INTRODUCTION TO PROGRAMMING BY EXAMPLES

Atanas Hristov, Naum Tuntev and Ninoslav Marina

INTRODUCTION TO PROGRAMMING BY EXAMPLES

Atanas Hristov, Naum Tuntev and Ninoslav Marina

UNIVERSITY OF INFORMATION SCIENCE AND TECHNOLOGY "ST. PAUL THE APOSTLE" OHRID

FACULTY OF INFORMATION AND COMMUNICATION SCIENCES

INTRODUCTION TO PROGRAMMING BY EXAMPLES

ATANAS HRISTOV, NAUM TUNTEV, NINOSLAV MARINA

OHRID, 2022

Authors:
Assoc. Prof. Atanas Hristov, PhD
Naum Tuntev, MsC
Prof. Ninoslav Marina, PhD

Title of the issue:
Introduction to programming by examples

Editor of publishing activity of UINT:
Prof. Ivan Bimbilovski, PhD

Editor of the issue:
Assoc. Prof. Atanas Hristov, PhD

Reviewers:
Prof. Cvetko Andreeski, PhD
Assoc. Prof. Aneta Velkoska, PhD

Cover design:
Goran Kunovski

Technical processing:
Assoc. Prof. Atanas Hristov, PhD

Proofreading in English language:
Assis. Prof. Lela Ivanovska, PhD

Publisher:
University of Information Science and Technology

Place and year of issue:
Ohrid, 2022

Abstract

This book contains code examples adopted for undergraduate students that explain key structured programming concepts: variables, data structure, functions, one and two-dimensional arrays, pointers, and strings. The code examples are written in the C programming language.

By reading this book, students will learn techniques for designing programs and will understand how to model real-world problems and managing complexity. Moreover, students will gain problem-solving skills in the meaning of efficient and elegant problem solving and will acquire debugging and other special programming skills.

Good programming style is another skill that the students will gain by reading this book. They will learn how to write robust, efficient, and readable code and what is the correct way to document and comment on the code.

Finally, students will learn how to use the main data structures: variables, arrays, matrices, strings. They will get familiar with the scope of the variables and basic arithmetical and logical operations. Also, they will learn how to control the flow of the program, how to use functions and will gain special programming skill - recursion. In the end, they will learn how to work with pointers and strings.

Acknowledgements

I would like to acknowledge the help of all the people involved in the writing of this book, more specifically, the authors and reviewers that took part in the review process. Without their support, this book would not have become a reality.

My sincere gratitude goes to the other authors who contributed their time and expertise to this book.

I would like to acknowledge the valuable contributions of the reviewers regarding the improvement of quality, coherence, and content presentation of chapters. The authors also served as referees; I highly appreciate their double task.

Finally, I would like to thank the University of Information Science and Technology "St. Paul the Apostle" - Ohrid and its administration for their support in publishing this book.

by Atanas Hristov

Contents

Abstract vii

Acknowledgements ix

Contents xi

List of Figures xv

List of Tables xvii

1 Introduction 1

2 Data types and data representation 3
2.1 Data types in C . . . 3
2.2 Ranges . . . 5
2.3 Structure of C program . . . 6
2.4 Constants . . . 7
2.5 Variables . . . 9
2.6 Data representation on the screen . . . 10
2.7 Data entry from the keyboard . . . 11

3 Operations 15
3.1 Arithmetic operations . . . 15
3.2 Assignment operations . . . 17
3.3 Relational operations . . . 19
3.4 Logical operations . . . 20
3.5 Cast operations . . . 21

4 Selection structures 23
4.1 if and if/else statements . . . 24
4.2 Switch statement . . . 30

5 Repetition structures 35
5.1 While loop . . . 35

5.2 Do-while loop 41
5.3 For loop 44
5.4 Statements for controlling program execution 52

6 Functions **55**
6.1 Functions in C 55
6.2 Scope of the variables 56
6.3 Defining and using functions 58
6.4 Recursive functions 67
6.5 Function prototype 71

7 Arrays **75**
7.1 Arrays in C 75
7.2 Declaring and initialization of arrays 76
7.3 Accessing elements of arrays 77
7.4 Passing arrays as parameters to a function 81

8 Matrices **99**
8.1 Matrices in C 99
8.2 Declaring and initialization of matrices 99
8.3 Accessing elements and passing matrices as parameters to a function 102

9 Pointers **119**
9.1 Pointers in C 119
9.2 Operations with pointers 121
9.3 Using pointers to functions and arrays 123

10 Strings **137**
10.1 Strings in C 137
10.2 Entering and printing strings 139
10.3 String manipulation - *strng.h* library 140
10.4 Testing and mapping characters - *ctype.h* library 142

11 Files **155**
11.1 Working with files in C 155

12 Structures **163**
12.1 Structures in C 163
12.2 Accessing structure members and initialization 164
12.3 Array of structures and nested structures 165
12.4 Pointers to structures and structures to function 166

Appendices **175**

A Commonly used C library functions **177**
A.1 Implementation of the functions from *string.h* library 177
A.2 Implementation of the functions from *ctype.h* library 184

A.3 Functions for numeric conversions and random number generation - *stdlib.h* library 189
A.4 Commonly used functions from *math.h* library 192
A.5 Functions for manipulating date and time - *time.h* library 194

B Using command-line arguments in C **197**

C C keywords **199**

References **201**

Short excerpts from the reviewers **203**

Copyright **205**

List of Figures

2.1 Standard ASCII code table[1]. 4

4.1 Flow chart of the if statement. 25
4.2 Flow chart of the switch statement. 31

5.1 Flow chart of the while loop. 36
5.2 Flow chart of the do-while loop. 42
5.3 Flow chart of the for loop. 46
5.4 Flow chart of the break statement. 53
5.5 Flow chart of the continue statement. 54

7.1 An example of a one-dimensional array. 75

8.1 An example of a two-dimensional array. 99

9.1 Memory address concept. 119

10.1 Memory allocation when a string is initialized 138

List of Tables

2.1 Standard C data types 4
2.2 Ranges of the datatypes in C 5
2.3 Escape sequences in C 11
2.4 Type specifiers in C 12

3.1 Arithmetic operations 15
3.2 Shorthand assignment operators 18
3.3 Relational operators in C 19
3.4 Logical operators in C 20
3.5 Truth table of logical AND, OR, and NOT 20

10.1 Functions from *strng.h* library 141
10.2 Functions from *ctype.h* library 142

11.1 Functions for file management 156
11.2 File access modes 156

A.1 Functions from *stdlib.h* library 189
A.2 Functions from *math.h* library 192
A.3 Datatypes and functions from *time.h* library 194

C.1 List of keywords in C 199

CHAPTER 1

Introduction

This book introduces students with programming experience and problem-solving using the C programming language. It's written in a step-by-step tutorial style that makes programming available even to beginners. The book results from the author's years of experience in teaching introduction to programming.

The book contains lecture notes and code examples adopted for undergraduate students that explain key structured programming concepts: variables, data structure, functions, one and two-dimensional arrays, pointers, and strings. The code examples are written in the C programming language.

By reading this book, students will learn techniques for designing programs and will understand how to model real-world problems and managing complexity. Moreover, students will gain problem-solving skills in the meaning of efficient and elegant problem solving and will acquire debugging and other special programming skills.

Good programming style is another skill that the students will gain by reading this book. They will learn how to write robust, efficient, and readable code and what is the correct way to document and comment on the code.

Finally, students will learn how to use the main data structures: variables, arrays, matrices, strings. They will get familiar with the scope of the variables and basic arithmetical and logical operations. They will learn how to control the flow of the program, how to use functions and will gain special programming skill - recursion. In the end, they will learn how to work with pointers, strings and files.

Each chapter ends with exercises and proposed solutions. The authors collect exercises from their lecture notes, tutorials, labs and exams in the last ten years.

The book contains twelve chapters and three appendices. Chapter 2 provides an overview of basic data types and data representation. Chapter 3 describes basic arithmetic, assignment, relational, cast and logical operations. Chapter 4 describes the selection statements for choosing among two (if/else) and multiple(switch) statements. Chapter 5 describes the selection statements for choosing among two (if/else) and multiple(switch) statements. Chapter 6 explains the need to use functions in C, gives an overview of how to use functions, and introduces recursion. Chapter 7 presents the usage of one-dimensional arrays. Chapter 8 provides an overview of two-dimensional arrays (matrices) and explains how to work with them. Chapter 9 deals with memory addresses and introduce a special type of variables - pointers.

Chapter 10 describes how to work with an array of characters (strings) in C. Chapter 11 provides an overview of working with files. Chapter 12 introduces the concept of creating data structures and explains how to work with data structures. Finally, the book finishes with three appendices. Appendix A gives an overview and examples of commonly used C library functions. Appendix B describes how to use command-line arguments in C. Appendix C provides an overview of the C keywords.

CHAPTER 2

Data types and data representation

Data in computers are stored in a binary number system. The binary number system is a positional numeral system with two as its base and uses only the digits 0 and 1 to represent all data types. The binary number system follows the same set of mathematics rules as the commonly used decimal number system. The main reason behind using a binary number system is that digits 0 and 1 are stored as two voltage levels. On the other hand, every data can be represented with digits 0 and 1.

Each digit in the binary number system is called bit. The bits are the basic unit for storing information in digital information systems. For example, the number 01010111 is 8 bits long or one byte. The byte is the smallest addressable unit of memory; hence, it is the main unit to measure memory capacity.

2.1 Data types in C

In structural programming, a data type is an attribute of data that tells the compiler how the programmer intends to use it. The data type determines how data is stored in memory, which operations can be performed over the data, and which values can be set.

The data types can be classified as follows:

- Standard data types;
- Derived data types - user-defined, composite, etc.;
- Simple data types;
- Complex data types.

In the C programming language, there are four standard data types and four modifiers. The modifiers are used to change the range of the data type, i.e., to define how they are stored in the memory. The range of the data types will be discussed in detail in the next section. The standard data types, followed by the modified ones, are given in Table 2.1.

In the C programming language, **characters** (textual data) are encoded and stored as integers. The encoding is done based on the values in the **ASCII** (American

Data type	Description
int	Integer type
float	Single-precision floating-point type
double	Double-precision floating-point type
char	Character type
short or short int	Short integer type
long or long int	Long integer type
signed or signed int	Signed integer type
unsigned or unsigned int	Unsigned integer type
signed short	Signed short integer type
unsigned short	Unsigned short integer type
signed long	Signed long integer type
unsigned long	Unsigned long integer type
long double	Long double-precision floating-point type
signed char	Signed character type
unsigned char	Unsigned character type

Table 2.1: List of standard C data types

Standard Code for Information Interchange) code table. The standard *ASCII code table* is given on Figure 2.1.

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Figure 2.1: Standard ASCII code table[1].

[1] Source: https://simple.wikipedia.org/wiki/ASCII#/media/File:ASCII-Table-wide.svg

2.2 Ranges

The range in a set of numbers is the difference between the maximum and minimum values. In the C programming language, range refers to the value stored inside the variable of a given type. Besides the type of a variable, the range defers based on whether the variable is unsigned or signed.

The number of numerical values that can be represented by unsigned numbers with n-bits is 2^n. **The range of unsigned numbers** represented by n-bit words is calculated as follows:

$$0 \leqslant x \leqslant 2^n - 1$$

For example, if the number of bits n is 8 (1 byte), the range will be from 0 to 255. It means that there will be 2^8 (256) numerical values in the range.

The range of signed numbers represented by n-bit words is calculated as follows:

$$-2^{n-1} \leqslant x \leqslant 2^{n-1} - 1$$

For example, if the number of bits n is 8 (1 byte), the range will be from -128 to 127. It means that there will be 2^8 (256) numerical values in the range, including 0.

In table 2.2, ranges of some commonly used datatypes in C is given.

Datatype	**Size**	**Range**
char, unsigned char	1 byte (8 bits)	0 to 255
signed char	1 byte (8 bits)	-128 to 127
short, signed short	2 bytes (16 bits)	-32,768 to 32,767
unsigned short	2 bytes (16 bits)	0 to 65,535
int, signed int	4 bytes (32 bits)	-2,147,483,648 to 2,147,483,647
unsigned int	4 bytes (32 bits)	0 to 4,294,967,295
long, signed long	8 bytes (64 bits)	-9223372036854775808 to 9223372036854775807
unsigned long	8 bytes (64 bits)	0 to 18446744073709551615
float	4 bytes (32 bits)	1.2E-38 to 3.4E+38 Precision is 6 decimal places
double	8 bytes (64 bits)	2.3E-308 to 1.7E+308 Precision is 15 decimal places
long double	10 bytes (80 bits)	3.4E-4932 to 1.1E+4932 Precision is 19 decimal places

Table 2.2: Ranges of the datatypes in C

The above ranges are approximate and depend of the platform. Different hardware and software platforms can have different values for the datatypes ranges. There is a predefined operator in the C programming language that returns the exact size of the datatype on a particular platform. The operator's structure is *sizeof(type)* and returns the amount of memory allocated to that datatype in bytes.

The negative part of the range is stored in the memory as a two's complement of the positive value. The **two's complement** is calculated in two steps as follows:

- Step 1: Find the inverse binary value of the number. The inverse number is calculated by converting each bit into the opposite one. For example, the inverse value of 10011101 is 01100010.
- Step 2: Add 1 to the inverted number. In the example above, it means 01100010+1=01100011. Hence, the two's complement of 10011101 is 01100011.

An **overflow** occurs when an arithmetic operation attempts to create a numeric value outside of the range of the specific datatype. It is a programmer's responsibility to ensure that all intermediate and final results are in a datatype range. For example, if the datatype range is 1bytes (0 to 256), and you try to calculate 200 + 100, both operands (200 and 100) are in the range, but the result (300) will be out of the range.

