A` as a local variable. If `f1` calls another function, say `f2`, to perform operations on `A`, `f1` must pass the array to `f2`. Instead of passing all the elements of `A`, only the address of the first element (`&A[0][0]`) is passed from `f1` to `f2`.

As an additional information, `f1` must also pass the number of rows and columns of `A` to `f2`, allowing it to determine the size of the array. Given that the address of `A[0][0]` is `x`, `f2` can calculate the address of any element `A[i][j]` using Equation 9.4. It can then use this address to access or modify the value of `A[i][j]`.

Below is a small code example to illustrate the concept. We assume that `A` has 3 rows and 4 columns.

Code 9.27: Working with a **static 2D array** named `marks`.

[2dArrayFunCallsf1f2.c](#)

```

1 void f2(int A[3][4]){
2     ...
3 }
4
5 void f1(int A[3][4]){
6     ...
7     f2(A); // the address of the 1st element of A is passed to f2
8     ...
9 }
10
11 int main(){
12     int A[3][4];
13     ...
14     f1(A); // the address of the 1st element of A is passed to f1
15     ...
16     return 0;
17 }
```

To understand the above concept for a specific problem, let us refer back to Code 9.24. In that code, we have seen how a 4-by-5 2D array can be declared and subsequently used in `main()` for specific tasks like initialization and computing the total marks. We now modify that code to make it modular by writing user-defined functions and calling them from `main()` to perform the same tasks. The modified code is given in Code 9.28. Observe in this code that the argument passed to either of `fillArray` and `printTotalMarks` is the array name `marks`, which basically works as a pointer to the first element of `marks`. This is just similar to the convention followed for 1D arrays, as discussed in Chapter 8.

## 9.7 Dynamic memory allocation for 2D array

The array `marks` used in Code 9.28 was a **fixed-size array**. We now see what happens if it's a **dynamic array**, i.e., its size, specified by the number of rows and the number of columns, are given as input. It's given in Code 9.29. In this code, `malloc` is used to allocate the optimum space for the 2D array. The function `malloc` could be called in `main()` for the allocation. Instead of that, it uses a user-defined function named `allocateArray` to handle the dynamic allocation.

In Code 9.29, you have to observe carefully the portion on dynamic memory allocation (`allocateArray` function). It first allocates memory for the **array of row pointers**, and then it allocates memory for each row. After every call of `malloc`, it checks for memory allocation failures.

The function prototypes and their calls in Code 9.29 are discussed below.

- `int **allocateArray(int rows, int cols)`: This function dynamically allocates memory for a 2D array of integers using pointers. It takes the number of rows and columns as arguments and returns a pointer to a dynamically allocated array. The array is a pointer to pointers (i.e., `int**`), where each row is allocated individually. Inside `main()`, it is called as `marks = allocateArray(m, n)`.
- `void fillArray(int **marks, int rows, int cols)`: This function fills the 2D array with user input. It takes the pointer to the 2D array (i.e., `marks`), the number of rows, and the number of columns as arguments. It uses nested loops to prompt the user to enter values for each element in the array. In `main()`, it is called as `fillArray(marks, m, n)` after memory allocation.
- `void printTotalMarks(int **marks, int rows, int cols)`: This function calculates and prints the total marks for each student. It takes the pointer to the 2D array, the number of rows, and the number of columns as arguments. For each student (i.e., row), it calculates the sum of marks across all subjects (i.e., columns) and prints the total. In `main()`, it is called as `printTotalMarks(marks, m, n)`.

**Code 9.28:** Working with a static 2D array named `marks`. `2dArray-4by5-StudentMarksFunctions.c`

```

1  #include <stdio.h>
2
3  #define ROWS 4
4  #define COLS 5
5
6  void fillArray(int marks[ROWS][COLS]){
7      int row, col;
8      for (row = 0; row < ROWS; row++){
9          for (col = 0; col < COLS; col++){
10             printf("Enter marks for student %d, course %d: ", row + 1, col + 1);
11             scanf("%d", &marks[row][col]);
12         }
13     }
14
15 void printTotalMarks(int marks[ROWS][COLS]){
16     int row, col, total;
17     for (row = 0; row < ROWS; row++){
18         total = 0;
19         for (col = 0; col < COLS; col++){
20             total += marks[row][col];
21         }
22         printf("Total marks for student %d: %d\n", row + 1, total);
23     }
24
25 int main(){
26     int marks[ROWS][COLS];
27     fillArray(marks);
28     printTotalMarks(marks);
29     return 0;
30 }

```

- `void freeArray(int **array, int rows)`: This function frees the dynamically allocated memory for the 2D array. It takes the pointer to the 2D array and the number of rows as arguments. It first frees each individual row, then frees the array itself. In `main()`, it is called as `freeArray(marks, m)` after the array has been used.
- `int main()`: The `main()` function prompts the user for the number of rows (students) and columns (subjects), allocates the 2D array using `allocateArray`, calls `fillArray` to populate the array, and then calls `printTotalMarks` to display the total marks for each student. After that, it calls `freeArray` to release the allocated memory.

**Code 9.29:** Working with a dynamic 2D array `marks`. `2dArray-dynamic-StudentMarksFunctions.c`

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int **allocateArray(int rows, int cols){
5      int **array = (int **)malloc(rows * sizeof(int *));
6      if (array == NULL){
7          printf("Memory allocation failed for row pointers.\n");
8          exit(1);
9      }
10     for (int i = 0; i < rows; i++){

```



```
11     array[i] = (int *)malloc(cols * sizeof(int));
12     if (array[i] == NULL){
13         printf("Memory allocation failed for row %d.\n", i);
14         exit(1);
15     }
16 }
17 return array;
18 }
19
20 void fillArray(int **marks, int rows, int cols){
21     for (int row = 0; row < rows; row++){
22         for (int col = 0; col < cols; col++){
23             printf("Enter marks for student %d, course %d: ", row + 1, col + 1);
24             scanf("%d", &marks[row][col]);
25         }
26     }
27 }
28 void printTotalMarks(int **marks, int rows, int cols){
29     for (int row = 0; row < rows; row++){
30         int total = 0;
31         for (int col = 0; col < cols; col++){
32             total += marks[row][col];
33         }
34         printf("Total marks for student %d: %d\n", row + 1, total);
35     }
36 }
37 void freeArray(int **array, int rows){
38     for (int i = 0; i < rows; i++){
39         free(array[i]);
40     }
41     free(array);
42 }
43 int main(){
44     int m, n;
45     printf("Enter number of rows (students): ");
46     scanf("%d", &m);
47     printf("Enter number of columns (courses): ");
48     scanf("%d", &n);
49
50     int **marks = allocateArray(m, n);
51     fillArray(marks, m, n);
52     printTotalMarks(marks, m, n);
53     freeArray(marks, m);
54     return 0;
55 }
```

## 9.8 Declaration (dynamic): A summary

A 2D array can be declared in several ways, as shown in Code 9.30. The meanings are as follows:

1. `int A[MAXROW][MAXCOL];` ⇒ A is a **statically allocated 2D array** with fixed dimensions. This refers to memory allocated at compile time, contrasting with dynamically allocated arrays that are created at runtime.
2. `int (*B)[MAXCOL];` ⇒ B is a **pointer to an array** of MAXCOL integers. The parentheses are necessary to bind the pointer to the array of integers, not just to an individual integer.

- `int *C[MAXROW];`  $\Rightarrow$  `C` is an array of `MAXROW` pointers to integers. Each element of `C` can point to the start of a separate array of integers.
- `int **D;`  $\Rightarrow$  `D` is a pointer to a pointer to an integer. This is typically used for dynamic memory allocation for 2D arrays.
- The last three arrays support dynamic memory allocation; the most commonly used style is `int **D;`. When properly allocated memory, any of them can be used to represent a `MAXROW`-by-`MAXCOL` array.

Code 9.30: Different Ways of Declaring 2D Arrays.

2dArrayWaysDeclaration.c

```

1 | #include <stdio.h>
2 | #include <stdlib.h>
3 |
4 | #define MAXROW 4
5 | #define MAXCOL 5
6 |
7 | int main(){
8 |     int A[MAXROW][MAXCOL];
9 |     int (*B)[MAXCOL];
10 |    int *C[MAXROW];
11 |    int **D;
12 |    ...
13 |    return 0;
14 | }

```

Let's now see how memory is allocated dynamically for `MAXROW`-by-`MAXCOL` arrays using `B`, `C`, and `D`.

- `int (*B)[MAXCOL];`  $\Rightarrow$  `B` is a pointer to an array of `MAXCOL` integers.

So, it can be allocated `MAXROW` rows in the following way:

```
B = (int (*)[MAXCOL])malloc(MAXROW * sizeof(int[MAXCOL]));
```

- `int *C[MAXROW];`  $\Rightarrow$  `C` is an array of `MAXROW` `int` pointers. Therefore, `C` itself cannot be allocated memory. The individual rows of `C` should be allocated memory.

```

int i;
for (i=0; i<MAXROW; ++i)
    C[i] = (int *)malloc(MAXCOL * sizeof(int));

```

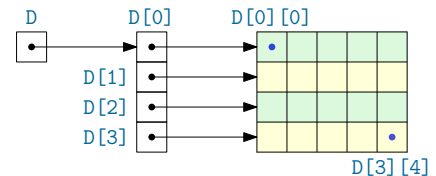
- `int **D;`  $\Rightarrow$  `D` is a pointer to an `int` pointer. So, `D` is dynamic in both directions.

First, it should be allocated memory to store `MAXROW` `int` pointers, each meant for a row of the 2D array. Each row pointer, in turn, should be allocated memory for `MAXCOL` `int` data.

```

int i;
D = (int **)malloc(MAXROW * sizeof(int *));
for (i=0; i<MAXROW; ++i)
    D[i] = (int *)malloc(MAXCOL * sizeof(int));

```



## 9.9 Arrays with higher dimension

3D and higher-dimensional arrays allow data to be stored and accessed in more complex structures. A 3D array can be visualized as an array of 2D arrays. For example, an array `int volume[3][4][2]` defines a 3D array with 3 blocks, each containing 4 rows and 2 columns. Accessing an element like `volume[1][2][0]` retrieves the value in the 2nd block, 3rd row, and 1st column. These types of arrays are useful in applications like modeling matrices for computer simulations in diverse applications of science and engineering.

## 9.10 Solved problems

Note the following points regarding the problems stated in this section and in §9.11.

1. By  $m$ -by- $n$  or  $m \times n$  array (or matrix), we mean a 2D array with  $m$  rows and  $n$  columns.
2. Unless mentioned, assume that  $m, n \leq 10$ .
3. Unless mentioned, assume that all elements are integers.

1. **[Matrix addition: Version 1]** Compute the addition of two matrices with compatible dimensions and elements as input, store it in a new matrix, and print its elements on the terminal.

If  $A$  and  $B$  are two compatible matrices (i.e., have  $m$  rows and  $n$  columns each), then in their sum matrix  $C$ , the element at row  $i$  and column  $j$  is denoted by  $C[i][j]$  and evaluated as  $A[i][j] + B[i][j]$ . Here is an example:

$$A + B = \begin{bmatrix} 1 & 3 & 2 & 1 \\ 3 & 2 & 1 & 3 \\ 2 & 1 & 3 & 2 \end{bmatrix} + \begin{bmatrix} 3 & 2 & 1 & 3 \\ 1 & 3 & 2 & 1 \\ 2 & 1 & 3 & 2 \end{bmatrix} = \begin{bmatrix} 4 & 5 & 3 & 4 \\ 4 & 5 & 3 & 4 \\ 4 & 2 & 6 & 4 \end{bmatrix}$$

**Code 9.31:** Matrix addition: Version 1  
(static memory allocation, without user-defined functions).

[matrixAdd.c](#)

```

1  #include<stdio.h>
2
3  int main(){
4      int m, n, a[10][10], b[10][10], c[10][10], i, j;
5
6      printf("\nEnter #rows & #columns of the matrices: ");
7      scanf("%d%d", &m, &n);
8
9      printf("Enter Matrix 1:\n");
10     for(i=0; i<m; i++)
11         for(j=0; j<n; j++)
12             scanf("%d", &a[i][j]);
13
14     printf("Enter Matrix 2:\n");
15     for(i=0; i<m; i++)
16         for(j=0; j<n; j++)
17             scanf("%d", &b[i][j]);
18
19     for(i=0; i<m; i++)
20         for(j=0; j<n; j++)
21             c[i][j] = a[i][j] + b[i][j]; // scalar addition
22
23     printf("Resultant matrix (after addition):\n");
24     for(i=0; i<m; i++, printf("\n"))
25         for(j=0; j<n; j++)
26             printf("%3d ", c[i][j]);
27     printf("\n");
28
29     return 0;
30 }
```

**Q10** How many scalar additions are used in Code 9.31? (A **scalar addition** means the addition between two numbers.)

2. **[Matrix addition: Version 2]** Rewrite Version 1 of matrix addition with user-defined functions for input of the elements of the matrices, for adding the matrices, and for printing the result.

**Code 9.32:** Matrix addition: Version 2

(static memory allocation, with user-defined functions).

[matrixAddFun.c](#)

```

1  #include<stdio.h>
2
3  // Function to input matrix elements
4  void inputMatrix(int matrix[10][10], int rows, int cols, int num) {
5      printf("Enter Matrix %d:\n", num);
6      for (int i = 0; i < rows; i++)
7          for (int j = 0; j < cols; j++)
8              scanf("%d", &matrix[i][j]);
9  }
10
11 // Function to add two matrices
12 void addMatrices(int a[10][10], int b[10][10], int c[10][10], int rows, int cols) {
13     for (int i = 0; i < rows; i++)
14         for (int j = 0; j < cols; j++)
15             c[i][j] = a[i][j] + b[i][j];
16 }
17
18 // Function to print matrix
19 void printMatrix(int matrix[10][10], int rows, int cols) {
20     printf("Resultant matrix (after addition):\n");
21     for (int i = 0; i < rows; i++, printf("\n"))
22         for (int j = 0; j < cols; j++)
23             printf("%3d ", matrix[i][j]);
24 }
25
26 int main() {
27     int m, n, a[10][10], b[10][10], c[10][10];
28
29     printf("Enter #rows & #columns of the matrices: ");
30     scanf("%d%d", &m, &n);
31     inputMatrix(a, m, n, 1);
32     inputMatrix(b, m, n, 2);
33     addMatrices(a, b, c, m, n);
34     printMatrix(c, m, n);
35
36     return 0;
37 }

```

3. **[Matrix addition: Version 3 (with dynamic memory allocation)]** Rewrite Version 2 of matrix addition with user-defined functions with the provision for taking input for number of rows and columns in `main()` and for memory allocation of the matrices.

In Code 9.33, you should understand the use of `freeMatrix` function that frees allocated memory for a 2D array:

- (i) In `freeMatrix` function, memory is freed in two stages. First, each row's memory is freed using `free(matrix[i]);`. Second, the memory allocated for the array of row pointers is freed using `free(matrix);`.
- (ii) Using `free(matrix);` directly only frees the memory allocated for the array of pointers, leaving the memory for the rows still allocated. This causes **memory leaks**, as the memory is no longer needed but remains occupied. Such leaks can degrade performance, particularly when the program needs to allocate other large arrays later.

**Code 9.33:** Matrix addition: Version 3 (with dynamic memory allocation).

`matrixDynAddFun.c`

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int **allocateMatrix(int m, int n){ // allocate memory for m-by-n matrix
5     int **matrix = (int **)malloc(m * sizeof(int *));
6     if (matrix == NULL) {
7         printf("Memory allocation failed for row pointers.\n"); exit(1); }
8     for (int i = 0; i < m; i++){
9         matrix[i] = (int *)malloc(n * sizeof(int));
10        if (matrix[i] == NULL) {
11            printf("Memory allocation failed for row %d.\n", i); exit(1); }
12        }
13    return matrix;
14 }
15
16 void inputMatrix(int **matrix, int m, int n, int num){
17     printf("Enter Matrix %d:\n", num);
18     for (int i = 0; i < m; i++)
19         for (int j = 0; j < n; j++)
20             scanf("%d", &matrix[i][j]);
21 }
22
23 void addMatrices(int **a, int **b, int **c, int m, int n){
24     for (int i = 0; i < m; i++)
25         for (int j = 0; j < n; j++)
26             c[i][j] = a[i][j] + b[i][j];
27 }
28
29 void printMatrix(int **matrix, int m, int n){
30     printf("Resultant matrix (after addition):\n");
31     for (int i = 0; i < m; i++, printf("\n"))
32         for (int j = 0; j < n; j++)
33             printf("%3d ", matrix[i][j]);
34 }
35
36 void freeMatrix(int **matrix, int m){ // free allocated memory for matrix with m rows
37     for (int i = 0; i < m; i++)
38         free(matrix[i]); // free all cells of row i
39     free(matrix); // free m row-pointers
40 }
41
42 int main(){
43     int m, n;
44     printf("Enter #rows and #columns of the matrices: ");
45     scanf("%d%d", &m, &n);
46     int **a = allocateMatrix(m, n);
47     int **b = allocateMatrix(m, n);
48     int **c = allocateMatrix(m, n);
49
50     inputMatrix(a, m, n, 1); inputMatrix(b, m, n, 2); // take input
51     addMatrices(a, b, c, m, n);
52     printMatrix(c, m, n);
53     freeMatrix(a, m); freeMatrix(b, m); freeMatrix(c, m); // free all three matrices
54     return 0;
55 }
```

4. **[Matrix multiplication: Version 1]** Compute the multiplication of two matrices with compatible dimensions and elements as input, store it in a new matrix, and print its elements on the terminal. Consider static memory allocation and do not use user-defined functions.

If  $A$  and  $B$  are two compatible matrices (i.e., with sizes  $m \times n$  and  $n \times p$ , respectively), then in their product matrix  $C$ , the element at row  $i$  and column  $j$  is denoted by  $C[i][j]$  and evaluated as

$$C[i][j] = \sum_{k=1}^n A[i][k] \cdot B[k][j].$$

Here is an example with  $m = 4, n = 5, p = 4$ :

$$A \times B = \begin{bmatrix} 1 & 2 & 3 & 1 & 2 \\ 2 & 1 & 3 & 2 & 1 \\ 3 & 2 & 1 & 3 & 2 \\ 1 & 3 & 2 & 1 & 3 \end{bmatrix} \times \begin{bmatrix} 2 & 1 & 3 & 2 \\ 1 & 3 & 1 & 3 \\ 2 & 2 & 3 & 1 \\ 1 & 1 & 2 & 2 \\ 3 & 2 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 20 & 16 & 20 & 14 \\ 14 & 16 & 22 & 14 \\ 18 & 17 & 22 & 16 \\ 20 & 17 & 17 & 14 \end{bmatrix}$$

**Code 9.34:** Matrix multiplication: Version 1 (without dynamic memory allocation). [matrixMult.c](#)

```

1  #include<stdio.h>
2
3  int main(){
4      int m, n, p;
5      int a[10][10], b[10][10], c[10][10];
6      int i, j, k;
7
8      printf("\nEnter #rows & #columns of 1st matrix: ");
9      scanf("%d%d", &m, &n);
10     printf("Enter #columns of 2nd matrix: ");
11     scanf("%d", &p);
12
13     printf("Enter Matrix 1:\n");
14     for(i=0; i<m; i++)
15         for(j=0; j<n; j++)
16             scanf("%d", &a[i][j]);
17
18     printf("Enter Matrix 2:\n");
19     for(i=0; i<n; i++)
20         for(j=0; j<p; j++)
21             scanf("%d", &b[i][j]);
22
23     for(i=0; i<m; i++)
24         for(j=0; j<p; j++)
25             for(c[i][j]=0; k<n; k++)
26                 c[i][j] += a[i][k] * b[k][j]; // scalar multiplication
27
28     printf("Output matrix:\n");
29     for(i=0; i<m; i++, printf("\n")) // see a correct nuance here: printf("\n") [laugh]
30         for(j=0; j<p; j++)
31             printf("%3d ", c[i][j]);
32     printf("\n");
33
34     return 0;
35 }
```

**Q11** How many scalar multiplications are used in Code 9.34? (A **scalar multiplication** means the multiplication between two numbers.)

5. **[Saddle point]** An element  $x$  is said to be a **saddle point** in a 2D array if it is smallest in its row and largest in its column. Given a 2D array with 4 rows and 5 columns, find whether it has any saddle point. Write a C program with dynamic memory allocation. It need not have any user-defined function. Assume that all elements are distinct.

In the example below,  $A$  has no saddle point, but  $B$  has exactly one saddle point:  $B[3][0] = 16$ .

$$A = \begin{bmatrix} 3 & 8 & 7 & 6 & 4 \\ 14 & 5 & 9 & 10 & 11 \\ 13 & 17 & 2 & 15 & 16 \\ 12 & 18 & 19 & 1 & 20 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 28 & 19 & 20 \end{bmatrix}$$

**Code 9.35:** Finding saddle points in a matrix.

saddlePoint.c

```

1  #include <stdio.h>
2
3  int main(){
4      int m = 4, n = 5; // Fixed for a 4-by-5 matrix
5      int a[m][n];
6      int saddleFound = 0; // Flag to check if a saddle point is found
7      int i, j, k, minRowIndex, isSaddlePoint;
8
9      printf("Enter the elements of the 4-by-5 matrix:\n");
10     for (i = 0; i < m; i++)
11         for (j = 0; j < n; j++)
12             scanf("%d", &a[i][j]);
13
14     // Find the saddle point, if any, for each row.
15     // Because there will be at most one saddle point in each row.
16     for (i = 0; i < m; i++){
17         // Find the smallest-element index (minRowIndex) in row i
18         for (j = 0, minRowIndex = 0; j < n; j++)
19             if (a[i][j] < a[i][minRowIndex]) // element-to-element comparison
20                 minRowIndex = j;
21
22         // Check if a[i][minRowIndex] is the largest in its column
23         for (isSaddlePoint = 1, k = 0; k < m; k++)
24             if (a[k][minRowIndex] > a[i][minRowIndex]){ // element-to-element comparison
25                 isSaddlePoint = 0;
26                 break;
27             }
28
29         if (isSaddlePoint)
30             saddleFound = 1,
31             printf("\nSaddle point: a[%d][%d]: %d\n", i, minRowIndex, a[i][minRowIndex]);
32     }
33
34     if (!saddleFound)
35         printf("\nNo saddle point found.\n");
36
37     return 0;
38 }
```

**Q12** At most how many comparisons are used in Code 9.35? (Comparison means element-to-element comparison.) At most how many comparisons will be used if Code 9.35 is generalized for an  $m \times n$  matrix?

**Q13** Can you revise Code 9.35 to find saddle points in a different way? How?

6. **[4-adjacent local max]** An element  $x$  in a 2D array is said to be a **local max** if  $x$  is larger than all its four adjacent elements—left, right, above, and below. If  $x$  lies on the array boundary, then it may be disregarded due to insufficient adjacency. Given such an array with  $m \geq 3$  rows and  $n \geq 3$  columns, find all its local maxima.

**Code 9.36:** 4-adjacent local max.

adj4Max2D.c

```

1 // Local max in 2D array
2
3 #include<stdio.h>
4 #define MAX 10
5
6 void readArray(int a[MAX][MAX], int m, int n){
7     int i, j;
8     printf("Enter elements:\n");
9     for(i=0; i<m; i++)
10        for(j=0; j<n; j++)
11            scanf("%d", &a[i][j]);
12 }
13
14 void adj4Max2D(int a[MAX][MAX], int m, int n){
15     int i, j, found = 0;
16     printf("Local max: ");
17
18     for(i=1; i<m-1; i++)
19         for(j=1; j<n-1; j++)
20             if((a[i][j] > a[i][j-1]) && (a[i][j] > a[i][j+1])) // left and right
21                 if((a[i][j] > a[i-1][j]) && (a[i][j] > a[i+1][j])) // above and below
22                     printf("%d ", a[i][j]), found = 1;
23
24     if(!found)
25         printf("None");
26     printf("\n");
27 }
28
29 int main(){
30     int m, n, a[MAX][MAX];
31     printf("\nEnter #rows & #columns: ");
32     scanf("%d%d", &m, &n);
33
34     readArray(a, m, n);
35     adj4Max2D(a, m, n);
36
37     return 0;
38 }

```

**Q14** How many comparisons are used in Code 9.36? (Comparison means element-to-element comparison.)

**Q15** At most how many local maxima can be present in the 2D array? Justify.

**Q16** Suppose we redefine local max as follows:

An element  $a[i][j]$  in a 2D array  $a$  is said to be a **local max** if it is larger than  $a[i+p][j+q]$ , where  $p$  and  $q$  are in  $\{+1, -1\}$ . If  $a[i][j]$  lies on the array boundary, then it may be disregarded.

Given such an array with  $m \geq 3$  rows and  $n \geq 3$  columns, find all its local maxima, by modifying Code 9.36.

**Q17** How many comparisons will be needed in the modified code of Question 16?



7. **[Image fading]** An **image** of width  $c$  and height  $r$  is a 2-dimensional array with  $r$  rows and  $c$  columns in which each element represents the color of a pixel. For a **gray-scale image**, the color is in gray shade, expressed as an integer ranging from 0 (absolute black) to 255 (absolute white). It is stored as a **PGM (portable gray map)** file, with the following content:

- (i) Line 1: P2
- (ii) Line 2: values of  $c$  and  $r$  (in this order)
- (iii) Line 3: 255
- (iv) Line 4 to Line  $(3 + r \times c)$ : values of the pixel colors in row-major order

Optionally, it may contain one or more comment lines (e.g. `# created by ...`) just after Line 1. We assume that in our PGM files, there is no comment line.

Given a PGM image as input, our task is to fade the image and to save it as PGM and PNG files (Figure 9.1). Below are the commands to run your code on two input files: `m1a.pgm` and `m1.pgm`.

```
gcc a07-0.c
./a.out < m1a.pgm > m1a_0.pgm
convert m1a_0.pgm m1a_0.png

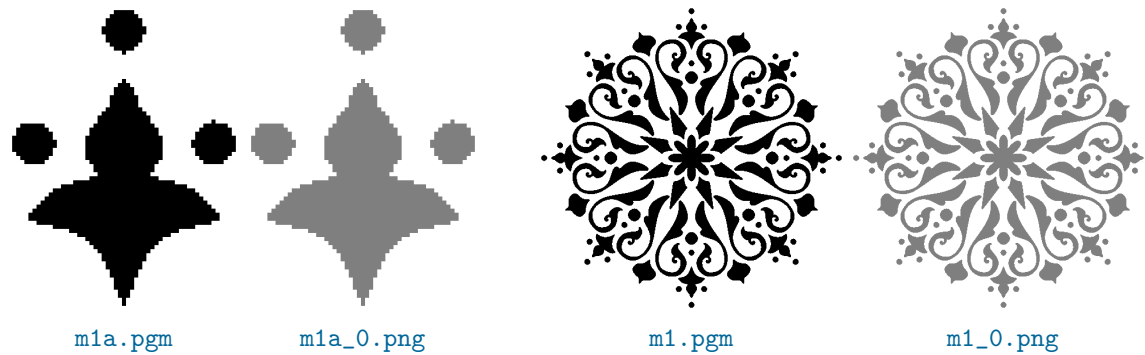
./a.out < m1.pgm > m1_0.pgm
convert m1_0.pgm m1_0.png
```

The sign `<` instructs to take input from the file `m1a.pgm`, and the sign `>` instructs to write the output to the file `m1a_0.pgm`. To convert from PGM to PNG, the `convert` command in **Linux** is used.

**Code 9.37:** Image fading.

`imageFading.c`

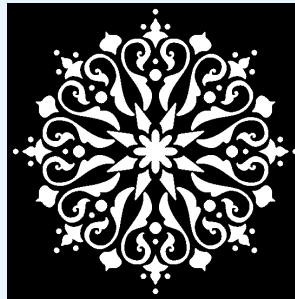
```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(){
5      char imageType[3]; // imageType is in the 1st line
6      int r, c, maxColor; // r = #rows, c = #columns, maxColor = 255
7      unsigned char **a; // image array
8
9      a = (unsigned char **)malloc(r * sizeof(unsigned char *));
10     scanf("%s%d%d", imageType, &c, &r, &maxColor);
11     if (a == NULL){ printf("Memory allocation failed for row pointers.\n"); exit(1); }
12     for (int i = 0; i < r; i++){
13         a[i] = (unsigned char *)malloc(c * sizeof(unsigned char));
14         if (a[i] == NULL){ printf("Memory allocation failed for row %d.\n", i); exit(1); }
15     }
16     printf("%s\n%d %d\n%d\n", imageType, c, r, maxColor);
17     int i, j, x;
18
19     for(i=0; i<r; i++)
20         for(j=0; j<c; j++){
21             scanf("%d", &x);
22             a[i][j] = (x==0) ? 127 : 255;
23         }
24
25     for(i=0; i<r; i++)
26         for(j=0; j<c; j++)
27             printf("%d\n", a[i][j]);
28
29     return 1;
30 }
```



**Figure 9.1:** Image fading. Left image has 77 rows and 59 columns. Right image has 549 rows and 549 columns. The color black (value 0) in the input image is changed to the value 127 in the output image.

**Q18** Modify exactly one statement of Code 9.37 so that the modified code inverts an image and saves it as a PNG file.

For example, the image `m1.pgm` in Figure 9.1, after being inverted, will look as follows:



## 9.11 Exercise problems

- [Matrix multiplication: Version 2]** Rewrite Version 1 of matrix multiplication with user-defined functions for input of the elements of the matrices, for adding the matrices, and for printing the result.
- [Matrix multiplication: Version 3]** Rewrite Version 2 of matrix multiplication with user-defined functions with the provision for taking input for number of rows and columns in `main()` and for memory allocation of the matrices.
- [Max-sum  $3 \times 3$  submatrix]** Given a matrix with  $m \geq 3$  rows and  $n \geq 3$  columns, find a  $3 \times 3$  submatrix such that the sum of its elements is maximum.
- [Reverse all primes in 2D array]** Given a 2D array of integers with 3 rows and 4 columns, where each element is a prime number greater than 10, the task is to perform two operations on the array. First, create a function `f1` that reverses the digits of each number in the array and updates the array with these reversed numbers. Second, create a function `f2` that takes the modified array from `f1` and checks whether each number is a prime. If a number is prime, it should remain unchanged; otherwise, it should be replaced with 0. The functions should be called in sequence from the `main()` function, and the array should be printed before and after each function call to show the transformations.
- [Doubling image size]** Modify Code 9.37 so that it doubles the size of an image and saves the result as a PNG file. For instance, the image `m1.pgm` in Figure 9.1, which has 549 rows and 549 columns, should have 1098 rows and 1098 columns after resizing.

6. **[Counting submatrices with a given sum]** Given a matrix of size  $m \times n$  filled with integers and a target sum  $t$ , find all its submatrices that sum up to  $t$ . Assume that  $2 \leq m, n \leq 50$ . A **submatrix** is defined as a contiguous rectangular block within the matrix. In the example below, there are 4 submatrices that sum up to  $t = 7$ . The last submatrix contains just one element (7).

Input matrix:

```
1 -2 1 -3 2
-2 2 4 5 -6
0 0 -3 -2 7
```

Output submatrices:

```
-2 1 -3          1 -3          2 4 5 -6          7
 2 4 5          4 5          0 -3 -2 7
```

**Q19** How many submatrices exist in a matrix of size  $m \times n$ ?

7. **[Maximum-Sum Path in a Grid]** You are given a 2D grid of size  $m \times n$  where each cell contains a positive integer. Assume that  $2 \leq m, n \leq 50$ . You start from the bottom-left cell and need to reach the top-right cell. You can only move rightward or upward at each step. Your task is to find a/the path that maximizes the sum of the numbers along the path.

For example, for the following grid, the maximum sum is 26.

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

**Application:** This problem can be applied in resource optimization, such as finding the most profitable path in a grid-based system, and in game development, where it helps to determine the highest-scoring path through a grid of values.

8. **[Minimum-Sum Path without Obstacles]** Given a 2D grid where some cells may contain obstacles, represented by the value 0, and other cells contain positive integers less than 10. Assume that the width and height of the grid are at most 10. Find the minimum cost to reach the bottom-right cell from the top-left cell while avoiding obstacles. You are allowed to move right, down, or diagonally (right-down). The goal is to compute the least-cost path from the start to the destination, considering only valid moves.

For example, for the following grid, the least-cost path has a cost of 10.

1	1	2	1	3
0	3	1	0	1
2	1	5	9	4

**Application:** This problem is relevant in robotics for path planning, where a robot needs to navigate through a grid-like environment while avoiding obstacles and minimizing travel cost.