2.3 Structure of C program

Every C program has similar basic parts. Some of the parts are mandatory; some are optional. The C program can be seen as a composition of one or many components, called functions. In the C programming language, every statement finishes with a semicolon "**;**". The compiler did not recognize the newline terminator. The basic structure of a C program is as follows:

The structure of C program

```
/* header - name, author, version */
/* INCLUDE section - contains #include statements */
/* Defining constants and data types - #define */
/* GLOBAL variables*/
/* definition of functions */
int main()
{
    /* declaration of variables */
    /* statements */
    return 0;
}
```

The first part of each program is the header of the program. The header usually is optional, in which the programmer provides information about the program.

Another important part of the program is the include section. The programmer includes all libraries with pre-defined functions that the programmer is going to use in the program in the include section.

In a defined section, the programmer defines his own constants and data types.

The programmer also needs to define the global variables and functions used in the program.

Each C program must have at least one function - ***main()***.

2.4 Constants

Constants are data that do not change their value during program execution, i.e., the value is constant.

In the C programming language, constants can be of any of the basic datatypes presented in Table 2.1. The compiler treats constants as a regular variable except that their values can not be modified after their definition. Based on their datatype, the constants can be classified as follows:

- **Integer constants**. In this group belongs all decimal integers data types, including modified versions: short, long, signed, and unsigned. To specify the modified version of an integer, the suffix L for long, U for unsigned, or a combination of both needs to be added. Also, this group covers octal and hexadecimal integers. The prefix 0 for Octal, or 0x or 0X for hexadecimal integer, needs to be added to specify the octal and hexadecimal constant.

  Examples:

  - 189 is integer.
  - 29L is long integer.
  - 398U is unsigned integer.
  - 199UL is unsigned long integer.
  - 0361 is octal integer.
  - 0x591AF7 is hexadecimal integer.
  - 0x123AAL is long hexadecimal integer.

- **Floating-point constants**. In this group belongs float and double datatypes. The floating-point constants are composed of: an integer part, a decimal point, a fractional part, and an exponent part. In the C programming language, floating-point constants can be represented either in decimal form or exponential form. To represent in decimal form, the programmer needs to include the decimal dot (.), the exponent, or both. To represent in exponential form, the programmer needs to include the integer part, the fractional part, or both. Floating-point constants without a suffix are from type double. The suffix L in floating-point constants refers to long double datatype. The suffix F refers to float datatype. The integer part of the floating-point constant can be omitted.

  Examples:

  - 31.123 is double.
  - 29.0L is long double.
  - 127.34F is float.
  - 2.435E1 is double. It is same as 24.35
  - 13E-3L is long double. It is same as 0.013L.
  - .05E1 is double. It is same as .5; 0.5; .005E2; 500E-3; etc.

- **Character constants**. This group includes char datatypes. Character constants are inserted in single quotes (' '). A character constant can be some character from the ASCII code table presented in Figure 2.1, some escape sequence from Table 2.3, or some universal character.

  Examples:

  - 'A' is a character from the ASCII code table.
  - '\n' is an escape sequence.
  - '\u02C0' is universal character.

- **String constants**. String constants are similar to character constants. They are composed of two or more characters, either from the ASCII code table, escape sequence, universal character, or a combination of the above. String constants are inserted in double quotes (" ").

  Examples:

  - "Some string" characters from the ASCII code table.
  - "Some string. \n This text goes in a new line."

In the C programming language, there are two ways to define constants: by using ***#define*** preprocessor directive and by using ***const*** keyword.When a constant is defined by using ***#define*** preprocessor directive, the name of the constant and the value should be provided. The type of the constant is automatically detected by the compiler based on the value provided by the programmer by following the above-explained methods. The structure of the ***#define*** preprocessor directive is as follows:

The structure of ***#define*** preprocessor directive

```
#define CONSTANTNAME value
```

where:
CONSTANTNAME is the name of the constant;
value is the value assigned to the constant.

When a constant is defined by using ***const*** keyword, the name, the type, and the value of the constant should be provided. The structure of the statement with a textbf*const* keyword is as follows:

The structure of a statement with ***#define*** keyword

```
const type CONSTANTNAME = value;
```

where:
type is the datatype of the constant;
CONSTANTNAME is the name of the constant;

value is the value assigned to the constant.

Additional comment: It is a good programming style to define constants in CAPITAL letters.

Examples:

```
#define SIZE 50  /* Will create an integer constant with value 50 */
#define NEWLINE '\n'  /* Will create a character constant with value \n */
#define PI 3.14  /* Will create a floating-point constant with value 3.14 */

const int SIZE = 50;  /* same as #define SIZE 50 */
const char NEWLINE = '\n';  /* same as #define NEWLINE '\n' */
const double PI = 3.14;  /* same as #define PI 3.14 */
```

2.5 Variables

A variable provides the user with named storage that programs can manipulate. The names of variables correspond to locations in the computer memory. Each variable in C has a type. The type determines the size and layout of the variable's memory, the range of values that it can store, and the set of operations applied to it. Whenever a new value is placed in a container (variable), the previous value is erased. Reading values from a variable do not change its value.

In the C programming language, variables are declared by defining the name and the type of the variable. The structure of a statement for declaring a variable is as follows:

The structure of a statement for declaring a variable

```
type variableName;
```

where:
type is the datatype of the variable;
variableName is the name of the variable;

Multiple variables from the same type can be declared separated by comma (,), as follows:

```
type name1, name2, name3, name4;
```

The name of the variable must begin with either a letter or an underscore. A variable name can only have letters, digits and underscore. Variable names are case-sensitive, i.e., *name1* and *Name1* are two different variables. The C keywords defined in table C.1 cannot be used as a variable name.

The variable must always be declared before it is used.
Additional comment: Always define the variable at the beginning of the main function or the beginning of a block of statements or outside a function.

To assign a value to a variable, an assignment operator (=) should be used. Assigning a value to a variable means storing a value to a variable. The structure of a statement for assigning a value to a variable is as follows:

```
variableName = value;
```

The assigning of the value to a variable can also be done during a declaration of the variable, as follows:

```
type variableName = value;
```

Examples:

```
int a;  /* Will declare an integer variable with name a */
double b = 10; /* Will declare and initialize a variable b with value 10 */
float c, d = 7.2;

c = 15.8 /* Will assign value 15.8 to c */
b = 11.5 /* Will assign value 11.5 to b. The old value is deleted */
```

2.6 Data representation on the screen

There is a predefined function for a visual representation of data on screen in the C programming language. The function is part of the *"stdio.h"* library. The structure of the function is as follow:

The structure of *printf()* function

```
printf(format, exp1, exp2, ...);
```

where:
format is a text shown on the screen
exp1, exp2,... are individual values that are shown on the screen

For better output and formatting of the text on the screen, the C language supports several escape sequences. The escape sequences are sequences of characters that produce some effect in the C programming language. In table 2.3, a list of some commonly used escape sequences is given, together with a description of them.

Exercises:

1. Write a C program that will display Hello World! on the screen.

Solution:

```
#include <stdio.h>

int main(){
```

Escape sequence	Description
\n	New line
\t	Tab
\b	Backspace
\r	Carriage Return
\"	Quotation mark
\'	Single quote
\\	Backslash

Table 2.3: List of escape sequences

```
    printf("Hello World!");
    return 0;

}
```

2. Write a C program that will create and initialize three variables, one real number, one decimal point number, and one single character. Use a printf() function to display the values of these variables on the screen.

Solution:

```
#include <stdio.h>

int main(){

    int real = 10;
    double decimal = 15.2;
    char singleChar = 'c';

    printf("The real number is %d\n", real);
    printf("The decimal point number is %.2f\n", decimal);
    printf("The character is %c\n", singleChar);

    return 0;

}
```

2.7 Data entry from the keyboard

There is a predefined function for entering data from the keyboard in C programming language. The function is a part of the *"stdio.h"* library. The structure of the function is as follows:

The structure of **scanf() function**

```
scanf(format, &val1, &val2, ...);
```

where:
format is a text containing the information for the data that will be entered
& mark referees that the value of the variables val1, val2, ... will be hanged by values from the keyboard;
val1, val2, ... are variables that will store the data entered from the keyboard.

A list of some commonly used data types specifiers is given in Table 2.4.

Type specifier	Data type
%c	Character (char)
%d	Integer (int)
%f	Double and float
%h	Short integer (short)
%l	Long integer (long)
%u	Unsigned integer (unsigned)
%o	Octal notation (octal)
%x	Hexadecimal (hexadecimal)

Table 2.4: List of data types specifiers

Exercises:

1. Write a C program that read three numbers from the keyboard and print their sum on the screen.

Solution:

```c
#include <stdio.h>

int main(){

    int a, b, c;
    int sum;

    printf("Enter the numbers: ");
    scanf("%d %d %d", &a, &b, &c);

    sum = a + b + c;

    printf("The total sum is %d\n", sum);

    return 0;

}
```

2. Write a C program that calculates the area of a rectangle. The sides of the rectangle are entered from the keyboard.

Solution:

```c
#include <stdio.h>

int main(){

    float a, b;
    float area;

    printf("Enter the sides of the rectangle: ");
    scanf("%f %f", &a, &b);

    area = a * b;

    printf("The area of a rectangle with sides %.2f and %.2f is %.2f\n
         ", a, b, area);

    return 0;

}
```

3. Write a C program that calculates the average of five integers entered by the user.

Solution:

```c
#include <stdio.h>

int main(){

    int n1, n2, n3, n4, n5;
    float average;

    printf("Enter the numbers: ");
    scanf("%d %d %d %d %d", &n1, &n2, &n3, &n4, &n5);

    average = (n1 + n2 + n3 + n4 + n5)/5.0;

    printf("The average is %.2f\n", average);

    return 0;

}
```

CHAPTER 3

Operations

In this chapter, the basic operations in the C programming language, and the rules for creating and using them will be presented. The presented operations are as follows:

- Arithmetic operations;
- Assignment operations;
- Relational operations;
- Logical operations and;
- Cast operations.

3.1 Arithmetic operations

In the C programming language, the arithmetic operator is a symbol that performs mathematical operations on a value or a variable. Those operations include mathematical operations on numerical values such as addition, subtraction, multiplication, division, modulo, etc. The C programming language has a wide range of operators to perform various operations. Some of the most common operator used in C and their equivalence with the corresponding mathematical operation is given in Table 3.1.

Operation	**Operator**	**C example**
Addition	+	a + b
Subtraction	-	a - b
Multiplication	*	a * b
Division	/	a / b
Modulo	%	a % b

Table 3.1: Arithmetic operations in C

.

The arithmetic operations in C following the same rules as in mathematics. The operations multiplication, division, and modulo are with higher priority, and they are calculated first. The addition and subtraction are with lower priority, and they are calculated last. If there are more operations with the same priority, they are executed from left to right. The brackets are used to change the priority of the operations.

The operand in the arithmetic operations can be from the same type or a different type. If the variables are from different types, the result will be in the type with a wider range.

Examples:

- int + double = double
- double + float = double
- float * int = float
- char + int = int + int = int

C example with arithmetic operation

```
int main()
{
    int whole;
    float real;
    real = 1.0 / 2.0;
    whole = 1 / 3;
    real = (1 / 2) + (1 / 2);
    real = 3.0 * 2.0;
    whole = real;
    whole = whole +1;
    return (0);
}
```

Exercises:

1. Write a C program that converts Celsius into Fahrenheit degrees. The Celsius degrees are entered from the keyboard. The conversion formula is: f = 1.8C + 32

Solution:

```
#include <stdio.h>

int main(){

    float celsius, fahrenheit;

    printf("Enter the Celsius degrees: ");
    scanf("%f", &celsius);

    fahrenheit = 1.8 * celsius + 32;
```

```
    printf("%.1f Celsius are %.1f Fahrenheit\n", celsius, fahrenheit);

    return 0;

}
```

2. Write a C program that asks the user to enter the amount of money, debt period (in years), and the interest rate per year. The program should tell the user how much money he will return in total at the end of the debt period.

   Solution:

```
#include <stdio.h>

int main(){

    float amount, interest;
    int years;

    float result;

    printf("How much money? ");
    scanf("%f", &amount);

    printf("How many years? ");
    scanf("%d", &years);

    printf("Interest rate per year (percentage): ");
    scanf("%f", &interest);

    result = amount + (amount * interest / 100) * years;

    printf("Total money to return after %d years is %.2f\n", years,
         result);

    return 0;

}
```

3.2 Assignment operations

Assignment operators are used for assigning a value to a variable. In the C programming language, the assign operator is =.
Example: a=5;.

The assignment operator returns a value that can be assigned to another variable. The assignment operator is flexible. In this line of code:

```
int a, b, c, d;
a=b=c=d=10;
```

d=10 is the first operation that is executed, making the value 10 available to the next assignment operation of assigning 10 to c, etc.

In the C programming language, the assignment operators can also be applied to assign the result of an expression to a variable, i.e., to shorthand the operators. Some of the most common shorthand operator used in C and their equivalence with the corresponding mathematical operation is given in Table 3.2.

Operator	**Example**	**Equivalent**
+=	a+=2;	a=a+2;
-=	a-=2;	a=a-2;
*=	a*=2;	a=a*2;
/=	a/=2;	a=a/2;
%=	a%=2;	a=a%2;
++	a++; or ++a;	a=a+1;
--	a--; or --a;	a=a-1;

Table 3.2: Shorthand assignment operators in C

. Operators ++ is called increment, and the operator -- is called decrement. Increment increases the value of the variable by 1. Decrement, decrease the value of the variable by 1. The increment and decrement operators can be used as a prefix or as a postfix. If they are used as a prefix, firstly, the value of the variable is changed by 1, and after that, it returns the result. If they are used as a postfix, firstly, the value of the variable is returned, and after that, it is changed by 1.