# 10 | Searching in 1D array

## 10.1 Linear search

**Linear search**, also known as **sequential search**, is commonly used to search for a key in a 1D array when the array is unsorted, i.e., the elements of the array are not arranged in any particular order. This method works by sequentially checking each element until a match is found or the end of the array is reached. Linear search is typically applied to small datasets, unsorted collections, or situations where simplicity and ease of implementation are prioritized over performance efficiency. In case the array is sorted (i.e., its elements are arranged in increasing or decreasing order), then binary search is used, which we shall study in §10.2.

Code 10.38 and Code 10.39 show how linear search can be implemented and used. In Code 10.38, the function `linearSearch` takes as argument the pointer to the array `a`, the number of elements the array has, denoted by `n`, and the key to be searched in the array. The function returns the index of the array where the match is found. If no match is found, it returns `-1`.

---

**Code 10.38:** Linear search of a number `key` in a 1D array `a` having `n` numbers.

`linearSearch.c`

```
1 | int linearSearch(int a[], int n, int key){
2 |     int i = 0;
3 |     while ((i < n) && (key != a[i]))
4 |         i++;
5 |     if (i < n)
6 |         return i    // SUCCESSFUL search (match found at i)
7 |     return -1;    // UNSUCCESSFUL search (no match found)
8 | }
```

---

### Time complexity of linear search

**Time complexity** of an algorithm (or function) refers to the number of major operations performed by the algorithm. In general, to evaluate the efficiency of an algorithm across different test cases, we analyze three types of time complexity: **best case**, **worst case**, and **average case**.

In the case of linear search, the major operation is the **comparison** (between the search key and the elements of the array), i.e., `key != a[i]` in Code 10.38. Assuming all elements of `a` are distinct, the best-, worst-, and average-case time complexities of linear search are given below.

1. **Best case:** This occurs when the search ends at the first element. This happens when `key = a[0]`, so the function returns `0`. Only one comparison is needed in this case, resulting in a constant time complexity, denoted as  $\mathcal{O}(1)$ .

Any constant time is denoted by  $\mathcal{O}(1)$  (pronounced “big O of one”), no matter how large or small the constant is. We call it a ‘constant’ because its value is independent of `n`.

**2. Worst case:** This occurs when the search takes the maximum number of comparisons. There are two worst-case scenarios:

- (i) **Unsuccessful search:** The search key is not found in the entire array, i.e.,  $\text{key} \neq a[i]$  for all  $i$  from 0 to  $n-1$ .
- (ii) **Successful search:** The match occurs at the very last position of the array, i.e.,  $\text{key} = a[n-1]$ .

In both cases,  $n$  comparisons are required, which results in a worst-case time complexity of  $\mathcal{O}(n)$  (pronounced “big O of  $n$ ”).

For any linear function (e.g.,  $n/2+100$  or  $8n+5$ ) or linear-dominant function (e.g.,  $n/5 + 2 \log n + 100$ ), we disregard constants and non-dominant terms and say the time complexity is  $\mathcal{O}(n)$ . The notation  $\mathcal{O}(\cdot)$  captures the overall growth rate of the function.

**3. Average case:** In this scenario, we calculate the number of comparisons across all possible positions where  $\text{key}$  could be found. For the case  $\text{key} = a[i]$ , the number of comparisons is  $i+1$  since  $\text{key}$  is compared with  $a[0], a[1], \dots, a[i]$  until the match is found. The total number of comparisons for all  $n$  cases is

$$\sum_{i=0}^{n-1} (i+1) = \sum_{i=1}^n i = \frac{n(n+1)}{2}.$$

Hence, the average number of comparisons is

$$\frac{1}{n} \left( \frac{n(n+1)}{2} \right) = \frac{n+1}{2},$$

which implies that the average-case time complexity is  $\mathcal{O}(n)$ .

**Code 10.39:** The `main()` from where the linear search is called.

`linearSearchMain.c`

```

1  #include <stdio.h>
2
3  int linearSearch(int a[], int n, int key); // Function prototype
4
5  int main() {
6      int a[1000]; // Declare the array
7      int n, key, result;
8
9      printf("Enter the size of the array (at most 1000): ");
10     scanf("%d", &n);
11     printf("Enter %d elements of the array:\n", n);
12     for (int i = 0; i < n; i++)
13         scanf("%d", &a[i]);
14
15     printf("Enter the key to search: ");
16     scanf("%d", &key);
17
18     result = linearSearch(a, n, key);
19
20     if (result != -1)
21         printf("Key found at index %d\n", result);
22     else
23         printf("Key not found in the array\n");
24
25     return 0;
26 }
```

## 10.2 Binary search

Binary search is applicable only if the array is sorted, i.e., its elements are arranged in increasing or decreasing order. If the elements are not distinct but arranged in non-increasing or non-decreasing order (e.g., the array {3, 5, 5, 6, 7, 7, 7, 9} is in non-decreasing order), then also binary search can be done.

The basis idea is as follows:

1. Look at the middle of the array.
2. If the key is not at the middle, ignore half of the array, and repeat the process with the other half.

Thus, in every step, the search space is halved (that's why it is said to be 'binary'), and hence the searching converges very fast.

Binary search can be implemented in several ways, which are given in Codes 10.40, 10.41, and 10.42. For Code 10.40, a demonstration is presented in Figure 10.1. If the search key is found, then its index is returned; otherwise -1 is returned to the caller function. The convention of returning the index, and not the element, is same as in linear search.

**Code 10.40:** Version 1 of binary search.

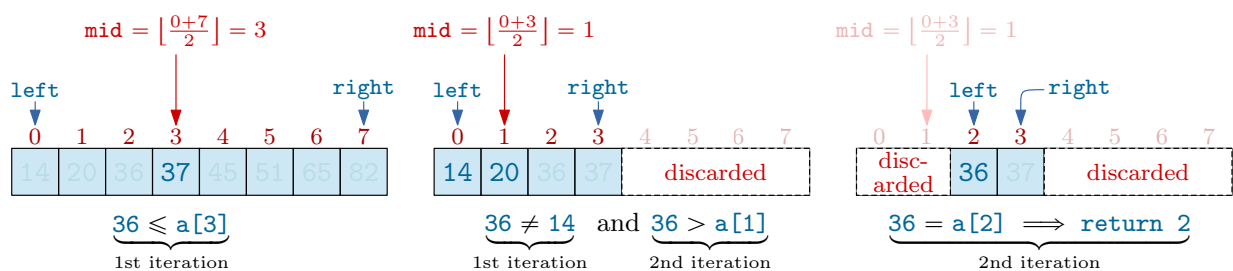
The `key` is searched for in a sorted array `a` having `n` elements.

[binarySearch.c](#)

```

1 | int binarySearch(int a[], int n, int key) {
2 |     int left, right, mid;
3 |     left = 0;
4 |     right = n - 1;
5 |
6 |     while (left != right) {
7 |         mid = (left + right) / 2;
8 |         if (key <= a[mid])
9 |             right = mid;
10 |        else
11 |            left = mid + 1;
12 |    }
13 |
14 |    if (key == a[left])
15 |        return left;
16 |    else
17 |        return -1;
18 | }
```

**Q20** Observe that exactly four elements (37, 14, 20, 36) are accessed by Code 10.40 while searching for the key 36, as shown in Figure 10.1. Calculate the number of accesses for keys 10, 14, 25. Do it for both Code 10.40 and Code 10.41.



**Figure 10.1:** Execution of binary search (Code 10.40) with `key = 36` returns the index 2. Only the accessed elements are shown; the rest are dimmed or hidden, as they are irrelevant to the search function.

**Q21** Consider the array shown in Figure 10.1. For each element in the array, treated as a search key, determine which keys will require the maximum number of comparisons when using Code 10.40.

**Code 10.41:** Version 2 of binary search.

binarySearchVer2.c

```
1 int bin_search_1(int a[], int n, int key) {
2     int left, right, mid;
3     left = 0;
4     right = n - 1;
5
6     while (left <= right) {
7         mid = (left + right) / 2;
8
9         if (key == a[mid])
10            return mid;
11
12        if (key < a[mid])
13            right = mid - 1;
14        else
15            left = mid + 1;
16    }
17
18    return -1;
19 }
```

**Code 10.42:** Recursive version of binary search.

binarySearchRecursive.c

```
1 #include <stdio.h>
2
3 int binarySearchRec(int a[], int left, int right, int key) {
4     int mid;
5     if (left <= right) {
6         mid = (left + right) / 2;
7         if (key == a[mid]) // Element is present at the middle
8             return mid;
9         if (key < a[mid]) // Look into the left subarray
10            return binarySearchRec(a, left, mid - 1, key);
11        else // Look into the right subarray
12            return binarySearchRec(a, mid + 1, right, key);
13    }
14    return -1; // Element is not present in the array
15 }
16
17 int main() {
18     int a[] = {14, 20, 36, 37, 45, 51, 65, 82}; // Sorted array
19     int n = sizeof(a) / sizeof(a[0]); // Size of the array
20     int key = 36;
21     int result = binarySearchRec(a, 0, n - 1, key);
22
23     if (result != -1)
24         printf("Element %d found at index %d.\n", key, result);
25     else
26         printf("Element %d not found in the array.\n", key);
27     return 0;
28 }
```

## Recursive version of binary search

The idea behind binary search leads to a recursive formulation. The full code including the `main()` is given in Code 10.42. The function recursively calls itself by adjusting the left or right pointers, as applicable. It stops when the element is found, or the left and right pointers cross each other.

## Time complexity of binary search

In each iteration (or recursive call), binary search eliminates half of the remaining elements, and this process continues until either the key is found or no more elements remain. The maximum number of iterations required is the number of times  $n$  can be divided by 2 until it is less than 1. This value is given by  $\lceil \log_2 n \rceil + 1$ . Each iteration involves a constant number of operations, including key-to-element comparisons, which are all performed in constant time. Therefore, the worst-case time complexity of binary search is  $\mathcal{O}(\log_2 n)$ , or logarithmic in  $n$ . (**Note:** For any positive real number  $x$ ,  $\lfloor x \rfloor$  denotes the largest integer less than or equal to  $x$ .)

**Q22** What is the base case for the recursive binary search function to terminate?

**Q23** Describe the difference in memory usage between iterative and recursive binary search.

**Q24** Given the following array and key, use the recursive binary search to find the key's index:  
`a[] = {3, 7, 12, 18, 22, 27}`, `key = 21`.



## 10.3 Solved problems

1. **[Deletion from an unsorted array]** Given an unsorted array `a` with `n` distinct elements, delete an element `x` if it exists in `a`. Assume that `n` is 100.

For example, if the initial array is {14, 10, 36, 17, 45, 51, 40, 82}, and the element 10 is deleted, then the new array will be {14, 36, 17, 45, 51, 40, 82}.

**Code 10.43:** Deletion from an unsorted array.

`deleteUnsortedArray.c`

```

1  #include <stdio.h>
2
3  #define MAX_SIZE 100
4
5  void deleteElement(int arr[], int *n, int x) {
6      int i, j;
7
8      // Find the element to be deleted
9      for (i = 0; i < *n; i++) {
10         if (arr[i] == x) {
11             // Shift elements to the left to remove the element
12             for (j = i; j < *n - 1; j++)
13                 arr[j] = arr[j + 1];
14             (*n)--; // Decrease the size of the array
15             return;
16         }
17     }
18     printf("Element %d not found in the array.\n", x);
19 }
20
21 int main() {
22     int n, arr[MAX_SIZE], x, i;
23
24     // Input the number of elements
25     printf("Enter the number of elements (max 100): ");
26     scanf("%d", &n);
27
28     // Input the elements of the array
29     printf("Enter %d elements:\n", n);
30     for (i = 0; i < n; i++)
31         scanf("%d", &arr[i]);
32
33     // Input the element to be deleted
34     printf("Enter the element to delete: ");
35     scanf("%d", &x);
36
37     // Delete the element
38     deleteElement(arr, &n, x);
39
40     // Print the updated array
41     printf("Array after deletion:\n");
42     for (i = 0; i < n; i++)
43         printf("%d ", arr[i]);
44     printf("\n");
45
46     return 0;
47 }

```

2. **[Insertion in a sorted array]** Given a sorted array `a` with `n` (potentially non-distinct) elements, insert a new element `x` such that `a` remains sorted after the insertion. Assume `n` is less than 100, and `a` can accommodate up to 100 elements. Also assume that `x` is not present in the array before insertion. Use binary search to find the position where `x` has to be inserted.

For example, if the initial array is {14, 20, 36, 37, 45, 51, 65, 82}, and the element 27 is inserted, then the new array will be {14, 20, 27, 36, 37, 45, 51, 65, 82}.

Code 10.44: Insertion in a sorted array.

[insertSortedArray.c](#)

```

1  #include <stdio.h>
2
3  void insertSorted(int a[], int *n, int x) {
4      int low = 0, high = *n - 1, mid, pos;
5
6      // Perform binary search to find the correct position
7      while (low <= high) {
8          mid = (low + high) / 2;
9          if (a[mid] < x)
10             low = mid + 1;
11         else
12             high = mid - 1;
13     }
14
15     pos = low; // Position to insert the new element
16
17     for (int i = *n; i > pos; i--) // Shift elements to the right
18         a[i] = a[i - 1];
19
20     a[pos] = x; // Insert the new element
21
22     (*n)++; // Increase the size of the array
23 }
24
25 int main() {
26     int a[100], n, x;
27     printf("Enter number of elements (at most 99): ");
28     scanf("%d", &n);
29
30     printf("Enter sorted array elements: ");
31     for (int i = 0; i < n; i++)
32         scanf("%d", &a[i]);
33
34     printf("Enter the element to insert: ");
35     scanf("%d", &x);
36
37     insertSorted(a, &n, x);
38
39     printf("Updated array: ");
40     for (int i = 0; i < n; i++)
41         printf("%d ", a[i]);
42
43     return 0;
44 }
```

3. **[Deletion from a sorted array]** Given a sorted array `a` with `n` (potentially non-distinct) elements, delete an element `x` if it exists in `a` such that `a` remains sorted after the deletion. Use linear search to find the position of `x`. Assume that `n` is 100.

For example, if the initial array is {14, 20, 36, 37, 45, 51, 65, 82}, and the element 20 is deleted, then the new array will be {14, 36, 37, 45, 51, 65, 82}.

**Code 10.45:** Deletion from a sorted array.

deleteSortedArray.c

```
1  #include <stdio.h>
2
3  // Function to delete an element from a sorted array
4  int deleteElement(int a[], int n, int x) {
5      int i, pos = -1;
6
7      for (i = 0; i < n; i++) {
8          if (a[i] == x) {
9              pos = i;
10             break;
11         }
12     }
13
14     if (pos == -1)
15         return n; // If element is not found, return the original size
16
17     // Shift the elements to the left to fill the gap
18     for (i = pos; i < n - 1; i++)
19         a[i] = a[i + 1];
20     return n - 1; // Return the new size of the array
21 }
22
23 int main() {
24     int a[100], n, x;
25     printf("Enter number of elements (at most 99): ");
26     scanf("%d", &n);
27
28     printf("Enter sorted array elements: ");
29     for (int i = 0; i < n; i++)
30         scanf("%d", &a[i]);
31
32     printf("Enter the element to be deleted: ");
33     scanf("%d", &x);
34     n = deleteElement(a, n, x);
35
36     printf("Array after deleting %d: ", x);
37     for (int i = 0; i < n; i++)
38         printf("%d ", a[i]);
39     printf("\n");
40
41     return 0;
42 }
```

**Q25** What are the best, worst, and average-case time complexities for inserting an element into a sorted array? Assume binary search is used to find the insertion position.

**Q26** What are the best, worst, and average-case time complexities for deleting an element from a sorted array? Assume binary search is used to find the element to delete.

**Q27** Will the time complexities for insertion and deletion be worse if linear search is used instead of binary search? Explain why or why not.

## 10.4 Exercise problems

Write C programs with appropriate user-defined functions for the following problems. Elements in the input array need not be distinct, unless mentioned.

1. **[Find a pair with a given sum in an unsorted array]** Given an unsorted array `a` of `n` elements and a target sum `target`, find a pair of elements in the array that adds up to `target`. If no such pair exists, return `-1`. The function should run in quadratic time (i.e.,  $\mathcal{O}(n^2)$  time) in the worst case.

For example, if `a = {9, 15, 5, 16, 2}`, `target = 20`, the output will be `(5, 15)`.

**Q28** Provide arguments about the worst-case time complexity of your algorithm.

2. **[Find a pair with a given sum in a sorted array]** Given a sorted array `a` of `n` elements and a target sum `target`, find a pair of elements in the array that adds up to `target`. If no such pair exists, return `-1`. The function should run in linear time (i.e.,  $\mathcal{O}(n)$  time) in the worst case.

For example, if `a = {2, 5, 9, 15, 16}`, `target = 20`, the output will be `(5, 15)`.

**Q29** Provide arguments about the worst-case time complexity of your algorithm.

3. **[Interval search in a sorted array]** Given a sorted array `a` of `n` (possibly non-distinct) elements, and given two integers `p` and `q`, where `p ≤ q`, find all the elements of `a` that lie within the interval `[a, b]`.

For example, if `a = {2, 5, 6, 9, 15, 16}` and `a = 5`, `b = 10`, then the output will be `{5, 6, 9}`.

**Q30** Provide arguments about the worst-case time complexity of your algorithm.

4. **[Peak in a bitonic array]** A **bitonic array** is defined as follows:

- The elements from the start of the array up to a peak index `p` are in increasing order.
- The elements from the peak index `p` to the end of the array are in decreasing order.

For example, in `a = [1, 3, 8, 12, 4, 2]`, the peak element is `12`.

Given a bitonic array `a` of `n` elements, write a function to find the peak element in `a`.

The function should run in logarithmic time (i.e.,  $\mathcal{O}(\log n)$  time) in the worst case.

Assume that all elements in `a` are distinct.

**Q31** Provide arguments about the worst-case time complexity of your algorithm.

- ♣ 5. **[Ternary search in a sorted array]** Write a function `ternarySearch` that searches for a target element (`key`) in an array sorted in ascending order. The function should have the prototype:

```
int ternarySearch(int arr[], int left, int right, int key);
```

The function should return the index of the `key` if it is found in the array. If the `key` is not present in the array, the function should return `-1`.

It should perform the following steps:

- (i) Divide the array:** In each iteration, divide the search interval into three parts. This is done by computing two mid-points: `mid1` and `mid2`. The array is then split into three segments based on these mid-points.
- (ii) Determine the search region:** Compare the target element with the values at `mid1` and `mid2`. Based on these comparisons, decide which segment of the array the target element might be in. This helps in narrowing down the search region efficiently.
- (iii) Recursively Search the Region:** Recurse on the segment where the target element might be located. This process continues until the element is found or the search interval is reduced to an empty range.

**Q32** What will be the worst-case time complexity of your algorithm? Is this time complexity better than that of binary search? Justify.

- ♣ 6. [Exponential search in a sorted array] In this problem, you are required to implement a search function to find a target element (**key**) in a sorted array. Assume that the array is sorted in ascending order. The function should perform the following steps:

- (i) **Determine the range:** Begin at the start of the array and repeatedly double the index to find a range in which the target element might be located. For instance, if the target element is less than the value at the current index, then the element must lie in the range from the previous index to the current index.
- (ii) **Binary search within the range:** Once the range is identified, use Binary Search to find the exact position of the target element within this range.

Exponential search is particularly efficient for unbounded or infinite lists. The time complexity is  $\mathcal{O}(\log n)$ , similar to binary search.

- ♠ 7. [Number of palindromic subsequences] A **subsequence** of a sequence is derived by deleting some or none of the elements from the original sequence without changing the order of the remaining elements. A sequence or subsequence is **palindromic** if it remains the same when read from left to right and from right to left. Thus, a **palindromic subsequence** of a given sequence is a subsequence that reads the same forwards and backwards.

For example, the sequence **abcba** has 13 palindromic subsequences, which are:

a	b	c	b	a
aa	bb			
aba	aca	aba	bc b	
abba				
abcba				

Your task is to find the number of palindromic subsequences in a given sequence. You need not print these subsequences.

Your code should treat two subsequences as different only if the original indices of their constituent characters are different, even if they consist of the same sequence of characters. For example, in the 1st row of the above table, **a** appears twice; its first occurrence is for the first **a** in **abcba**, while its second occurrence is for the second **a** in **abcba**. Similarly, in the 3rd row of the above table, **aba** appears twice; its first occurrence is for the first **b** in **abcba**, while its second occurrence is for the second **b** in **abcba**.

- ♠ 8. [Max length of monotonic subsequence] A **subsequence** of a sequence is derived by deleting some or none of the elements from the original sequence without changing the order of the remaining elements. A sequence or subsequence is **monotonic** if its characters are in non-decreasing order when read from left to right.

You are given a sequence **s** consisting of 0, 1, and 2 only. Your task is to find the maximum length (**maxlen**) of a monotonic subsequence in **s** such that it contains at least one from each of 0, 1, and 2. You need not print that subsequence. Here are some examples:

- **s** = 2011: **maxlen** = 0
- **s** = 2012: **maxlen** = 3
- **s** = 001120201: **maxlen** = 6
- **s** = 1021002110210011012210: **maxlen** = 11
- **s** = 10210021102100110122101021002110210011012210: **maxlen** = 20

# 11 | Sorting

**Sorting** means ordering. Given an array  $a[]$  with  $n$  distinct elements arranged in an arbitrary order, the task of sorting involves rearranging them in increasing or decreasing order. If the elements are not distinct, the ordering will be non-decreasing or non-increasing. After sorting, the elements are usually stored in the same array  $a[]$  or in a different array.

For example, if the given array is  $\{5, 2, 8, 5, 3\}$ , then after sorting in non-decreasing order, we get  $\{2, 3, 5, 5, 8\}$ . If we sort the same array in non-increasing order, we get  $\{8, 5, 5, 3, 2\}$ .

Henceforth, we will assume that sorting refers to arranging elements in increasing or non-decreasing order. Specifically, the array  $a[]$  is considered sorted if and only if  $a[0] \leq a[1] \leq a[2] \leq \dots \leq a[n-1]$ . This assumption does not affect any subsequent reasoning or arguments. With minor adjustments, any sorting algorithm can be adapted to handle decreasing or non-increasing order as well. So, we proceed with the following definition of sorted array:

## Definition 11.1 (Sorted array)

An array  $a[]$  is sorted **if and only if** every two consecutive elements are in the correct order, i.e.,

$$a[i] \geq a[i-1] \text{ for every } i \text{ from } 1 \text{ to } n-1.$$

**Q33** Write a code that checks whether a given array  $a[]$  is sorted or not. How many comparisons will be needed by your code if the array has  $n$  elements, which are not necessarily distinct?

Question 33 is conceptually very simple if we temporarily set aside coding syntax. What will we do? We'll simply apply Definition 11.1: compare every pair of consecutive elements to verify if they are in the correct order. If not, the array is unsorted. However, Definition 11.1 cannot be directly used to sort an unsorted array. We have to dive a little deeper...

Is there any other way to determine if the array is sorted? Think about it! ☺

We can think of splitting the array into two parts: the left part is called a **prefix**, and the right part is called a **suffix**. If one of them is the entire array, the other is empty. Since  $a[]$  contains  $n$  elements, there are exactly  $n$  nonempty prefixes and exactly  $n$  nonempty suffixes. Specifically, any subarray from  $a[0]$  to  $a[i]$ , where  $i$  ranges from  $0$  to  $n-1$ , is considered a nonempty **prefix** of  $a[]$ , and the remainder is a **suffix**. We make the following definition:

## Definition 11.2 (Tail-Max Prefix)

The subarray  $a[0], \dots, a[i]$  is called a **tail-max prefix** if and only if  $a[i] = \max \{a[j] : 0 \leq j \leq i\}$ .

The above definition plays a crucial role in determining whether an array is sorted, as stated below.

## Fact 11.1 (Sorted array)

An array  $a[]$  is sorted if and only if all its prefixes are tail-max.

As an example, consider the array  $\{2, 3, 5, 5, 8\}$ . It has five elements and hence five prefixes:  $\{2\}$ ,  $\{2, 3\}$ ,  $\{2, 3, 5\}$ ,  $\{2, 3, 5, 5\}$ ,  $\{2, 3, 5, 5, 8\}$ . Observe that, for each of these prefixes, the largest element appears at the end of that prefix, which implies the original array is sorted. On the contrary, for the array  $\{5, 2, 8, 5, 3\}$ , which has the following five prefixes:  $\{5\}$ ,  $\{5, 2\}$ ,  $\{5, 2, 8\}$ ,  $\{5, 2, 8, 5\}$ , and  $\{5, 2, 8, 5, 3\}$ , the prefixes  $\{5, 2\}$ ,  $\{5, 2, 8, 5\}$ , and  $\{5, 2, 8, 5, 3\}$  do not satisfy the **Tail-Max** property, and hence it is not sorted.

Try to prove why Fact 11.1 holds. Don't miss the **if and only if**. You can use the definitions to prove it.

## 11.1 Bubble Sort: A basic sorting algorithm

The heart of Bubble Sort is Fact 11.1. It basically establishes the **Tail-Max** property for each prefix, starting from the longest prefix (i.e., the entire array). For every prefix, it iteratively checks every two consecutive elements and swaps them if they are not in the correct order. As a result, each prefix ultimately becomes a tail-max prefix.

Code 11.46: Bubble Sort.

`bubbleSort.c`

```

1 void bubbleSort(int a[], int n){ // Full code: bubbleSortMain.c
2   for (int i = n-1; i >= 1; i--){ // Longest to smallest prefix
3     int swapped = 0; // To track if a swap occurred
4     for (int j=0; j <= i-1; j++){ // Check each prefix {a[0],...,a[i]}
5       if (a[j] > a[j + 1]){ // Swap a[j] and a[j + 1]
6         int temp = a[j];
7         a[j] = a[j + 1];
8         a[j + 1] = temp;
9         swapped = 1; // A swap occurred
10      }
11    }
12
13    if (!swapped) // No swap => prefix is sorted => all its prefixes are sorted
14      break; // Returns to the caller, e.g. main()
15  }
16 }
```

i	j	Unsorted Part	Sorted Part	Swap
4	0	{5, 2, 8, 5, 3}	{}	Swap 5 and 2
4	1	{2, 5, 8, 5, 3}	{}	No swap for 5 and 8
4	2	{2, 5, 8, 5, 3}	{}	Swap 8 and 5
4	3	{2, 5, 5, 8, 3}	{}	Swap 8 and 3
3	0	{2, 5, 5, 3}	{8}	No swap for 2 and 5
3	1	{2, 5, 5, 3}	{8}	No swap for 5 and 5
3	2	{2, 5, 5, 3}	{8}	Swap 5 and 3
2	0	{2, 5, 3}	{5, 8}	No swap for 2 and 5
2	1	{2, 5, 3}	{5, 8}	Swap 5 and 3
1	0	{2, 3}	{5, 5, 8}	No swap for 2 and 3
-	-	{}	{2, 3, 5, 5, 8}	-

Figure 11.1: Bubble Sort on  $\{5, 2, 8, 5, 3\}$ .

An implementation of Bubble Sort is given in Code 11.46, and its demonstration is shown in Figure 11.1. Its outer `for`-loop considers all prefixes, starting from the longest one. For each prefix, the inner `for`-loop examines every two consecutive elements and swaps them if needed. If, for a particular prefix  $\{a[0], \dots, a[i]\}$ , no swap is needed, the flag `swapped` is found to be 0 (i.e., false) after the checking is complete for  $\{a[0], \dots, a[i]\}$ , indicating that  $\{a[0], \dots, a[i]\}$  is sorted. This, in turn, implies that all the prefixes of  $\{a[0], \dots, a[i]\}$  are also sorted. Since the suffix  $\{a[i+1], \dots, a[n-1]\}$  is already sorted from previous iterations of the outer `for`-loop, the algorithm terminates successfully with no further iterations. Thus, the `swapped` flag in the code enhances the efficiency of the algorithm.

During execution, Bubble Sort implicitly partitions the original array into two parts: the `unsorted part`, which lies at the front end and gradually diminishes in size, and the `sorted part` that follows, as illustrated in Figure 11.1. The sorted part contains the largest elements of the prefixes processed so far. The algorithm begins with an empty sorted part and continues until the unsorted part becomes empty.

## Time Complexity

**Best case:** The best-case scenario occurs when the array is already sorted, resulting in a time complexity of  $\mathcal{O}(n)$  since no swaps are required.

**Worst case:** In the worst-case scenario, the array is sorted in reverse order, leading to a time complexity of  $\mathcal{O}(n^2)$  as all elements must be compared and swapped.

**Average case:** On average, Bubble Sort performs about  $n^2/4$  comparisons and swaps, resulting in an average time complexity of  $\mathcal{O}(n^2)$ .

## Comments

Bubble Sort is a straightforward sorting algorithm that repeatedly traverses the list, compares adjacent elements, and swaps them if they are in the wrong order. The algorithm derives its name from the way lighter elements “bubble” to the top of the array with each pass, while heavier elements settle at the bottom.

Although Bubble Sort is an in-place sorting algorithm, it is not efficient for large datasets due to its quadratic time complexity. Despite its inefficiency compared to more advanced algorithms, this algorithm is studied primarily for historical interest.

**Q34** What will be the number of comparisons for Bubble Sort, as per Code 11.46, for an array of 100 distinct elements if the array is already sorted in increasing order? What happens if it is already sorted in decreasing order? What will be these two values if the flag `swapped` is not used?

**Q35** Can you initialize with `i = 0` in Code 11.46 and suitably adjust the initialization of `j` so that your code is correct?

**Q36** Can you write a provable fact similar to Fact 11.1 in which ‘prefix’ is replaced by ‘suffix’? Can you use it to design a sorting algorithm?

**Q37** Given an array of 10 elements, the task is to find its smallest 5 elements. Write a code for this by modifying Code 11.46.



## 11.2 Selection Sort (Version 1)

Just like Bubble Sort, Selection Sort is also in-place, comparison-based, and works with the **Tail-Max** principle. The only difference is that it repeatedly selects the largest element from the unsorted part (prefix) of the array and swaps it with its last element. The last element `a[maxIndex]` of the prefix now serves as the first element of the updated suffix, which, being already sorted (with elements no less than `a[maxIndex]`), remains sorted. See its implementation in Code 11.47.

Recall that Bubble Sort has a special flag `swapped` that enhances its efficiency in the best and average cases. However, Selection Sort does not utilize any such flag, resulting in a time complexity of  $\mathcal{O}(n^2)$  for all cases (best, average, and worst) due to the nested loops.

**Code 11.47:** Selection Sort.

`selectionSort.c`

```
1 void selectionSort(int a[], int n) { // Full code: selectionSortMain.c
2   for (int i = n - 1; i >= 1; i--) { // Longest to smallest prefix
3     int maxIndex = 0;                // Index of the maximum element
4     for (int j = 1; j <= i; j++) {   // Find the maximum element in each prefix
5       if (a[j] > a[maxIndex])
6         maxIndex = j;
7     }
8
9     // Make the prefix tail-max
10    if (maxIndex != i) {              // Only swap if maxIndex is not i
11      int temp = a[maxIndex];
12      a[maxIndex] = a[i];
13      a[i] = temp;                   // Place the maximum element at the end
14    }
15  }
16 }
```

**Q38** Justify whether true or false: “There is no practical way to optimize the performance of Selection Sort with a flag for early termination.”

**Q39** Justify whether true or false: “Selection Sort is a stable sorting algorithm.”

**Q40** Under what circumstances might Selection Sort be preferred over more efficient algorithms?

## 11.3 Selection Sort (Version 2)

Selection Sort is a simple sorting algorithm. The algorithm divides the input array into two parts: the sorted part and the unsorted part. **The sorted part grows at the beginning of the array itself.** Initially, the sorted part is empty, and the unsorted part contains all the elements. On each iteration, the algorithm selects the smallest element (or one of the smallest, if there is more than one) from the unsorted portion and swaps it with the element at the current position, extending the sorted portion by one.

**Code 11.48:** Selection Sort.

`selectionSortVer2.c`

```

1 void selectionSort(int a[], int n){
2     for (int i = 0; i < n - 1; i++){
3         int minIndex = i;
4         for (int j = i + 1; j < n; j++){
5             if (a[j] < a[minIndex])
6                 minIndex = j;
7         }
8         int temp = a[i];
9         a[i] = a[minIndex];
10        a[minIndex] = temp;
11    }
12 }
```

Step	Array State	Sorted Part	Unsorted Part	Min	Swap
0	{5, 2, 8, 6, 5, 3}	{}	{5, 2, 8, 6, 5, 3}	2	Swap 5 and 2
1	{2, 5, 8, 6, 5, 3}	{2}	{5, 8, 6, 5, 3}	3	Swap 5 and 3
2	{2, 3, 8, 6, 5, 5}	{2, 3}	{8, 6, 5, 5}	5	Swap 8 and 5
3	{2, 3, 5, 6, 5, 8}	{2, 3, 5}	{6, 5, 8}	5	Swap 6 and 5
4	{2, 3, 5, 5, 6, 8}	{2, 3, 5, 5}	{6, 8}	6	No swap needed
5	{2, 3, 5, 5, 6, 8}	{2, 3, 5, 5, 6}	{8}	8	No swap needed
6	{2, 3, 5, 5, 6, 8}	{2, 3, 5, 5, 6, 8}	{}	—	—

**Figure 11.2:** Selection Sort on the array {5, 2, 8, 6, 5, 3} (Min = smallest element in the unsorted part).

An implementation of Selection Sort is given in Code 11.48. To illustrate the behavior of the Selection Sort algorithm, a demonstration of the algorithm is given in Figure 11.2. Although Selection Sort is an in-place sorting algorithm, it is not stable. It can be made stable but that requires advanced data structures.

The core operation is selecting the smallest element (from the unsorted part), which gives Selection Sort its name. Despite its simplicity, this algorithm provides a good starting point for understanding basic sorting principles.

### Proof of correctness

The correctness of Selection Sort can be proved through an inductive argument:

- **Base Case:** After the first pass, the smallest element is correctly placed in the first position.
- **Inductive Step:** Assume that the first  $k$  elements are sorted after  $k$  iterations. In the  $(k+1)$ -th iteration, the algorithm selects the smallest element from the unsorted portion and swaps it with the first unsorted element. Thus, the first  $k + 1$  elements are sorted.

Since the unsorted portion decreases by one element in each iteration and the sorted portion grows, eventually the entire array is sorted.

## Time Complexity

**Best case:** The best-case scenario occurs when the array is already sorted. However, Selection Sort does not take advantage of this and still performs  $\mathcal{O}(n^2)$  comparisons. Thus, the best-case time complexity is  $\mathcal{O}(n^2)$ .

**Worst case:** In the worst case, the array is sorted in reverse order, but the number of comparisons remains the same. Therefore, the worst-case time complexity is also  $\mathcal{O}(n^2)$ .

**Average case:** In the average case, Selection Sort performs approximately  $n^2/2$  comparisons, leading to an average time complexity of  $\mathcal{O}(n^2)$ .

**Q41** Justify whether true or false: “Selection Sort’s primary inefficiency is the large number of comparisons, even when the array is already sorted.”

**Q42** Deduce the result that the average-case time complexity of Selection Sort is  $\mathcal{O}(n^2)$ .

**Q43** Explain why Selection Sort is not a stable sorting algorithm. Provide an example to support your answer.

**Q44** Compare Selection Sort with Bubble Sort. What are the advantages and disadvantages of each?

**Q45** Under what conditions might Selection Sort be preferred over more efficient algorithms like Merge Sort or Quick Sort?

## 11.4 Insertion Sort: Another basic sorting algorithm

Insertion Sort is based on the following fact:

### Fact 11.2 (Sorted array)

An array `a[]` is sorted if and only if all its prefixes are sorted.

As an example, consider the array `{2, 3, 5, 5, 8}`. It has five prefixes: `{2}`, `{2, 3}`, `{2, 3, 5}`, `{2, 3, 5, 5}`, and `{2, 3, 5, 5, 8}`, all of which are sorted, implying that the original array is sorted. On the contrary, for the array `{5, 2, 8, 5, 3}`, the prefix `{5, 2}` is not sorted, thus violating the **Sorted Prefix** property stated in Fact 11.2, indicating that the array is not sorted.

Fact 11.2 sounds obvious, but you should try to prove it. You can use previous definitions or facts for your proof.

An implementation of Insertion Sort is given in Code 11.49. Unlike Bubble Sort, the **unsorted part** in Insertion Sort is a suffix, whereas the **sorted part** is a prefix, as illustrated in Figure 11.3. After the *i*-th iteration of the outer loop, the sorted part contains all elements of the *i*-th prefix in sorted order. The algorithm begins with the sorted part containing just `a[0]` (1st prefix) and continues until the unsorted part becomes empty. For the prefix `{a[0], ..., a[i]}`, the inner loop inserts `a[i]` (denoted by `lastElem` in the code) at a suitable position so that the prefix remains sorted.

Code 11.49: Insertion Sort.

`insertionSort.c`

```

1 void insertionSort(int a[], int n){ // Full code: insertionSort.c
2     for (int i = 1; i < n; i++){
3         int lastElem = a[i];
4         int j = i - 1;
5
6         /* Move elements of a[0..i-1], that are greater than lastElem,
7            to one position ahead of their current position */
8         while (j >= 0 && a[j] > lastElem){
9             a[j + 1] = a[j];
10            j = j - 1;
11        }
12        a[j + 1] = lastElem;
13    }
14 }
```

i	Prefix (Sorted)	Suffix (Unsorted)	lastElem	Move
1	{5}	{2, 8, 5, 3}	2	Move 2 before 5
2	{2, 5}	{8, 5, 3}	8	No move
3	{2, 5, 8}	{5, 3}	5	Move 5 before 8
4	{2, 5, 5, 8}	{3}	3	Move 3 before 5
–	{2, 3, 5, 5, 8}	{}	–	End

Figure 11.3: Insertion Sort on `{5, 2, 8, 5, 3}`.

**Q46** What is the worst-case time complexity of Code 11.49? When does it occur? Can you modify Code 11.49 so that its worst-case time complexity is improved? How?

**Differences between Bubble Sort and Insertion Sort:**

1. Bubble Sort establishes the Tail-Max property for each prefix, whereas Insertion Sort establishes the Sorted Prefix property.
2. Bubble Sort starts with the longest prefix, whereas Insertion Sort starts with the smallest prefix, and maintains Sorted Prefix property for each subsequent prefix, until all prefixes are sorted.
3. In Bubble Sort, the sorted part appears at the end and it never changes when the unsorted part is gradually sorted. In Insertion Sort, the sorted part receives the first element of the unsorted part in each iteration of the outer loop, and that element gets inserted at a suitable position of the sorted part.

## 11.5 Quick Sort

Quick Sort stands out among sorting algorithms due to its remarkable efficiency and elegance, particularly in handling large datasets. Unlike Bubble Sort, Selection Sort, and Insertion Sort—which operate with a time complexity of  $\mathcal{O}(n^2)$  in their worst cases—Quick Sort achieves an average-case time complexity of  $\mathcal{O}(n \log n)$ . This is **pseudo-linear**, i.e., almost linear even for very large values of  $n$ . As you can check, when  $n$  is around one billion, those three algorithms take around a billion of billions comparisons on the average, whereas Quick Sort takes just ten billions!

Before looking at the exact steps of Quick Sort, let us first comprehend an interesting fact presented next. In this fact, by **prefix-suffix pair** of the array  $a[]$  with  $n$  elements, we mean any prefix and its resultant suffix, assuming that none is empty. Thus, the pair can be expressed as  $(\{a[0], \dots, a[i]\}, \{a[i+1], \dots, a[n-1]\})$ , where  $i$  ranges from  $0$  to  $n-2$ . For brevity, we denote this pair by  $(p[], s[])$ .

### Fact 11.3 (Sorted array)

An array is sorted if and only if the following relation holds for every prefix-suffix pair  $(p[], s[])$ :

**Relation PSR:** Every element of  $p[]$  is at most as large as every element of  $s[]$ .

*Proof.* (Forward) We prove by contradiction. Suppose that PSR holds for every prefix-suffix pair but the array is not sorted. Then, there must exist at least two elements  $a[i]$  and  $a[j]$  such that  $i < j$  and  $a[i] > a[j]$ . Since  $i < j$ , we can construct a prefix-suffix pair  $(p[], s[])$  such that  $p[]$  contains  $a[i]$  and  $s[]$  contains  $a[j]$ . This specific pair violates PSR, resulting to contradiction.

(Converse) If the array is sorted, then  $a[i] \leq a[j]$  for every pair  $(i, j)$  with  $i < j$ . So, for every prefix-suffix pair, PSR holds.  $\square$

As an illustration, consider two arrays:  $\{2, 3, 5, 5, 8\}$  and  $\{5, 2, 8, 5, 3\}$ , the former being sorted, but the latter not. For the sorted array, PSR holds for every prefix-suffix pair. For instance, take the prefix  $\{2, 3, 5\}$  and suffix  $\{5, 8\}$ . Notice that every element of the prefix is at most as large as every element of the suffix. On the contrary, for the unsorted array, if we take the prefix  $\{5, 2, 8\}$  and suffix  $\{5, 3\}$ , the prefix element  $8$  is greater than the suffix element  $5$ , which violates PSR.

### PS-Sort

Based on Fact 11.3, we can design an algorithm for sorting, named **PS-Sort** ('P' for prefix, 'S' for suffix; we are sorting based on prefix-suffix pairs, whence the name), as shown in Code 11.50. It has a nested loop. The outer loop iterates over each possible prefix of the array, starting from the smallest one. The inner loop the last element  $a[i]$  of the prefix  $p[]$  with every subsequent element  $a[j]$  in the suffix  $s[]$ , where  $j > i$ . The formal proof of correctness is given below.

Code 11.50: PS-Sort.

`prefixSuffixSort.c`

```

1 void PS_Sort(int a[], int n){
2     for (int i = 0; i < n - 1; i++){
3         for (int j = i + 1; j < n; j++){
4             if (a[i] > a[j]){ // Swap a[i] and a[j] to ensure CPS
5                 int temp = a[i];
6                 a[i] = a[j];
7                 a[j] = temp;
8             }
9         }
10    }
11 }
```

### Proof of correctness of PS-Sort

Proving that PS-Sort follows Fact 11.3 suffices. Let's prove by induction on the index  $i$  that works as a variable of the outer loop.

**Basis:** For  $i = 0$ , the prefix  $p[]$  consists of the single element  $a[0]$ , and the suffix  $s[]$  consists the rest. The inner loop compares  $a[0]$  with all elements of  $s[]$ , ensuring that if any element of  $s[]$  is smaller than  $a[0]$ , they are swapped, thus ensuring PSR.

**Hypothesis:** Assume that PSR is true for every prefix-suffix pair up to iteration  $i - 1$ .

**Inductive Step:** By hypothesis, PSR holds for every prefix-suffix pair up to iteration  $i - 1$ , which we refer to as **previous iteration**. We need to prove that PSR holds for the prefix-suffix pair at iteration  $i$  (current iteration). The current prefix includes  $a[i]$  as its last element, which happened to be the first element of the previous suffix. The inner loop ensures  $a[i]$  in the current prefix is the smallest element of the previous suffix, so that no element of the current suffix is smaller than  $a[i]$ . During this process,  $a[i]$  is compared with each element  $a[j]$  of the set  $\{a[i+1], \dots, a[n-1]\}$ , and if  $a[j]$  is smaller, the two are swapped. This guarantees that PSR is maintained for the current prefix-suffix pair.  $\square$

### Time complexity of PS-Sort

The outer loop runs from  $i = 0$  to  $i = n - 2$ , which is  $\mathcal{O}(n)$ . For each iteration of the outer loop, the inner loop runs from  $j = i + 1$  to  $j = n - 1$ , and so it has  $n - i - 1$  iterations. Summing up, the total number of iterations in the PS-Sort algorithm is

$$\sum_{i=0}^{n-2} (n - i - 1) = \sum_{i=0}^{n-2} (n - 1) - \sum_{i=0}^{n-2} i = (n - 1)^2 - \frac{(n - 2)(n - 1)}{2} = \frac{n(n - 1)}{2} = \mathcal{O}(n^2).$$

Each of the above iterations takes exactly one comparison and at most one swap. Thus, the time complexity in best, worst, and average cases comes  $\mathcal{O}(n^2)$ .

### How is PS-Sort?

If you examine carefully, the algorithm PS-Sort is essentially a variant of the Selection Sort algorithm, but with a different mechanism for locating and placing the smallest element in the correct position. In the Selection Sort algorithm, for each iteration, the smallest element in the unsorted portion of the array is identified and swapped with the first unsorted element, ensuring that the new prefix is sorted after each iteration. However, in PS-Sort, the last element of the current prefix is compared with all other elements in the suffix, and swaps are performed whenever necessary to maintain the Prefix-Suffix Relation (PSR). This approach results in multiple swaps, in contrast to Selection Sort, which only swaps the smallest element once per iteration.

Despite the difference in implementation, both algorithms have the same time complexity of  $\mathcal{O}(n^2)$ , but Selection Sort typically performs fewer swaps since it minimizes them to one per iteration of the outer loop, whereas PS-Sort may swap elements more frequently.

Nevertheless, the idea of using Prefix-Suffix Relation (PSR) can be used to improve the average-case time complexity to  $\mathcal{O}(n \log n)$ . To achieve this, we need the following fact, which is even deeper than the previous one.

#### Fact 11.4 (Sorted array)

An array is sorted if and only if PSR holds for some prefix-suffix pair  $(p[], s[])$ , and recursively holds also for some prefix-suffix pair of  $p[]$  and some prefix-suffix pair of  $s[]$ .

*Proof.* (Forward) We proceed by induction on  $n$ , with the base case  $n = 1$ , where the proof is trivial. Assume as the inductive hypothesis that the fact holds when the array has 2 to  $n - 1$  elements.

In the inductive step, let PSR hold for some prefix-suffix pair  $(p[], s[])$  in an array of size  $n$ . By the inductive hypothesis,  $p[]$  and  $s[]$  are sorted, as each contains at most  $n - 1$  elements and admits PSR recursively. Further, since PSR holds for  $(p[], s[])$ , no element of  $p[]$  is larger than any element of  $s[]$ . Thus, the entire array is sorted.

(Converse) Assume the array is sorted. By Fact 11.3, PSR holds for every prefix-suffix pair  $(p[], s[])$ , including recursively defined prefix-suffix pairs within  $p[]$  and  $s[]$ . This confirms the converse.  $\square$

Fact 11.4 underpins the principle of Quick Sort. Based on this principle, Quick Sort utilizes the **divide-and-conquer** technique to sort an array or subarray. Here are the main steps:

1. **Divide:** Choose an element called **pivot** from the array. This can be the first, last, or a random element. We choose the last element in Code 11.51. Partition the array into three parts: **prefix**, **pivot**, **suffix**, so that no element of **prefix** is larger than the pivot and no element of **suffix** is smaller than the pivot.
2. **Conquer:** Recursively apply Quick Sort to the prefix and the suffix. (The pivot is already in its correct position.)
3. **Combine:** Because the prefix and the suffix are already sorted, and the pivot is sandwiched between them, no work is needed to combine them! The entire array is now sorted.

The **divide-and-conquer** paradigm involves three steps at each level of the recursion:

1. **Divide** the problem into smaller sub-problems.
2. **Conquer** the sub-problems by solving them recursively. Define the basis of recursion so that when a sub-problem is small enough, the solution is trivial.
3. **Combine** the sub-problem solutions to get the solution for the original problem.

The Quick Sort code is presented in Code 11.51, and its demonstration on two arrays is illustrated in Figure 11.4. Its recursion tree for one array is given in Figure 11.5.

Consider the array or subarray  $a[\text{low}], \dots, a[\text{high}]$ . If it is the original array  $a[]$  containing  $n$  elements, then the indices **low** and **high** are 0 and  $n - 1$ , respectively. A subarray typically represents either a prefix or suffix of the original array or of a larger prefix or suffix. Quick Sort has several variations, which are minor modifications of one another. In Code 11.51, the pivot is always the last element of the array or subarray to be partitioned into a prefix and suffix. Moreover, the pivot is neither included in the prefix nor the suffix. In some variants, the pivot is the first element and is placed in the prefix if the suffix is nonempty, and in the suffix otherwise.

## Time complexity of Quick Sort

**Worst-case time complexity:** The worst-case scenario for Quick Sort occurs when the pivot is always the smallest or largest element in the array or subarray at every recursive step. This usually happens when the original array is already sorted in ascending or descending order. As a result, the partitions become highly unbalanced, where the prefix (or suffix) contains all elements except the pivot, leaving the suffix (or prefix) empty. In these cases, Quick Sort requires  $\mathcal{O}(n)$  time at each recursive step, and since there are  $n$  steps, the overall time complexity becomes  $\mathcal{O}(n^2)$ .

**Best-case time complexity:** The best case occurs when the pivot always divides the array or subarray into two roughly equal halves at every step of the recursion. When this happens, Quick Sort converges faster with almost  $\log_2(n)$  depth in the recursion tree. At every level of the recursion tree,  $n$  or fewer elements are processed in linear time. As a result, the overall time complexity becomes  $\mathcal{O}(n \log n)$ .

**Average-case time complexity:** Most of the time, the pivot will divide the array or subarray into reasonably balanced parts, though not necessarily perfectly equal. Thus, the expected height of the recursion tree will be logarithmic in  $n$ , similar to the best case. Although the base of the logarithm is a constant factor larger compared to the best case, the average time complexity remains  $\mathcal{O}(n \log n)$ .



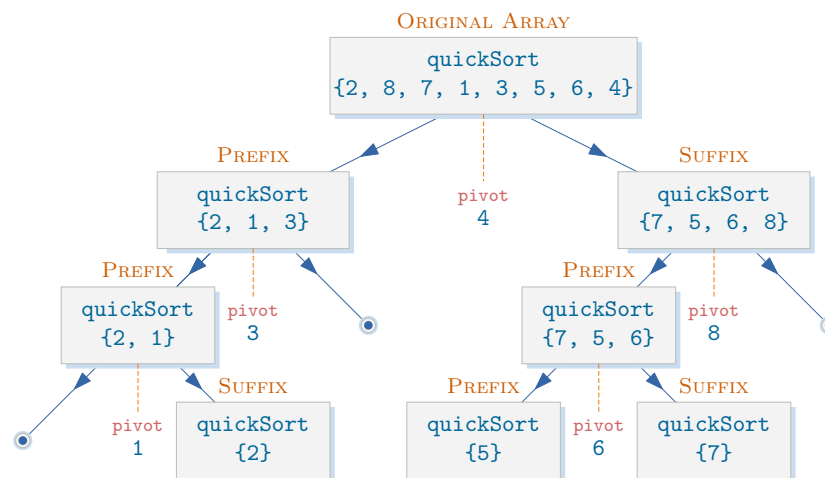
Code 11.51: Quick Sort.

quickSortCLRSMain.c

```
1 #include <stdio.h> // CLRS Book: Edition 3 (with some revisions mentioned below)
2
3 void printArray(int a[], int s, int t){
4     for (int i = s; i <= t; i++){
5         printf("%d ", a[i]);
6     }
7
8 int partition(int a[], int left, int right){ // CLRS Book: Edition 3
9     int pivot = a[right]; // Choose the last element as pivot
10    int i = left-1; // Index of the smaller element
11
12    for (int j = left; j <= right - 1; j++){
13        if (a[j] <= pivot){
14            i++;
15            if(i != j){ // Prevents self-swap (this "if" is not there in CLRS book)
16                int temp = a[i];
17                a[i] = a[j];
18                a[j] = temp;
19            }
20        }
21    }
22    // Place pivot at its correct position
23    if(i+1 != right){ // Prevents self-swap (this "if" is not there in CLRS book)
24        int temp = a[i+1];
25        a[i+1] = a[right];
26        a[right] = temp;
27    }
28    return i+1; // Return the partitioning index
29 }
30
31 void quickSort(int a[], int left, int right){ // CLRS Book: Edition 3
32     if (left < right){
33         int pivot = partition(a, left, right);
34         quickSort(a, left, pivot - 1); // Sort prefix
35         quickSort(a, pivot + 1, right); // Sort suffix
36     }
37 }
38
39 int main(){
40     int a[100], n;
41     printf("Enter the number of elements (at most 100): ");
42     scanf("%d", &n);
43
44     printf("Enter the elements: ");
45     for (int i = 0; i < n; i++){
46         scanf("%d", &a[i]);
47     }
48     quickSort(a, 0, n - 1); // Perform Quick Sort
49
50     printf("Sorted array: ");
51     printArray(a, 0, n-1);
52
53     return 0;
54 }
```

Step	Before partition	pivot (element)	left (index)	right (index)	Swaps	Prefix	Suffix	Array state
<b>Input array: {2, 8, 7, 1, 3, 5, 6, 4}</b>								
1	{2, 8, 7, 1, 3, 5, 6, 4}	4	0	7	8-1, 7-3, 8-4	{2, 1, 3}	{7, 5, 6, 8}	{2, 1, 3, 4, 7, 5, 6, 8}
2	{2, 1, 3}	3	0	2	-	{2, 1}	-	{2, 1, 3, 4, 7, 5, 6, 8}
3	{2, 1}	1	0	1	2-1	-	{2}	{1, 2, 3, 4, 7, 5, 6, 8}
4	{7, 5, 6, 8}	8	4	7	-	{7, 5, 6}	-	{1, 2, 3, 4, 7, 5, 6, 8}
5	{7, 5, 6}	6	4	6	7-5, 7-6	{5}	{7}	{1, 2, 3, 4, 5, 6, 7, 8}
<b>Input array: {5, 2, 8, 5, 3}</b>								
1	{5, 2, 8, 5, 3}	3	0	4	5-2, 5-3	{2}	{8, 5, 5}	{2, 3, 8, 5, 5}
2	{8, 5, 5}	5	2	4	8-5, 8-5	{5}	{8}	{2, 3, 5, 5, 8}

**Figure 11.4:** Demonstration of Quick Sort on two arrays. Note that in Step 2 and Step 4 of the first array, no swaps occur because the pivot is the largest element. As a result, their suffixes are empty, and the pivot does not belong to either the prefix or suffix. Thus, the prefix, followed by the pivot, and then the suffix, forms the array or subarray after each partition. The ‘array state’ column shows the arrangement of elements in the entire array after each step.



**Figure 11.5:** The recursion tree of Quick Sort for the input array {2, 8, 7, 1, 3, 5, 6, 4}.

The symbol  $\bullet$  stands for empty prefix or suffix.

An interesting fact: If you collect the elements of leaf nodes (pivots are also leaves) from left to right, you get the sorted sequence.

## Comments

Quick Sort is an in-place algorithm but uses the recursion stack. It does not admit tail recursion, so it cannot be implemented as an iterative function, as seen in previous algorithms such as Bubble Sort and Insertion Sort. However, by utilizing a user-defined stack, the recursion can be avoided, although this adds some complexity to the implementation.

## 11.6 Merge Sort

Merge Sort is a highly time-efficient, comparison-based sorting algorithm that uses the **divide-and-conquer** strategy. It works by recursively splitting an array or list into smaller subarrays until each subarray contains a single element, which is inherently sorted. Once the array is divided, the algorithm merges the subarrays back together in a way that results in a sorted array. See its implementation in Code 11.52.

During the merging phase, two sorted subarrays are compared element by element, and the smaller element is placed into the sorted array. This process is repeated until all elements are merged into a fully sorted array. The worst-case time complexity of merge sort is  $\mathcal{O}(n \log n)$ , where  $n$  is the number of elements to be sorted, making it highly efficient even for large datasets. Unlike other sorting algorithms such as Quick Sort, Merge Sort guarantees a stable sort, meaning that the relative order of equal elements is preserved. It also performs well with linked lists and is widely used in various applications that require efficient, large-scale sorting. However, it requires additional  $\mathcal{O}(n)$  space for the temporary subarrays.

Code 11.52: Merge Sort.

mergeSort.c

```
1 #include <stdio.h>
2
3 void merge(int a[], int left, int mid, int right){
4     int i = left, j = mid + 1, k = 0;
5     int temp[100];
6
7     while (i <= mid && j <= right){ // Merging the prefix and suffix
8         if (a[i] <= a[j])
9             temp[k++] = a[i++];
10        else
11            temp[k++] = a[j++];
12    }
13
14    while (i <= mid) // Copy remaining elements of prefix
15        temp[k++] = a[i++];
16    while (j <= right) // Copy remaining elements of suffix
17        temp[k++] = a[j++];
18
19    // Copy the sorted subarray back to the original array
20    for (i = left, k = 0; i <= right; i++, k++)
21        a[i] = temp[k];
22 }
23
24 void mergeSort(int a[], int left, int right){
25     if (left < right){
26         int mid = (left + right) / 2;
27         mergeSort(a, left, mid); // Sort the prefix
28         mergeSort(a, mid + 1, right); // Sort the suffix
29         merge(a, left, mid, right); // Merge the prefix and suffix
30     }
31 }
32
33 int main(){
34     int n, a[100]; // Assume that there are at most 100 elements
35     ... // Take as input n and the elements
36     mergeSort(a, 0, n - 1);
37     ... // Print and do other things if needed
38     return 0;
39 }
```

Step	Before partition or merging	left (index)	right (index)	mid (index)	Prefix	Suffix	After merging
Input array: {2, 8, 7, 1, 3, 5, 6, 4}							
1	{2, 8, 7, 1, 3, 5, 6, 4}	0	7	3	{2, 8, 7, 1}	{3, 5, 6, 4}	-
2	{2, 8, 7, 1}	0	3	1	{2, 8}	{7, 1}	-
3	{2, 8}	0	1	0	{2}	{8}	{2, 8}
4	{7, 1}	2	3	2	{7}	{1}	{1, 7}
5	{2, 8, 1, 7}	0	3	1	{2, 8}	{1, 7}	{1, 2, 7, 8}
6	{3, 5, 6, 4}	4	7	5	{3, 5}	{6, 4}	-
7	{3, 5}	4	5	4	{3}	{5}	{3, 5}
8	{6, 4}	6	7	6	{6}	{4}	{4, 6}
9	{3, 5, 4, 6}	4	7	5	{3, 5}	{4, 6}	{3, 4, 5, 6}
10	{1, 2, 7, 8, 3, 4, 5, 6}	0	7	3	{1, 2, 7, 8}	{3, 4, 5, 6}	{1, 2, 3, 4, 5, 6, 7, 8}
Input array: {5, 2, 8, 5, 3}							
1	{5, 2, 8, 5, 3}	0	4	2	{5, 2, 8}	{5, 3}	-
2	{5, 2, 8}	0	2	1	{5, 2}	{8}	-
3	{5, 2}	0	1	0	{5}	{2}	{2, 5}
4	{2, 5, 8}	0	2	1	{2, 5}	{8}	{2, 5, 8}
5	{5, 3}	3	4	3	{5}	{3}	{3, 5}
6	{2, 5, 8, 3, 5}	0	4	2	{2, 5, 8}	{3, 5}	{2, 3, 5, 5, 8}

Figure 11.6: Demonstration of Merge Sort on two arrays.

## 11.7 Classification of sorting algorithms

All the commonly used sorting algorithms are discussed in this chapter. The only one left is Heap Sort. **Heap Sort** is a comparison-based sorting algorithm that uses a **binary heap** data structure to sort elements. It works by first building a **max heap** from the input array, ensuring that the largest element is placed at the root. The algorithm then repeatedly extracts the maximum element from the heap, reducing the heap size by one. After each extraction, the heap property is restored to maintain order. Heap Sort has a worst-case time complexity of  $\mathcal{O}(n \log n)$  and is an in-place algorithm, making it space-efficient.

**Note:** A **binary tree** is a hierarchical structure with a root node. Each of its nodes has up to two children, known as the left and right child. A **binary heap** is a special type of binary tree where each parent has at most two children, and the tree is completely filled from top to bottom, left to right. It has a clever-yet-simple implementation using 1D array. In a **max heap**, the value of each parent is greater than or equal to its children.

All the algorithms discussed in this chapter, including Heap Sort, fall under **comparison-based sorting**, as they rely on comparing elements of the array to perform the sorting task. Non-comparison-based algorithms, such as Radix Sort or Counting Sort, avoid direct comparisons and may run faster under certain conditions; however, they are suitable for some specific types of data only.

|| Heap Sort, Radix Sort, and Counting Sort are not in your syllabus.

**Table 11.1:** Properties and complexities of comparison-based sorting algorithms.

Algorithm	In-place	Stable	Time Complexities			Auxiliary Space
			Best Case	Worst Case	Average Case	
Selection Sort	✓	✗	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Bubble Sort	✓	✓	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Insertion Sort	✓	✓	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Merge Sort	✗	✓	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$
Quick Sort	✓	✗	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(\log n)$
Heap Sort	✓	✗	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(1)$

Each algorithm has its own strengths and weaknesses, and the choice of which to use depends on the nature and size of the input data, and the specific application requirements. Among the algorithms listed above, only Merge Sort requires an auxiliary array, while the others perform **in-place sorting**, meaning they sort the data without needing extra space proportional to the input array size. However, it is important to note that Quick Sort also utilizes a stack to manage recursive calls, which results in additional space usage, typically  $\mathcal{O}(\log n)$  on average. There are certain advanced techniques to make Merge Sort work in-place, but those are not easily implementable. Quick Sort and Heap Sort stand out for their intriguing analyses and tricky implementations. Both algorithms are widely studied for their elegance and practical efficiency. Similarly, Merge Sort has received significant attention for its stability and effective performance, particularly in external sorting scenarios. The key attributes of a sorting algorithm are evaluated based on their properties, as well as their time and space complexities. The important ones are as follows:

- **Comparison-based sorting:** These algorithms rely on comparing elements to determine their order. It can be proved that the worst-case time complexity for such algorithms cannot be reduced below  $\mathcal{O}(n \log n)$ .
- **In-place sorting:** An in-place algorithm sorts data without requiring additional space proportional to the input size. It typically uses only a constant amount of extra space, making it memory-efficient.
- **Stable sorting:** A sorting algorithm is stable if it preserves the relative order of elements with equal keys. For  $i < j$ , if  $a[i]$  and  $a[j]$  are equal in the input array, a stable sort ensures that  $a[i]$  appears before  $a[j]$  in the output.

**Example:** Consider an array of pairs  $\{(5,A), (1,B), (5,C)\}$ . After sorting by the first value (key), a stable algorithm would return  $\{(1,B), (5,A), (5,C)\}$ , preserving the order of  $(5,A)$  and  $(5,C)$ .

Table 11.1 provides a concise comparison of the above sorting algorithms based on their type (comparison-based, in-place, stable) and their time complexities across best, worst, and average cases. This allows for an easy selection of the most appropriate algorithm depending on the nature of the data and the requirements of the task at hand. By comparing attributes such as stability and efficiency, the table helps highlight the trade-offs involved, especially for large datasets.

## 11.8 Recursive vs Iterative Algorithms

Sorting algorithms may be **recursive** (e.g., Quick Sort, Merge Sort) or **iterative** (e.g., Bubble Sort). Recursive algorithms often use function calls to divide the problem into smaller sub-problems, while iterative ones rely on loops to process elements.

Both Quick Sort and Merge Sort can be implemented iteratively but the coding is complex. To implement Quick Sort iteratively, an explicit stack is needed to manage the subarrays, making it iterative. Merge Sort can also be implemented iteratively by using a bottom-up approach, where the array is repeatedly merged in pairs until fully sorted.

## 11.9 Exercise problems

1. Identify the sorting algorithms that require constant amount of additional space for sorting.
2. How many swaps and how many comparisons are used in Code 11.46 (Bubble Sort) for the input array {3, 2, 6, 5, 8, 5}?
3. What is the best-case time complexity for Bubble Sort? Justify.
4. How many swaps and how many comparisons are used in Code 11.48 (Selection Sort) for the input array {3, 2, 6, 5, 8, 5}?
5. How many assignments (to array elements) are used in Code 11.49 (Insertion Sort) for the input array {3, 2, 6, 5, 8, 5}?
6. How many times is the function `partition` called in Code 11.51 (Quick Sort) for the input array {3, 2, 6, 5, 8, 5}?
7. How many swaps are required by Code 11.51 (Quick Sort) to sort the array {1, 2, 3, 4, 5, 6, 7, 8}?
8. Which sorting algorithms have the same time complexity for best, worst, and average cases?
9. Which sorting algorithms perform in-place sorting but are not stable?
10. Explain why Merge Sort is often preferred over Selection Sort for larger datasets.
11. Name a sorting algorithm which is adaptive, i.e., it takes advantage of existing order in the input.
12. Modify Code 11.51 (Quick Sort) to sort an array in non-increasing order.
13. Modify Code 11.51 (Quick Sort) to sort an array with the pivot as the first element during every partition.
14. What are the scenarios where Quick Sort performs poorly?
15. What happens when Quick Sort is applied to a nearly sorted array?
16. Write a Quick Sort algorithm that sorts an array of pairs based on the first element, and if the first elements are equal, sorts based on the second element?  
For example, it will sort {(3, a), (5, e), (3, c), (2, d), (5, b)} to:  
{(2, d), (3, a), (3, c), (5, b), (5, e)}.
17. Explain why Merge Sort is a stable sort.
- ♣ 18. Explain the space complexity of Quick Sort.
- ♣ 19. Write a program that finds the median of an array using a modified version of Quick Sort.
- ♣ 20. How would you implement a Quick Sort variant where the pivot is chosen randomly?  
**Hint:** `rand()` can be used to generate a random pivot. To do this, first, call `srand(time(NULL))` to initialize the random seed. Then, use `rand()` to get a random index `r` in the desired range. Swap the element at the index `r` with the usual pivot. This ensures that a random element is used as the pivot in each recursion. `rand()` generates a pseudo-random number, and `srand()` ensures different sequences across program runs. You have to include `stdlib.h` and `time.h` for these functions.
- ♠ 21. Can Quick Sort be made stable? If so, how?
- ♠ 22. How would you implement Quick Sort in-place using a three-way partitioning scheme to handle duplicate elements efficiently?