Exercises:

1. What will be the output from the following program?

```c
#include <stdio.h>
int main()
{
    int a=1;
    printf ("a = %d\n", a);
    printf ("a = %d\n", ++a);
    printf ("a = %d\n", a);
    return 0;
}
```

Solution: The program will print on the screen:

```
a = 1
a = 2
a = 2
```

2. What will be the output from the following program?

```c
#include <stdio.h>
int main()
```

```
{
    int a=1;
    printf ("a = %d\n", a);
    printf ("a = %d\n", a++);
    printf ("a = %d\n", a);
    return 0;
}
```

Solution: The program will print on the screen:

```
a = 1
a = 1
a = 2
```

3.3 Relational operations

Relational operators are used to comparing two operands. If the result of the comparison is true, it returns value 1; if the relation is false, it returns value 0. Relational operators are mainly used in selection structures and repetitions. Some of the most common relational operators used in C and their meaning is given in Table 3.3.

Operator	Example	Meaning
>	a>b;	a is **greater than** b
<	a<b;	a is **less than** b
>=	a>=b;	a is **greater or equal to** b
<=	a<=b;	a is **less or equal to** b
==	a==b;	a is **equal to** b
!=	a!=b;	a is **not equal to** b

Table 3.3: Relational operators in C

Please note that there is no boolean type in the C programming language. Hence, every nonzero value is true. The value 0 means false, including 0.0, -0, +0, etc.

Examples:

- 5>6 is false.
- 3==3 is true.
- 20!=20 is false.
- 8<8 is false.

3.4 Logical operations

Logical operators are used for connecting two or more operands to create a logical expression. If the logical expression is true, the compiler will return 1. If the logical expression is false, the compiler will return 0. The logical expressions are commonly used to test expression that controls program flow. In the C programming language, there are three common logical operators. A description of these operators is given in Table 3.4.

Operator	Operation	Description
&&	Logical AND	The expression's value will be true (nonzero) if and only if all operands are true.
||	Logical OR	The expression's value will be true (nonzero) if at least one operand is true.
!	Logical negation (NOT)	The expression's value will be true (nonzero) if the operand is false.

Table 3.4: Logical operators in C

.

A common way to present the evaluation of logical expressions is by using truth tables. The truth table of logical AND, OR, and NOT is given in Table 3.5.

A	B	A && B	A || B	!A
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

Table 3.5: Truth table of logical AND, OR, and NOT

.

In the C programming language, all logical expressions are evaluated from left to right. Evaluation is performed as long as the compiler is not certain of the result. For example, for a=50, when evaluating (a<20) && (a>5) only the first part (a<20) will be evaluated. Since the evaluation result of the first part is false, the further evaluation will stop, and the compiler will return 0 (false).

When we evaluate the logical expressions, the following rules regarding the **priorities of the logical operators** should be taken into consideration:

- Operator **! - NOT** is with the highest priority, and it is executed first. Also, its priority is higher than multiplication and division. It has the same priority as increment and decrement operations.

- Operator **&& - AND** has higher priority than **|| - OR**, and it is executed second. Both operators, && and ||, have lower priority than relational operators.

3.5 Cast operations

Cast operation refers to converting a variable of one data type into another. The compiler will automatically change one type of data into another if it makes sense. For example, if you assign an integer value to a decimal variable, the compiler will convert the int to double.

Format of the cast operation

```
(data type)value;
```

Examples:

- (int)6.72 will convert 6.72 into 6.
- (double)62 will convert 62 into 62.0.
- (char)87 will convert 87 into W (87 is ASCII code for the W character).
- (long)337.23 will convert 337.23 into 337L.

C example with cast operation

```
int i;
double d = 6.28;
i = (int) d;
```

Exercises:

1. What will be the output from the following program?

```
#include <stdio.h>

int main()
{
    int i = 65;
    float f = 22.8;
    char c;
    c = (char)i + (char)f;
    printf("c = %c \n",c);
    return 0;
}
```

Solution: The program will print on the screen: c = W

2. What will be the output from the following program?

```c
#include <stdio.h>

int main()
{
    int a=10, b=4;
    double c;
    c = (double)a/b;
    printf("c = %f, ",c);
    c = a / (double)b;
    printf("c = %f, ",c);
    c = (double)a / (double)b;
    printf("c = %f, ",c);
    c = (double)(a/b);
    printf("c = %f, ",c);
    return 0;
}
```

Solution: The program will print on the screen:
c = 2.5, c = 2.5, c = 2.5, c = 2.0,

CHAPTER 4

Selection structures

In structured programming, the program can be defined as a sequence of steps. Generally, the steps are executed one after another in a strictly predefined order by the programmer. In some special cases, it is also possible to transfer the control, i.e., the next statement being executed is NOT the next in the given order.

In the Böhm–Jacopini theorem from the programming language theory, it is stated that all programs can be written with three control structures. These structures are:

1. **Sequential structures:** expressions are executed sequentially one after another if there is no transfer control.

2. **Selection structures:** choosing among one of two subprograms according to the value of a boolean expression, i.e., *if/else*; or choosing among multiple subprograms according to the value of a boolean expression, i.e., *switch/case*.

3. **Repetition structures:** the subprogram is executed as long as a boolean expression is true.

The sequential structures can be single expressions, a block of statements, or a linear structure. The sequential statements are always executed one after another.

In the C programming language, the sequential statements are separated from the rest of the code with brackets {...}.

Example of sequential structure

```
{
     double temp, a=0, b=15;
     temp=a;
     a=b;
     b=temp;
}
```

4.1 if and if/else statements

The *if* structure enables choosing among one of two statements or block of statements according to the value of a boolean expression. The general structure of the *if* statement is as follows:

Structure of the if statement

```
if (condition)
    statement_for_a_true_condition;
else
    statement_for_a_false_condition;
```

In case of a block of statements, the beginning and the end of the block are marked with brackets. The general structure of the *if* statement in case of block of statements is as follows:

Structure of the if statement in case of block of statements

```
if (condition)
    {
    block_of_statements_for_a_true_condition;
    }
else
    {
    block_of_statements_for_a_false_condition;
    }
```

The *else* part is not mandatory. In case the *else* part is not necessary, the structure of the *if* statement as follows:

Structure of the if statement without else part

```
if (condition)
    statement_for_a_true_condition;
```

In case of a block of statements, the beginning and the end of the block are marked with brackets. The general structure of the *if* statement in case of block of statements is as follows:

Structure of the if statement in case of block of statements without else part

```
if (condition)
    {
    block_of_statements_for_a_true_condition;
    }
```

The **condition** in the brackets can be any arithmetical or logical expression.

The flow diagram of the *if* statement is given on Figure 4.1.

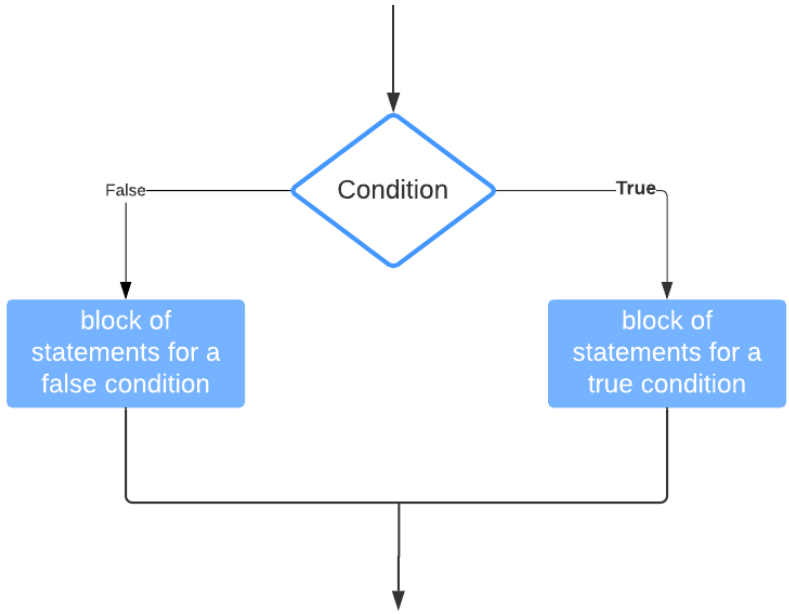

Figure 4.1: Flow chart of the if statement.

Multiple *if* statements can be nested together to allow multiple different criteria. **Nested *if* statement** is when one *if* statement is inside another, allowing to the programmer to test multiple criteria. Creating a nested *if* statement will increase the number of possible outcomes. The general structure of the nested *if* statement is as follows:

Structure of the nested if statement

```
if (condition1)
    {
    block_of_statements_for_a_true_condition1;
    }
else
    if (condition2)
    {
    block_of_statements_for_a_true_condition2;
    }
    else
    {
    block_of_statements_for_a_false_condition2;
    }
```

Exercises:

1. Write a C program that checks whether the letter entered by the user is a vowel.

   Solution:

```c
#include <stdio.h>

int main(){

    char c;

    printf("Enter the character: ");
    scanf("%c", &c);

    if(c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u' || c
        == 'y'){
        printf("The character is a vowel\n");
    }
    else{
        printf("The character is NOT a vowel\n");
    }

    return 0;

}
```

2. Write a C program that asks the user to enter the points that he gains on the exam and return some message based on the input:

   - If the input is in the range 51-100, print "You pass the exam."
   - If the input is in the range 0-50, print "You fail the exam."
   - For any other number, print "The points that you entered are not correct."

   Solution:

```c
#include <stdio.h>

int main(){

    int points;

    printf("Enter the exam points: ");
    scanf("%d", &points);

    if(points >= 0 && points <= 50){
        printf("You failed the exam.\n");
    }
    else if(points >= 51 && points <= 100){
        printf("You passed the exam.\n");
    }
    else{
        printf("The points you entered are not correct.\n");
    }
```

```
    return 0;

}
```

3. Write a C program that checks whether a number entered from the keyboard is even or odd.

   Solution:

```
#include <stdio.h>

int main(){

    int number;

    printf("Enter the number: ");
    scanf("%d", &number);

    if(number % 2 == 0){
        printf("The number is even\n");
    } else{
        printf("The number is odd\n");
    }

    return 0;

}
```

4. Write a C program that asks the user to enter a number and print: “positive” if the number is positive, “negative” if the number is negative, “zero” if the number is zero.

   Solution:

```
#include <stdio.h>

int main(){

    int number;

    printf("Enter the number: ");
    scanf("%d", &number);

    if(number > 0){
        printf("Positive number\n");
    }
    else if(number < 0){
        printf("Negative number\n");
    }
    else{
        printf("Zero\n");
```

```
    }

    return 0;

}
```

5. Write a C program that checks if the year is a leap or not.

   Solution:

```
#include <stdio.h>

int main(){

    int year;

    printf("Enter the year: ");
    scanf("%d", &year);

    if(((year % 4 == 0) && (year % 100 != 0)) || (year % 400 == 0)){
        printf("%d is a leap year", year);
    }else{
        printf("%d is a NOT leap year", year);
    }

    return 0;

}
```

6. Write a C program that finds the smallest of three numbers entered from the keyboard.

   Solution:

```
#include <stdio.h>

int main(){

    float n1, n2, n3;

    printf("Enter the numbers: ");
    scanf("%f %f %f", &n1, &n2, &n3);

    if(n1 < n2) {
        if(n1 < n3){
            printf("The smallest number is %.2f\n", n1);
        } else {
            printf("The smallest number is %.2f\n", n3);
        }
    } else {
        if(n2 < n3) {
            printf("The smallest number is %.2f\n", n2);
```

```
        } else {
            printf("The smallest number is %.2f\n", n3);
        }
    }

    return 0;

}
```

7. Write a C program that asks the user to enter three numbers and checks whether they can be sides of a triangle or not.

   Solution:

```
#include <stdio.h>

int main(){

    float a, b, c;

    printf("Enter the numbers: ");
    scanf("%f %f %f", &a, &b, &c);

    if((a + b) > c && (a + c) > b && (b + c) > a){
        printf("The numbers can be sides of triangle\n");
    } else {
        printf("The numbers can NOT be sides of triangle\n");
    }

    return 0;

}
```

8. Write a C program that asks the user to enter the price of the product and calculate the new discounted price based on the following criteria:

   - If the price is less than 1.000 denars - 0% discount,
   - If the price is in the range of 1.001-10.000 denars - 5% discount
   - If the price is over 10.001 denars - 10% discount

   Solution:

```
#include <stdio.h>

int main(){

    int price;
    int discount;

    printf("Enter the price of the product: ");
    scanf("%d", &price);
```

```
    if(price <= 1000){
        discount = 0;
    } else if(price > 1000 && price <= 10000){
        discount = 5;
    } else if(price > 10000){
        discount = 10;
    }

    printf("The discount price is: %d\n", (price - (price * discount)
         / 100));

    return 0;

}
```

4.2 Switch statement

The *switch* structure enables choosing among multiple statements or block of statements according to the value of a boolean expression. The general structure of the *switch* statement is as follows:

Structure of the switch statement

```
switch(expresion)
{
    case constant1: block_statements1; break;
    case constant2: block_statements2; break;
    ...
    case constantN: block_statementsN; break;
    default:        block_statements;
}
```

Note: The result of **expression** must be from type int or char.

When a *switch* statement is created, the following rules and conditions must be taken into account:

- There are no two *case* expressions with the same value.
- The program continues with the evaluation after the case statement by receiving the value of the evaluated expression from the switch statement.
- All further statements are executed until a *break* statement has been reached or until the last *case* block.
- If there is no *case* statement with a given value, the default block's statements are executed.
- If there is no *default* block, then none of the statements will be executed. The execution will continue from the end of the *switch* statement.

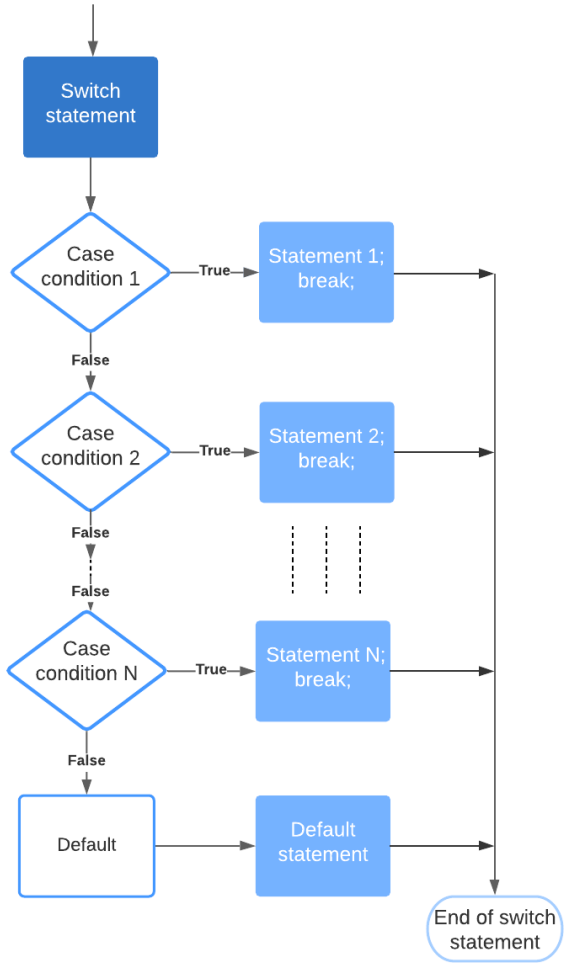

Figure 4.2: Flow chart of the switch statement.

The flow diagram of the *switch* statement is given on figure 4.2.
Exercises:

1. What does this program do?

```
#include <stdio.h>
int main()
{
  char c;
  c = getchar();
  switch (c)
  {
    case '0': printf("zero");  break;
    case '1': printf("one");   break;
    case '2': printf("two");   break;
```

```
    case '3': printf("three"); break;
    case '4': printf("four");  break;
    case '5': printf("five");  break;
    case '6': printf("six");   break;
    case '7': printf("seven"); break;
    case '8': printf("eight"); break;
    case '9': printf("nine");  break;
    default:  printf("not a digit");
  }
return(0);
}
```

Solution: This program reads a character from the keyboard, and if the input is a digit, it will print the name of the digit. In case the input is not a digit, it will print "not a digit" on the screen.

2. Write a C program that asks the user to enter the month number and print the month's name on the screen. If the number is not in the range of 1-12, print "Incorrect number.".

Solution:

```
#include <stdio.h>

int main(){

    int month;
    char monthName[10];

    printf("Enter month number: ");
    scanf("%d", &month);

    switch(month){
        case 1:
            printf("Month name is January");
            break;
        case 2:
            printf("Month name is February");
            break;
        case 3:
            printf("Month name is March");
            break;
        case 4:
            printf("Month name is April");
            break;
        case 5:
            printf("Month name is May");
            break;
        case 6:
            printf("Month name is June");
            break;
        case 7:
            printf("Month name is July");
            break;
        case 8:
```

```
            printf("Month name is August");
            break;
        case 9:
            printf("Month name is September");
            break;
        case 10:
            printf("Month name is October");
            break;
        case 11:
            printf("Month name is November");
            break;
        case 12:
            printf("Month name is December");
            break;
        default:
            printf("Incorrect number");
    }

    printf("\n");

    return 0;

}
```

3. Write a C program that implements a simple calculator (no priorities, only integers)

Solution:

```
#include <stdio.h>

int main(){

    char operator;
    float nr1, nr2, result = 0;

    printf("Enter operator ('+', '-', '*', '/'):");
    scanf("%c", &operator);

    printf("Enter two numbers:");
    scanf("%f %f", &nr1, &nr2);

    switch (operator) {
        case '+':
            result = nr1 + nr2;
            printf("The result of the operation is: %.2f %c %.2f = %f"
                 , nr1,
            operator, nr2, result);
            break;
        case '-':
            result = nr1 - nr2;
            printf("The result of the operation is: %.2f %c %.2f = %f"
                 , nr1,
            operator, nr2, result);
            break;
```

```
        case '*':
            result = nr1 * nr2;
            printf("The result of the operation is: %.2f %c %.2f = %f"
                 , nr1,
            operator, nr2, result);
            break;
        case '/':
            if (nr2 == 0) {
                printf("Error: Division with 0.\n");
            }
            else {
                result = nr1 / nr2;
                printf("The result of the operation is: %.2f %c %.2f =
                      %.2f", nr1,
                operator, nr2, result);
            }
            break;
        default:
            printf("Unknown operator %c\n", operator);
            break;
        }

    printf("\n");

    return 0;

}
```

CHAPTER 5

Repetition structures

In the programming language theory, repetition structures are structures where a statement or a block of statements are repeated as long as a boolean expression is true. The repetition structures are also called iterative structures, looping structures, or loops.

In the C programming language, there are three types of loops:

- While loop.
- Do-while loop.
- For loop.

While and for loop, check the boolean expression on the entry. The do-while loop checks the boolean expression on the exit.

All three types loops are working almost similar. However, they can be used in different scenarios. It is up to the programmer to choose the loop based on the requirement of the program.

5.1 While loop

In the *while* loop, the boolean expression is tested at the beginning (before entering the loop). The statements of the loop are repeated multiple times, or never at all.

The general structure of the *while* loop is as follows:

Structure of the while loop

```
while (expresion)
    statement;
```

In case of a block of statements, the beginning and the end of the block are marked with brackets. The general structure of the *while* loop in case of block of statements is as follows:

Structure of the while loop in case of block of statements

```
while (espresion)
{
    block_statements;
}
```

The flow diagram of the *while* loop is given on Figure 5.1.

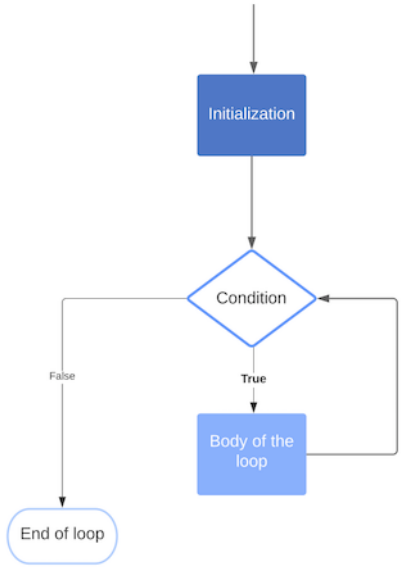

Figure 5.1: Flow chart of the while loop.

Exercises:

1. Write a C program that calculates the factorial of a number.

Solution:

```
#include <stdio.h>

int main(){

    int num, digit, temp;
    int factorial = 1;

    printf("Enter the number: ");
    scanf("%d", &num);

    temp = num;

    while(temp > 0){
        factorial *= temp;
        temp--;
    }
```

```
    printf("The factorial of %d is %d\n", num, factorial);

    return 0;

}
```

2. Write a C program that counts how many dividers have a number entered by the user.

   Solution:

```
#include <stdio.h>

int main(){

    int num, totalDivisors = 0;
    int i = 2;

    printf("Enter the number: ");
    scanf("%d", &num);

    while(i < num){
        if(num % i == 0){
            totalDivisors++;
        }
        i++;
    }

    printf("%d has %d divisors\n", num, totalDivisors);

    return 0;

}
```

3. Write a C program that will find the sum of digits from a number entered by the user.

   Solution:

```
#include <stdio.h>

int main(){

    int num, sumOfDigits = 0, temp;
    int digit;

    printf("Enter the number: ");
    scanf("%d", &num);

    temp = num;
```

```
    while(temp > 0){
        digit = temp % 10;
        sumOfDigits += digit;
        temp /= 10;
    }

    printf("The sum of digits of %d is %d\n", num, sumOfDigits);

    return 0;

}
```

4. Write a C program that from an unknown number of numbers will print the total sum of all number's most significant digit.

   Solution:

```
#include <stdio.h>

int main(){

    int num, sum = 0;
    int digit;

    printf("Start entering numbers. Enter any non-digit character to
         teminate.\n");

    while (scanf("%d", &num)){
        while (num > 10){
            num /= 10;
        }
        sum += num;
    }

    printf("The total sum of all most significant digits is %d\n", sum
         );

    return 0;

}
```

5. Write a C program that checks if the number can be divided with its sum of even digits.

   Do not use %2 in your code, and solve the problem using a WHILE loop.

   Solution:

```
#include <stdio.h>

int main(){
```

```
    int num, sumOfEvenDigits = 0, temp;
    int digit;

    printf("Enter the number: ");
    scanf("%d", &num);

    temp = num;

    while(temp > 0){
        digit = temp % 10;

        if((digit / 2) * 2 == digit){
            sumOfEvenDigits += digit;
        }

        temp /= 10;
    }

    if(sumOfEvenDigits > 0 && num % sumOfEvenDigits == 0){
        printf("The number can be divided with it\'s sum of even
            digits\n");
    } else {
        printf("The number can\'t be divided with it\'s sum of even
            digits\n");
    }

    return 0;

}
```

6. Write a C program that from an unknown number of numbers will print and count the numbers larger than 10000 that satisfy the condition: the first two digits are smaller than the last two digits.

   Solution:

```
#include <stdio.h>

int main(){

    int num, firstTwo, lastTwo, count = 0, temp;

    printf("Start entering numbers. Enter any non-digit character to
        teminate.\n");

    while (scanf("%d", &num)){

        if(num > 10000){
            lastTwo = num % 100;
        }

        firstTwo = num;

        while (firstTwo > 100){
            firstTwo /= 10;
```

```
        }

        if(firstTwo < lastTwo){
            printf("Number found: %d\n", num);
            count++;
        }
    }

    printf("The total count is %d\n", count);

    return 0;

}
```

7. Write a C program that, from an unknown number of numbers, will print the total sum of all numbers factorials.

   Solution:

```
#include <stdio.h>

int main(){

    int num, totalSum = 0, factorial;

    printf("Start entering numbers. Enter any non-digit character to
        teminate.\n");

    while (scanf("%d", &num)){

        factorial = 1;

        while(num > 0){
            factorial *= num;
            num--;
        }

        totalSum += factorial;
    }

    printf("The total sum is %d\n", totalSum);

    return 0;

}
```

8. Write a program that from an unknown number of numbers will print and count all numbers greater than 100, that its sum of digits is divisible with the number of digits.

   Solution:

```c
#include <stdio.h>

int main(){

    int num, sumOfDigits, numOfDigits, count = 0, temp;

    printf("Start entering numbers. Enter any non-digit character to
         teminate.\n");

    while (scanf("%d", &num)){

        if(num > 100){
            temp = num;
            sumOfDigits = numOfDigits = 0;

            while (temp > 0){
                sumOfDigits += temp % 10;
                numOfDigits++;
                temp /= 10;
            }

            if(sumOfDigits % numOfDigits == 0){
                printf("Number found: %d\n", num);
                count++;
            }
        }

    }

    printf("The total count is %d\n", count);

    return 0;

}
```

5.2 Do-while loop

The *do-while* loop is similar to the while loop. The difference is that the boolean expression is tested at the end of the loop. That means the body of the loop, i.e., the statement or the block of statements, is executed at least once.

The general structure of the *do-while* loop is as follows:

Structure of the do-while loop

```c
do
    statement;
while (expresion);
```

In case of a block of statements, the beginning and the end of the block are marked with brackets. The general structure of the *do-while* loop in case of block of statements is as follows:

Structure of the do-while loop in case of block of statements

```
do
{
    block_statements;
}
while (expresion);
```

The flow diagram of the *do-while* loop is given on Figure 5.2.

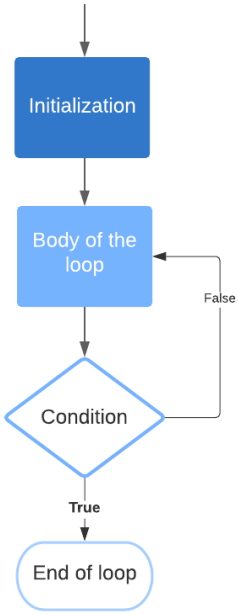

Figure 5.2: Flow chart of the do-while loop.

Exercises:

1. Write a C program that asks the user to enter numbers until they enter something different from a number. The program should display the largest and smallest entered numbers.

Solution:

```
#include <stdio.h>

int main(){

    int num, min, max;

    printf("Start entering numbers. Enter any non-digit character to
        teminate.\n");
```

```
    scanf("%d", &num);

    min = max = num;

    do {
        if(num < min){
            min = num;
        }

        if(num > max){
            max = num;
        }
    } while(scanf("%d", &num));

    printf("The minumin number is %d, and the maximum number is %d\n",
          min, max);

    return 0;

}
```

2. Write a C program that reverses a number entered from the keyboard.

   Solution:

```
#include <stdio.h>

int main(){

    int num, digit, reversed = 0, temp;

    printf("Enter the number: ");
    scanf("%d", &num);

    temp = num;

    do {
        digit = temp % 10;
        reversed = reversed * 10 + digit;

        temp /= 10;
    } while(temp > 0);

    printf("The reversed number of %d is %d\n", num, reversed);

    return 0;

}
```

3. Write a C program that from an unknown number of numbers will count and print all numbers that satisfy the condition: the number is divisible by the sum of its first and last digit.