# 12 | Number systems

Numbers are the foundation of mathematics and science, providing a way to quantify, measure, and describe the universe. From ancient tally marks to the sophisticated binary codes of modern computing, number systems have evolved to meet the demands of different eras. For computers, number systems are essential—they enable machines to process, store, and communicate data efficiently using only binary digits. Understanding these systems, whether decimal, binary, or hexadecimal, is key to grasping how computers perform calculations and represent complex information. This chapter is to give an idea about the representation and significance of number systems in the digital age.

## 12.1 Representation of Integers

Integers can be represented in various number systems, each serving specific purposes. The most common systems include the decimal, binary, octal, and hexadecimal systems. Each representation has its applications in computing, digital electronics, and mathematics, influencing how integers are stored, manipulated, and processed in various systems.

The binary number system is the most important out of all these systems in the domain of digit arithmetic. There are several fundamental techniques to represent the binary number system. These techniques are mostly required to simplify subtraction operations, so that addition and subtraction can be handled using the same circuitry by merely complementing and adding. Understanding all these techniques is fundamental for grasping the evolution of binary arithmetic in computing.

### 12.1.1 Decimal number system

The **decimal number system**, also known as **base 10**, is the standard numerical system used in everyday life. It consists of ten digits, ranging from 0 to 9. Each digit's position represents a power of 10, facilitating the representation of numbers through combinations of these digits. Here are some important points:

- Basic symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
- **Radix-10** positional number system. The radix is also called the **base** of the number system.
- Example:  $12304 = 1 \times 10^4 + 2 \times 10^3 + 3 \times 10^2 + 0 \times 10^1 + 4 \times 10^0$ .

### 12.1.2 Octal number system

The **octal number system**, also known as **base 8**, is a numeral system that uses eight digits, ranging from 0 to 7. Each digit's position represents a power of 8, which allows for the compact representation of binary numbers. Here are some important points:

- Basic symbols: 0, 1, 2, 3, 4, 5, 6, 7
- **Radix-8** positional number system. The radix is also known as the **base** of the number system.
- Example:  $1947_8 = 1 \times 8^3 + 9 \times 8^2 + 4 \times 8^1 + 7 \times 8^0 = 1035_{10}$ .

### 12.1.3 Hexadecimal number system

The **hexadecimal number system**, also referred to as **base 16**, utilizes sixteen distinct symbols, ranging from 0 to 9 and A to F. Each digit's position represents a power of 16, making it useful for representing binary data more compactly. Here are some important points:

- Basic symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
- **Radix-16** positional number system. The radix is commonly referred to as the **base** of the number system.
- Example:  $1DA7_{16} = 1 \times 16^3 + 13 \times 16^2 + 10 \times 16^1 + 7 \times 16^0 = 7615_{10}$ .

### 12.1.4 Conversion from decimal number system

The methods for converting a number from any radix to another radix system are based on integer division and are relatively straightforward. In this section, we illustrate these methods using two decimal numbers, 1947 and 1757, to derive their representations in hexadecimal, octal, and binary systems. Each method involves repeated integer divisions until the quotient reaches zero. The sequence of digits, arranged from the highest to the lowest position, is formed by collecting the remainders in reverse order (see 12.1.5).

#### Decimal to Hexadecimal Conversion

Step	Division	Quotient	Remainder	Step	Division	Quotient	Remainder
1	$1947 \div 16$	121	11 (B)	1	$1757 \div 16$	109	13 (D)
2	$121 \div 16$	7	9	2	$109 \div 16$	6	13 (D)
3	$7 \div 16$	0	7	3	$6 \div 16$	0	6
Result: $1947_{10} = 79B_{16}$				Result: $1757_{10} = 6DD_{16}$			

#### Decimal to Octal Conversion

Step	Division	Quotient	Remainder	Step	Division	Quotient	Remainder
1	$1947 \div 8$	243	3	1	$1757 \div 8$	219	5
2	$243 \div 8$	30	3	2	$219 \div 8$	27	3
3	$30 \div 8$	3	6	3	$27 \div 8$	3	3
4	$3 \div 8$	0	3	4	$3 \div 8$	0	3
Result: $1947_{10} = 3633_8$				Result: $1757_{10} = 3355_8$			

#### Decimal to Binary Conversion

Step	Division	Quotient	Remainder	Step	Division	Quotient	Remainder
1	$1947 \div 2$	973	1	1	$1757 \div 2$	878	1
2	$973 \div 2$	486	1	2	$878 \div 2$	439	0
3	$486 \div 2$	243	0	3	$439 \div 2$	219	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
9	$7 \div 2$	3	1	9	$6 \div 2$	3	0
10	$3 \div 2$	1	1	10	$3 \div 2$	1	1
11	$1 \div 2$	0	1	11	$1 \div 2$	0	1
Result: $1947_{10} = 11110011011_2$				Result: $1757_{10} = 11011011101_2$			

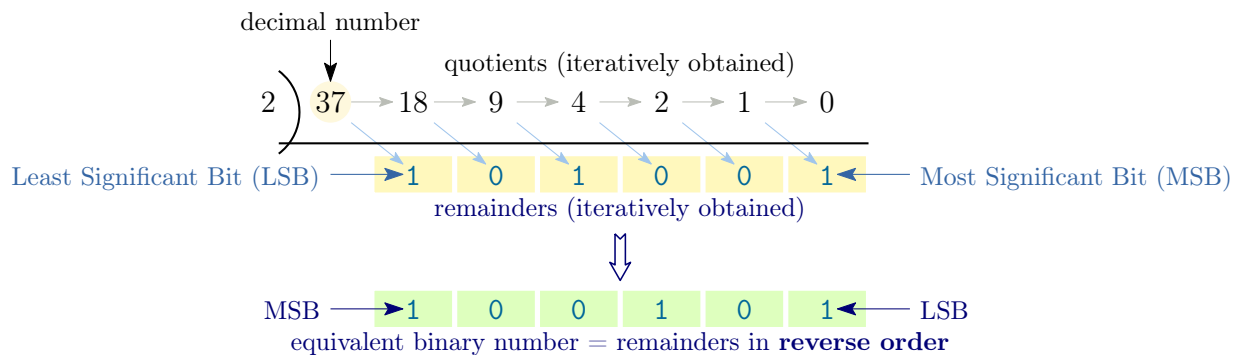


### 12.1.5 Conversion from decimal to binary

To convert a decimal number to binary, follow these steps:

1. Divide the decimal number by 2.
2. Record the remainder.
3. Update the decimal number to the quotient.
4. Repeat steps 1–3 until the quotient is 0.
5. Read the recorded remainders in reverse order for the binary representation.

Here is an example that shows why  $37_{10} = 2^0 + 2^2 + 2^5 = 100101_2$ .



Clearly, this method works for converting decimal numbers to other number systems, such as octal and hexadecimal, with minor adjustments. Instead of dividing by 2, you divide by 8 for octal and by 16 for hexadecimal. The process remains the same: record the remainders and read them in reverse order.

### 12.1.6 Unsigned binary number system

The **unsigned binary number system** is a method of representing non-negative integers using only two digits: 0 and 1. Each digit's position represents a power of 2, with the rightmost digit being  $2^0$ , the next  $2^1$ , and so on. For example, the binary number 1011 represents the decimal value  $1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 8 + 0 + 2 + 1 = 11$ . This system is widely used in electronics and computer science, as it aligns with the binary nature of electronic circuits. Here are some important points:

- Basic symbols: 0, 1
- Radix-2 positional number system.
- Example:  $10110 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 22$  in decimal.

### 12.1.7 Word of CPU

A **word** refers to the binary string the CPU can process in a single instruction. In a **32-bit** architecture, the CPU handles 32 bits simultaneously, allowing for efficient data manipulation. This word size determines the maximum value an integer can hold, which is  $2^{32} - 1 = 4,294,967,295$  for unsigned integers, the minimum being 0. Moreover, it affects memory addressing, allowing access to  $2^{32}$  unique memory locations, equating to 4 GB of RAM.

Understanding the word size is crucial for optimizing software and hardware performance in computing systems. Table 12.1 lists the 32-bit words for some decimal numbers.

If the words consist of just 3 or 4 bits each, their corresponding range of decimal values is as follows:

Unsigned 3-bit words							
Binary			Decimal	Binary			Decimal
0	0	0	0	1	0	0	4
0	0	1	1	1	0	1	5
0	1	0	2	1	1	0	6
0	1	1	3	1	1	1	7

Unsigned 4-bit words																	
Binary		Decimal	Binary		Decimal	Binary		Decimal	Binary		Decimal						
0	0	0	0	0	0	0	0	1	0	0	0	8	1	1	0	0	12
0	0	0	1	0	1	0	1	1	0	0	1	9	1	1	0	1	13
0	0	1	0	0	1	1	0	1	0	1	0	10	1	1	1	0	14
0	0	1	1	0	1	1	1	1	0	1	1	11	1	1	1	1	15

### 12.1.8 Signed number

We use the + and – symbols to indicate the sign of a decimal number. But, in a binary system, only 0 and 1 are available to encode **any information**. So, leftmost bit is used to denote the sign of a number. This bit is called the **sign bit**. There are three schemes for signed number representation in binary number system: Sign-magnitude, 1’s complement, 2’s complement, which are discussed below.

### 12.1.9 Sign-magnitude

In the sign-magnitude representation, the leftmost bit (which is the MSB in unsigned numbers) is the **sign bit**. This bit is 0 for positive numbers and 1 for negative numbers. The remaining bits represent the **magnitude** of the number. For example, in a 4-bit word:  $[b_3 | b_2 | b_1 | b_0]$ ,  $b_3$  is the sign bit, while the other three bits define the **magnitude**. Key aspects of this representation include:

**Table 12.1:** 32-bit words for some decimal numbers.

Decimal	Binary																																
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
1947	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	1	1	0	1	1
1948	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	1	1	1	0	0	
4000000000	1	1	1	0	1	1	1	0	0	1	1	0	1	0	1	1	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	
4000000010	1	1	1	0	1	1	1	0	0	1	1	0	1	0	1	1	0	0	1	0	1	0	0	0	0	0	0	0	0	1	0	1	0
4294967295	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	

- Two representations of 0: +0 and -0.
- Range for an  $n$ -bit word:  $-(2^{n-1} - 1)$  to  $(2^{n-1} - 1)$ .

For example, with 4-bit words, the range is  $-7, \dots, -1, 0, +1, \dots, +7$ . In this case, both 0000 and 1000 represent 0. The full set is listed below.

Sign-magnitude of 4-bit words							
Binary	Decimal	Binary	Decimal	Binary	Decimal	Binary	Decimal
0 0 0 0	+0	0 1 0 0	+4	1 0 0 0	-0	1 1 0 0	-4
0 0 0 1	+1	0 1 0 1	+5	1 0 0 1	-1	1 1 0 1	-5
0 0 1 0	+2	0 1 1 0	+6	1 0 1 0	-2	1 1 1 0	-6
0 0 1 1	+3	0 1 1 1	+7	1 0 1 1	-3	1 1 1 1	-7

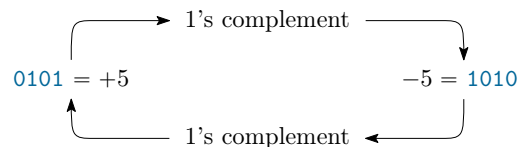
### 12.1.10 1's complement numeral

The **1's complement** of an integer  $k$  is obtained by toggling every bit of  $k$ . For example, for 4-bit words, the 1's complement of 0101 is 1010. In 1's complement representation, this yields  $-k$  when  $k \neq 0$ , and produces a separate representation for +0 and -0 when  $k = 0$ .

In the context of binary strings, **toggling** or **inverting** refers to flipping a bit from 0 to 1, or from 1 to 0. For example, in the binary string 1010, toggling the second bit changes it to 1110. Toggling can be achieved using the XOR (exclusive OR) operation. Recall that XOR of two bits is 1 if and only if they are unequal. So, when XORing a bit with 1, the bit is flipped, whereas XORing with 0 leaves the bit unchanged. This process is important in bitwise manipulation and digital logic circuits.

Key aspects of the 1's complement representation are outlined below. For simplicity, examples are taken from 4-bit words; however, these explanations apply to any word of at least two bits.

1. Positive numbers have  $\text{MSB} = 0$ , and negative numbers have  $\text{MSB} = 1$ .  
Example: 0101 is a positive number, whereas 1101 is a negative number.
2. As per the above convention, we have two representations of 0:  $\underbrace{0000 \dots 0}_{+0}$  and  $\underbrace{1000 \dots 0}_{-0}$ .
3. Since 1's complement is simply toggling of bits, it follows that the 1's complement of 1's complement of any number is the number itself, as illustrated below.



4. If  $k$  is a positive integer (i.e., its  $\text{MSB} = 0$ ), then its value is same as its sign-magnitude representation.  
Example: The value of 0101 is  $1 + 2^2 = 5$ .
5. If  $k$  is a negative integer (i.e., its  $\text{MSB} = 1$ ), then its magnitude is same as that of its 1's complement.  
Example: The magnitude of 1101 is given by the value of 0010, which is 2. Since the MSB of 1101 is 1, its signed value is  $-2$ .
6. Range for an  $n$ -bit word:  $-(2^{n-1} - 1)$  to  $(2^{n-1} - 1)$ , which is same as sign-magnitude representation.  
Example: With 4-bit words, the range is  $-7, \dots, -1, 0, +1, \dots, +7$ . In this case, both 0000 and 1000 represent 0. Notice its difference from sign-magnitude representation. The full set is listed below.

1's complement numerals of 4-bit words																				
Binary		Decimal	Binary		Decimal	Binary		Decimal												
0	0	0	0	0	+0	0	1	0	0	+4	1	0	0	0	-7	1	1	0	0	-3
0	0	0	1		+1	0	1	0	1	+5	1	0	0	1	-6	1	1	0	1	-2
0	0	1	0		+2	0	1	1	0	+6	1	0	1	0	-5	1	1	1	0	-1
0	0	1	1		+3	0	1	1	1	+7	1	0	1	1	-4	1	1	1	1	-0

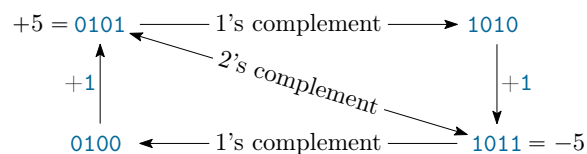
### 12.1.11 2's complement numeral

The 2's complement numeral system is the most widely used method for representing signed integers in modern computing. It simplifies binary arithmetic, particularly subtraction, by eliminating the need for separate hardware for subtraction operations. In this system, a negative number is represented by inverting all the bits of its positive counterpart and adding 1. One of the key advantages of 2's complement is that it has a unique representation for zero, unlike 1's complement, which has both +0 and -0. This consistency reduces the complexity in handling numerical data. Furthermore, addition and subtraction operations between positive and negative numbers become seamless, as the same binary addition process is used for both. The 2's complement numeral system is critical to understanding how computers efficiently perform arithmetic operations.

Key aspects of this representation include the following. For explanations, examples are taken from 4-bit words, as done in §12.1.10, but all these explanations apply for any words of at least two bits.

1. Only one representation of zero, e.g., 0000 in 4-bit word.
2. Positive numbers have  $\text{MSB} = 0$ , and negative numbers have  $\text{MSB} = 1$ .  
Example: 0101 is a positive number, whereas 1101 is a negative number.
3. For any integer  $k$ ,  $-k$  is obtained by first toggling every bit of  $k$ , no matter whether  $k < 0$ ,  $k = 0$ , or  $k > 0$ , and then adding 1 to it. The result is called the **2's complement** of  $k$ .

Example: The 2's complement of 0101 is  $1010 + 1 = 1011$ . Similarly, the 2's complement of 1011 is 0101.



4. The 2's complement of a negative number is a positive number, and vice versa.  
**Only exception:**  $1000 = -2^3 = -8$  whose 2's complement is itself and not 8.
5. A positive number ( $\text{MSB} = 0$ ) evaluates to its sign-magnitude or 1's complement.  
Example: 0101 has  $\text{MSB} = 0$ , so it is a positive number having the value +5.
6. For evaluating a negative number ( $\text{MSB} = 1$ ), compute its 2's complement to get its magnitude. Apply a negative sign to get its signed value.  
Example: 1011 is a negative number and its 2's complement is 0101, whose decimal value is 5; so the signed value of 1011 is -5.
7. Range for 4 bits:  $-8, \dots, 7$ .
8. Range for  $n$  bits:  $-2^{n-1}, \dots, (2^{n-1} - 1)$ .
9. Range of `int` (32 bits):  $-2147483648$  to  $2147483647$ .

2's complement numerals of 4-bit words																				
Binary		Decimal	Binary		Decimal	Binary		Decimal												
0	0	0	0	0	+0	0	1	0	0	+4	1	0	0	0	-8	1	1	0	0	-4
0	0	0	1		+1	0	1	0	1	+5	1	0	0	1	-7	1	1	0	1	-3
0	0	1	0		+2	0	1	1	0	+6	1	0	1	0	-6	1	1	1	0	-2
0	0	1	1		+3	0	1	1	1	+7	1	0	1	1	-5	1	1	1	1	-1

### 12.1.12 Word extension in 2's complement

To extend a word without altering the signed value of the stored number, append bits the same as its MSB to its left side. That is, add 0's to its left side if the number is positive, otherwise add 1's.

For example, if the number 0101 (which represents +5) is stored in a 4-bit word, its extension to an 8-bit word would be 0000 0101. Conversely, the extension of the 4-bit number 1101 (representing -3) to an 8-bit word results in 1111 1101. This method ensures that the signed values remain unchanged after the extension.

### 12.1.13 Carry-out and overflow in 2's complement

In the context of binary arithmetic, particularly in 2's complement representation, understanding carry-out and overflow is crucial for correctly interpreting results. They are two distinct concepts that arise during arithmetic operations.

**Carry-out** occurs when the result of an addition operation cannot be accommodated within the given number of bits. For example, in a 4-bit system, adding 1111 with 1000 will produce the 5-bit binary string 10111, in which the leftmost bit is the carry-out.

**Overflow** occurs during addition if and only if the two numbers have the same sign but their sum has the opposite sign. If two numbers have opposite signs, there is no overflow. For example, in a 4-bit system, adding 0110 (= +6) with 0100 (= +4) will produce the 4-bit binary string 1010, which is a negative number because its MSB is 1. You can check that 1010 evaluates to  $-0110 = -6$ .

The overflow and carry-out can occur independently. In unsigned numbers, carry-out is equivalent to overflow. But in general, carry-out tells you nothing about overflow. The possible cases with examples of 4-bit systems are discussed below.

#### No Carry-Out, No Overflow

	Word				Value	
	0	0	1	1	+3	
+	0	0	1	0	+2	
Sum:	0	1	0	1	+5	Correct

The result is within 4 bits, indicating that there is no carry-out from the most significant bit (MSB). There is no overflow since both summands and the sum are positive.

#### Overflow without Carry-Out

	Word				Value	
	0	1	1	1	+7	
+	0	0	1	1	+1	
Sum:	1	0	0	0	-8	Incorrect

No carry-out occurs because the sum fits within 4 bits. However, despite both operands being positive, MSB of the result indicates a negative number, leading to an overflow.

**Overflow with Carry-Out**

	Word				Value	
	1	0	0	1	-7	
+	1	0	1	0	-6	
Sum:	1	0	0	1	+3	Incorrect

The sum exceeds 4 bits, resulting in a carry-out. Although both operands are negative, the sum is positive, as indicated by the MSB. Therefore, overflow occurs.

**Carry-Out without Overflow**

	Word				Value	
	1	1	0	0	-4	
+	1	1	1	1	-1	
Sum:	1	0	0	1	-5	Correct

The sum exceeds 4 bits, resulting in a carry-out. However, since both operands are negative and the result is also negative, there is no overflow. In fact, this indicates that the 4-bit result is correct, as the 5th bit (carry-out) is ignored.

**12.1.14 10's complement**

The 2's complement numeral is nothing special. In fact, in general, we can use **radix-complement** numerals for any radix to represent signed numbers without using any sign symbol. Let's see how for radix 10 or decimal system, we can construct 10's complement numerals.

For simplicity, let's consider 3-digit words in decimal system. There are one thousand patterns (000 to 999) for this. In this system of 3-digit words, we interpret any integer  $k$  in the following way:

1. If the most significant digit is 0 to 4, then  $k$  is a usual positive number.  
Example: 341 is same as the usual decimal number 341.
2. If the most significant digit is from 5 to 9, then  $k$  is treated as a negative number.  
Example: 725 is a negative number with the actual value  $-(1000 - 725) = -275$ .
3.  $-k$  is obtained by  $1000 - k$ .  
Example:  $-725 = 1000 - 725 = 275$ .
4. The range of numbers represented in 3 digits is  $-10^3/2$  to  $+10^3/2 - 1$ , i.e.,  $-500$  to  $499$ .  
In general, for  $n$ -digit 10's complement numbers, the range is  $-10^n/2$  to  $+10^n/2 - 1$ .

**Word extension in 10's complement:** The idea of extending a word without altering the signed value of the stored number is similar to that in 2's complement. We need to add 0's to its left side if the number is positive, otherwise add 9's. For example, the 3-digit word 273 (which represents +273) should be extended to 000273 for its equivalent 6-digit word. On the contrary, the 3-digit word 673 (which represents -327) should be extended to 999673.

**Carry-out and overflow in 10's complement:** There will overflow problem here too, as it happened for 2's complement. Let's assume 3-digit words to see how it happens.

**No Carry-Out, No Overflow**

	Word			Value	
	1	2	7	+127	
+	2	0	5	+205	
Sum:	3	3	3	+332	Correct

The result is within 3 digits, indicating that there is no carry-out from the most significant digit (MSD). There is no overflow since both summands and the sum are positive.

**Overflow without Carry-Out**

	Word			Value	
	4	2	7	+427	
+	3	0	5	+305	
Sum:	7	3	2	-268	Incorrect

No carry-out occurs because the sum fits within 3 digits. However, despite both operands being positive, MSD (7) of the result indicates a negative number, leading to an overflow.

**Overflow with Carry-Out**

	Word			Value	
	7	2	7	-273	
+	6	0	5	-395	
Sum:	1	3	3	+332	Incorrect

The sum exceeds 3 digits, resulting in a carry-out. Although both operands are negative, the sum is positive, as indicated by the MSD (3). Therefore, overflow occurs.

**Carry-Out without Overflow**

	Word			Value	
	8	2	7	-173	
+	7	0	5	-295	
Sum:	1	5	3	-468	Correct

The sum exceeds 3 digits, resulting in a carry-out. However, since both operands are negative and the result is also negative, there is no overflow. In fact, this indicates that the 3-digit result is correct, as the 4th digit (carry-out) is ignored.

## 12.2 Decimal numbers: Standard floating-point representation

The decimal number system, also known as the base-10 or radix-10 system, is the usual standard for representing real numbers. We are introduced to it in childhood and understand how to write any real number, no matter how large or precise. However, representing, storing, and working with such numbers in a computer requires a standardized scientific format. This raises the question of efficient representation. In a computer, only the binary digits 0 and 1 are available to accomplish all tasks.

Let's begin by exploring how to create a standard representation for decimal numbers using only the digits 0 to 9, while intelligently accounting for the decimal point (i.e., the dot symbol) and the sign of the number. In the standard floating-point representation of real numbers, the position of the decimal point can "float", which is why it is called "floating point". The following example shows how the number 31415.9 can be expressed in various forms:

$$\begin{aligned}
 31415.9 &= 3141.59 \times 10^1 \\
 &= 314.159 \times 10^2 \\
 &= 31.4159 \times 10^3 \\
 &= 3.14159 \times 10^4 \quad \leftarrow \text{standard form} \\
 &\approx 0.31416 \times 10^5 \\
 &\approx 0.03142 \times 10^6 \\
 &\vdots
 \end{aligned}$$

In this representation, a real number is expressed in the form:

$$s = \text{sign} \quad e = \text{signed exponent} \quad m = \text{mantissa}$$

where,

$$\text{value of the real number} = \text{sign} \times \text{mantissa} \times \text{base}^{\text{exponent}} = s \times m \times 10^e.$$

- **sign**  $s$  is either  $+1$  or  $-1$ .
- **mantissa**  $m$  (also called ‘significand’) is an unsigned number in the interval  $[1, 10)$ .
- **exponent**  $e$  is a signed integer.

In the previous example, only  $3.14159 \times 10^4$  is the standard or normalized form, with sign  $s = +1$ , mantissa  $m = 3.14159$ , and exponent  $e = +4$ . For the number  $-31415.9$ ,  $s$  will be  $-1$ , while  $m$  and  $e$  remain the same.

Now, suppose a word of 10 digits can be used to store any real number, with the following convention:

- First digit is for the sign of the number (0 if positive and 1 if negative).
- Second digit is for the sign of the exponent (0 if positive and 1 if negative).
- Third and fourth digits are for the unsigned value of the exponent.
- Last six digits are for the mantissa.

Then, the word storing the number  $3.14159 \times 10^4$  will be:

0	0	0	4	3	1	4	1	5	9
---	---	---	---	---	---	---	---	---	---

Note that since the decimal point (i.e. the dot sign) is implicit, because in the standard representation,  $3$  is the one and only digit before the decimal point and the rest, i.e.,  $14159$ , follows after it. That is, the decimal point is located as follows:

0	0	0	4	3	.	1	4	1	5	9
---	---	---	---	---	---	---	---	---	---	---

Similarly, for the number  $-3.14159 \times 10^4$ , we get

1	0	0	4	3	1	4	1	5	9
---	---	---	---	---	---	---	---	---	---

For  $-3.14159 \times 10^{-4}$ , we get

1	1	0	4	3	1	4	1	5	9
---	---	---	---	---	---	---	---	---	---

Since the mantissa can accommodate 6 digits, the extra digits in a real number will be lost in this 10-digit representation. For example, if we need to store  $3.1415927 \times 10^4$ , it will be

0	0	0	4	3	1	4	1	5	9
---	---	---	---	---	---	---	---	---	---

where, as you see, the last two digits (2 and 7) could not be stored. This example shows we have to compromise with the precision for a large mantissa.

### 12.2.1 Floating-point representation versus fixed-point representation

Fixed-point representation and floating-point representation are two ways to represent numbers, each with distinct characteristics.

**Example:** In a fixed-point system of 10-digit word with three decimal places, the number 31415.9 would be stored as **0031415900**. Whenever needed, the system multiplies **0031415900** by  $10^{-3}$  to get 31415.9. Thus, when stored, the number appears as **0031415900**, and when interpreted by the system, it is understood to represent 31415.900.

Fixed-point representation has a limited range, as the position of the decimal point is fixed. In fixed-point representation, the components are just the sign and the mantissa, and there is no exponent. Floating-point representation has a much wider range because the decimal point can "float" based on the exponent.



**Example:** Suppose we have to represent only non-negative real numbers that we have to store in 6-digit words. Suppose that three digits are after the decimal point in fixed-point representation. Then, the range of numbers in fixed-point representation will be:

$$000.000, 000.001, 000.002, \dots, 000.999, 001.000, 001.001, \dots, 999.999.$$

For floating-point representation, we don't require the sign bit because it's given that all are non-negative numbers. Let's reserve four digits for the mantissa and two for the exponent, assuming that the exponent is non-negative. Then, the range of numbers in floating-point representation will be:

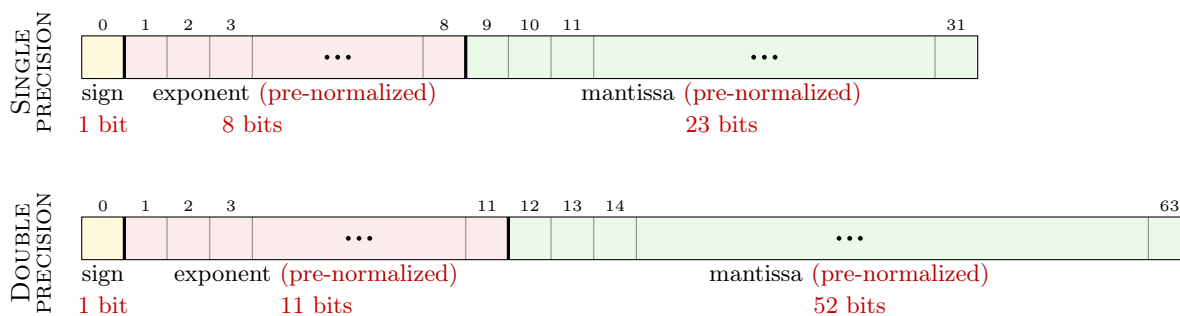
The mantissa can take values from 0000 to 9999, while the exponent  $e$  can take values from 00 to 99. Suppose that  $e \leq 49$  represents a positive exponent with the value same as  $e$ , while  $e \geq 50$  represents a negative exponent with the value  $-100 + e$ . That is,  $e$  ranges from  $-50$  to  $+49$ .

This means that the smallest non-zero number represented is  $0.001 \times 10^{-50}$ , and the largest number represented is  $9.999 \times 10^{49}$ .

## 12.3 Binary numbers: Normalized floating-point representation

The IEEE 754 Floating-Point Standard<sup>1</sup> is widely used for representing real numbers in a **normalized form** of binary number system. It defines how real numbers are stored using a combination of a sign bit, a signed exponent, and a mantissa, enabling representation of a vast range of values.

The IEEE standard supports two primary formats: single precision (32 bits) and double precision (64 bits). These are denoted by `float` and `long double` respectively, in the C language. It also specifies rules for rounding, special values like `inf` (infinity) and `nan` (not-a-number), and handling exceptions, contributing to reliable numerical computations in different applications.



### 12.3.1 Data classes and normalization

The different classes or categories of floating-point numbers are determined by the values of the exponent and the mantissa, as shown in Table 12.2. The numbers we typically work with fall under the class of normalized numbers. In addition to this class, there are four other categories, as outlined below:

1. Normalized numbers
2. Zeros (`+0` and `-0`)
3. Subnormal (denormal) numbers
4. Infinity (`inf`)
5. Not-a-Number (`nan`)

<sup>1</sup>IEEE stands for **Institute of Electrical and Electronics Engineers**.

**Table 12.2:** Classes of floating-point numbers.

Precision			
Single	Double		
Exponent (pre-normalized)		Mantissa	Data Class
0	0	0	$\pm 0$
0	0	$\neq 0$	$\pm$ subnormal
1 – 254	1 – 2046	anything	$\pm$ normalized
255	2047	0	$\pm\infty$
255	2047	$\neq 0$	nan

Let us understand the composition of the above classes for single-precision floating-point numbers. The composition for double precision will be similar and summarized at the end. Following are the rules of composition for single precision:

- **Special exponents:** 00000000 (0) and 11111111 (255) are reserved for  $\pm 0$ , nan, and inf.
- **Sign:**  $s = 0$  means positive,  $s = 1$  means negative.
- **Pre-normalized exponent:** For normalized numbers,  $e$  ranges from 00000001 (1) to 11111110 (254).
- **Normalized exponent:**  $e - 127$ , ranging from  $1 - 127 = -126$  to  $254 - 127 = 127$ . Here, 127 is referred to as **bias** and used for normalization.
- **Normalized range:**  $2^{-126}$  to  $2^{127}$ .
- **Normalized mantissa:** The mantissa  $m$ , normalized to  $1.m$ , ranges from 1.0 to  $2.0 - 2^{-23}$ . For example, if  $m = \underbrace{11000\dots0}_{\text{two 1s, rest 0}}$ , then its normalized value will be exactly  $1 + \frac{1}{2} + \frac{1}{4} = 1.75$ .
- **Normalized number:**  $(-1)^s \times 1.m \times 2^{e-127}$

**Subnormal (or denormal numbers)** represent a specific class of values in floating-point arithmetic that can approximate values closer to zero than the smallest normal numbers. In binary, a subnormal number has an exponent of all zeros (the smallest possible exponent), while its mantissa is non-zero. These numbers fill the gap between zero and the smallest positive normal number, which is essential for ensuring numerical stability in computations and effectively handling **underflow** conditions. Subnormal numbers facilitate gradual underflow, allowing values to decrease smoothly as they approach zero instead of abruptly becoming zero. This is vital for maintaining precision in numerical calculations, particularly in scientific and geometric computing.