Solution:

```c
#include <stdio.h>

int main(){

    int num, firstDigit, lastDigit, count = 0;

    printf("Start entering numbers. Enter any non-digit character to
         teminate.\n");

    while (scanf("%d", &num)){

        lastDigit = num % 10;

        firstDigit = num;

        while (firstDigit >= 10){
            firstDigit /= 10;
        }

        if(num % (firstDigit + lastDigit) == 0){
            printf("Number found: %d\n", num);
            count++;
        }

    }

    printf("The total count is %d\n", count);

    return 0;

}
```

5.3 For loop

In the C programming language, the for loop is very effective for statements that need to be executed a specific number of times. It is composed of three parts:initialization, condition, and increment or decrement.

The general structure of the *for* loop is as follows:

Structure of the for loop

```c
for(initialization; conditions; increment_or_decrement)
    statement;
```

In case of a block of statements, the beginning and the end of the block are marked with brackets. The general structure of the *for* loop in case of block of statements is as follows:

Structure of the for loop in case of block of statements

```
for(initialization; conditions; increment_or_decrement)
{
    block_statements;
}
```

The initial values of the counters are usually set in the first part - initialization.

```
for(i = 1; ...
```

In the case of multiple counters, the initializations are separated by commas.

```
for(i = 1, j = 100, k = 5; ...
```

In the *for* loop, the condition is a boolean expression tested after the initialization part. While the condition is satisfied, the loop is incremented or decremented, and the loop statements are repeated. The statements of the loop are repeated multiple times, or never at all.

```
for(i = 1; i < 100; i++)
```

In the third part, one or more variables can be incremented or decremented.

```
for(i = 1, j = 100; i < 100; i++, j--)
```

In each of parts, any other statement can be placed. It is also possible, some of the parts or all parts to be left empty.

```
for( ; scanf("%d", &num); )
```

In each of three parts, various statements can be placed, but the order of their executions and interpretation of results are strictly defined as follows:

- First, the *initialization* part is executed at the beginning. This part is executed before entering the loop.
- Next, the statements in the second part - *conditions* are executed. They are executed before the beginning of every new iteration of the loop. They result in a boolean value. If the boolean value is a logical truth, it will cause another execution of the statements in the loop. If the boolean value is a logical false, the loop ends, and the statements after the loop are executed.
- The statements from the third part - *increment_or_decrement* are executed at the end of every iteration (once all statements from the *block_statements* part have been executed), followed by the part conditions. If the conditions are still true, the cycle is repeated.

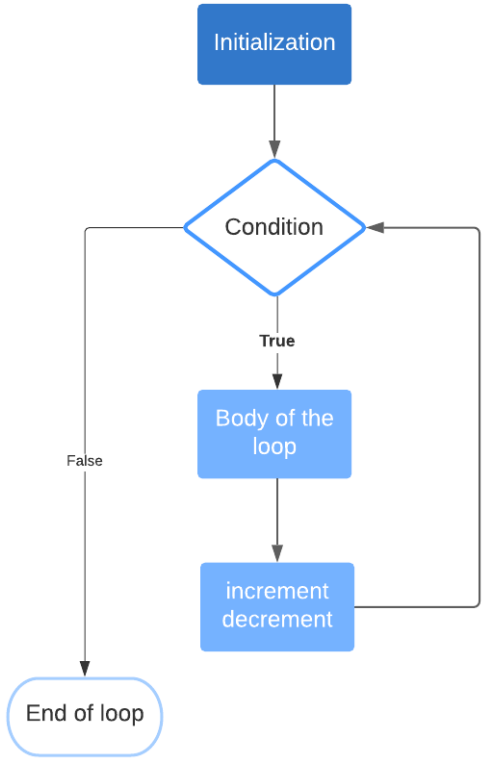

Figure 5.3: Flow chart of the for loop.

The flow diagram of the *for* loop is given on Figure 5.3.

Multiple *for* loops can be nested together to allow two-dimensional execution. **Nested *for* loop** is when one *for* loop is inside another.When a loop is nested inside another loop, the inner loop is executed multiple times inside the outer loop. On every iteration of the outer loop, the inner loop will be reinitialized. The inner loop must finish all of its iterations before the outer loop can continue to its next iteration.

The general structure of the nested *for* loop is as follows:

Structure of the nested for loop

```
for(initialization_outer; conditions_outer; increment_or_decrement_outer)
{
   block_statements_outer_loop;
   for(initialization_inner; conditions_inner; increment_or_decrement_inner)
   {
       block_statements_inner_loop;
   }
   block_statements_outer_loop;
```

```
}
```

Exercises:

1. Write a C program that calculates the sum of the first N numbers. N is entered from the keyboard.

   Solution:

```c
#include <stdio.h>

int main(){

    int n, i, sum = 0;

    printf("Enter the number N: ");
    scanf("%d", &n);

    for(i = 1; i <= n; i++){
        sum += i;
    }

    printf("The total sum is %d\n", sum);

    return 0;

}
```

2. Write a C program that checks whether a number entered from the keyboard is a prime number.

   Solution:

```c
#include <stdio.h>

int main(){

    int num, j = 2, isPrime = 1;

    printf("Enter the number: ");
    scanf("%d", &num);

    while(j <= num / 2){
        if(num % j == 0){
            isPrime = 0;
            break;
        }
        j++;
    }

    if(isPrime){
        printf("The number %d is prime number\n", num);
    }
```

```
    else{
        printf("The number is %d NOT prime number\n", num);
    }

    return 0;

}
```

3. Write a C program that will find and print the number of digits of a number entered by the user.

   Solution:

```
#include <stdio.h>

int main(){

    int num, numOfDigits = 0, temp;

    printf("Enter the number: ");
    scanf("%d", &num);

    temp = num;

    while(temp > 0){
        numOfDigits++;
        temp /= 10;
    }

    printf("The number of digits of %d is %d\n", num, numOfDigits);

    return 0;

}
```

4. Write a C program that prints on the screen the multiplication table from 1 to 10. The output should be similar to:
   1 x 1 = 1
   1 x 2 = 2
   etc.

   Solution:

```
#include <stdio.h>

int main(){

    int i, j;

    for(i = 1; i <= 10; i++){
        for(j = 1; j<= 10; j++){
```

```
            printf("%d x %d = %d\n", i, j, (i*j));
        }
        printf("-------------\n");
    }

    return 0;

}
```

5. Write a C program that checks whether a number entered from the keyboard is a palindromic number.

   Solution:

```
#include <stdio.h>

int main(){

    int num, reversed = 0, temp;

    printf("Enter the number: ");
    scanf("%d", &num);

    temp = num;

    while (temp > 0){
        reversed = reversed * 10 + temp % 10;
        temp /= 10;
    }

    if(num == reversed){
        printf("The number %d is a palindromic number\n", num);
    }
    else{
        printf("The number %d is a NOT palindromic number\n", num);
    }

    return 0;

}
```

6. Write a C program that prints and counts all numbers from a certain range divisible by 6. **Do not use %6 in your code**

   Solution:

```
#include <stdio.h>

int main(){

    int start, end, number, count = 0;
```

```
    printf("Enter the start and the end of the range: ");
    scanf("%d %d", &start, &end);

    for(number = start; number <= end; number++){
        if((number / 6) * 6 == number){
            printf("Number found: %d\n", number);
            count++;
        }
    }

    printf("The total count is %d\n", count);

    return 0;

}
```

7. Write a C program that finds all numbers in a given range that its most significant digit is larger than the number next to the most significant digit.

   Solution:

```
#include <stdio.h>

int main(){

    int start, end, number, temp;

    printf("Enter the start and the end of the range: ");
    scanf("%d %d", &start, &end);

    for(number = start; number <= end; number++){
        temp = number;

        while (temp > 10){
            temp /= 10;
        }

        if(temp / 10 > temp % 10){
            printf("Number found %d\n", number);
        }
    }

    return 0;

}
```

8. Write a C program that finds all perfect numbers in a given range.

   Note: Perfect number is a positive integer that is equal to the sum of its proper divisors. The smallest perfect number is 6, which is the sum of 1, 2, and 3.

   Solution:

```c
#include <stdio.h>

int main(){

    int start, end, num, j, sumOfDivisors;

    printf("Enter the start and the end of the range: ");
    scanf("%d %d", &start, &end);

    for(num = start; num < end; num++){
        sumOfDivisors = 0;

        j = 1;

        while(j < num){
            if(num % j == 0){
                sumOfDivisors += j;
            }
            j++;
        }

        if(num == sumOfDivisors){
            printf("Perfect number found %d\n", num);
        }
    }

    return 0;

}
```

9. Write a program that checks if a number entered by the user is an Armstrong number.

   Note: Armstrong number is a number where the sum of cubes of each digit of the number is equal to the number itself.

   Example: 153 = (1 * 1 * 1) + (5 * 5 * 5) + (3 * 3 * 3)

   Solution:

```c
#include <stdio.h>

int main(){

    int num, temp, digit, totalSum = 0;

    printf("Enter the number: ");
    scanf("%d", &num);

    temp = num;

    while (temp > 0){
        digit = temp % 10;
        printf("%d %d\n", digit, digit*digit*digit);
```

```
        totalSum += digit*digit*digit;
        temp /= 10;
    }

    if(totalSum == num){
        printf("%d is an Armstrong number\n", num);
    }else{
        printf("%d is NOT an Armstrong number\n", num);
    }

    return 0;

}
```

5.4 Statements for controlling program execution

In the C programming language, three statements can interrupt the program's sequential execution, i.e., to transfer the program control. Those statements are:

- break
- continue
- goto

The **break** statement enables exiting a *for, while, do-while* loop or a *switch* statement by immediately terminated it. The program control resumes at the next statement after the loop. **Example:**

```
#include <stdio.h>
int main()
{
  int i = 0;
  for (;;++i)
  {
    if(i > 9999)
    break;
  }
  printf("%d",x);
  return(0);
}
```

The flow diagram of the *break* statement is given on Figure 5.4.

The **continue** statement passes control to the next iteration of the loop by jumping the remaining statements after the continue. The program control resumes at the next iteration of the loop.

Example:

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

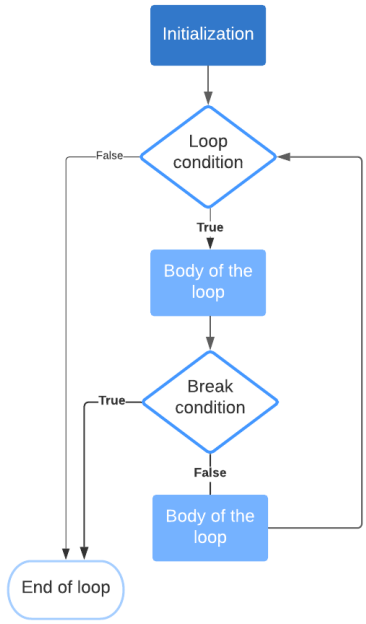

Figure 5.4: Flow chart of the break statement.

```
    int i;
    for(i=0; i<10; i++)
    {
        if(i<5)continue;
        printf("%d\n", i);
    }
    return 0;
}
```

The output of the above example will be:

```
5
6
7
8
9
```

The flow diagram of the *continue* statement is given on Figure 5.5.

The **goto** statement passes control to code location where a *label* of the goto statement is put. The program control resumes at the next iteration after the label. The basic structure of the goto statement is as follows:

```
goto (label);
```

and somewhere in the code, there is:

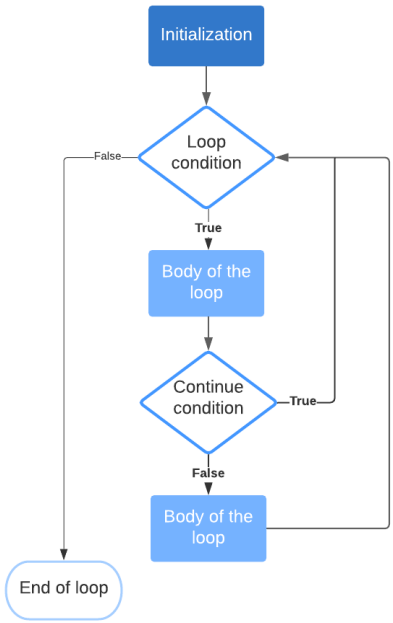

Figure 5.5: Flow chart of the continue statement.

```
label:
```

CHAPTER 6

Functions

Function is a block of code that is used to perform a specific task. Functions usually receive some data as an input, process it, and return some result. Once a function is written, it can be reused multiple times. Functions make programming simpler. Every function executes a strictly defined job, useful for other parts of the program and other programs. The rest of the program does not need to "know" how the job is performed.

Using functions is similar to a boss giving a task to an employee. Every worker gets information, does the job, performs the task, and gives results back. Information hiding: the boss does not need to know the details of how the job was performed. Only the results are of interest to the boss.

The advantages of using functions are as follows:

- Simpler maintenance of the program;
- Possibility of using previously developed functions, i.e., existing functions are used as building blocks for the new programs;
- Abstraction – the inner details are hidden;
- Avoid repeating the same code in multiple places in the program.

6.1 Functions in C

In the C programming language, programs are combinations of user-defined functions and predefined functions that are part of the libraries. Standard C libraries contain a large number of various functions.

Using functions in a program is called **function calls**. When a function is called, a name and arguments for the function are provided.

The format of the **function call** is as follows:

```
nameOfFunction(list of arguments)
```

If there is more than one argument, they are separated by commas.

Example:

```
printf("The square root of 81 is %f", sqrt(81.0));
```

In the example above, the function *sqrt* is called, calculating the square root of the argument - 81.

In the C programming language, the arguments of the function can be:

- Constant
- Variables
- Expressions

6.2 Scope of the variables

Variables can differ per type, scope, and places of declaration. Different types of variables are briefly described in section 1 of this book. In this section, the other two aspects will be considered. According to the place where they are declared, the variables can be:

- **Global** - defined outside of functions and can be used by all functions in the program. These variables exist as long as the program runs.
- **Local** - exists only in functions or blocks of code where they are declared. They didn't exist in any other functions and blocks of code. These variables are destroyed when the function ends or the block of code where the variable is declared ends. They are recreated each time the function is called, or the block of code is executed.

When variables are created, several rules should be taken into consideration:

- Global variables can be used anywhere in the program but always after they are declared;
- Local variables declared in a block of code can be used only in that block;
- Local variables declared in one function cannot be used in another function,
- The variable cannot be used outside of its scope;
- The variable can be hidden in some part of its scope;
- Two variables cannot have the same name in the same scope.

Local variables can be declared in any block of code. Local variables are recreated at the beginning of every block's repetition and are accessible in the block and all nested blocks. Once the block ended, the local variables are destroyed. For example, in the following code,

```
for(i=0; i<50; i++)
{
int sum=0;
...
}
```

the local variable **a** exists only within the *for* loop. In each iteration of the loop, memory is reserved, the variable is initialized and destroyed eventually.

According to the duration, the variables can be:

- **Permanent.** Permanent variables are created and initialized at the beginning of each program and exist until the end of the program.Usually, global variables are permanent.
- **Temporary.** Temporary variables are created and initialized when the block of code is executed. They are terminated when the block in which they are declared ends. Usually, local variables are temporary.

The following keywords are used to determine when and where variables will be created and destroyed:

- **static**
- **auto**
- **extern**
- **register**

Variables declared as *static* are created and initialized once, with the first call of the function. During the consecutive calls to the function, the static variables are not recreated or reinitialized. When the function ends, the variable still exists, but the other variables cannot access it.

The variables declared as *auto* behave completely differently. They are created when the function starts and are destroyed when the function ends. Temporary variables are called automatic because memory space is automatically allocated for them. The auto qualifier can be used to mark this type of variable, although it is omitted in praxis.

The variables declared as *extern* are variables that are only declared but not defined. It means that there is no memory allocated for these variables.

The variables declared as *register* are variables that are stored in the CPU registers instead of a memory. This allows the *register* variables to be faster accessible compared to the other types of variables.

In the listing below, it is given an example of using static and auto variables.

```
#include <stdio.h>
void StaticAndAuto(void)
{
```

```
    auto int a = 0;
    static int s=0;
    printf("auto = %d, static = %d\n", a, s);
    a++;
    s++;
}
int main()
{
    for(int i=0;i<3;i++)
    {
        StaticAndAuto();
    }
    return 0;
}
```

The output of the above listing will be:

```
"auto = 0, static = 0
 auto = 0, static = 1
 auto = 0, static = 2
```

6.3 Defining and using functions

When a function is defined, the type of value returned by the function, the name of the function, and the arguments for the function are provided. Also, the body of the function should be defined.

The general structure of the function definition is as follows:

```
Type_of_returned_value nameOfFunction(type var1, type var2, ...)
{
    /* body of the function */
}
```

The type of returned value by the function can be:

- The standard data type (int, float, char, etc.);
- Complex data type;
- User-defined type;
- Void

If the return type is *void*, the function doesn't return a value. If the type of returned value is not defined, the default type is *int*. But to avoid confusion, always define the type.

Example:

```
int SomeFunction(){...}
```

is same as:

```
SomeFunction(){...}
```

type var1, type var2, ... is list of formal arguments/parameters of the function. If there is more than one argument/parameter, they are separated by commas. Variables are declared in a list. Functions can be without formal arguments/parameters, i.e., the list can be empty. Variables declared in the list of arguments/parameters can be used only in the function's body. Theoretically, the number of formal arguments/parameters is unlimited, but it is limited in reality. The general recommendation is to avoid using a large number of parameters.

The body of a function contains the same elements as the *main()* function. It includes variables declaration and statements. Variables that are declared in the function are local to that function.

Example: C program that uses a function to calculates the factorial of a number.

```
#include <stdio.h>
void calc_factoriel(int n)
{
   int i, fact_num = 1;
   for( i = 1; i <= n; ++i )
   fact_num *= i; printf("The factorial of %d is %d\n", n, fact_num);
}
int main()
{
   int number = 0;
   printf("Enter a number\n"); scanf("%d", &number); calc_factoriel(number);
   return 0;
}
```

The output of the above example will be:

```
Enter a number
3
The factorial of 3 is 6
```

If the function is declared *void*, then the function ends when the ***return;*** statement is met, or a closing bracket ***}*** is met. **Example:**

```
void print_positive_number(int number)
{
    if(number<0)
    {
        printf("Number is negative\n");
        return;
    }
    printf("%d\n is positive number", number);
}
```

If the function is declared as returning a value, then the function ends when **return *expression;*** statement is met. The type of ***expression*** value returned must be

of the type as a defined type returned by the function.

Example:

```
#include<stdio.h>
int add_numbers(int a, int b)
{
    int c=a+b;
    return c;
}
int main()
{
    int sum;
    sum=add_numbers(3, 5);
    printf("The sum is %d\n",sum);
    return 0;
}
```

The expression *sum=add_numbers(3, 5);* is a call to the function *add_numbers*. The variable ***a*** will be assigned with value 3, and ***b*** with value 5. The function returns the value 8, placed in the variable sum. The formal arguments accept the values passed when the function is called. **Whenever a call to a function is made, the number and type of arguments passed is checked if:**

- **The number of real arguments is equal to the number the formal arguments.** If they are not equal, the compiler will give an error.
- **The type of real arguments is the same as the type of formal arguments.** If they are not the same, the compiler will first try to "translate" the passed value into the type of formal arguments. If the "translation" is possible, the call can be made, but the result might be wrong. If the "translation" is not possible, the compiler will give an error.

For every formal argument, when **passing by value**, a new variable is created. The variable is initialized with the value of the real argument. Any change to this variable will not affect the value of the real argument passed. This will avoid any unwanted effects produced by the changes to the formal parameter.

It is also possible for a function to have several return statements. **Example:**

```
int check_operator(char operator)
{
    switch(operator)
    {
        case '+' :   case '-' : return 1;
        case '*' :   case '/' : return 2;
        default  : return 0;
    }
}
```

Exercises:

1. Write a C function that prints “My first C Function!” on the screen.

   Solution:

```c
#include <stdio.h>

void printHello(){
    printf("My first C Function!\n");
}

int main(){
    printHello();
    return 0;
}
```

2. Write a C function that calculates a square of a number.

   Solution:

```c
#include <stdio.h>

int numberSquared(int n){
    return n*n;
}

int main(){

    int n;

    printf("Enter the number n = ");
    scanf("%d", &n);

    printf("The square of number %d is %d\n", n ,numberSquared(n));
    return 0;

}
```

3. Write a C function that calculates the area of a rectangle.

   Solution:

```c
#include <stdio.h>

float area(float a, float b){
    return a*b;
}
```

```c
int main(){

    float a, b;

    printf("Enter the sides of the rectangle: ");
    scanf("%f %f", &a, &b);

    printf("The area of the rectangle with sides %.2f and %.2f is %.2f
        \n", a, b ,area(a,b));
    return 0;

}
```

4. Write a C function that prints some informative message on the screen.

   Solution:

```c
#include <stdio.h>

void printMessage(char s[]){
    printf("%s\n", s);
}

int main(){

    char text[100];

    printf("Enter the message: ");
    scanf("%s", text);

    printMessage(text);

    return 0;

}
```

5. Write a C function to check if the number is divisible by 6. The function returns 1 or 0.

   Solution:

```c
#include <stdio.h>

int isDivisibleBySix(int n){
    return (n % 6 == 0);
}
```

```c
int main(){

    int n;

    printf("Enter the number: ");
    scanf("%d", &n);

    if(isDivisibleBySix(n)){
        printf("The number is divisible by 6\n");
    }
    else{
        printf("The number is NOT divisible by 6\n");
    }

    return 0;

}
```

6. Write a C function that checks if the number can be divided with its sum of even digits.

Solution:

```c
#include <stdio.h>

int sunOfEvenDigits(int n){

    int digit, sum = 0;

    while (n > 0){

        digit = n % 10;

        if(digit % 2 == 0){
            sum += digit;
        }

        n /= 10;
    }

    return sum;
}

int checkNumber(int n){

    int sumOfEventDigits = sunOfEvenDigits(n);

    return sumOfEventDigits > 0 ? n % sunOfEvenDigits(n) == 0 : 0;

}

int main(){

    int n;
```

```
    printf("Enter the number: ");
    scanf("%d", &n);

    if(checkNumber(n)){
        printf("The number can be divided with it\'s sum of even
            digits\n");
    }
    else{
        printf("The number can NOT be divided with it\'s sum of even
            digits\n");
    }

    return 0;

}
```

7. Write a C function that determines if the number inputted from the keyboard is larger than 500, is prime, and its last digit is 7.

   Solution:

```
#include <stdio.h>

int isPrime(int n){

    int y = 2;

    while (y <= n/2){
        if(n % y++ == 0){
            return 0;
        }
    }

    return 1;
}

int checkNumber(int n){
    return (n > 500) && (n % 10 == 7) && isPrime(n);
}

int main(){

    int n;

    printf("Enter the number: ");
    scanf("%d", &n);

    if(checkNumber(n)){
        printf("The number satisfies the condition\n");
    }
    else{
        printf("The number does not satisfy the condition\n");
    }
```

```
    return 0;

}
```

8. Write a C program that uses a function to find all numbers in a given range that its most significant digit is larger than the number next to the most significant digit. Write a *main()* function where the user enters the start and the end of the range and tests the previously defined function.

   Solution:

```
#include <stdio.h>

int checkNumber(int n){
    while (n > 100){
        n /= 10;
    }

    return (n / 10) > (n % 10);
}

void findNumbers(int start, int end){

    int i;

    for(i = start; i <= end && i >= 100; i++){
        if(checkNumber(i)){
            printf("Number found %d\n", i);
        }
    }

}

int main(){

    int start, end;

    printf("Enter the start and the end of the interval: ");
    scanf("%d %d", &start, &end);

    findNumbers(start, end);

    return 0;

}
```

9. Write a C program that from an unknown number of numbers will print and count the numbers larger than 10000 that satisfy the condition: the first two digits are smaller than the last two digits. The checking is done by a separate function called *checkNumber(int)*. The program ends when the user enters something different than a number.

Solution:

```c
#include <stdio.h>

int checkNumber(int n){

    if(n > 10000){
        int lastTwo = n % 100;

        while (n > 100){
            n /= 10;
        }

        return n < lastTwo;
    }
    else{
        return 0;
    }

}

int main(){

    int n;

    printf("Start entering numbers. Enter non-digit character to
         terminate the program: ");

    while(scanf("%d", &n)){
        if(checkNumber(n)){
            printf("Number found %d\n", n);
        }

        printf("Enter new number: ");
    }

    return 0;

}
```

10. Write a C program that from an unknown number of numbers will print the total sum of all number's most significant digits. Use a function for extracting the most significant digit of the number. Write a main function to test your code.

Solution:

```c
#include <stdio.h>

int extractFirstDigit(int n){

    while (n >= 10){
        n /= 10;
    }
```

```
    return n;

}

int main(){

    int n, sum = 0;

    printf("Start entering numbers. Enter non-digit character to
         terminate the program: ");

    while(scanf("%d", &n)){
        sum += extractFirstDigit(n);

        printf("Enter new number: ");
    }

    printf("The total sum is %d\n", sum);

    return 0;

}
```

6.4 Recursive functions

In structural programming, there are two ways for the program to repeat code or algorithm:

- **Iteration.** During the iterations, the algorithm is not repeated. Only the parameters of the algorithm are repeated. The iterations are detailed explained in the previous chapter.
- **Recursion.** During the recursion, the algorithm is repeated.

We have a recursion, usually when the return value of the function is the function itself. For example, the function below is recursive:

```
void RecursionExample(int a)
{
    if (a<0)
    return;
    else
    printf("%d", a);
    return RecursionExample(a--);
}
```

When a recursive function is created, the algorithm movement is in both directions. Firstly, we go from top to bottom by simplifying the input algorithm. Once we reach the base case, i.e., the algorithm can not be further simplified, we start with the second approach, solve the problem by going from bottom to top. For example, if we need to find the sum of all numbers from 0 to N, the recursive approach will be as follows:

```
I. Top to bottom approach - simplify the initial algorithm.

sum(n)=n+sum(n-1)
sum(n-1)=(n-1)+sum(n-1)
.............
sum(0)=0   <---- Basic case


II. Bottom to top approach - replacing all intermediate sums.

sum(0)=0
sum(1)=1+sum(0)=1+0=1
sum(2)=2+sum(1)=2+1=3
sum(3)=3+sum(2)=3+3=6
sum(4)=4+sum(3)=4+6=10
sum(5)=5+sum(4)=5+10=15
......
```

In order to create a correct recursive algorithm, two rule of thumb must be followed:

1. Every recursive algorithm MUST have a **basic case**, i.e., an expression that solves the problem. In the previous example, it was *sum(0)=0*.
2. The other part of the algorithm is called **general case**, i.e., a recursive connection. This part MUST simplify the initial algorithm.

In order to write a recursive function, firstly, the basic case and the general case needs to be determined. Every recursive call must solve part of the problem or to reduce the problem complexity.