You can write code to define, print, and compare a subnormal number with zero, normal numbers, or another subnormal. The following code defines and prints a subnormal number's value.

```

1 #include <stdio.h>
2 int main() {
3     double x = 1.0e-40; // A subnormal number with higher precision
4     printf("%e\n", x); // This prints 1.000000e-40
5     return 0;
6 }
```

### 12.3.2 Examples

#### Example 1 (binary to decimal):

Suppose we are given a single-precision number:

11011011 00110000 00000000 00000000

It can be broken into three components (1-bit sign, 8-bit exponent, and 23-bit mantissa) as follows:

1	1 0 1 1 0 1 1 0	0 1 1 0
---	-----------------	---

So, here is how the normalization is done:

- Sign =  $(-1)^1 = -1$ .
- Pre-normalized exponent is  $10110110 = 2^7 + 2^5 + 2^4 + 2^2 + 2 = 182$ .  
So, the normalized exponent is  $182 - \text{bias} = 182 - 127 = 55$ .
- Pre-normalized mantissa is  $011000\dots 0$ .  
After being normalized, it becomes  $1.011000\dots 0 = 1 + 2^{-2} + 2^{-3} = 1.375$ .

Thus, the signed value of the given single-precision number in decimal number system is  $-1.375 \times 2^{55}$ .

#### Example 2 (decimal to exact binary):

Consider the decimal number  $x = 105.625$ .

Since  $105 = 2^6 + 2^5 + 2^3 + 2^0 = 1101001$  and  $0.625 = 2^{-1} + 2^{-3} = 0.101$ , the single-precision floating-point representation of  $x$  is

$$\begin{aligned}
 &+ 1101001.101 \\
 = &+ 1.101001101 \times 2^6 \\
 = &+ 1.101001101 \times 2^{133-127} \\
 = &+ 1.101001101 \times 2^{10000101-01111111} \\
 = &0 \mid 1000 \ 0101 \mid 101 \ 0011 \ 0100 \ 0000 \ 0000 \ 0000
 \end{aligned}$$

#### Example 3 (decimal to approximate binary):

Consider  $+2.7$ . Its single-precision floating-point representation is

$$\begin{aligned}
 &+ 10.10 \ 1100 \ 1100 \ 1100 \dots \text{(repeating } 1100) \\
 = &+ 1.010 \ 1100 \ 1100 \dots \times 2^1 \\
 = &+ 1.010 \ 1100 \ 1100 \dots \times 2^{128-127} \\
 = &+ 1.010 \ 1100 \dots \times 2^{10000000-01111111} \\
 = &0 \mid 1000 \ 0000 \mid 010 \ 1100 \ 1100 \ 1100 \ 1100 \ 1100
 \end{aligned}$$

Observe that the exact value of 2.7 or 0.7 or 0.2 does not exist in binary number system. Hence, we get

$$2.7 = 10.10 \underbrace{1100 \ 1100 \ 1100 \dots}_{\text{infinite sequence of } 1100} = 10.10(1100)^\infty.$$

In fact, it is not difficult to prove that **a finite-precision decimal number has a finite-length binary representation if and only if the denominator of its corresponding fraction (in reduced form) is a power of 2.**

For 2.7, the corresponding fraction is  $\frac{27}{10}$ , whose denominator is not a power of 2, which implies it has not a finite-length binary representation.

On the contrary,  $2.5 = \frac{25}{10} = \frac{5}{2}$  and  $2.25 = \frac{225}{100} = \frac{9}{4}$  have their denominators 2 and 4 respectively, which are powers of 2, and hence they have finite-length binary representations.

### 12.3.3 nan

There are two types of **nans**: **quiet nan** and **signaling nan**. A quiet **nan** is produced by operations that do not trigger any exceptions. It generally represents an indeterminate form or the result of operations that cannot yield a valid number without causing a disruption. For example, dividing zero by zero ( $0.0/0.0$ ) or performing an operation that involves infinity (e.g.,  $\pm\infty/\pm\infty$ ) can yield a quiet **nan**.

A signaling **nan** is intended to signal an error or exception during computations. When a signaling **nan** is encountered in an operation, it typically raises a trap or generates a signal to alert the system that an invalid operation has occurred. For instance, attempting to perform a mathematical operation involving an invalid input (like  $\sqrt{-1.0}$ ) can produce a signaling **nan**.

In summary, quiet **nans** do not raise exceptions, while signaling **nans** are meant to indicate an error condition that should be addressed. Here is a code:

```

1  #include <stdio.h>
2  #include <math.h>
3
4  int main(){ // Each printf prints -nan (gcc: Ubuntu 9.4.0-1ubuntu1~20.04.2 9.4.0)
5      printf("0.0/0.0: %f\n", 0.0/0.0);
6      printf("inf/inf: %f\n", (1.0/0.0)/(1.0/0.0));
7      printf("0.0*inf: %f\n", 0.0*(1.0/0.0));
8      printf("-inf + inf: %f\n", (-1.0/0.0) + (1.0/0.0));
9      printf("sqrt(-1.0): %f\n", sqrt(-1.0));
10     printf("log(-1.0): %f\n", log(-1.0));
11     return 0;
12 }
```

### 12.3.4 About inf, +0, -0

Here are some key points:

1. **Infinity:** Without the sign bit, **inf** means:

1111 1111000 0000 0000 0000 0000 0000.

Without the sign bit, the largest normalized number (as unsigned integer) means:

1111 1110111 1111 1111 1111 1111 1111.

Hence, if we treat the largest normalized number as an **int** and add 1 to it, then we get **inf**.

2. **Using infinity:** **inf** can be used in computation, e.g., to compute  $\tan^{-1} \infty$ .
3. **Zeros:** There are two zeros, **+0** and **-0**, in IEEE representation, but they are treated as equal.

## 12.4 Solved problems

1. **[Positive integer to unsigned binary (recursive)]** Given a positive integer `n` as input (as `unsigned int`), write a recursive function to print its unsigned binary representation. Assume that we have 32-bit words.

```

1 #include <stdio.h>
2
3 void printBinary_Recursive(unsigned int n){
4     if (n > 1)
5         printBinary_Recursive(n >> 1); // Recursively shift right
6     printf("%d", n & 1); // Print the least significant bit
7 }
8
9 int main(){
10     unsigned int n;
11     printf("Enter a positive integer: ");
12     scanf("%u", &n);
13     printBinary_Recursive(n);
14     printf("\n");
15     return 0;
16 }

```

2. **[Integer to sign-magnitude (recursive)]** Given an integer `n` as input in the interval  $[-127, 127]$ , write a recursive function to print its sign-magnitude binary representation. Assume that we have 8-bit words.

```

1 #include <stdio.h>
2
3 // Recursive function to print the magnitude part (7 bits)
4 void signedMag_Rec(unsigned int n, int bits) {
5     if (bits > 1)
6         signedMag_Rec(n >> 1, bits - 1); // Recursively shift right
7     printf("%d", n & 1); // Print the least significant bit
8 }
9
10 int main() {
11     int n;
12     printf("Enter an integer (-127 to 127): ");
13     scanf("%d", &n);
14
15     if (n < 0) {
16         printf("Sign-magnitude binary representation: 1 ");
17         signedMag_Rec(-n, 7);
18     } else {
19         printf("Sign-magnitude binary representation: 0 ");
20         signedMag_Rec(n, 7);
21     }
22
23     printf("\n");
24
25     return 0;
26 }

```

3. **[Real to binary (finite precision)]** Given a positive real number  $x$  less than 1, scan it using the format specifier "%Lf" (long double) and compute its floating-point representation in binary up to  $n$  bits of precision after the decimal point. The value of  $n$  is also provided by the user.

As the number  $x$  may not have a finite binary representation, its floating-point binary value may not be exact when printed, as it will be truncated after  $n$  decimal places.

```

1 #include <stdio.h>
2
3 void real2Binary(long double x, int n) {
4     printf("0.");
5
6     for (int i = 1; i <= n; i++) { // Compute i-th bit
7         x = x * 2; // Multiply by 2
8         if (x != 0){
9             if (x >= 1.0) {
10                printf("1");
11                x = x - 1.0; // Subtract 1 to get the new fractional part
12            }
13            else
14                printf("0");
15            // printf( " (%0.50Lf)\n", x); // print to see why it's approx
16        }
17        else break;
18    }
19    printf("\n");
20 }
21
22 int main() {
23     long double x;
24     int n;
25     printf("Enter x and n: ");
26     scanf("%Lf%d", &x, &n);
27     real2Binary(x, n);
28     return 0;
29 }

```

Output for some values of  $x$  and  $n$  is given below. Observe that for  $n = 100$ , not all 100 bits of 0.7 are printed. This occurs because the value of  $x$  in the code becomes zero before reaching the 100-th bit.

In each iteration,  $x$  is doubled and decremented by 1 in some cases, ensuring it remains less than 1. Since 0.7 does not have a finite-length binary representation but is stored in a finite-length space, it eventually becomes zero as bits are left-shifted one position each time it is doubled. This results in an approximate output.

#### Output:

```

Enter x and n: .7 50
0.10110011001100110011001100110011001100110011001100110011001100
Enter x and n: .7 100
0.1011001100110011001100110011001100110011001100110011001100110011
Enter x and n: .75 50
0.11

```



- ♣ 5. **[Fraction to binary (infinite precision)]** Extend your previous code to print the floating-point binary value of  $a/b$  as a sequence of non-periodic bits, if any, followed by the smallest periodic sequence without repeating it. Enclose the periodic sequence within brackets. For example, represent  $1/3$  as  $0.(01)$  and  $7/10$  as  $0.1(0110)$ .

```

1  #include <stdio.h>
2
3  int checkPeriodic(int p[], int q[], int j){
4      int i, r, s, u, v, periodic = 0;
5
6      for(r=0; r<j-1 && !periodic; r++)
7          for(s=r+1; s<j && !periodic; s++)
8              if(p[r] == p[s] && q[r] == q[s])
9                  u = r, v = s, periodic = 1;
10
11     if(periodic){
12         for(i = 0; i < u; i++)
13             printf("%c", q[i]);
14         printf("(");
15         for(; i < v; i++){
16             printf("%c", q[i]);
17             if((i+1)%50 == 0) // It's long sequence, so print in the new line
18                 printf("\n ");
19         }
20         printf(")\n");
21     }
22     return 1;
23 }
24
25
26 void fraction2binaryPeriodic(int a, int b) {
27     char bit;
28     int j = 0, n = 2*b;
29     int p[n], q[n]; // p[] stores the changing numerator, q[] stores the bits
30     printf("0.");
31
32     for (int i = 1; i <= n; i++) { // Compute i-th bit
33         a = a * 2;
34         if (a != 0){
35             if (a >= b) {
36                 q[j] = bit = '1';
37                 a = a - b; // Get the new proper fraction
38             }
39             else
40                 q[j] = bit = '0';
41             p[j] = a;
42             j++;
43             if (checkPeriodic(p, q, j))
44                 break;
45         }
46         else
47             break;
48     }
49 }

```



```

50 |
51 | int main() {
52 |     int a, b;
53 |     printf("Enter a/b: ");
54 |     scanf("%d/%d", &a, &b);
55 |     fraction2binaryPeriodic(a, b);
56 |     return 0;
57 | }

```

**Output:**

```

Enter a/b: 1/3
0.(01)
Enter a/b: 7/10
0.1(0110)
Enter a/b: 7/100
0.00(01000111101011100001)
Enter a/b: 7/10000
0.0000(000000101101111000000001101000110110111000101
11010110001110001000011001011001010010101111010011
11000011011000010001001101000000010011101010010010
10100011000001010101001100100110000101111100000110
11110110100101000100011001110011100000011101011111
0110111111010010000111111110010111001001000111010
00101001110001110111100110100110101101010000101100
0011110010011110111011001011111101100010101101101
01011100111110101010110011011001111010000011111001
00001001011010111011100110001100011111100010100000
1001)
Enter a/b: 123/321
0.(01100010000101111110110011011100000111001011010111
01010011101111010000001001100100011111000110100101
000101)

```

To understand the mathematical reasoning of the above code, let's prove the following statement: **The binary floating-point representation of any fraction  $a/b$ , where  $0 < a < b$ , is either of finite length or eventually periodic.**

The representation is finite if and only if  $b$  is a power of 2. This is because  $a/b$  can be expressed as a sum of unit fractions of the form  $1/2^k$  only when  $b$  is a power of 2.

Now, let's prove that in the infinite case, the representation will be periodic. The division algorithm used to compute the binary representation of  $a/b$  involves repeated subtraction and multiplication by 2. Since there are at most  $b - 1$  nonzero remainders when dividing by  $b$ , a particular remainder with a specific bit value will eventually recur with the same bit value after at most  $2(b - 2) + 1$  iterations.\* This recurrence leads to the same sequence of bits being generated thereafter. Consequently, once the non-finite part begins, it will repeat periodically.

\*In binary representation, each division step produces 0 or 1. Thus, any remainder  $r$  will repeat after at most  $b - 2$  iterations, though its corresponding bit may differ. However, among three consecutive occurrences of  $r$ , at least two will correspond to the same bit. In the worst-case scenario, these two occurrences will be the first and third instances of  $r$ , with at most  $2(b - 2) + 1$  remainders between them, which include other remainders as well as the second occurrence of  $r$ .

## 12.5 Exercise problems

1. **[Normalized representation]** Given a positive real number  $x$  as input, write an iterative function to print its normalized representation. Both the input and the output are in the decimal number system. Don't use the math library.

Examples:  $3.14159 \rightarrow 3.141590 \text{ e}(0)$  |  $314.159 \rightarrow 3.141590 \text{ e}(2)$  |  $0.00314259 \rightarrow 3.142590 \text{ e}(-3)$

2. **[Positive integer to binary (iterative)]** Given a positive integer  $n$  as input (as `unsigned int`), write an iterative function to print its unsigned binary representation. Assume that we have 32-bit words.
3. **[Integer to sign-magnitude (iterative)]** Given an integer  $n$  as input in the interval  $[-127, 127]$ , write an iterative function to print its sign-magnitude binary representation. Assume that we have 8-bit words.
4. **[Integer to 1's complement (iterative)]** Given an integer  $n$  as input in the interval  $[-127, 127]$ , write an iterative function to print its 1's complement. Since `00000000` and `11111111` are both 0 in 1's complement, printing either of the two is fine when  $n$  is 0. Assume that we have 8-bit words.
5. **[Positive integer to hex (iterative)]** Given a positive integer  $n$  as input, write an iterative function to print its unsigned hexadecimal representation.
6. **[Whether exact representation exists]** Given a positive real number  $x$  less than 1, scan it using the format specifier `"%Lf"` and check whether its exact floating-point representation exists. For example, 0.34375 has an exact binary representation (`0.01011`), but 0.3 has not.

Refer to the fact in §12.3.2: a finite-precision decimal number has a finite-length binary representation if and only if the denominator of its corresponding fraction (in reduced form) is a power of 2.

- ♣ 7. **[Real to hexadecimal (finite precision)]** Given a positive real number  $x$  less than 1, scan it using the format specifier `"%Lf"` (`long double`) and compute its floating-point representation in hexadecimal with  $n$  digits of precision after the decimal point. The value of  $n$  is also provided by the user.

For example, if  $x = 0.7$  and  $n = 10$ , your code should print `0.B333333333`. If  $n = 5$ , then it should print `0.B3333`. As shown earlier, the decimal number  $x = 0.7$  has no finite-precision binary value, so its floating-point hexadecimal representation is truncated after  $n$  digits, with the digit `3` being recurring.

On the contrary, if  $x = 0.34375$ , which has a finite-precision binary or hexadecimal value, it should print `0.58` for any  $n \geq 2$ .

- ♣ 8. **[Fraction to hexadecimal (finite precision)]** Given any positive proper fraction  $a/b$  (not necessarily in the reduced form), scan it using the format specifier `"%d/%d"` and compute its floating-point representation in hexadecimal system up to  $n$  bits of precision. The value  $n$  is provided as input by the user.

As the number  $a/b$  may not have a finite binary representation, its floating-point hex value may not be exact when printed, as it will be truncated after  $n$  decimal places.

# 13 | Structures

A **structure** is a useful construct for grouping logically related data items, regardless of whether they are of the same or different types. Since it mirrors the nature of the group it represents, it is called a "structure".

For example, a **student** record might include a **name** (string), **roll number** (string), and **marks** (integer), all encapsulated within one entity. All the three datatypes here are not same, we can define the **student** as a structure. This approach simplifies managing multiple student records, especially in large arrays.

**Structures** provide a natural way to organize related data, irrespective of type. They can be used to define larger structures. Hence, the significance of **structures** extends beyond simple cases like students. For example, we can define a 2D **point** as a structure with  $x$  and  $y$  coordinates. It can subsequently be used to form geometric shapes like **line segments**, **circles**, and **polygons**, which require points for their definitions. For more advanced designs, take the example of defining a **vehicle** structure for a video game. A vehicle might be composed of several parts, such as an **engine**, **wheels**, and **chassis**. Each part could be its own **structure** containing different fields, and in turn, these fields could themselves be **structures**. This modular design naturally leads to **hierarchical structures**, ensuring all components work together in harmony and synchrony.

## 13.1 Structure declaration

Let's now see how a **student** can be defined as a structure containing **name** (string), **rollNum** (string), and **marks** (integer). Here is one of the possible ways of defining it:

```
struct student {
    char name[30];
    char rollNum[12];
    int marks;
};
```

With the above declaration, **student** becomes a **struct** datatype. You can declare variables with the type of a structure after declaring the structure. For example, after declaring the structure **student** as shown above, you can declare two variables **s1** and **s2** of datatype **struct student**, as follows:

```
struct student s1, s2;
```

Then, you can fill up the individual fields of the variables **s1** and **s2** by scanning input data or by direct assignment. For example, you can assign the fields of **s1** as follows:

```
strcpy(s1.name, "Raman");
strcpy(s1.rollNum, "24AB001");
s1.marks = 95;
```

## 13.2 Structure variable declaration

A variable of structure type can be defined in either of the following ways, depending on requirements.

Separately	Together (with reuse option)	Together (without reuse)
<pre>struct student {     char name[30];     char rollNum[12];     int marks; };  struct student s1, s2;</pre>	<pre>struct student {     char name[30];     char rollNum[12];     int marks; } s1, s2;</pre>	<pre>struct {     char name[30];     char rollNum[12];     int marks; } s1, s2;</pre>
<p><code>struct student</code> can be reused. For example, somewhere later in the code, you can write:</p> <pre>struct student s3, s4;</pre>	<p><code>struct student</code> can be reused later, as the first one.</p>	<p><code>struct</code> cannot be reused later because it has no name. In the previous two cases, it has the name <code>student</code>.</p>

## 13.3 The typedef construct

The `typedef` construct can be used to give new names to (existing) datatypes in C. For example, to define a new datatype named `speed` (say, to denote speed in kilometers per hour), we can write:

```
typedef float speed;
speed u = 0, v = 42.50;
```

Some more examples of typedef are:

```
typedef int intarray[50];
intarray A,      // A is an array of 50 integers
        B[20], // B is an array of 20 arrays of 50 integers (a 20 x 50 array)
        *C;    // C is a pointer to an array of 50 integers

typedef int *intptr;
intptr p,      // p is an int pointer
        q[10], // q is an array of 10 int pointers
        *r;    // r is a pointer to an int pointer
```

The above declaration styles are usually not practiced in programming but given here to show how things can be expressed in complicated ways. They are basically equivalent to:

```
int A[50], B[20][50], (*C)[50];
int *p, *q[10], **r;
```

## 13.4 Structures and typedef

The `typedef` keyword in C allows you to create an alias for a **structure**, making it easier to declare variables of that structure type. By combining `typedef` with **structures**, we can simplify code, improve readability, and avoid writing `struct` repeatedly when defining variables. Here are a couple of examples:

Without typedef	With typedef
<pre>struct student {     char name[30];     char rollNum[12];     int marks; };  struct student s1, s2;</pre>	<pre>typedef struct {     char name[30];     char rollNum[12];     int marks; } student;  student s1, s2;</pre>
Here <code>struct student</code> is a new datatype.	Here <code>student</code> is a new datatype. It has be addressed just by its name <code>student</code> .

Without typedef	With typedef
<pre>struct complex{     float real;     float imag; };  struct complex a, b;</pre>	<pre>typedef struct {     float real;     float imag; } complex;  complex a, b;</pre>
Here <code>struct complex</code> is a new datatype.	Here <code>complex</code> is a new datatype. It has be addressed just by its name <code>complex</code> .

## 13.5 Operations with structure

In C, structures enable efficient data management by grouping related variables. In this section, we shall see various operations that can be performed with structures, including initialization, assignment, and manipulation of structure members, enhancing the organization and functionality of data handling.

### 13.5.1 Accessing the members: dot operator

The members or fields of a structure are accessed individually, as separate entities. In the following example, the `x` and `y` coordinates of two points `p` and `q` are assigned individual values. To do this, the `x` and `y` coordinates of `p` are denoted by `p.x` and `p.y`, respectively, and similarly for `q`.

```
typedef struct{
    float x, y;
} point;

point p, q;
p.x = p.y = 0;
q.x = 3.5, q.y = 1.75.
```

The **dot operator** (`.`) is used to access members of a structure. There is another operator, denoted by `->` and referred to as **arrow operator**, which we shall see when dealing with pointer to structure.

### 13.5.2 Structure initialization

Structure variables may be initialized following similar rules as array. The values are provided within braces separated by commas. Here is an example that shows how two points are initialized during declaration:

```
typedef struct point{
    float x, y;
};

point p = {0, 0}, q = {3.5, 1.75};
```

Suppose now you want to compute the distance `d` between `p` and `q`. You can write:

```
float d = sqrt((p.x - q.x)*(p.x - q.x) + (p.y - q.y)*(p.y - q.y));
```

### 13.6 Assignment of structure variables

A structure variable can be directly assigned to another variable of the same structure. By this, all the individual members of the former get copied to the latter. Here is an example:

```
typedef struct point{
    float x, y;
};

point p = {0, 0};
point q = p;
```

However, two structure variables cannot be compared with each other. For example, this is not allowed:

```
if (p == q) // This will give compilation error
    printf("p and q are equal");
```

### 13.7 Array inside structure

A positive thing about structure is that although an array cannot be copied directly to another array, a structure variable can be copied directly to another variable of the same structure, even if they contain arrays. For example, this is not allowed:

```
int a[5] = {10, 20, 30, 40, 50};
int b[5];
b = a;    // This will give compilation error
```

However, this is allowed:

```
struct list {
    int x[5];
};
struct list a, b;

a.x[0] = 10; a.x[1] = 20; a.x[2] = 30; a.x[3] = 40; a.x[4] = 50;
b = a;    // This is okay
```

## 13.8 Size of a structure

Consider the following code:

```
struct student {
    char name[35];
    char rollNum[12];
    int marks;
} s;
```

Calculation shows a total space of  $35 + 12 + 4 = 51$  bytes is needed to store all the members of `struct student`. However, in practice, `sizeof(struct student)` or `sizeof(s)` may yield 52 or 56 (the nearest larger multiple of 4 or 8). The actual value depends on the computer architecture and the compiler, and it arises due to memory alignment and padding. To enhance access speed, the compiler aligns the structure members in memory to certain byte boundaries. In this case, the integer `marks` requires alignment, leading to padding that increases the overall size of the structure to the nearest multiple of the architecture's alignment requirement.

Now, consider the following variation of the above example:

```
struct student {
    char *name;
    char rollNum[12];
    int marks;
} s;
```

Assuming 64-bit addresses, the variable `name` requires 8 bytes to store a 64-bit pointer. The variable `rollNum` takes up 12 bytes, and `marks` occupies 4 bytes.

At first glance, it might seem that `sizeof(struct student)` or `sizeof(s)` would be  $8 + 12 + 4 = 24$  bytes. However, due to memory alignment and padding, the actual size of the structure may be larger, e.g.,  $8 + 16 + 8 = 32$  bytes. It's important to note that this size is independent of how much memory you allocate using `malloc` for `s.name`.

## 13.9 Array of structure

Structure is particularly useful when working with an array in which the elements are not simply a basic datatype but a combination of various fields or members. For example, we may need an array of the `student` structure that will contain a large number of students. Here is an example that shows how we can allocate an array `a[]` of 100 students:

```
typedef struct {
    char name[30];
    char rollNum[12];
    int marks;
} student;

student a[100];
```

Here is a simple code that scans the details of each student, writes them to the array `a[]`, and then prints all the records.



```
1 #include <stdio.h>
2 #include <string.h>
3
4 // Define the student structure
5 typedef struct {
6     char name[30], rollNum[12];
7     int marks;
8 } student;
9
10 int main() {
11     student a[100];
12     int n, i;
13
14     printf("Enter the number of students: ");
15     scanf("%d", &n);
16
17     for (i = 0; i < n; i++) {
18         printf("Enter details for student %d:\n", i + 1);
19
20         printf("Name: ");
21         scanf("%s", a[i].name);
22         printf("Roll Number: ");
23         scanf("%s", a[i].rollNum);
24         printf("Marks: ");
25         scanf("%d", &a[i].marks);
26     }
27
28     printf("\nStudent Records:\n");
29     for (i = 0; i < n; i++) {
30         printf("Student %d:\n", i + 1);
31         printf("Name: %s\n", a[i].name);
32         printf("Roll Number: %s\n", a[i].rollNum);
33         printf("Marks: %d\n\n", a[i].marks);
34     }
35
36     return 0;
37 }
```

### 13.10 Nested structure

In C programming, structures can be nested, allowing the creation of complex data types that reflect real-world entities more accurately. By defining a structure within another structure, developers can model intricate relationships and hierarchies, enabling better organization of data and improving code readability.

Here is an example that shows how a structure named `triangle` is declared using another structure named `point`:

```
typedef struct {
    float x, y;
} point;
```



```
typedef struct {
    point v[3]; // three vertices
    float area;
} triangle;
```

Suppose we need to scan the coordinates of the vertices of a triangle `t` and compute its area. Then, we can write as follows:

```
triangle t;

for (int i = 0; i < 3; i++) {
    printf("Enter the coordinates of vertex %d (x y): ", i + 1);
    scanf("%f %f", &t.v[i].x, &t.v[i].y);
}

t.area = 0.5 * (t.v[0].x * (t.v[1].y - t.v[2].y) +
               t.v[1].x * (t.v[2].y - t.v[0].y) +
               t.v[2].x * (t.v[0].y - t.v[1].y));
t.area = (t.area < 0) ? -t.area : t.area;
printf("The area of the triangle is: %.3f\n", t.area);
```

### 13.11 Self-referencing structure

Structure containment cannot be recursive, whether directly or indirectly. For example, `struct student` cannot contain `struct student`. Similarly, if `struct student` contains `struct hostel`, then `struct hostel` cannot contain `struct student`.

However, a structure can contain a pointer to another structure of the same type. This is known as **self-referencing**. Such pointers are extensively used to create chains and various types of linked data structures. One notable example is the **linked list**, which we will explore later. For now, consider the following example:

```
typedef struct nodeStudent { // A name after struct is mandatory for self-referencing
    char name[30];
    char rollNum[12];
    int marks;
    struct nodeStudent *next; // A self-referencing pointer named next
} student;
```

In the linked list, each node will be the structure `student`, and so it will contain a student's `name`, `rollNum`, and `marks`, along with `next` that will point to the next node in the list. This enables the formation of a chain, where the last node points to `NULL` to signify the end of the list.

### 13.12 Structure as function argument

In this section, we see how swapping two points can be done by passing them to a swap function. If they points are passed to the function, the desired result is not achieved in `main()`. The swapping performed in the swap function does not affect the points in the calling function `main()`, similar to the behavior observed with basic data types such as `int` or `float`. Below is the C code:

```
#include <stdio.h>

typedef struct {
    float x, y;
} point;

void swapPoints(point p1, point p2) {
    point temp = p1;
    p1 = p2;
    p2 = temp;
}

int main() {
    point p1 = {1.0, 2.0};
    point p2 = {3.0, 4.0};

    printf("Before swap: p1 = (%f, %f), p2 = (%f, %f)\n", p1.x, p1.y, p2.x, p2.y);
    swapPoints(p1, p2); // Won't swap
    printf("After swap: p1 = (%f, %f), p2 = (%f, %f)\n", p1.x, p1.y, p2.x, p2.y);
    return 0;
}
```

### 13.13 Pointer to structure as function argument

We observed in §13.12 that the `swap` function cannot exchange two entities in the caller function when they are structures and passed directly. However, by passing pointers to these structures instead, the swap operation can be achieved. This approach allows the function to modify the original values in the calling function, `main()`, effectively performing the swap. The code below demonstrates this concept.

```
#include <stdio.h>

typedef struct {
    float x, y;
} point;

void swapPoints(point *p1, point *p2) {
    point temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}

int main() {
    point p1 = {1.0, 2.0};
    point p2 = {3.0, 4.0};

    printf("Before swap: p1 = (%f, %f), p2 = (%f, %f)\n", p1.x, p1.y, p2.x, p2.y);
    swapPoints(&p1, &p2); // Will swap
    printf("After swap: p1 = (%f, %f), p2 = (%f, %f)\n", p1.x, p1.y, p2.x, p2.y);
    return 0;
}
```

Now consider the task of swapping the coordinates of a point. To get it done, we have to again use pointer to the structure as argument of the called function. Here is the function and its caller:

```
void swapCoords(point *p) {
    float temp = p->x;
    p->x = p->y;
    p->y = temp;
}

int main() {
    point p = {1.0, 2.0};
    printf("Before swap: p = (%f, %f)\n", p.x, p.y);
    swapCoords(&p); // Will swap the coordinates of p
    printf("After swap: p = (%.1f, %.1f)\n", p.x, p.y); // Will print (2.0, 1.0)
    return 0;
}
```

The symbol `->` is called **arrow operator**. It is used to access members of a structure, through a structure-type pointer. For example, `p->x` is needed to access the member `x` of `p`, and it is basically a shortcut for `(*p).x`.

### 13.14 Operator Precedence

When working with structure pointers, operator precedence must be carefully considered. The dot operator (`.`) has higher precedence than the dereferencing operator (`*`). While both `p->x` and `(*p).x` are syntactically correct and equivalent, `*p.x` is invalid.

The arrow operator (`->`) is among the highest precedence operators. For instance, `++p->x` increments the `x` member of `p`. This is equivalent to `++(p->x)`—you should verify this. However, if `p` points to a structure in an array of structures, `(++p)->x` will move `p` to the next structure in the array and access the `x` member of that next structure.

### 13.15 Solved problems

1. **[Addition and subtraction of complex numbers]** Recall the definition of structure `complex` given in §13.4. Use it to write a code for addition and subtraction of two complex numbers. You should write a single user-defined function that will perform both addition and subtraction. It has to be called from `main()`, and the results have to be printed from `main()`. The values of the complex numbers should be taken in as input from `main()`.

```
1 #include <stdio.h>
2
3 typedef struct {
4     float real;
5     float imag;
6 } complex;
7
8 void operateComplex(complex a, complex b, complex *sum, complex *dif) {
9     sum->real = a.real + b.real; sum->imag = a.imag + b.imag;
10    dif->real = a.real - b.real; dif->imag = a.imag - b.imag;
11 }
```

```

12
13 int main() {
14     complex a, b, sum, dif;
15
16     printf("Enter 1st complex number (real and imaginary parts): ");
17     scanf("%f %f", &a.real, &a.imag);
18     printf("Enter 2nd complex number (real and imaginary parts): ");
19     scanf("%f %f", &b.real, &b.imag);
20
21     operateComplex(a, b, &sum, &dif);
22
23     printf("Sum = %.3f + %.3fi\n", sum.real, sum.imag);
24     printf("Difference = %.3f + %.3fi\n", dif.real, dif.imag);
25
26     return 0;
27 }

```

2. Recall the definition of structure `student` given in §13.4. Use it to write a code for the following tasks:

- (i) In `main()`, declare an array of 100 `students`. Read an integer `n` (at most 100) and then read in the details of `n` students in this array.

During insertion of each new record, use insertion sort so that the records are sorted by roll numbers in lexicographic order.

- (ii) Write a function to print the array.
- (iii) Write a function to search the array for a student by roll number using binary search. If it is found, return the corresponding index to `main()` and print the details from `main()`. If it is not found, return `-1`.



```

1 #include <stdio.h>
2 #include <string.h>
3
4 typedef struct {
5     char name[30];
6     char rollNum[12];
7     int marks;
8 } student;
9
10 void insertStudent(student a[], int n) {
11     student key;
12     for (int i = 1; i < n; i++) {
13         key = a[i];
14         int j = i - 1;
15         while (j >= 0 && strcmp(a[j].rollNum, key.rollNum) > 0) {
16             a[j + 1] = a[j];
17             j--;
18         }
19         a[j + 1] = key;
20     }
21 }
22
23 int binarySearch(student a[], int n, const char* rollNum) {
24     int left = 0, right = n - 1, mid;
25

```

```
26 while (left <= right) {
27     int mid = (left + right)/2;
28
29     if (strcmp(a[mid].rollNum, rollNum) == 0)
30         return mid;
31     if (strcmp(a[mid].rollNum, rollNum) < 0)
32         left = mid + 1;
33     else
34         right = mid - 1;
35 }
36 return -1;
37 }
38
39 void printStudents(student a[], int n) {
40     printf("Students List:\n");
41     printf("-----\n");
42     printf("%-30s %-12s %-5s\n", "Name", "Roll Number", "Marks");
43     printf("-----\n");
44
45     for (int i = 0; i < n; i++)
46         printf("%-30s %-12s %3d\n", a[i].name, a[i].rollNum, a[i].marks);
47 }
48
49 int main() {
50     student students[100];
51     int n;
52
53     printf("Enter the number of students (at most 100): ");
54     scanf("%d", &n);
55
56     for (int i = 0; i < n; i++) {
57         printf("Student %d (name, roll number, marks): ", i + 1);
58         scanf("%s %s %d", students[i].name, students[i].rollNum, &students[i].marks);
59     }
60
61     insertStudent(students, n);
62     printStudents(students, n);
63
64     char searchRollNum[12];
65     printf("Enter the roll number to search: ");
66     scanf("%s", searchRollNum);
67
68     int index = binarySearch(students, n, searchRollNum);
69     if (index != -1) {
70         printf("Student found:\n");
71         printf("Name: %s\n", students[index].name);
72         printf("Roll Number: %s\n", students[index].rollNum);
73         printf("Marks: %d\n", students[index].marks);
74     } else {
75         printf("Student with roll number %s not found.\n", searchRollNum);
76     }
77
78     return 0;
79 }
```

## 13.16 Exercise problems

1. **[Multiplication and division of complex numbers]** Recall the definition of structure `complex` given in §13.4. Use it to write a code for multiplication and division of two complex numbers. You should write a single user-defined function that will perform both multiplication and division. It has to be called from `main()`, and the results have to be printed from `main()`. The count and values of the complex numbers should be taken in as input from `main()`.
2. **[Sorting of points]** Write a function named `QuickSortPoints` that will take as arguments an array of points (structure `point` given in §13.5) and the number of points. Its task is to sort the complex numbers in lexicographic order of their radius and angle (in this order) in the polar coordinate system. It has to be called from `main()`, and the Cartesian as well as the polar coordinates of the sorted points have to be printed from `main()`. The count and the coordinates of the points should be taken in as input from `main()`. You can use `math.h`.

Example of 5 points (after sorting):

```
Point 1: ( 1.000,  2.000) = Radius:  2.236, Angle:  63.435 degrees
Point 2: (-2.000,  1.000) = Radius:  2.236, Angle: 153.435 degrees
Point 3: (-2.000, -2.000) = Radius:  2.828, Angle: 225.000 degrees
Point 4: (-5.000, -5.000) = Radius:  7.071, Angle: 225.000 degrees
Point 5: ( 7.000, -1.000) = Radius:  7.071, Angle: 351.870 degrees
```

3. **[Library system management]** Write a program to manage a library system. Define a structure `book` with fields like title, author, ISBN, and price. Declare a structure `library` containing an array of `books`. The program should allow the user to add new books, search for a book by ISBN, and display details of all books in the library.
4. **[Employee management system]** Write a program that stores employee details using a nested structure. The structure `employee` should contain fields like name, ID, and salary, as well as a nested structure `address` with fields like street, city, and postal code. The program should prompt the user to enter details for multiple employees, then print all details.
5. **[Shopping cart system]** Develop a program to simulate a shopping cart system. Define a structure `item` with fields for name, quantity, and price. Then, create a structure `cart` that contains an array of 100 `items`, along with the count for each item in a suitable manner. The program should calculate the total price of items (at most 100) in the cart and print the result. It should have the provision of choosing an item in multiple (at most 10, which is stored as its count).
- ♣ 6. **[Airline reservation system]** Write a program to manage an airline reservation system using nested structures. Define a structure `flight` with fields for flight number, departure time, and destination. Inside `flight`, include a nested structure `passenger` that holds passenger details, such as name, seat number, and a sub-structure for contact information (address, email ID, mobile phone number). Write a function to book a seat, ensuring that seats are not double-booked, and a function to search and display passengers by flight number. You should create an array of `flights`. For every flight, use sorting and binary search for efficiently managing the passengers by their seat numbers.
- ♣ 7. **[Multi-level bank management system]** Design a program for a multi-level bank management system. Define a structure `account` with fields like account number, balance, and type of account. Inside `account`, include a nested structure `transaction` with fields for transaction ID, date, and amount. Further, have another nested structure `branch` that holds branch details such as branch ID and branch address. Implement a function to perform deposits and withdrawals, update the account balance, and generate a detailed report of all transactions for a specific account. Use 1D arrays wherever necessary.
- ♠ 8. **[Polygon area]** Write a program to compute and print the signed area of a simple polygon. A positive value will indicate a counterclockwise orientation, while a negative value will denote a clockwise orientation. The program should take in as input the vertices in sequential order, either clockwise or counterclockwise, store them in an array of the structure `point`, given in §13.5. This problem can be challenging unless you take the hint.

**Hint:**

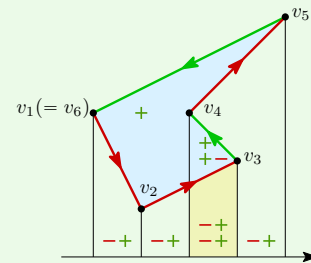
The **Shoelace Theorem** provides a formula to calculate the area of a simple (i.e., non-self-intersecting) polygon  $P$  whose vertices are known. Given  $n$  vertices  $v_1$  to  $v_n$ , either in clockwise or in counterclockwise order, the area can be computed using a determinant-like expression:

$$\text{area}(P) = \frac{1}{2} \begin{vmatrix} x_1 & x_2 & x_3 & \cdots & x_n & x_1 \\ y_1 & y_2 & y_3 & \cdots & y_n & y_1 \end{vmatrix} \quad (13.1)$$

which basically means:

$$\begin{aligned} \text{area}(P) &= \frac{1}{2} \left( \begin{vmatrix} x_1 & x_2 \\ y_1 & y_2 \end{vmatrix} + \begin{vmatrix} x_2 & x_3 \\ y_2 & y_3 \end{vmatrix} + \cdots + \begin{vmatrix} x_{n-1} & x_n \\ y_{n-1} & y_n \end{vmatrix} + \begin{vmatrix} x_n & x_1 \\ y_n & y_1 \end{vmatrix} \right) \\ &= \frac{1}{2} \left| \sum_{i=1}^n (x_i y_{i+1} - y_i x_{i+1}) \right|. \end{aligned} \quad (13.2)$$

The vertices of the polygon are represented as pairs of coordinates  $(x_i, y_i)$  for  $i = 1, 2, \dots, n$ , with  $v_{n+1}(x_{n+1}, y_{n+1})$  defined as  $v_1(x_1, y_1)$ . As depicted in the inset figure,  $v_6$  is same as  $v_1$ , as there are 5 vertices. In this figure, a red edge signifies that the area of the trapezoid below it is negative, while green indicates a positive area. The plus and minus signs for a region indicate how many times it has been accounted and discounted while summing up the trapezoidal areas. For example, the region highlighted in yellow has been accounted twice (for the edges  $(v_3, v_4)$  and  $(v_5, v_6)$ ) and also discounted twice (for  $(v_2, v_3)$  and  $(v_4, v_5)$ ), effectively contributing nothing to the area of  $P$ .



# 14 | Abstract Data Types

An **abstract data type (ADT)** is a specification of a set of organized data and the set of operations that can be performed on that data. It is abstract because it does not depend on specific implementations. The user only calls a function to perform an operation on the ADT, without needing to know the implementation details. Even if the internal implementation changes, the user can still use the ADT as long as the function interfaces remain the same. Each ADT defines a set of operations like insertion, deletion, and search, offering efficient ways to manage and manipulate data for specific use cases.

**Examples of ADT:** **List**, **Set**, **Stack**, and **Queue** are the most common examples of ADT, which we discuss here. Apart from these, there are some more, which include **Priority Queues**, **Double-Ended Queues**, **Circular Queues**, **Dictionaries** or **Maps**, **Trees**, **Graphs**, and **Hash Tables**.

**Implementation of ADT:** The ADTs that we shall study here can be implemented using either arrays or pointers (linked list, in particular). Array-based implementation is usually quicker and easier, but in some cases, pointer-based implementation is more efficient. We shall first study the array-based implementation.

## 14.1 List

**List** is an ADT representing a sequence of data items, usually of the same type. It is indexed, meaning that it has a first element, a second element, a third element, and so on. Thus, identical elements, if present, are assigned different indexes. Here are some key points about list:

1. An array is one way to represent a list.
2. Advantage of array-based list: Arrays are compact, with no wastage of space, and provide easy and constant-time access to an element if its index is known.
3. Limitations of array-based list: The size of an array must be statically or dynamically allocated. Inserting or deleting an element requires shifting other elements. **Linked list** is an alternative implementation to address these limitations.

A 1D array with single-valued elements, as discussed in Chapter 5, can be treated as a list. When the array contains elements of a structure type, as explained in Chapter 13, it can also be treated as a list. However, the typical operations on list, shown in Table 14.1, must be defined for the array.

## 14.2 Set

**Set** is an ADT representing an unordered collection of unique elements. For example, the sets  $\{1, 2, 3\}$  and  $\{3, 1, 2\}$  are considered identical, although they are not so if treated as lists. Similar to list, set can be implemented using a linear datatype like array. However, array-based implementation of set is sometimes not a good option, and then more efficient implementations are done using nonlinear structures. The common operations on a set include insertion, deletion, checking the size, and performing set operations like union, intersection, and difference, which are shown in Table 14.2.



**Table 14.1:** Operations on array-based list (elements are of type `char`).

Operation	Meaning
<code>list init()</code>	Initialize an empty list (say, <code>L</code> , declared and allocated in <code>main()</code> )
<code>int isEmpty(L)</code>	Return <code>1</code> (True) if the list <code>L</code> is empty, otherwise return <code>0</code> (False)
<code>int isFull(L)</code>	Return <code>1</code> (True) if the list <code>L</code> holds the maximum number of elements that it can, otherwise return <code>0</code> (False)
<code>char get(L, i)</code>	Return the element at index <code>i</code> of the list <code>L</code> if it is valid, otherwise return an error value or message
<code>list insert(L, i, x)</code>	Insert the element <code>x</code> at index <code>i</code> of the list <code>L</code>
<code>list delete(L, i)</code>	Delete the element at index <code>i</code> of the list <code>L</code>
<code>int size(L)</code>	Return the number of elements in the list <code>L</code>
<code>void print(L)</code>	Print the elements of the list <code>L</code> from first to last

**Table 14.2:** Operations on array-based set.

Operation	Meaning
<code>void insert(a, x)</code>	Insert element <code>x</code> into set <code>a</code>
<code>void delete(a, x)</code>	Remove element <code>x</code> from set <code>a</code>
<code>int size(a)</code>	Return number of elements in the set <code>a</code>
<code>set union(a, b)</code>	Return union of sets <code>a</code> and <code>b</code>
<code>set intersection(a, b)</code>	Return intersection of sets <code>a</code> and <code>b</code>
<code>set difference(a, b)</code>	Return difference of sets <code>a</code> and <code>b</code>

## 14.3 Stack

**Stack** is a linear data structure that follows the **last-in, first-out (LIFO)** principle, where elements are added and removed from the top. The stack is specified by the operations listed in Table 14.3, which are implemented as functions. Stack elements can be of any datatype, similar to a list or set. A demonstration of push and pop operations on a stack is provided in Fig. 14.1.

In an array-based implementation of stack, the following steps are performed:

1. Declare an array `S[]` of fixed size, say `n`, to serve as the stack, where `n` is the maximum stack-size. The allocation can be static or dynamic.
2. Store stack elements in `S[]` starting from index `0`.
3. Use a variable `top` that points to the top of the stack. If the stack contains `k` elements, then `top = k-1`. The value of `k` can be at most `n` when the stack is full. `top = k-1` indicates that elements `S[0]` to `S[k-1]` belong to the stack, while `S[k]` to `S[n-1]` are not part of the stack.
4. Both **push** and **pop** operations occur at the top of the stack. The most recently pushed element will be at the top. When an element is popped, it is removed from the stack, and `top` is decremented by one.

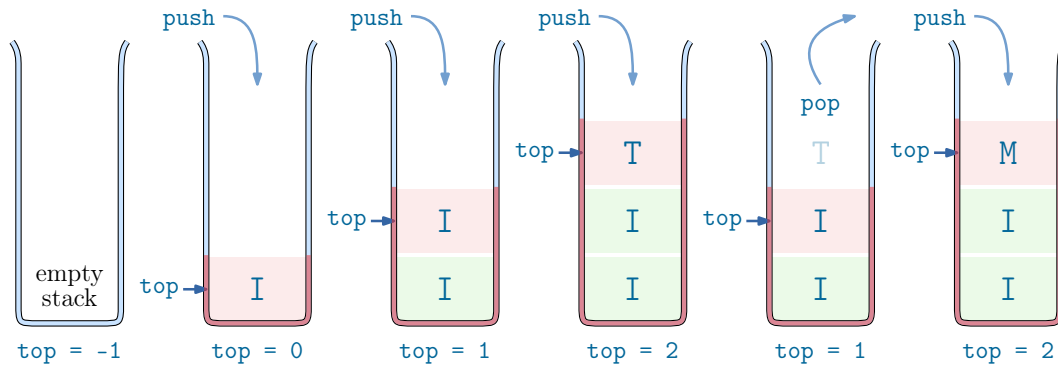


Figure 14.1: Effect of `push` and `pop` on a stack.

Table 14.3: Operations on array-based stack (elements are of type `char`). The actual prototypes are shown in Code 14.54.

Operation	Meaning
<code>void init(S)</code>	Initialize an empty stack <code>S</code> (with its pointer as argument)
<code>int isEmpty(S)</code>	Return 1 (True) if the stack <code>S</code> is empty, otherwise return 0 (False)
<code>int isFull(S)</code>	Return 1 (True) if the stack <code>S</code> holds the maximum number of elements that it can, otherwise return 0 (False)
<code>int push(S, x)</code>	Push the element <code>x</code> to the top of the stack <code>S</code> and adjust its top index
<code>char pop(S)</code>	Pop the element from the top of the stack <code>S</code> and adjust its top index
<code>char peek(S)</code>	Return the element at top of the stack <code>S</code> if it is nonempty, otherwise return an error value or message

Q47 Can we change the convention to fix the stack-top always at index 0?

Q48 Let `S[top] = x`. What is the change in `S[top]` just after `x` is popped?

### 14.3.1 Header Files

Header files, such as `stackArray.h` shown in Code 14.53, enhance modularity by organizing code into reusable components. They act as **guards** to prevent multiple inclusions of the same file, avoiding redefinition errors and compilation issues. The preprocessor directives used in Code 14.53 and their purposes are explained below.

1. `#ifndef _STACK_H`: This directive checks whether the macro `_STACK_H` is not defined. If it is not, the code within the guard will be processed.
2. `#define _STACK_H`: This defines the macro `_STACK_H`. Once defined, any further inclusion of the same header file will skip its content.
3. `#endif`: This marks the end of the conditional block that began with `#ifndef`.

The header file `stackArray.h` is included in `stackArrayOperations.c` to define all its functions that simulate the stack operations. This is shown in Code 14.54. The functions are called from `main()`, as shown in Code 14.55.

Code 14.53: Stack macro written as a header (.h) file.

stackArray.h

```
1 #ifndef _STACK_H
2 #define _STACK_H
3 #define MAXSIZE 100
4
5 typedef struct {
6     char data[MAXSIZE];
7     int top;
8 } stack;
9
10 void init(stack *);
11 int isEmpty(stack *);
12 int isFull(stack *);
13 int push(stack *, char);
14 char pop(stack *);
15 char peek(stack *);
16
17 #endif
```

Code 14.54: Stack operations.

stackArrayOperations.c

```
1 #include "stackArray.h" // header file included
2
3 void init(stack *S) {
4     S->top = -1; }
5
6 int isFull(stack *S) {
7     return S->top == MAXSIZE - 1; }
8
9 int isEmpty(stack *S) {
10    return S->top == -1; }
11
12 int push(stack *S, char c) {
13    if (isFull(S)) {
14        printf("Cannot be pushed, the stack is full!\n");
15        return 0; // Indicates failure
16    }
17    S->data[++S->top] = c;
18    return 1; // Indicates success
19 }
20
21 char pop(stack *S) {
22    if (isEmpty(S)) {
23        printf("The stack is empty.\n");
24        return '\0'; // Indicates failure
25    }
26    return S->data[S->top--];
27 }
28
29 char peek(stack *S) {
30    if (isEmpty(S)) {
31        printf("The stack is empty.\n");
32        return '\0'; // Indicates failure
33    }
34    return S->data[S->top];
35 }
```

Code 14.55: The main function operating on an array-based stack.

`stackArrayMain.c`

```
1  #include <stdio.h>
2  #include "stackArrayOperations.c"
3
4  int main() {
5      stack S;
6      char c, x;
7      int success;
8
9      init(&S);
10
11     while (1) {
12         printf("Choose operation (P = push | p = pop | k = peek | e = exit)\n");
13         c = getchar();
14
15         switch (c) {
16             case 'e':
17                 return 0;
18             case 'P': {
19                 scanf(" %c", &x);
20                 success = push(&S, x);
21                 break;
22             }
23             case 'p':
24                 success = pop(&S);
25                 if (success != '\0')
26                     printf("Popped element: %c\n", success);
27                 break;
28             case 'k':
29                 success = peek(&S);
30                 if (success != '\0')
31                     printf("Peeked element: %c\n", success);
32                 break;
33             default:
34                 printf("Invalid choice, please provide a valid choice...\n");
35                 break;
36         }
37
38         while (getchar() != '\n')
39             ; // It's an empty statement, used to skip leftover newline or white space
40     }
41
42     return 0;
43 }
```

## 14.4 Queue

**Queue** is a linear data structure that follows the **first-in, first-out (FIFO)** principle. Elements are inserted at the rear and removed from the front. Insertion refers to adding an element, which is known as the **enqueue** operation, while removal refers to deleting an element, also known as the **dequeue** operation.

In addition to enqueue and dequeue, checking whether the queue is empty or full, similar to stacks, is essential. Another important operation is determining the number of elements currently present in the queue.

In a stack, both insertion and deletion occur at the top, managed by a single pointer named **top**. In contrast, a queue uses two pointers: **front**, which tracks the first element, and **rear**, which tracks the last. A macro containing the structure and operations for the queue is provided in Code 14.56. This macro can be included as a header file in different implementations of queues, as discussed in §14.4.2 and §14.4.3.

---

**Code 14.56:** Queue macro as a header file (for both linear and circular arrays).

[queueArray.h](#)

```
1 | #ifndef _QUEUE_H
2 | #define _QUEUE_H
3 |
4 | #define MAXSIZE 12
5 |
6 | typedef struct {
7 |     char data[MAXSIZE];
8 |     int front;
9 |     int rear;
10 | } queue;
11 |
12 | void init(queue *q);
13 | int add(queue *q, char val);
14 | int delete(queue *q);
15 | int front(queue *q, char *val);
16 |
17 | #endif
```

---

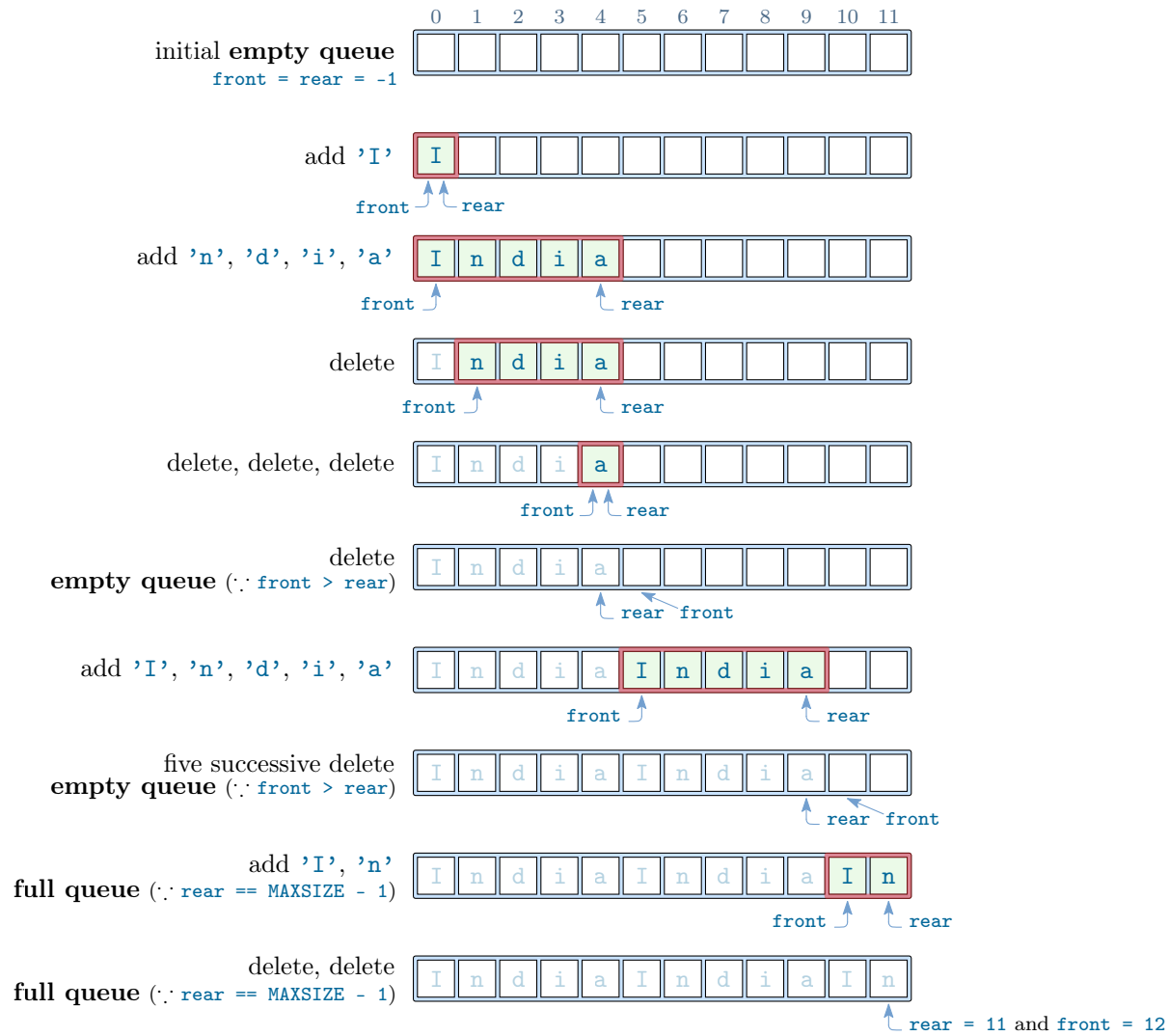
### 14.4.1 Applications

Queues have a broad range of applications, often outnumbering those of stacks, as the principle of waiting in turn is fundamental in both computing and daily life. In computer systems, queues manage tasks awaiting shared resources, such as print jobs pending on a printer, processes waiting for disk storage access, or tasks in time-sharing systems waiting for CPU execution.

Within individual programs, queues facilitate sequential processing of multiple requests. For instance, one task may spawn additional tasks that must be completed in order, all managed by a queue.

Moreover, queues are crucial in various computational problems. For example, they are employed in **Breadth-First Search (BFS)** (beyond the scope of this course) for traversing all nodes in a graph, layer by layer. Practical applications include packet scheduling in network routers and managing customer service requests, where clients are served based on their position in the queue. They also simulate call centers by organizing customer requests in the order of arrival.

In coding challenges, queues are essential for processing elements systematically. In the **sliding window maximum** problem, a **deque** (double-ended queue) helps identify the maximum element in each window of a specified size within an array. Queues are also valuable in **job scheduling** scenarios, such as implementing round-robin task management.



**Figure 14.2:** Add (enqueue) and delete (dequeue) operations on a queue implemented with a linear array of 12 elements. The faded elements are present in the array but are not part of the active queue. Note that in the last two configurations, the queue is considered full, preventing further enqueues, even though the array is empty. This illustrates a serious limitation of this implementation.

### 14.4.2 Linear queue

We can implement a queue using a linear array, say `q[]`, equipped with the pointers `front` and `rear`. Both pointers are initialized to `-1` when `q[]` is first created. To add an element to `q[]`, we increment `rear` by one and store the element in `q[rear]`. To remove an element from `q[]`, we retrieve it from `q[front]` and then increment `front` by one. This queue is known as a **linear queue**, as it is implemented using a linear array.

This implementation has a significant limitation due to the way the pointers are managed. Since both `front` and `rear` are only incremented and never reset, they will eventually reach the end of `q[]`. As a result, the queue will not accept any new elements, even though `q[]` may have free space. This highlights an inherent flaw in linear array-based queue implementations. See an illustration in Figure 14.2.

Code 14.57: Queue operations on linear array.

queueArrayOperations.c

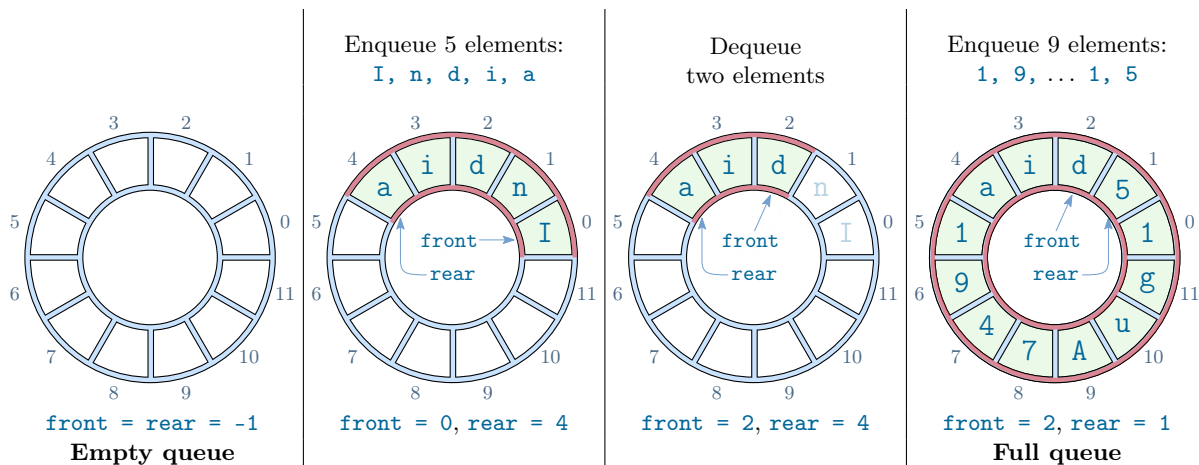
```
1 #include "queueArray.h"
2
3 void init(queue *q) {
4     q->front = q->rear = -1;
5 }
6
7 int add(queue *q, char val) {
8     if (q->rear == MAXSIZE - 1)
9         return 0; // Queue is full
10
11     if (q->front == -1)
12         q->front = 0; // Set front to 0 if the queue was empty
13
14     q->data[++q->rear] = val;
15     return 1; // Success
16 }
17
18 int delete(queue *q) {
19     if (q->front == -1 || q->front > q->rear)
20         return 0; // Queue is empty
21
22     q->front++; // Move the front pointer forward
23     return 1; // Success
24 }
25
26 int front(queue *q, char *val) {
27     if (q->front == -1 || q->front > q->rear)
28         return 0; // Queue is empty
29
30     *val = q->data[q->front]; // Retrieve front character
31     return 1; // Success
32 }
```

1. `#include "queueArray.h"`: Includes the header file defining the `queue` structure and constants, such as `MAXSIZE`.
2. `void init(queue *q)`: Initializes a queue by setting `front` and `rear` to `-1`, marking it as empty.
3. `int add(queue *q, char val)`:
  - Checks if the queue is full by comparing `rear` with `MAXSIZE - 1`.
  - If the queue was empty, sets `front` to `0`.
  - Adds `val` to the queue by incrementing `rear` and storing the value in `q->data`.
  - Returns `1` for success or `0` if the queue is full.
4. `int delete(queue *q)`:
  - Checks if the queue is empty by verifying if `front` is `-1` or `front > rear`.
  - If not empty, increments `front` to remove an element.
  - Returns `1` for success or `0` if the queue is empty.
5. `int front(queue *q, char *val)`:
  - Checks if the queue is empty.
  - If not, retrieves the front element in `val`.
  - Returns `1` for success or `0` if the queue is empty.

**Code 14.58:** The main function for a queue (for both linear and circular arrays). [queueArrayMain.c](#)

```
1  #include <stdio.h>
2  #include "queueArrayOperations.c"
3
4  int main() {
5      queue q;
6      char c, x;
7      int success;
8
9      init(&q);
10
11     while (1) {
12         printf("Choose operation (a = add | d = delete | f = front | e = exit):\n");
13         c = getchar();
14
15         switch (c) {
16             case 'e':
17                 return 0;
18
19             case 'a': {
20                 printf("Enter character to add: ");
21                 scanf(" %c", &x);
22                 success = add(&q, x);
23                 if (success)
24                     printf("%c' added to the queue\n", x);
25                 else
26                     printf("Queue is full. Could not add '%c'\n", x);
27                 break;
28             }
29
30             case 'd':
31                 success = delete(&q);
32                 if (success)
33                     printf("Deleted an element from the queue\n");
34                 else
35                     printf("Queue is empty. Nothing to delete\n");
36                 break;
37
38             case 'f':
39                 success = front(&q, &x);
40                 if (success)
41                     printf("Front element: %c\n", x);
42                 else
43                     printf("Queue is empty. Nothing at the front\n");
44                 break;
45
46             default:
47                 printf("Invalid choice, please provide a valid choice...\n");
48                 break;
49         }
50
51         while (getchar() != '\n') ; // Empty loop to skip leftover input
52     }
53
54     return 0;
55 }
```





**Figure 14.3:** Enqueue (`add`) and dequeue (`delete`) operations on a circular queue of 12 elements. The faded elements (after dequeue) are present in the array but are not part of the active queue. Note that in the last configuration, when the queue is considered full, the array is also fully occupied by the elements of queue. In this example, this is the first time when the value of `rear` becomes less than the value of `front`.

### 14.4.3 Circular queue

The limitation of a linear array can be overcome by using a **circular array**. Various queue implementations based on circular arrays, such as those in [1, 2], efficiently utilize memory by allowing the `rear` and `front` pointers to wrap around, minimizing wasted space. This type of queue, called a **circular queue**, maintains the FIFO principle and enables efficient queue management.

Here, we illustrate a simple and efficient implementation of a circular queue that fully utilizes the array. A demonstration of the enqueue and dequeue operations is shown in Figure 14.3.

The circular queue builds on the same header (Code 14.56) and main function (Code 14.58), which are used in linear array-based queue. The `main()` function simply includes the modified C file in place of the previous one: `#include "queueCirArrayOperations.c"`

Only the functions implementing queue operations (Code 14.57) are adjusted to handle the array in a circular manner. The modified code is shown in Code 14.59, with an outline of its functions provided below.