Exercises:

1. Write a recursive function in C that calculates the factorial of a number.

Solution:

```c
#include <stdio.h>

long int factorial(int n){

    if(n == 1){
        return 1;
    }
    else{
        return n * factorial(n-1);
    }

}

int main(){
```

```
    int n;

    printf("Enter the number ");
    scanf("%d", &n);

    printf("Factorial of %d is %ld\n", n, factorial(n));

    return 0;

}
```

2. Write a recursive function in C that calculates X power N.

$$X^n$$

Solution:

```
#include <stdio.h>

long int power(int x, int n){

    if(n == 1){
        return x;
    }
    else{
        return x * power(x, n-1);
    }

}

int main(){

    int n, x;

    printf("Enter the number and the power: ");
    scanf("%d %d", &n, &x);

    printf("%d on power of %d is %ld\n", n, x, power(n,x));

    return 0;

}
```

3. Write a recursive function in C that prints the Fibonacci sequence up to N.

Solution:

```
#include <stdio.h>

int fibonnacci(int n){
```

```
    if(n == 0){
        return 0;
    }
    else if(n == 1){
        return n;
    }
    else{
        return fibonnacci(n-1) + fibonnacci(n-2);
    }

}

int main(){

    int i, n;

    printf("Enter the number N: ");
    scanf("%d", &n);

    for(i = 0; i < n; i++){
        printf("%d\n", fibonnacci(i));
    }

    return 0;

}
```

4. Write a recursive function in C that calculates the sum of digits of a number.

   Solution:

```
#include <stdio.h>

int sumOfDigits(int n){

    if(n < 10){
        return n;
    }
    else{
        return n % 10 + sumOfDigits(n/10);
    }

}

int main(){

    int n;

    printf("Enter the number: ");
    scanf("%d", &n);

    printf("Sum of digits of %d is %d\n", n, sumOfDigits(n));

    return 0;
```

```
}
```

5. Write a recursive function in C that finds the number of digits of a number.

   Solution:

```c
#include <stdio.h>

int numberOfDigits(int n){

    if(n < 10){
        return 1;
    }
    else{
        return 1 + numberOfDigits(n/10);
    }

}

int main(){

    int n;

    printf("Enter the number: ");
    scanf("%d", &n);

    printf("Number of digits of %d is %d\n", n, numberOfDigits(n));

    return 0;

}
```

6.5 Function prototype

A function prototype is a declaration of a function that specifies the function's name, parameters, and return type. If the function is used before being completely declared, it should be declared similarly to every other variable. In this way, the compiler will be informed that the function may later be used in the program. The function prototype doesn't contain a body.

The format of the function prototype is as follows:

```
type nameOfFunction(listOfParameters);
```

where:

- **type** is the data type of the value returned by the function. If the type is not defined, the default is *int*.
- **nameOfFunction** is the name of the function.

- **listOfParameters** is the list of formal arguments passed to the function. In the function prototype, only the type of the parameters is enough to be defined.

The prototype ends with a semicolon **";"**, and it marks that a function is only declared and not completely defined.

Below, examples of function prototypes are given.

```
float max(float a, float b, float c);
    - Three decimal values are sent, a decimal value is returned.

float max(float, float, float);
    - same as previous example.

void printarray(void);
    - No values are sent, no value is returned.

int find(void);
    - No values are sent, an integer value is returned.

void smaller(int, int);
    - Two integer values are sent, no value is returned.
```

Exercises:

1. Find the problem in the following C program:

```c
#include <stdio.h>
int main()
{
    int a, b;
    printf("Enter two integer numbers: ");
    scanf("%d%d", &a, &b);
    printf("The smaller is: %d\n", smaller(a, b) );
    return 0;
}
int smaller(int a, int b)
{
    if (a > b)
        return b;
    else
        return a;
}
```

Solution: The function *smaller* is defined after the *main* function, i.e., after it was used. Hence, a declaration (prototype) of a function is required before the *main* function. The program should be written like this:

```c
#include <stdio.h>
int smaller(int, int);
int main()
{
    int a, b;
    printf("Enter two integer numbers: ");
    scanf("%d%d", &a, &b);
```

```
    printf("The smaller is: %d\n", smaller(a, b) );
    return 0;
}
int smaller(int a, int b)
{
    if (a > b)
        return b;
    else
        return a;
}
```

CHAPTER 7

Arrays

Arrays are homogeneous, linear data structures that can store collections of the same type. An array is a variable containing more elements of the same type. It can be seen as a collection of individual elements with a common name, enabling access to the elements.

Arrays can have multiple dimensions. Therefore the arrays can be:

- One-dimensional (vectors)
- Two-dimensional (matrices)
- Multi-dimensional (three, four, etc.)

7.1 Arrays in C

In the C programming language, an array is a structure with related data where the number of elements is known in advance. All elements of the array are variables of the same type. By definition, an array is a collection of variables of the same type placed in an array of consecutive memory locations, given a unique common name.

To access an element of the array, the name and the position of the element is used. The position is determined by an index. Each element of the array is a variable that can be assigned with a value. Figure 7.1 presents an example of a one-dimensional array.

Figure 7.1: An example of a one-dimensional array.

7.2 Declaring and initialization of arrays

To declare an array in the C programming language, the programmer specifies the name of the array, the type of the elements (variables), and the number of elements. Declaring means allocating memory space.

The format of the statement for declaring an array is as follows:

```
type nameOfArray[NumberOfElements];
```

where:

- **type** is the data type of the elements of the array. All elements must be of the same type.
- **nameOfArray** is the name of the array.
- **NumberOfElements** is the number of elements of the array.

Exercises:

1. Declare: one-dimensional array consists of 50 integers; one-dimensional array consists of 100 double-precision floating-point numbers; one-dimensional array consists of 10 single-precision floating-point numbers, and two one-dimensional arrays consist of 40 and 35 characters.

   Solution:

   ```
   int A[50];
   double temp[100];
   float b[10];
   char c[40], temp1[35];
   ```

2. Why the declarations presented below, are not the correct way to declare an array?

   ```
   char c;
   int a;
   int a[i];
   float b[j];
   double d[];
   ```

   Solution:

   - ***char c;*** and ***int a;*** are valid declaration for variables, not for an array. For array declaration we need to specify the number of elements of the array in [ ] (square brackets).
   - In ***int a[i];*** and ***float b[j];*** the number of elements of the array is not defined, i.e., i and j are variables from unknown type with unknown value. If i and j already have some integer value, this will be correct way to declare an array.

- In ***d[ ];*** the number of elements of the array is not defined.

The initialization of an array can be done within the declaration. For example, the initialization can be done as follows:

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
double b[5] = {10.0, 10.3, 2.22, 4.51, 23.94};
char c[5] = {'A', 'B', 'C', 'D', 'E'};
```

If the number of initialization values is less than the array's size, the remaining elements are initialized with 0.

```
int a[10] = {1, 2, 3, 4, 5}; /* the remaining elements are 0. */
double b[50] = {10.0}; /* The first element is initialized to 10.0,
                          the remaining to 0. */
int c[100] = {0}; /* The easiest way to initialize an array with all 0's. */
```

During the initialization, the length of the array can be omitted. The compiler will automatically determine it based on the number of values provided in the initialization.

```
int a[] = {1, 2, 3, 4, 5, 6, 7};/*The size will be automatically set to 7*/
double b[] = {10.0, 10.3, 2.22};/*The size will be automatically set to 3*/
char c[] = {'A', 'B', 'C', 'D'};/*The size will be automatically set to 4*/
```

Another way to initialize an array is by using a *for* loop. This is the most commonly used way to initialize an array. In the example below, the initialization of an array with elements entered from the keyboard is presented:

```
#include <stdio.h>

int main()
{
    int a[10], i;

    for(i = 0; i < 10; i++)
    {
        printf("a[%d]=", i);
        scanf("%d", &a[i]);
    }

}
```

7.3 Accessing elements of arrays

To access an element of the array, the name of the array and the position of the element in the array needs to be provided. The format of the statement for accessing an element of the array is as follows:

```
nameOfArray[IndexPosition];
```

where:

- **nameOfArray** is the name of the array.
- **IndexPosition** is the index of the element.

In the C programming language, the index of the first element is always 0. In an array of N elements, the index of the last element is N-1. The index of an array can only be an integer value. For an array with a name a and N elements, the elements will be as follows:

- a[0], a[1], a[2], ..., a[N-1].

When accessing the elements of an array, C compilers do not check the boundaries of the array. It is a programmer's responsibility for the index to be in the interval of [0] to [N-1]. For example, if this definition is given:

```
int a[10];
```

the following expressions are possible:

```
a[5] = a[-5];
a[50] = a[180] + a[-92];
```

But using an index outside of the boundaries of the array causes the program to access location outside of the array.

The operator that determines the index of the array has the highest priority. Therefore the statement a[0]++ will increase the value of the first element of the array for 1.

Exercises:

1. Write a C program that reads an array from the keyboard. If the user enters less than 100 numbers, fill the remaining elements with 1. Print the array to test your code.

   Solution:

```
#include <stdio.h>

int main(){

    int a[100], i, n;

    printf("Enter the size of the array, n = ");
    scanf("%d", &n);

    for(i = 0; i < n; i++){
        printf("a[%d]=", i);
        scanf("%d", &a[i]);
    }
```

```
    for(i = n; i < 100; i++){
        a[i] = 1;
    }

    for(i = 0; i < 100; i++){
        printf("a[%d]=%d\n", i, a[i]);
    }

    return 0;

}
```

2. Write a C program that prints the number of days in each month of the year.

   Solution:

```
#include <stdio.h>

int main() {

    int i,
    months[] = {31,28,31,30,31,30,31,31,30,31,30,31};

    for(i = 1; i < 13; i++){
        printf("Month nr %d has %d days\n", i, months[i-1]);
    }

    return 0;

}
```

3. Write a C program that calculates the sum of two arrays and puts the result in a new array.

   Solution:

```
#include <stdio.h>

int main() {

    int i,c[5],
    a[] = {1,2,3,4,5},
    b[] = {1,1,1,1,1};

    for(i = 0; i < 5; i++){
        c[i]=a[i]+b[i];
    }

    return 0;

}
```

4. Write a program that calculates the average of 10 numbers. For every number print, whether it is bigger or smaller than the average.

   Solution:

```c
#include <stdio.h>

int main() {

    int num[10], n;
    float avg=0;

    printf("Enter numbers:\n");
    for(n=0; n<10; n++)
        scanf("%d",&num[n]);

    for(n=0; n<10; n++)
        avg+=num[n];

    avg/=n;

    printf("The average of the numbers entered is: %f\n", avg);
    for(n=0; n<10; n++)
        printf("%d %s %f\n", num[n], num[n]>avg?"> ":"<=", avg);

    return 0;


}
```

5. Write a C program that checks if two arrays are completely identical. **Note:** Two arrays are identical if they have the same number of elements and if elements on identical positions in both arrays have identical values.

   Solution: Please note that there is no statement like A==B in the C programming language, which can check if two arrays have identical contents. This statement exists, but the meaning is completely different.

```c
#include <stdio.h>

int main() {

    int i, AreSame,
    a[] = {1,2,3,4,5},
    b[] = {1,2,3,4,5},
    n1 = sizeof(a) / sizeof(a[0]), n2 = sizeof(b) / sizeof(b[0]);

    if(n1 == n2) {
        for(AreSame = 1, i=0; AreSame && (i<n1); i++)
            if(a[i]!=b[i])
                AreSame = 0;
    }
    else
```

```
            AreSame = 0;

        if (AreSame)
            printf("Arrays are identical");
        else
            printf("Arrays are NOT identical");

        return 0;

    }
```

7.4 Passing arrays as parameters to a function

C programming language allows the programmer to pass the entire array as a parameter to function. It also allows passing individual elements of an array to function.The individual elements are passed as arguments to a function similar to any other variables.

The individual elements of an array are passed as arguments to a function by passing the array's name and the element's position. The format is as follows:

```
someFunction(nameOfArray[IndexPosition])
```

To pass the entire array to a function, the programmer needs to pass only the array's name as an argument. The format is as follows:

```
someFunction(nameOfArray)
```

When an entire array is passed as an argument to a function, the programmer needs to use square brackets [] in the function definition. The format is as follows:

```
Type_of_returned_value someFunction(type_of_array nameOfArray[])
{
/* body of the function */
...
}
```

Programmer can pass arrays as a parameter to a function in the C programming language, but the function can not return an array.

Exercises:

1. Write a C program that calculates the average of ten numbers.

   Solution:

```
#include <stdio.h>

#define SIZE 10
```

```c
float averageOfNumbers(float a[]){
    int i;
    float average = 0;

    for(i = 0; i < SIZE; i++){
        average += a[i];
    }

    return average/SIZE;
}

int main(){

    int i;
    float a[SIZE];

    printf("Enter the elements: ");

    for(i = 0; i < SIZE; i++){
        scanf("%f", &a[i]);
    }

    printf("The average is %.2f\n", averageOfNumbers(a));


}
```

2. Write a C program that reads elements of an array, and after that, print them on the screen in reverse order.

Solution:

```c
#include <stdio.h>

void enterArray(int a[], int n){

    int i;

    for(i = 0; i < n; i++){
        printf("a[%d] = ", i);
        scanf("%d", &a[i]);
    }

}

void printInReverse(int a[], int n){

    int i;

    for(i = n-1; i >= 0; i--){
        printf("a[%d] = %d   ", i, a[i]);
    }

}
```

```c
int main(){

    int a[100], n;

    printf("Enter the size of the array, n = ");
    scanf("%d", &n);

    enterArray(a, n);

    printInReverse(a, n);

    return 0;
}
```

3. Write a C program that finds the maximum element of an array. The size of the array and the elements are entered from the keyboard.

Solution:

```c
#include <stdio.h>

void enterArray(float a[], int n){

    int i;

    for(i = 0; i < n; i++){
        printf("a[%d] = ", i);
        scanf("%f", &a[i]);
    }

}

float findMax(float a[], int n){

    int i;
    float max = a[0];

    for(i = 1; i < n; i++){
        if(a[i] > max){
            max = a[i];
        }
    }

    return max;