- **Initialization (`init`):**  
Sets both `front` and `rear` to `-1` to indicate an empty queue, initializing the queue at the start.
- **Enqueue (`add`):**  
Checks if the queue is full by verifying if the next position of `rear` matches `front`:  
`if ((q->rear + 1) % MAXSIZE == q->front).`  
When adding the first element, it sets `front` to `0` and updates `rear` with:  
`q->rear = (q->rear + 1) % MAXSIZE;`  
This moves `rear` to the next position in a circular fashion, placing the new value in `data[q->rear]`.
- **Dequeue (`delete`):**  
First, checks if the queue is empty: `if (q->front == -1).`  
If only one element remains, both pointers are reset to `-1`. Otherwise, it updates `front` with:  
`q->front = (q->front + 1) % MAXSIZE;`  
This moves `front` to the next position, maintaining the circular arrangement.
- **Accessing the front element (`front`):**  
This function retrieves the value at the front index. It checks if the queue is empty and returns the element if it is not.

Code 14.59: Queue operations on a circular array.

queueCirArrayOperations.c

```
1 #include "queueArray.h" // header file is same as linear array
2
3 void init(queue *q) {
4     q->front = q->rear = -1;
5 }
6
7 int add(queue *q, char val) {
8     if ((q->rear + 1) % MAXSIZE == q->front)
9         return 0; // Queue is full
10
11     if (q->front == -1) {
12         q->front = 0; // Set front to 0 if the queue was empty
13     }
14
15     q->rear = (q->rear + 1) % MAXSIZE; // Move rear circularly
16     q->data[q->rear] = val;
17     return 1; // Success
18 }
19
20 int delete(queue *q) {
21     if (q->front == -1)
22         return 0; // Queue is empty
23
24     if (q->front == q->rear) // Check if the queue is empty after deletion
25         q->front = q->rear = -1; // Reset the queue
26     else
27         q->front = (q->front + 1) % MAXSIZE; // Move front circularly
28
29     return 1; // Success
30 }
31
32 int front(queue *q, char *val) {
33     if (q->front == -1)
34         return 0; // Queue is empty
35
36     *val = q->data[q->front]; // Retrieve front character
37     return 1; // Success
38 }
```

## 14.5 Conceptual problems

**14.5.1 Linear Queue Operations:** In an array-based linear queue of size 10, explain the behavior of the queue when performing the following operations in sequence:

- Enqueue 5 elements.
- Dequeue 2 elements.
- Enqueue 3 more elements.
- Dequeue all elements.

Describe the state of the front and rear pointers after each operation.

**Answer:** The sequence of operations is as follows:

- After enqueueing 5 elements, the queue will have elements at positions 0 to 4, and the rear will be 4.
- After dequeuing 2 elements, the front will be at position 2, and the queue will have elements at positions 2 to 4.
- After enqueueing 3 more elements, the queue will have elements at positions 2 to 4 and 5 to 7, and the rear will be 7.
- After dequeuing all elements, the front and rear will both be  $-1$ , indicating the queue is empty.

**14.5.2 Queue Overflow and Underflow:** Explain what happens when you try to enqueue an element into a full queue and when you try to dequeue from an empty queue. Assume a queue implemented with a fixed-size array.

**Answer:** In a fixed-size array-based queue:

- If an attempt is made to enqueue an element into a full queue, the operation will result in a "Queue Overflow" error.
- If an attempt is made to dequeue from an empty queue, the operation will result in a "Queue Underflow" error.

**14.5.3 Circular Queue vs. Linear Queue:** Compare and contrast the circular queue and linear queue in terms of space utilization, memory usage, and handling of overflow.

**Answer:**

- A linear queue has a limitation that the rear pointer can only move forward and is unable to reuse the spaces freed by dequeued elements, leading to inefficient space utilization.
- A circular queue, on the other hand, reuses the spaces freed by dequeued elements, resulting in better space utilization. It uses modular arithmetic to wrap the pointers around when they reach the end of the array.

**14.5.4 Circular Queue Operations:** Consider a circular queue of size 6. You enqueue 5 elements and then dequeue 3 elements. After that, you enqueue 2 more elements. Describe the state of the queue at the end (contents of the queue and positions of front and rear).

**Answer:** After the operations:

- After enqueueing 5 elements, the queue will have elements at positions 0 to 4, with the **rear** at 4 and the **front** at 0.
- After dequeuing 3 elements, the queue will have elements at positions 3 to 4, with the **front** moved to 3 and the **rear** still at 4.
- After enqueueing 2 more elements, the queue will have elements at positions 3 to 5 and 0 to 1, with the **rear** wrapping around to position 1 and the **front** remaining at 3.

**14.5.5 Queue and Stack Differences:** Describe the primary differences between a queue and a stack in terms of their structure, order of element access, and typical use cases.

**Answer:**

- A queue follows a First-In-First-Out (FIFO) order, meaning the element that is enqueued first will be dequeued first.
- A stack follows a Last-In-First-Out (LIFO) order, meaning the element that is pushed last will be popped first.
- Queues are typically used in situations like task scheduling and resource management, while stacks are used in recursive function calls, undo operations, and syntax parsing.

**14.5.6 Stack Overflow and Underflow:** Explain what happens when you try to push an element into a full stack and when you try to pop an element from an empty stack. Assume the stack is implemented with a fixed-size array.

**Answer:** In a fixed-size array-based stack:

- If an attempt is made to push an element into a full stack, the operation will result in a "Stack Overflow" error.
- If an attempt is made to pop an element from an empty stack, the operation will result in a "Stack Underflow" error.

**14.5.7 Stack Operations: Push and Pop:** Describe the sequence of stack operations for the following actions:

- Push 4 elements onto an empty stack.
- Pop 2 elements.
- Push 3 more elements.
- Pop all elements.

What will be the stack content and top pointer after each operation?

**Answer:** After the operations:

- After pushing 4 elements, the stack will have elements at positions 0 to 3, and the top pointer will be 3.
- After popping 2 elements, the stack will have elements at positions 0 to 1, and the top pointer will be 1.
- After pushing 3 more elements, the stack will have elements at positions 0 to 5, and the top pointer will be 5.
- After popping all elements, the stack will be empty, and the top pointer will be  $-1$ .

**14.5.8 Evaluating Postfix Expressions:** Describe how to evaluate a postfix expression using a stack. For example, given the expression  $6\ 5\ 2\ 3\ +\ 8\ *\ +\ 3\ +\ *$ , show the sequence of operations and the final result.

**Answer:** To evaluate a postfix expression:

- Traverse each element of the expression.
- Push numbers onto the stack, say  $S$ .
- When an operator is encountered, pop the required number of operands, apply the operator, and push the result back onto the stack.
- Continue until the end of the expression. The final result will be the only element left on the stack.

For  $6\ 5\ 2\ 3\ +\ 8\ *\ +\ 3\ +\ *$ , the sequence of operations is:

- Push 6, 5, 2, and 3:  $S = \{6, 5, 2, 3\}$  with  $S.top = 3$ .
- Encounter  $+$ : Pop 3 and 2, push 5:  $S = \{6, 5, 5\}$  with  $S.top = 5$ .
- Push 8:  $S = \{6, 5, 5, 8\}$  with  $S.top = 8$ .

- Encounter `*`: Pop twice to get 8 and 5, multiply them to get 40, push 40:  $S = \{6, 5, 40\}$  with  $S.top = 40$ .
- Encounter `+`: Pop twice to get 40 and 5, push their sum 45:  $S = \{6, 45\}$  with  $S.top = 45$ .
- Push 3, then encounter `+`: Pop 3 and 45, push their sum 48:  $S = \{6, 48\}$  with  $S.top = 48$ .
- Encounter `*`: Pop twice to get 48 and 6, push their product 288:  $S = \{288\}$ .
- No more elements in the sequence, pop to get the final result: 288.

**14.5.9 Detecting a Circular Queue Overflow Condition:** In a circular queue implementation (without any extra slot or cell), explain how you can differentiate between a full and empty queue when using a fixed-size array.

**Answer:** To detect overflow and underflow conditions in a circular queue using a fixed-size array (without any extra slot or cell), see §14.4.3 and Code 14.59. Here is the idea:

- When the queue is empty, both `front == -1` and `rear == -1`.
- When the queue is full, the condition `(rear + 1) % size == front` is used. This indicates that the next position after `rear` will overwrite `front`, meaning the queue is full.
- When an element is dequeued, `front` is updated using `(front + 1) % size`.
- When the queue is empty after a dequeue operation, both `front` and `rear` are reset to `-1`.

**14.5.10 Checking Palindromes Using Stack:** Explain how a stack can be used to check if a string is a palindrome. For example, show the process for a string like `level`.

**Answer:** To check if a string is a palindrome using a stack  $S$ :

- First, find the length of the string, denoted as  $n$ .
- Push the first  $\frac{n}{2}$  characters of the string onto the stack.
- If  $n$  is odd, skip the next character.
- For the second half of the string, pop each character from the stack and compare it with the corresponding character in the second half.
- If all characters match, the string is a palindrome.

For the string `level`, where  $n = 5$ :

- First, push the first  $\frac{n}{2} = 2$  characters onto the stack:
  - Push `l`: Now  $S = [l]$ .
  - Push `e`: Now  $S = [l, e]$ .
- Next, pop and compare the characters with the second half of the string:
  - Pop `e`: Compare with the leftover character (`e`). They match.
  - Pop `l`: Compare with the next leftover character (`l`). They match.

Since all characters match, the string `level` is a palindrome.

♣ **14.5.11 Simulating Browser Back Button and Forward Button with Stacks:** Describe how two stacks can be used to implement the back and forward buttons of a web browser.

**Answer:** To simulate browser navigation:

- Use one stack for the history (back stack) and another for the forward stack.
- When a page is visited, push it onto the back stack and clear the forward stack.
- For the back button, pop from the back stack and push it onto the forward stack, displaying the previous page.
- For the forward button, pop from the forward stack and push it onto the back stack.

♣ **14.5.12 Implementing a Min Stack:** Describe how to design a stack that supports retrieving the minimum element in constant time, along with regular push and pop operations. How would you

handle both efficiency and memory usage?

**Answer:** To implement a min stack:

- Maintain two stacks: the main stack and a min stack.
- For push, add the element to the main stack. Also, push it onto the min stack if it is smaller than or equal to the current top of the min stack.
- For pop, remove the element from the main stack. Also pop the min stack if the popped element matches its top.
- Retrieve the minimum element by looking at the top of the min stack.
- This approach uses  $\mathcal{O}(n)$  space for the additional min stack, but all operations are  $\mathcal{O}(1)$ .

Consider a sequence of operations with duplicates:

- (i) Push 2 → Main Stack: [2], Min Stack: [2]
- (ii) Push 1 → Main Stack: [2, 1], Min Stack: [2, 1] (since 1 is a new minimum)
- (iii) Push 1 → Main Stack: [2, 1, 1], Min Stack: [2, 1, 1] (duplicate minimum)
- (iv) Push 3 → Main Stack: [2, 1, 1, 3], Min Stack: [2, 1, 1] (no change in minimum)
- (v) Pop → Main Stack: [2, 1, 1], Min Stack: [2, 1, 1]
- (vi) Pop → Main Stack: [2, 1], Min Stack: [2, 1]
- (vii) Pop → Main Stack: [2], Min Stack: [2]

## 14.6 Solved problems

- 14.6.1 [Balanced Parentheses]** Given a string containing only three types of parentheses, determine whether it is balanced. For example, "[(){}]" is balanced, but "([]){}" and "[{}]" are not. Use a stack to push open parentheses and ensure each has a corresponding close by popping them appropriately.

```

1  #include <stdio.h>
2  #include <stdbool.h>
3
4  #define MAXSIZE 100
5
6  typedef struct {
7      char data[MAXSIZE];
8      int top;
9  } Stack;
10
11 void init_stack(Stack *s) { s->top = -1; }
12 void push(Stack *s, char c) { s->data[++(s->top)] = c; }
13 char pop(Stack *s) { return s->data[(s->top)--]; }
14
15 bool is_balanced(const char *exp) {
16     Stack s;
17     init_stack(&s);
18
19     for (int i = 0; exp[i] != '\0'; i++) {
20         char current = exp[i];
21
22         if (current == '(' || current == '[' || current == '{')
23             push(&s, current);
24         else if (current == ')' || current == ']' || current == '}') {
25             if (s.top == -1)
26                 return false;
27
28             char top = pop(&s);
29             if ((current == ')' && top != '(') ||
30                 (current == ']' && top != '[') ||
31                 (current == '}' && top != '{'))
32                 return false;
33         } // end else-if
34     } // end for
35     return s.top == -1;
36 }
37
38 int main() {
39     char exp[MAXSIZE];
40     printf("Enter an expression: ");
41     scanf("%s", exp);
42     printf("%s.\n", is_balanced(exp) ? "Balanced" : "Not balanced");
43
44     return 0;
45 }

```

♣ **14.6.2 [Infix to Postfix]** An **infix expression** is an arithmetic expression in which an operator is positioned between its operands, for example,  $1+2$ . Generally, if `op1` and `op2` are two operands and `opr` is the operator, the infix expression can be represented as `(op1 opr op2)`. Note that enclosing an infix expression in parentheses is always safe, as it allows the expression to be treated as an operand in another expression.

A **postfix expression**, also known as **Reverse Polish Notation**, places operators after their respective operands. Therefore, the structure is represented as `op1 op2 opr`, such as in the example  $12+$ . The advantage of postfix notation is that it eliminates the need for parentheses to indicate operator precedence, allowing for evaluation from left to right using a stack. We shall see it soon.

The objective is to convert a given infix expression into a postfix expression. Assume that the infix expression will contain the digits 1 through 9 as operands, along with the operators `+`, `-`, `*`, and `/`, as well as parentheses `(` and `)`. Also assume that the provided infix expression is properly parenthesized and valid.

Here are some examples:

Infix	Postfix
$1+2*3$	$123*+$
$(1+2)*(3-4)$	$12+34-*$
$1+2*3-4$	$123*+4-$
$((1*9-4)/(7*8)+1/5)/((1-9)*(4+7))$	$19*4-78*/15/+19-47+*/$

**Algorithm:** Use a stack to handle operators and parentheses as you scan the infix expression from left to right. Here are the steps:

- (i) Send any operand directly to the postfix.
- (ii) For operators, pop elements from the stack until the stack top has an operator with lower precedence. Then, push the current operator.
- (iii) Push `(` directly onto the stack, and on encountering `)`, pop all elements until a `(` is encountered.

This approach ensures that the expression is correctly converted to postfix with proper operator precedence. The algorithm operates in linear time with respect to the length of the infix expression.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  #define MAX 100
6
7  char stack[MAX]; // It's declared globally to make the code smaller.
8  int top = -1;    // Not a conventional way of defining a stack.
9
10 int isEmpty() {
11     return top == -1;
12 }
13
14 void push(char c) {
15     stack[++top] = c;
16 } // assuming no overflow
17
18 char pop() {
19     if (top >= 0)
20         return stack[top--];
21     return '\0'; // Return null character if stack is empty
22 }
23

```



```
24 int precedence(char c) { // Function to check the precedence of operators
25     if (c == '+' || c == '-') return 1;
26     if (c == '*' || c == '/') return 2;
27     return 0; // non-operators
28 }
29
30 int isDigit(char c) {
31     return c >= '1' && c <= '9';
32 }
33
34 void infixToPostfix(char* infix, char* postfix) {
35     int i, j = 0;
36
37     for (i = 0; infix[i]; i++) {
38         if (isDigit(infix[i]))
39             postfix[j++] = infix[i];
40
41         else if (infix[i] == '(')
42             push(infix[i]);
43
44         else if (infix[i] == ')') {
45             while (!isEmpty()) {
46                 char topChar = pop(); // Pop the top element
47                 if (topChar == '(') break; // Stop if it's '('
48                 postfix[j++] = topChar; // Otherwise, add it to output
49             }
50         }
51
52         else { // the character is an operator
53             while (!isEmpty() && precedence(stack[top]) >= precedence(infix[i]))
54                 postfix[j++] = pop();
55             push(infix[i]);
56         }
57     }
58
59     // Pop all the operators from the stack
60     while (!isEmpty())
61         postfix[j++] = pop();
62
63     postfix[j] = '\0'; // Null-terminate the postfix string
64 }
65
66 int main() {
67     char infix[MAX], postfix[MAX];
68
69     printf("Enter an infix expression (digits 1-9): ");
70     scanf("%s", infix);
71
72     infixToPostfix(infix, postfix);
73     printf("Postfix expression: %s\n", postfix);
74
75     return 0;
76 }
```

**14.6.3 [Evaluate postfix expression]** As stated in Problem 14.6.2, in a postfix expression, operators appear after their respective operands. Given a postfix expression, the task is to compute its real value as a floating point number.

Assume that the expression contains the digits 1 through 9 as operands, along with the operators +, -, \*, and /. Also assume that the provided expression is valid. Here are the same example expressions as in Problem 14.6.2 and their values computed from the postfix expressions using the code given below:

Infix	Postfix	Value
1+2*3	123*+	7.000000
(1+2)*(3-4)	12+34-*	-3.000000
1+2*3-4	123*+4-	3.000000
((1*9-4)/(7*8)+1/5)/((1-9)*(4+7))	19*4-78*/15/+19-47+*/	-0.003287

**Algorithm:** Use a stack to evaluate the postfix expression as you scan it from left to right. For each token in the expression:

- (i) If the token is an operand, push it onto the stack.
- (ii) If the token is an operator, pop the top two operands from the stack and apply the operator to these operands.
- (iii) Push the result back onto the stack.
- (iv) After processing all tokens, the stack will contain one element, which is the final value of the expression.

This approach guarantees that the expression is evaluated correctly based on the order of operations inherent in postfix notation. The algorithm operates in  $O(n)$  time, where  $n$  is the number of tokens in the postfix expression.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define MAX 100
5
6  float stack[MAX]; // It's declared globally to make the code smaller.
7  int top = -1;    // Not a conventional way of defining a stack.
8
9  int isEmpty() {
10     return top == -1;
11 }
12
13 void push(float x) {
14     stack[++top] = x;
15 } // assuming no overflow
16
17 float pop() {
18     if (top >= 0)
19         return stack[top--];
20     return 0; // Stack is empty
21 }
22
23 float evaluatePostfix(char* expression) {
24     for (int i = 0; expression[i]; i++) {
25         if (expression[i] >= '0' && expression[i] <= '9') {
26             push((float)(expression[i] - '0'));
27         } else { // It's an operator, pop the top two elements from the stack

```

```

28     float operand2 = pop();
29     float operand1 = pop();
30     float result;
31
32     switch (expression[i]) {
33     case '+': result = (float)operand1 + operand2; break;
34     case '-': result = (float)operand1 - operand2; break;
35     case '*': result = (float)operand1 * operand2; break;
36     case '/':
37         if (operand2 != 0) result = (float)operand1 / operand2;
38         else {
39             printf("Error: Division by zero\n"); return 0;}
40         break;
41     default: result = 0; // Invalid operator
42     }
43     push(result);
44 }
45 }
46 return pop();
47 }
48
49 int main() {
50     char expression[MAX];
51     printf("Enter a postfix expression (digits and operators): ");
52     scanf("%s", expression); // Use scanf to read input, avoids newline
53     float result = evaluatePostfix(expression);
54     printf("Result: %f\n", result);
55     return 0;
56 }

```

♣ **14.6.4 [Network Packet Scheduling]** Implement a packet scheduler using a circular queue of size 5 for a network interface. The scheduler should ensure that packets are sent in the correct order, and as new packets are added, old packets are removed as needed to manage memory efficiently within the circular queue.

The input data will consist of letters and digits and other characters, ending with a newline character, and each packet, except possibly the last, should contain exactly 12 characters from the input data.

**Explanation:** Network packet scheduling is essential for efficiently managing the order and timing of data transmission. A packet scheduler organizes packets so they can be transmitted smoothly over the network, avoiding delays. Here, a circular queue is used, allowing the scheduler to reuse space continuously as packets are processed. This approach ensures the correct ordering of packets and prevents memory overflow by discarding old packets when new data arrives.

Here is an example:

```

Enter data (end with newline):
India@1947-August-15-Independence&Division-J&K-Jinnah-Nehru-Mountbatten-Treaty...
Sending packet: India@1947-A
Sending packet: ugust-15-Ind
Sending packet: ependence&Di
Sending packet: vision-J&K-J
Sending packet: innah-Nehru-
Sending packet: Mountbatten-
Sending packet: Treaty...
--- End --- All packets despatched ---

```



```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <unistd.h> // to simulate sleep
5
6  #define MAX_PACKETS 5
7  #define PACKET_SIZE 12
8
9  typedef struct {
10     char packets[MAX_PACKETS][PACKET_SIZE + 1]; // +1 for null terminator
11     int front, rear;
12 } CircularQueue;
13
14 // Function to initialize the circular queue
15 void init_queue(CircularQueue *q) {
16     q->front = 0;
17     q->rear = 0;
18 }
19
20 // Function to add a packet to the circular queue
21 int add_packet(CircularQueue *q, const char *packet) {
22     int next_rear = (q->rear + 1) % MAX_PACKETS; // Calculate next rear position
23     if (next_rear == q->front) {
24         printf("Queue is full. Cannot add packet: %s\n", packet);
25         return 0; // Queue is full
26     }
27     strncpy(q->packets[q->rear], packet, PACKET_SIZE); // Copy the packet
28     q->packets[q->rear][PACKET_SIZE] = '\0'; // Ensure null termination
29     q->rear = next_rear; // Update rear
30     return 1; // Success
31 }
32
33 // Function to send (remove) a packet from the circular queue
34 void send_packet(CircularQueue *q) {
35     if (q->front == q->rear) {
36         printf("Queue is empty. No packet to send.\n");
37         return; // Queue is empty
38     }
39     printf("Sending packet: %s\n", q->packets[q->front]);
40     q->front = (q->front + 1) % MAX_PACKETS; // Update front to remove the sent packet
41 }
42
43 int main() {
44     CircularQueue queue;
45     init_queue(&queue);
46
47     char data[1000]; // To hold user input
48     printf("Enter data (end with newline):\n");
49
50     // Read input data
51     fgets(data, sizeof(data), stdin);
52
53     // Remove newline character from the input

```

```

54 | size_t len = strlen(data);
55 | if (len > 0 && data[len - 1] == '\n') {
56 |     data[len - 1] = '\0'; // Replace newline with null terminator
57 | }
58 |
59 | // Process input for packet creation
60 | size_t total_length = strlen(data);
61 | for (size_t i = 0; i < total_length; i += PACKET_SIZE) {
62 |     char packet[PACKET_SIZE + 1]; // +1 for null terminator
63 |
64 |     // Copy up to PACKET_SIZE characters from the input
65 |     size_t j;
66 |     for (j = 0; j < PACKET_SIZE; j++) {
67 |         if (i + j < total_length) {
68 |             packet[j] = data[i + j]; // Fill packet with characters
69 |         } else {
70 |             packet[j] = '\0'; // Ensure remaining space is null-terminated
71 |             break; // Exit loop if we reach the end of the string
72 |         }
73 |     }
74 |
75 |     // Null terminate the packet to avoid any garbage values
76 |     packet[j] = '\0';
77 |
78 |     // Add packet to the queue
79 |     if (add_packet(&queue, packet)) {
80 |         // Simulate sending a packet after a delay
81 |         sleep(1); // Wait for 1 second
82 |         send_packet(&queue); // Send the packet
83 |     }
84 | }
85 |
86 | // Send any remaining packets in the queue
87 | while (queue.front != queue.rear) {
88 |     send_packet(&queue);
89 |     sleep(1); // Optional delay for simulation
90 | }
91 | printf("--- End --- All packets despatched ---\n");
92 |
93 |
94 | return 0;
95 | }

```

♠ **14.6.5 [Next Greater Element]** Given an array of  $n$  nonzero integers, find the next greater element (NGE) for each element. Total time complexity should be  $\mathcal{O}(n)$ . Assume that  $n \leq 100$ .

For example, if the array contains 7 elements: 1 5 8 1 9 4 7, the output should be 5 8 9 9 - 7 -.

**Algorithm:** Use a stack and process elements from the last to the first. For each element  $a[i]$ , pop elements from the stack that are smaller than or equal to  $a[i]$ , as they cannot serve as the NGE for any preceding elements. After these elements are popped, the top of the stack represents the NGE of  $a[i]$ . Now push  $a[i]$  onto the stack to allow it to serve as the NGE for any preceding elements.

The operations of push and pop justify the linear time complexity. Each element is pushed onto the stack exactly once and, once popped, is not pushed again. Thus, the total number of push and pop operations is at most  $2n$ . Since each push and pop operation takes constant time, the overall complexity is linear.



```
1 #include <stdio.h>
2 #define MAX 100
3
4 int stack[MAX], top = -1; // It's declared globally to make the code smaller :)
5 // Not a conventional way of defining a stack :(
6
7 void push(int value) { stack[++top] = value; } // assuming no overflow
8 int pop() { return stack[top--]; } // assuming no underflow
9 int isEmpty() { return top == -1; }
10
11 void nextGreaterElement(int a[], int n) {
12     int nge[MAX];
13
14     for (int i = n - 1; i >= 0; i--) {
15         while (!isEmpty() && stack[top] <= a[i])
16             pop();
17         nge[i] = isEmpty() ? 0 : stack[top];
18         push(a[i]);
19         printf("top (%d) = %d\n", top, (top < 0) ? 0 : stack[top]);
20     }
21
22     for (int i = 0; i < n; i++)
23         if (nge[i] == 0)
24             printf("- ");
25         else
26             printf("%d ", nge[i]);
27     }
28
29 int main() {
30     int n, a[MAX];
31
32     printf("Enter the number of elements: ");
33     scanf("%d", &n);
34
35     printf("Enter the elements:\n");
36     for (int i = 0; i < n; i++)
37         scanf("%d", &a[i]);
38
39     nextGreaterElement(a, n);
40     return 0;
41 }
```

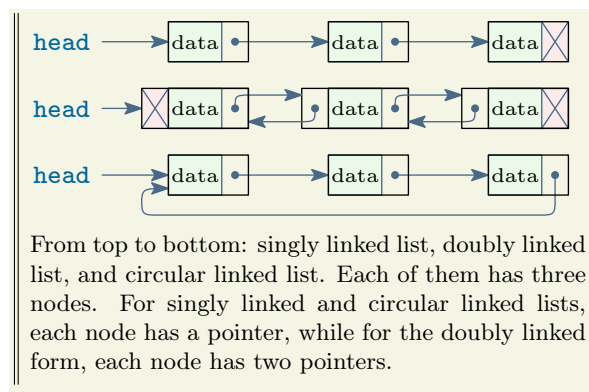
# 15 | Linked Lists

A **linked list** consists of a linear collection of entities, or **nodes**, all of the same type. Each node contains a **pointer** or **link** to the next node, establishing an ordered chain or sequence.

The physical placement of the nodes in memory is independent of their sequence. This provides flexibility to allocate space from available free memory as required. Linked lists are particularly useful in scenarios where the data size can change dynamically.

Linked lists can take various forms, as outlined below. Both doubly linked lists and circular linked lists include additional pointers that optimize specific operations.

- 1. Singly linked list:** The most basic form of linked list, where each node points to the next node only. (Unless mentioned otherwise, the phrase “linked list” typically refers to a “singly linked list”).
- 2. Doubly linked list:** Each node has two pointers—one to the next node and one to the previous, allowing bidirectional traversal.
- 3. Circular linked list:** Links the last node back to the first node, forming a loop, which enables continuous traversal. If it’s made doubly linked, we get **doubly linked circular list**, which allows continuous traversal in both directions.



## 15.1 Essence of linked lists

Linked lists are used for efficient implementation of any abstract data type (ADT), such as lists, stacks, and queues. For any such ADT, data may need to be arbitrarily inserted or removed, which may eventually increase in size to an extent where contiguous memory allocation is not possible, making arrays unsuitable. Linked lists, in such cases, serve the purpose because they do not require contiguous memory allocation.

We discuss below some relevant details of linked lists to understand their merits, demerits, and applicability.

### 15.1.1 Scope and programmability

The **C** language has particular strengths in working with linked lists due to its support for pointers, which allow direct access and manipulation of memory addresses. This enables a high degree of control over memory management, making **C** a suitable choice for creating and managing linked lists. Linked lists can also be implemented in other languages, including **C++**, **Java**, and **Python**. In these languages, linked lists are generally implemented with classes and object references.

However, not all programming languages provide direct support for linked lists as part of their standard library. For example, `JavaScript` does not offer built-in linked list support, although a linked list can still be created manually by defining nodes as objects. The flexibility of linked lists makes them a versatile data structure, adaptable across various programming languages.

### 15.1.2 Advantages of linked lists

1. **Ease of insertion and deletion:** Elements can be easily inserted or removed from any position in a linked list without requiring reallocation or reorganization of the entire structure, unlike arrays, where inserting or deleting an element typically involves shifting elements. In particular, inserting an element at the beginning of an array takes linear time because of shifting the existing elements, whereas for linked list it's a constant-time operation.
2. **No need for contiguous memory:** Linked lists do not require contiguous memory allocation, making them advantageous in cases where memory is fragmented.
3. **Dynamic size:** Linked lists can grow or shrink in size dynamically, which makes them more memory-efficient than arrays when the size of the dataset is unpredictable.

### 15.1.3 Disadvantages of linked lists

1. **Linear access time:** Accessing an element in a linked list requires iterating from the head of the list to the desired position, resulting in linear time complexity in the worst case and average case. In contrast, arrays offer constant-time access with indexing.

For a sorted array, binary search can be utilized to efficiently locate an element. However, in the case of a linked list, even if the elements are sorted, we must rely on linear search. This limitation poses a serious bottleneck for linked lists.

A potential solution is to use **linked trees**, such as **binary search trees (BST)**, which allow efficient binary search similar to that in sorted arrays. (The details of BSTs are beyond the scope of this course.)

2. **Higher memory usage:** Each node in a linked list must store a reference (link) to the next node, resulting in additional memory overhead compared to arrays.
3. **Poor cache locality:** Linked lists lack the contiguous memory layout of arrays, which can lead to poor cache performance during traversal since elements are scattered throughout memory.

### 15.1.4 Comparison with array-based datatypes

1. **Lists:** As elements are added or removed, a linked list resizes itself dynamically in constant time. In contrast, an array-based list requires reallocation or shifting of elements, which may take linear time.
2. **Stacks:** A stack implemented with a linked list can grow without limit (within available memory) and does not need to manage capacity. An array-based stack, however, requires an initial capacity, and resizing or expanding the array involves reallocating memory.
3. **Queues:** A doubly linked list with a rear pointer allows efficient insertion at the rear and removal from the front, making it ideal for implementing queues with minimal overhead. Array-based queues require handling wraparounds in circular implementations or shifting elements, which can be inefficient for large or frequently changing data sizes.



## 15.2 Node of a linked list

As mentioned in the very beginning of this chapter, each member of a linked list is referred to as a **node**. All nodes have the same type, defined using `struct` or `typedef struct`. The defined structure is a **self-referential structure** because the structure refers to itself in its definition.

Following are the usual ways to define a self-referential structure for a student record, modified from the structure definition of `student` given in Chapter 13. This self-referential can be utilized to define and create nodes in a linked list.

Without <code>typedef</code>	With <code>typedef</code>	
<pre>struct student{     char name[30];     char rollNum[12];     int marks;     struct student *next; };</pre>	<pre>typedef struct student{     char name[30];     char rollNum[12];     int marks;     struct student *next; } student;</pre>	<pre>typedef struct{     char name[30];     char rollNum[12];     int marks;     struct student *next; } student;</pre>
Here <code>struct student</code> is a new datatype with a self-referential structure.	Here <code>student</code> is a new datatype with a self-referential structure. This version is the traditional way of defining structures.	Here <code>student</code> is a new datatype with a self-referential structure. Available in newer versions of C (C99 and later).

All three approaches define a node in a linked list where each node contains student data and a pointer to the next node. The name `next` is conventional (but not mandatory) for the pointer that points each node to the next one in the list. We shall see soon how the pointers are used for different operations on linked lists. Before that, let's understand how the definition works with `typedef` for the traditional version.

- Structure definition:** When we define the structure with `typedef struct student { ... }`, we are creating a type called `struct student` and simultaneously giving it a `typedef` alias of `student`.
- Self-reference:** Inside the structure definition, `struct student *next;` refers to the type written in the very beginning: `struct student`, which we are currently defining. The compiler understands that we are referring to the same structure that is being defined at that moment.
- Use of `typedef`:** After this `typedef`, we can refer to this structure as `student` without needing to prefix it with `struct`. However, within the structure definition itself, we still need to use `struct student` to refer to the type because it hasn't been fully defined yet (to the compiler) in the context of the member declarations.

Nevertheless, to avoid the nuances of aliasing, many programmers prefer to define the structure in the most traditional way, as follows:

Most traditional	
<pre>typedef struct student_tag{     char name[30];     char rollNum[12];     int marks;     struct student_tag *next; } student;</pre>	In this style, <code>student_tag</code> acts as a tag for the structure definition, allowing the compiler to recognize the type during the definition of its members. On the other hand, <code>student</code> serves as the <code>typedef</code> -name that can be used to conveniently refer to the structure throughout the rest of the code.

---

**Code 15.60:** An example of `main()` function for creating a linked list.

`linkedList_minimal_main.c`

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 typedef struct {
6     char name[30];
7     char rollNum[12];
8     int marks;
9     struct student* next; // Pointer to the next node
10 } student;
11
12 int main() {
13     student *head = NULL; // Initialize an empty linked list
14
15     // Additional operations (e.g., inserting nodes) would follow here
16
17     return 0;
18 }
```

---

## 15.3 Operations on linear linked list

A linked list supports various operations to manage the data it stores. Here, we describe these operations, including creating a list, inserting into it, deleting elements, searching, and any additional operations such as displaying the elements.

1. **Creating:** Initializing an empty linked list by setting the head pointer to `NULL`.
2. **Inserting:** Adding new nodes to the linked list in various positions: at the beginning, at the end, or at a specific position.
3. **Deleting:** Removing nodes from the linked list, whether from the beginning, end, or a specified location.
4. **Searching:** Finding a specific node based on certain criteria, such as a matching roll number.
5. **Displaying:** Traversing the list to view all nodes' data.

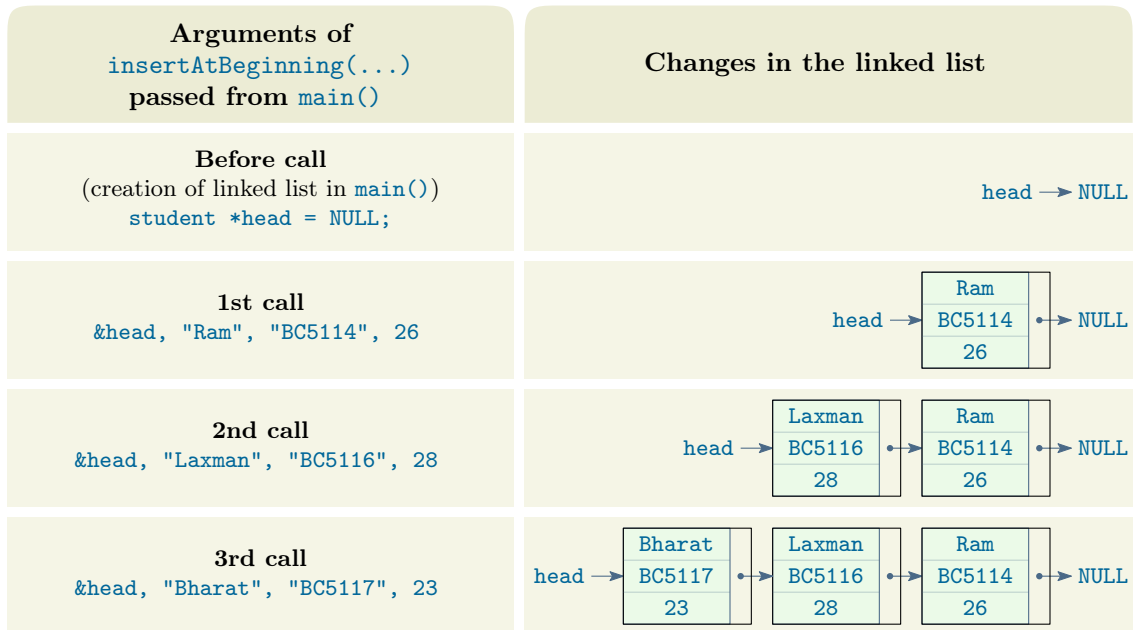
In the following subsections, we discuss each of these operations in detail.

### 15.3.1 Creating a linked list

To create a linked list, we first declare a pointer of the same type as its node structure. This pointer is named `head` (by usual convention, but it's not mandatory), because it points to the first node of a nonempty linked list. When the list is empty—that is, when it is just created but has no nodes—the `head` doesn't store any valid address and contains just `NULL`. To illustrate this, a minimal example of the `main()` function is given in Code 15.60. In this example, we initialize an empty linked list by setting the head pointer (declared as `student *head`) to `NULL`. This serves as the starting point for further operations, such as inserting nodes into the list.

### 15.3.2 Inserting at the beginning of a linked list

Insertion in a linked list can occur at different positions: at the beginning, at the end, or at a specific position. Insertion at the beginning is the simplest among these. The function is given in Code 15.61 and a detailed demonstration is provided in Figure 15.1 and Figure 15.2. The driver code `main()` is shown in Code 15.62, which is a modified version of Code 15.60.



**Figure 15.1:** Inserting nodes at the beginning of an initially empty linked list, by calling the function `insertAtBeginning(...)` from `main()`. See Figure 15.2 for further details.

**Code 15.61:** The function to insert a node at the beginning of a linked list. `linkedList_insertBeginFun.c`

```

1 void insertAtBeginning(student **head, char name[], char rollNum[], int marks) {
2     student *newNode = (student *)malloc(sizeof(student));
3     strcpy(newNode->name, name);
4     strcpy(newNode->rollNum, rollNum);
5     newNode->marks = marks;
6     newNode->next = *head;
7     *head = newNode;
8 }

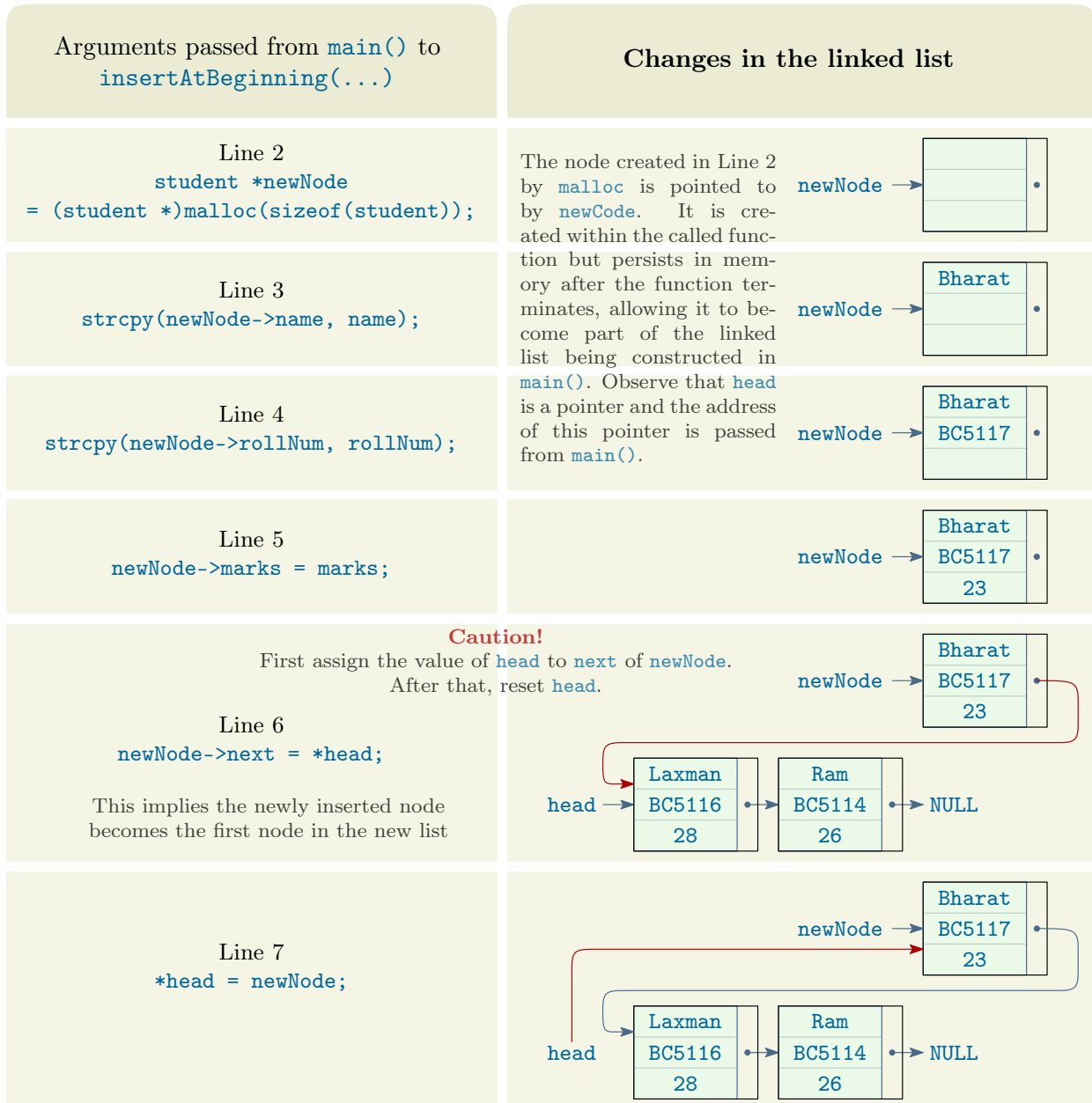
```

**Code 15.62:** The driver code `main()` for inserting nodes at the beginning of a linked list. `linkedList_Insert_Display_main.c`

```

1 int main() {
2     student *head = NULL; // Initialize an empty linked list
3     char name[30], rollNum[12], choice;
4     int marks;
5
6     do {
7         printf("Enter student name, roll number, marks: ");
8         scanf("%s%s%d", name, rollNum, &marks);
9         insertAtBeginning(&head, name, rollNum, marks);
10        printf("Do you want to add another student? (y/n): ");
11        scanf(" %c", &choice); // Mind the gap while scanning with %c
12    } while (choice == 'y' || choice == 'Y');
13
14    return 0;
15 }

```



**Figure 15.2:** Steps of inserting a new node at the beginning of a linked list with currently two nodes.

The function call from `main()` is:

```
insertAtBeginning(&head, "Bharat", "BC5117", 23);.
```

### 15.3.3 Inserting at the end of a linked list

Inserting a node at the end of a linked list involves creating a new node and initializing it with the given data fields. The process starts by checking if the list is empty. If it is, the new node is simply assigned as the head of the list. However, if the list already contains nodes, a temporary pointer is used to traverse from the head node to the last node in the list. Once the last node is identified (where its `next` pointer is `NULL`), the `next` pointer of this last node is updated to point to the new node. Finally, the new node's `next` pointer is set to `NULL` to mark it as the end of the list. The function is given in Code 15.63.

**Code 15.63:** The function to insert a node at the end of a linked list. [linkedList\\_insertEndFun.c](#)

```

1 void insertAtEnd(student **head, char name[], char rollNum[], int marks) {
2
3     // Allocate memory for the new node
4     student *newNode = (student *)malloc(sizeof(student));
5
6     // Set node field values
7     strcpy(newNode->name, name);           // Set name
8     strcpy(newNode->rollNum, rollNum);     // Set roll number
9     newNode->marks = marks;               // Set marks
10    newNode->next = NULL;                  // New node will be last, so next is NULL
11
12    // If the list is empty, new node becomes the head
13    if (*head == NULL) {
14        *head = newNode;
15        return;
16    }
17
18    // Traverse to the end of the list
19    student *endNode = *head;
20    while (endNode->next != NULL) {
21        endNode = endNode->next;
22    }
23
24    // Link the last node's next to the new node
25    endNode->next = newNode;
26 }

```

### 15.3.4 Inserting in an ordered linked list

The function is given in Code 15.64. This is a little more complex than the previous two cases. Here are the steps:

1. **Create** a new node with the specified data value to be inserted.
2. **Check** if the list is empty or if the new data is smaller than or equal to the head node's data. If either condition holds, insert the new node at the beginning of the list by making it the new head. This step ensures that the list remains in ascending order.
3. **Traverse** the list to find the correct position for insertion if the new data is greater than the head's data. Start from the head and move through each node until you find a node with a data value greater than the new data or reach the end of the list. This position will maintain the ordered sequence.
4. **Insert** the new node by adjusting the pointers so that the current node points to the new node, and the new node points to the next node in the list.

**Code 15.64:** The function to insert a node in an ordered linked list. [linkedList\\_insertOrdFun.c](#)

```

1 void insertInOrder(student **head, char name[], char rollNum[], int marks) {
2     // Create a new node
3     student *newNode = (student *)malloc(sizeof(student));
4     strcpy(newNode->name, name);
5     strcpy(newNode->rollNum, rollNum);
6     newNode->marks = marks;
7     newNode->next = NULL;

```

```

8
9 // Case 1: Insert at the beginning if list is empty or if new data is smallest
10 if (*head == NULL || (*head)->marks >= marks) {
11     newNode->next = *head;
12     *head = newNode;
13     return;
14 }
15
16 // Case 2: Find the insertion point
17 student *current = *head;
18 while (current->next != NULL) {
19     if (current->next->marks < marks)
20         current = current->next;
21     else
22         break;
23 }
24
25 // Insert the new node
26 newNode->next = current->next;
27 current->next = newNode;
28 }

```

### 15.3.5 Deleting from a linked list

Deletion in a linked list, similar to insertion, can occur at various positions: the beginning, the end, or a specific node. Among these, deletion of a specific node requires a little deeper understanding. Let's concentrate on this scenario, while the other cases are left as exercises. Its function is provided in Code 15.65, along with an explanation below. The input consists of a roll number corresponding to the node that needs to be deleted and the head of the linked list.

1. **Check** if the list is empty. If it is, there is no node to delete.
2. **Traverse** the list to identify the node with the specified roll number. Start from the head and move through each node, comparing the roll number of each node with the specified roll number until you find a match or reach the end of the list.
3. **Delete** the identified node by adjusting the pointers. If the node to be deleted is the head, update the head pointer to point to the next node in the list. If the node is found elsewhere in the list, update the previous node's next pointer to skip over the node being deleted and point to the subsequent node.
4. **Free** the memory allocated for the deleted node to prevent memory leaks.

**Code 15.65:** The function to delete a node from a linked list.

[linkedList\\_delete\\_rollNumFun.c](#)

```

1 void deleteByRollNum(student **head, char rollNum[]) {
2     student *current = *head;
3     student *previous = NULL;
4
5     // Traverse the list to find the node to delete
6     while (current != NULL && strcmp(current->rollNum, rollNum) != 0) {
7         previous = current;
8         current = current->next;
9     }
10
11     // If rollNum not found
12     if (current == NULL) {
13         printf("Student with roll number %s not found.\n", rollNum);

```

```

14     return;
15 }
16
17 // If node to be deleted is the head node
18 if (previous == NULL)
19     *head = current->next;
20 else
21     previous->next = current->next;
22
23 free(current);
24 printf("Student with roll number %s deleted successfully.\n", rollNum);
25 }

```

### 15.3.6 Searching in a linked list

To search for a node in the linked list, we traverse the list and compare each node's data with the target value. The function `searchByRollNum` to search for a student node in a linked list by roll number is provided in Code 15.66. The function takes the head of the linked list and a roll number as parameters. It returns a pointer to the `student` node if found, or `NULL` if not. Here are the steps:

1. **Traversal:** A pointer `current` is initialized to traverse the list starting from the head.
2. **Comparison:** Inside the `while` loop, the function compares the `rollNum` of the `current` node with the specified roll number using `strcmp`. If a match is found, the function returns the pointer to that `student` node.
3. **Moving to next node:** If the roll number does not match, the `current` pointer is moved to the next node in the list.
4. **Return NULL:** If the end of the list is reached without finding a match, the function returns `NULL`, indicating that the student was not found.

**Code 15.66:** The function to search a node in a linked list.

`linkedList_search_rollNumFun.c`

```

1 student* searchByRollNum(student *head, char rollNum[]) {
2     student *current = head;
3
4     while (current != NULL) {
5         if (strcmp(current->rollNum, rollNum) == 0)
6             return current;
7         current = current->next;
8     }
9
10    return NULL; // Student is not found
11 }

```

### 15.3.7 Displaying a Linked List

Displaying a list is helpful to view the list after certain operations. The function `displayList` to display all records in the linked list is shown in Code 15.67. The head pointer, passed to this function from `main()`, is used by the function to access the first node of the list and print its data. Afterward, it uses the next pointer of the first node to move to the second node, if it exists, and repeats the process until it reaches the end of the list. The end of the linked list is identified when the address of the current node is equal to `NULL`.

**Code 15.67:** The function to display all records in the linked list.

`linkedList_displayFun.c`

```

1 void displayList(student *head) {
2     student *current = head;
3     while (current != NULL) {
4         printf("Name: %s, Roll Number: %s, Marks: %d\n",
5             current->name, current->rollNum, current->marks);
6         current = current->next;
7     }
8 }
```

The function `displayList` has the pointer to `head` as the argument. It uses a similar pointer variable named `current` and initializes its value with the value stored in `head`. The pointer `current` visits each node starting from the first, and prints the values stored in the data fields: `current->name`, `current->rollNum`, and `current->marks`. Since `current` is a pointer, the data fields in every node are accessed by the arrow operator (`->`) instead of the dot operator (`.`). The usages of these operators are mentioned earlier in §13.5.1 and §13.13.

## 15.4 Stack as a linked list

In §15.3, we explored how various operations on a linear list are implemented through functions. This provides a clear understanding of how a stack can be effectively implemented using a linked list. In this context, the top of the stack corresponds to the `head` of the list, with the `push` and `pop` operations realized as functions for the insertion and deletion of nodes at the beginning of the list. The code for stack based on linear linked list is given in Code 15.68.

**Code 15.68:** The function to search a node in a linked list.

`stack_pushpop_llist.c`

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 typedef struct Node {
5     int data;
6     struct Node *next; // Self-reference
7 } Node;
8
9 void push(Node **top, int value) {
10     Node *newNode = (Node *)malloc(sizeof(Node));
11     newNode->data = value;
12     newNode->next = *top;
13     *top = newNode;
14     printf("Pushed %d onto the stack.\n", value);
15 }
16
17 int pop(Node **top) {
18     if (*top == NULL) {
19         printf("Stack is empty. Cannot pop.\n");
20         return -1; // Return an invalid value or handle error appropriately
21     }
22     Node *temp = *top;
23     int poppedValue = temp->data;
24     *top = (*top)->next;
25     free(temp);
```



```
26     printf("Popped %d from the stack.\n", poppedValue);
27     return poppedValue;
28 }
29
30 int main() {
31     Node *stack = NULL; // Initialize an empty stack
32     char choice;
33     int value;
34
35     do {
36         printf("\nEnter 'P' to push, 'p' to pop, or 'e' to exit: ");
37         scanf(" %c", &choice); // Notice the space before %c to consume any leading whitespace
38
39         switch (choice) {
40             case 'P':
41                 printf("Enter a value to push: ");
42                 scanf("%d", &value);
43                 push(&stack, value);
44                 break;
45             case 'p':
46                 pop(&stack);
47                 break;
48             case 'e':
49                 printf("Exiting.\n");
50                 break;
51             default:
52                 printf("Invalid choice. Please try again.\n");
53         }
54     } while (choice != 'e');
55
56     return 0;
57 }
```

## 15.5 Queue as a linked list

The operations **enqueue** (to add an element) and **dequeue** (to remove an element) are implemented as insertion and deletion operations on the linked list. To optimize efficiency, we maintain an additional pointer, termed **rear**, which points to the last node of the list.

Dequeue operations occur at the front of the queue, tracked by the **head** pointer, while enqueue operations are performed at the rear end using the **rear** pointer. This design allows us to avoid traversing the entire list for each enqueue operation. The implementation of the queue based on a linear linked list is provided in Code 15.69.

**Code 15.69:** The function to search a node in a linked list.

`queue_enqdeq_llist.c`

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 typedef struct Node {
5     int data;
6     struct Node *next; // Self-reference
7 } Node;
8
9 void enqueue(Node **head, Node **rear, int value) {
```

```

10 Node *newNode = (Node *)malloc(sizeof(Node));
11 newNode->data = value;
12 newNode->next = NULL;
13
14 // If the queue is empty, both head and rear point to the new node
15 if (*rear == NULL)
16     *head = *rear = newNode;
17 else {
18     (*rear)->next = newNode; // Link the new node at the end of the queue
19     *rear = newNode;        // Move rear to point to the new node
20 }
21 printf("Enqueued %d to the queue.\n", value);
22 }
23
24 int dequeue(Node **head, Node **rear) {
25     if (*head == NULL) {
26         printf("Queue is empty. Cannot dequeue.\n");
27         return -1; // Return an invalid value or handle error appropriately
28     }
29
30     Node *temp = *head;
31     int dequeuedValue = temp->data;
32     *head = (*head)->next; // Move head to the next node
33
34     // If the queue becomes empty, reset rear to NULL
35     if (*head == NULL) {
36         *rear = NULL;
37     }
38
39     free(temp);
40     printf("Dequeued %d from the queue.\n", dequeuedValue);
41     return dequeuedValue;
42 }
43
44 int main() {
45     Node *head = NULL; // Initialize head of the queue
46     Node *rear = NULL; // Initialize rear of the queue
47     char choice;
48     int value;
49
50     do {
51         printf("\nEnter 'e' to enqueue, 'q' to dequeue, or 'x' to exit: ");
52         scanf(" %c", &choice); // Notice the space before %c to consume any leading whitespace
53
54         switch (choice) {
55             case 'e':
56                 printf("Enter a value to enqueue: ");
57                 scanf("%d", &value);
58                 enqueue(&head, &rear, value);
59                 break;
60             case 'q':
61                 dequeue(&head, &rear);
62                 break;
63             case 'x':
64                 printf("Exiting.\n");
65                 break;
66             default:
67                 printf("Invalid choice. Please try again.\n");
68         }

```

```

69     } while (choice != 'x');
70
71     return 0;
72 }

```

---

## 15.6 Solved problems

Unless mentioned, assume the following node structure for all problems here.

```

typedef struct Node {
    int data;
    struct Node* next;
} Node;

```

- 15.6.1 [Reverse a linked list]** Write a function to reverse a singly linked list. The function should take the head of the list as input and return the new head after reversing the list.

Example:

Original list: 1 -> 9 -> 4 -> 7 -> NULL

Reversed list: 7 -> 4 -> 9 -> 1 -> NULL

```

1 Node* reverseList(Node* head) {
2     Node* prev = NULL;
3     Node* current = head;
4     Node* next = NULL;
5
6     while (current != NULL) {
7         next = current->next;
8         current->next = prev;
9         prev = current;
10        current = next;
11    }
12    return prev;
13 }

```

- 15.6.2 [Merge two linked lists]** Write a function that merges two linked lists into a single linked list. The input will be the heads of the two linked lists, and the output should be the head of the first list, with the second list appended at the end. Assume both input lists are nonempty.

Example:

List 1: 1 -> 9 -> 4 -> 7 -> NULL

List 2: 0 -> 8 -> NULL

Merged List: 1 -> 9 -> 4 -> 7 -> 0 -> 8 -> NULL

```

1 Node* mergeLists(Node* head1, Node* head2) {
2     Node* tail = head1;
3     while (tail->next != NULL)
4         tail = tail->next;
5     tail->next = head2;
6     return head1;
7 }

```

- ♣ **15.6.3 [Polynomial addition]** Given two polynomials represented as linked lists where each node contains a coefficient and an exponent, write a function to add the two polynomials and return the resulting polynomial as a new linked list. For example, the polynomial  $3x^5 + 2x^2 + x + 1$  will have 4-node list for its three terms and  $x^9 + 4x^2$  will have a 2-node list for its two terms, and should return  $x^9 + 3x^5 + 6x^2 + x + 1$  in a linked list of five nodes. This is particularly useful when polynomials have large coefficients but just a few terms. Assume that the coefficients and exponents are all integers.

Example:

```
Enter the first polynomial:
Enter the number of terms in the polynomial: 4
Enter coefficient and exponent (e.g., 3 2 for 3x^2): 3 5
Enter coefficient and exponent (e.g., 3 2 for 3x^2): 2 2
Enter coefficient and exponent (e.g., 3 2 for 3x^2): 1 1
Enter coefficient and exponent (e.g., 3 2 for 3x^2): 1 0
Enter the second polynomial:
Enter the number of terms in the polynomial: 2
Enter coefficient and exponent (e.g., 3 2 for 3x^2): 1 9
Enter coefficient and exponent (e.g., 3 2 for 3x^2): 4 2
Resultant Polynomial: 1x^9 + 3x^5 + 6x^2 + 1x^1 + 1
```

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct Node {
5      int coef, exp;
6      struct Node* next;
7  } Node;
8
9  Node* createNode(int coef, int exp) {
10     Node* newNode = (Node*)malloc(sizeof(Node));
11     newNode->coef = coef, newNode->exp = exp, newNode->next = NULL;
12     return newNode;
13 }
14
15 void appendNode(Node** head, int coef, int exp) {
16     Node* newNode = createNode(coef, exp);
17     if (*head == NULL)
18         *head = newNode;
19     else {
20         Node* temp = *head;
21         while (temp->next != NULL)
22             temp = temp->next;
23         temp->next = newNode;
24     }
25 }
26
27 Node* addPolynomials(Node* poly1, Node* poly2) {
28     Node* result = NULL;
29     Node* p1 = poly1;
30     Node* p2 = poly2;
31
32     while (p1 != NULL && p2 != NULL) {
33         if (p1->exp == p2->exp) { // Exponents are the same, add coefficients

```

```
34     appendNode(&result, p1->coef + p2->coef, p1->exp);
35     p1 = p1->next;
36     p2 = p2->next;
37 } else if (p1->exp > p2->exp) { // Add the term from poly1
38     appendNode(&result, p1->coef, p1->exp);
39     p1 = p1->next;
40 } else { // Add the term from poly2
41     appendNode(&result, p2->coef, p2->exp);
42     p2 = p2->next;
43 }
44 }
45
46 // Append any remaining terms from poly1 or poly2
47 while (p1 != NULL) {
48     appendNode(&result, p1->coef, p1->exp);
49     p1 = p1->next;
50 }
51 while (p2 != NULL) {
52     appendNode(&result, p2->coef, p2->exp);
53     p2 = p2->next;
54 }
55
56 return result;
57 }
58
59 void printPolynomial(Node* head) {
60     Node* temp = head;
61     while (temp != NULL) {
62         if (temp->exp == 0) {
63             printf("%d", temp->coef);
64         } else {
65             printf("%dx^%d", temp->coef, temp->exp);
66         }
67         if (temp->next != NULL && temp->next->coef > 0) {
68             printf(" + ");
69         }
70         temp = temp->next;
71     }
72     printf("\n");
73 }
74
75 Node* createPolynomial() {
76     Node* head = NULL;
77     int n, coef, exp;
78
79     printf("Enter the number of terms in the polynomial: ");
80     scanf("%d", &n);
81
82     for (int i = 0; i < n; i++) {
83         printf("Enter coefficient and exponent (e.g., 3 2 for 3x^2): ");
84         scanf("%d %d", &coef, &exp);
85         appendNode(&head, coef, exp);
86     }
87 }
```

```

88     return head;
89 }
90
91 int main() {
92     printf("Enter the first polynomial:\n");
93     Node* poly1 = createPolynomial();
94     printf("Enter the second polynomial:\n");
95     Node* poly2 = createPolynomial();
96     Node* result = addPolynomials(poly1, poly2);
97     printf("Resultant Polynomial: ");
98     printPolynomial(result);
99     return 0;
100 }

```

## 15.7 Exercise problems

Unless mentioned, assume the following node structure for all problems here.

```

typedef struct Node {
    int data;
    struct Node* next;
} Node;

```

- 15.7.1 [Merge Two Sorted Linked Lists]** Write a function to merge two sorted linked lists into a single sorted linked list. The merged list can be a new list, requiring extra space. The input will be the heads of the two linked lists, and the output should be the head of the merged list. Assume neither input list is empty.
- 15.7.2 [Merging Two Sorted Linked Lists In-Place]** Improve your code for Problem **15.7.1** to merge two sorted linked lists into a single sorted linked list without using any additional space (other than pointers).
- 15.7.3 [Implement a doubly linked list]** Create a doubly linked list with the following functionalities: insert at the beginning, insert at the end, delete a node by value, and print the list in both forward and backward directions.
- 15.7.4 [Create a circular linked list]** Implement a function to create a circular linked list from an array of integers. The function should return the head of the new circular linked list.
- 15.7.5 [Rotate a doubly linked list]** Write a function to rotate a doubly linked list by  $k$  positions. The function should take the head of the list and the integer  $k$  as input and return the new head of the rotated list. For example, list  $3 \rightarrow 5 \rightarrow 8 \rightarrow 2 \rightarrow 1$  will become  $1 \rightarrow 3 \rightarrow 5 \rightarrow 8 \rightarrow 2$  if rotated by  $k = 1$ .
- 15.7.6 [Removing the  $n$ th node from the end]** Implement a function to remove the  $n$ th node from the end of a linked list. The function should handle cases where  $n$  is equal to the length of the list, as well as other scenarios. For example, for a linked list  $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$  and  $n = 2$ , the resulting list should be  $1 \rightarrow 2 \rightarrow 3 \rightarrow 5$ .
- 15.7.7 [Intersection of two linked lists]** Given two linked lists, determine the nodes at which the two lists intersect. Implement an efficient solution that traverses each list only once. The lists are singly linked and may have different lengths.
- ♣ **15.7.8 [Polynomial multiplication]** Implement a function to multiply two polynomials represented as linked lists. Each node should represent a term of the polynomial, as in Problem **15.6.3**. Assume that the coefficients and exponents are all integers.
- ♣ **15.7.9 [Reversing a linked list in groups]** Write a function to reverse a linked list in groups of a given size  $k$ . If the number of nodes is not a multiple of  $k$ , the remaining nodes at the end should remain

in their original order. For example, for a linked list  $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$  and  $k = 2$ , the resulting list should be  $2 \rightarrow 1 \rightarrow 4 \rightarrow 3 \rightarrow 5$ .

♣ **15.7.10 [Partitioning a linked list]** Given a linked list and a value  $x$ , partition the list such that all nodes less than  $x$  come before nodes greater than or equal to  $x$ . Assume that all have distinct data. The original relative order of nodes should be preserved. For example, for the linked list  $3 \rightarrow 5 \rightarrow 8 \rightarrow 7 \rightarrow 10 \rightarrow 2 \rightarrow 1$  and  $x = 5$ , the resulting list may be  $3 \rightarrow 2 \rightarrow 1 \rightarrow 5 \rightarrow 8 \rightarrow 7 \rightarrow 10$ .

♠ **15.7.11 [Detect a cycle in a linked list]** Write a function to determine if a given singly linked list contains a cycle (i.e., the last node points back to some node). The code is efficient if you do not use any auxiliary list but use a few variables only.

♠ **15.7.12 [Sparse matrix multiplication using linked lists]** A **sparse matrix** is one in which only a few elements are nonzero, and only those are represented by nodes in a linked list. Each node contains the following information:

- **row**: Row index of the element.
- **col**: Column index of the element.
- **val**: Value ( $\neq 0$ ) at the given row and column.

Given two compatible sparse matrices  $A$  and  $B$ , represented as linked lists, write a function that takes their heads as input, computes the product  $C = A \times B$ , and returns the head of the linked list for  $C$ .

As mentioned earlier, only non-zero entries should be stored in the linked list representation. You need to employ efficient traversal and multiplication for sparse structures, taking advantage of the linked list format to minimize operations on zero elements.

**Example:**

$$\underbrace{\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 2 & 0 & 3 \\ 0 & 0 & 0 & 5 \end{bmatrix}}_A \times \underbrace{\begin{bmatrix} 0 & 2 & 0 \\ 3 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}}_B = \underbrace{\begin{bmatrix} 0 & 6 & 3 \\ 12 & 0 & 8 \\ 15 & 0 & 0 \end{bmatrix}}_C$$

While running the code, the above input and output should look as follows:

Enter Matrix A:

```
0 0 1 0
0 2 0 3
0 0 0 5
```

Enter Matrix B:

```
0 2 0
3 0 0
0 1 0
```

Resulting Matrix C:

```
C =
(1, 2) -> 6
(1, 3) -> 3
(2, 1) -> 12
(2, 3) -> 8
(3, 1) -> 15
```

# Bibliography

- [1] R. L. Kruse, B. P. Leung, and C. L. Tondo. *Data Structures and Program Design in C*. Prentice-Hall of India Private Limited, New Delhi, 1991.
- [2] Y. Langsam, M. J. Augenstein, and A. M. Tenenbaum. *Data Structures using C and C++*. Prentice-Hall of India, New Delhi, 2000.