}

int main(){

    int n;
    float a[100];

    printf("Enter the size of the array, n = ");
    scanf("%d", &n);
```

```
    enterArray(a, n);

    printf("The max is %.2f\n", findMax(a, n));

    return 0;
}
```

4. Write a C program that finds the largest negative number of an array.

Solution:

```
#include <stdio.h>

void enterArray(float a[], int n){

    int i;

    for(i = 0; i < n; i++){
        printf("a[%d] = ", i);
        scanf("%f", &a[i]);
    }

}

float findLargestNegative(float a[], int n){

    int i;
    float maxNegative = 1;

    for(i = 0; i < n; i++){

        if(a[i] < 0){
            if(maxNegative == 1){
                maxNegative = a[i];
            }

            if(a[i] > maxNegative){
                maxNegative = a[i];
            }
        }
    }

    return maxNegative;

}

int main(){

    int n;
    float a[100];
    float largestNegative;

    printf("Enter the size of the array, n = ");
    scanf("%d", &n);
```

```
    enterArray(a, n);

    largestNegative = findLargestNegative(a, n);

    if(largestNegative < 0){
        printf("The largest negative number is %.2f\n",
            largestNegative);
    }
    else{
        printf("There are no negative numbers in the array\n");
    }

    return 0;
}
```

5. Write a C program that reads from the keyboard the daily temperatures for each day in the month. The program should print the average monthly temperature, the hottest and the coldest day in the month. The user enters the number of days in the month.

   Solution:

```
#include <stdio.h>

#define SIZE 30

void enterTemperatures(float a[]){

    int i;

    for(i = 0; i < SIZE; i++){
        printf("Day %d temeprature: ", i+1);
        scanf("%f", &a[i]);
    }

}

void findHottest(float a[]){

    int i, maxI = 0;
    float max = a[0];

    for(i = 1; i < SIZE; i++){
        if(a[i] > max){
            max = a[i];
            maxI = i;
        }
    }

    printf("The hottest day was day %d, and it was %.1fC\n", maxI+1,
         max);

}
```

```
void findColdest(float a[]){

    int i, minI = 0;
    float min = a[0];

    for(i = 1; i < SIZE; i++){
        if(a[i] < min){
            min = a[i];
            minI = i;
        }
    }

    printf("The coldest day was day %d, and it was %.1fC\n", minI+1,
         min);

}

float averageTemperature(float a[]){

    int i;
    float average = 0;

    for(i = 1; i < SIZE; i++){
        average += a[i];
    }

    return average/SIZE;
}

int main(){

    float a[SIZE];

    enterTemperatures(a);
    findColdest(a);
    findHottest(a);

    printf("The average temparature is %.1fC\n", averageTemperature(a)
         );

    return 0;
}
```

6. Write a C program that will sort an array in ascending order.

   Solution:

```
#include <stdio.h>

void enterArray(float a[], int n){
```

```
    int i;

    for(i = 0; i < n; i++){
        printf("a[%d] = ", i);
        scanf("%f", &a[i]);
    }

}

void sortArray(float a[], int n){

    int i, j;
    float temp;

    for(i = 0; i < n - 1; i++){
        for(j = i + 1; j < n; j++){
            if(a[i] > a[j]){
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }

}

void printArray(float a[], int n){
    int i;

    for(i = 0; i < n; i++){
        printf("a[%d]=%.2f\n", i, a[i]);
    }
}

int main(){

    int n;
    float a[100];

    printf("Enter the size of the array, n = ");
    scanf("%d", &n);

    enterArray(a, n);
    sortArray(a, n);
    printArray(a, n);

    return 0;
}
```

7. Write a C program that prints all duplicate numbers in an array.

Solution:

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

```c
void enterArray(float a[], int n){

    int i;

    for(i = 0; i < n; i++){
        printf("a[%d] = ", i);
        scanf("%f", &a[i]);
    }

}

void printDuplicates(float a[], int n){

    int i, j, k = 0, l;
    int found;

    float b[n];

    for(i = 0; i < n; i++){
        for(j = i + 1; j < n; j++){

            if(a[i] == a[j]){

                found = 0;

                for(l = 0; l < k; l++){
                    if(b[l] == a[j]){
                        found = 1;
                        break;
                    }
                }

                if(!found){
                    b[k++] = a[j];
                }
            }
        }
    }

    printf("The duplicates are: \n");

    for(i = 0; i < k; i++){
        printf("%.2f\n", b[i]);
    }

}

int main(){

    int n;
    float a[100];

    printf("Enter the size of the array, n = ");
    scanf("%d", &n);

    enterArray(a, n);
    printDuplicates(a, n);
```

```
    return 0;
}
```

8. Write a C program that prints all numbers from an array that are divisible with the number before them.

   Solution:

```
#include <stdio.h>

void enterArray(int a[], int n){

    int i;

    for(i = 0; i < n; i++){
        printf("a[%d] = ", i);
        scanf("%d", &a[i]);
    }

}

void findNumbers(int a[], int n){
    int i;

    for(i = 1; i < n; i++){
        if(a[i] % a[i-1] == 0){
            printf("Number found: %d\n", a[i]);
        }
    }
}

int main(){

    int a[100], n;

    printf("Enter the size of the array, n = ");
    scanf("%d", &n);

    enterArray(a, n);
    findNumbers(a, n);

    return 0;
}
```

9. Write a C program that reads 50 numbers. The program will print the number that appears most frequently in the array and the number of times this number appears.

   Solution:

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

```c
#define SIZE 5

void enterArray(int a[]){

    int i;

    for(i = 0; i < SIZE; i++){
        printf("a[%d] = ", i);
        scanf("%d", &a[i]);
    }

}

int inArray(int a[], int n, int number){
    int i;

    for(i = 0; i < n; i++){
        if(a[i] == number){
            return i;
        }
    }

    return -1;
}

void findAppearances(int a[]){
    int i, j, k = 0, foundIndex;

    int foundNumbers[SIZE];
    int count[SIZE];

    for(i = 0; i < SIZE; i++){
        count[i] = 0;
    }

    for(i = 0; i < SIZE; i++){

        foundIndex = inArray(foundNumbers, k, a[i]);

        if(foundIndex == -1){
            foundNumbers[k] = a[i];
            count[k]++;
            k++;
        }
        else{
            count[foundIndex]++;
        }

    }

    int max = foundNumbers[0], maxIndex = 0;

    for(i = 1; i < k; i++){
        if(a[i] > max){
            max = a[i];
            maxIndex = i;
        }
    }
```

```
    printf("Number %d appeared %d times\n", foundNumbers[maxIndex],
        count[maxIndex]);

}

int main(){

    int a[SIZE];

    enterArray(a);
    findAppearances(a);

    return 0;
}
```

10. Write a C program that finds and prints all pairs of elements in an array whose sum is greater than the smallest positive number of the array.

Solution:

```
#include <stdio.h>

void enterArray(float a[], int n){

    int i;

    for(i = 0; i < n; i++){
        printf("a[%d] = ", i);
        scanf("%f", &a[i]);
    }

}

float findSmallestPositiveNumber(float a[], int n){
    int i;
    float smallestPositive = -1;

    for(i = 0; i < n; i++){

        if(a[i] > 0){
            if(smallestPositive == -1){
                smallestPositive = a[i];
            }

            if(a[i] < smallestPositive){
                smallestPositive = a[i];
            }
        }
    }

    return smallestPositive;
}

void findAndPrintPairs(float a[], int n){
```

```
    int i;
    float smallestPositive = findSmallestPositiveNumber(a, n);

    if(smallestPositive == -1){
        printf("There are no positive numbers in the array.\n");
    }
    else{
        printf("Smallest positive number is %.2f\n", smallestPositive)
            ;
        for(i = 0; i < n - 1; i++){
            if(a[i] + a[i+1] > smallestPositive){
                printf("Pair found: %.2f and %.2f\n", a[i], a[i+1]);
            }
        }
    }
}

int main(){

    int n;
    float a[100];
    float smallestPositive;

    printf("Enter the size of the array, n = ");
    scanf("%d", &n);

    enterArray(a, n);

    findAndPrintPairs(a, n);

    return 0;
}
```

11. Write a C program that will swap the second smallest with the second biggest number of the array. Print the result to test your code.

Solution:

```
#include <stdio.h>

void enterArray(float a[], int n){

    int i;

    for(i = 0; i < n; i++){
        printf("a[%d] = ", i);
        scanf("%f", &a[i]);
    }

}

int findSecondBiggestIndex(float a[], int n){

    int i, initialIndex = 0;
```

```
    float initialValue = a[0];

    for(i = 1; i < n; i++){

        if(a[i] < initialValue){
            initialValue = a[i];
            initialIndex = i;
        }

    }

    float max1 = initialValue, max2 = initialValue;
    int maxIndex = initialIndex, secondMaxIndex = initialIndex;

    for(i = 0; i < n; i++){

        if(a[i] > max1){
            max2 = max1;
            secondMaxIndex = maxIndex;
            max1 = a[i];
            maxIndex = i;
        }
        else if(a[i] > max2 && a[i] != max1){
            max2 = a[i];
            secondMaxIndex = i;
        }

    }

    return secondMaxIndex;

}

int findSecondSmallestIndex(float a[], int n){

    int i, initialIndex = 0;
    float initialValue = a[0];

    for(i = 1; i < n; i++){

        if(a[i] > initialValue){
            initialValue = a[i];
            initialIndex = i;
        }

    }

    float min1, min2;
    min1 = min2 = initialValue;
    int min1Index, min2Index;
    min1Index = min2Index = initialIndex;

    for(i = 0; i < n; i++){

        if(a[i] < min1){
            min2 = min1;
            min2Index = min1Index;
            min1 = a[i];
```

```
            min1Index = i;
        }
        else if(a[i] < min2 && a[i] != min1){
            min2 = a[i];
            min2Index = i;
        }

    }

    return min2Index;

}

void swapElements(float a[], int n, int maxI, int minI){
    int i;
    float temp = a[maxI];
    a[maxI] = a[minI];
    a[minI] = temp;
}

void printArray(float a[], int n){
    int i;

    for(i = 0; i < n; i++){
        printf("a[%d]=%.2f\n", i, a[i]);
    }
}

int main(){

    int n, maxI, minI;
    float a[100];
    float smallestPositive;

    printf("Enter the size of the array, n = ");
    scanf("%d", &n);

    enterArray(a, n);

    maxI = findSecondBiggestIndex(a, n);
    minI = findSecondSmallestIndex(a, n);
    swapElements(a, n , maxI, minI);

    printf("The new array is:\n");
    printArray(a, n);

    return 0;
}
```

12. Write a C program that will remove all elements in an array smaller than some number entered by the user.

    Solution:

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

```c
void enterArray(int a[], int n){

    int i;

    for(i = 0; i < n; i++){
        printf("a[%d] = ", i);
        scanf("%d", &a[i]);
    }

}

int removeElements(int a[], int n, int number){

    int i = 0, j, p;

    while(i < n){
        if(a[i] < number){
            for(j = i; j < n-1; j++){
                a[j] = a[j+1];
            }

            n--;

            if(a[i] >= number){
                i++;
            }
        }
        else{
            i++;
        }
    }

    return n;
}

void printArray(int a[], int n){
    int i;

    for(i = 0; i < n; i++){
        printf("a[%d]=%d\n", i, a[i]);
    }
}

int main(){

    int a[100], n, number;

    printf("Enter the size of the array, n = ");
    scanf("%d", &n);

    enterArray(a, n);

    printf("Enter the number: ");
    scanf("%d", &number);

    n = removeElements(a, n, number);
```

```
    printf("The new array is:\n");
    printArray(a, n);

    return 0;
}
```

13. Write a C program that will remove all odd elements in an array.

    Solution:

```
#include <stdio.h>

void enterArray(int a[], int n){

    int i;

    for(i = 0; i < n; i++){
        printf("a[%d] = ", i);
        scanf("%d", &a[i]);
    }

}

int removeOddElements(int a[], int n){

    int i = 0, j, p;

    while(i < n){
        if(a[i] % 2 == 1){
            for(j = i; j < n-1; j++){
                a[j] = a[j+1];
            }

            n--;

            if(a[i] % 2 == 0){
                i++;
            }
        }
        else{
            i++;
        }
    }

    return n;
}

void printArray(int a[], int n){
    int i;

    for(i = 0; i < n; i++){
        printf("a[%d]=%d\n", i, a[i]);
    }
}
```

```c
int main(){

    int a[100], n;

    printf("Enter the size of the array, n = ");
    scanf("%d", &n);

    enterArray(a, n);

    n = removeOddElements(a, n);

    printf("The new array is:\n");
    printArray(a, n);

    return 0;
}
```

14. Write a C program that finds and prints the size of the longest consecutive sequence of elements from a given unsorted array of integers. **Sample array:** [9, 124, 13, 130, 16, 14, 179, 15]. The longest consecutive sequence of elements is [13, 14, 15, 16]. The size is 4.

    Solution:

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

void enterArray(int a[], int n){

    int i;

    for(i = 0; i < n; i++){
        printf("a[%d] = ", i);
        scanf("%d", &a[i]);
    }

}

void consecutiveSsequence(int a[], int n){

    int i, j, b[n], temp;
    int length = 0, tempLength, position;

    for(i = 0; i < n; i++){
        b[i] = a[i];
    }

    for(i = 0; i < n - 1; i++){
        for(j = i; j < n; j++){
            if(a[i] > a[j]){
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
```

```
    }

     for(i = 0; i < n; i++){

         tempLength = 1;
         j = i;

         while(abs(a[j] - a[j+1]) == 1){
             tempLength++;
             j++;
         }

         if(tempLength > length){
             position = i;
             length = tempLength;
         }
     }

     printf("The longest consecutive sequence of elements is: ");

     for(i = position; i <= position+length-1; i++)
        printf("%d ",a[i]);

     printf("\nIt\'s length is %d\n", length);

}

int main(){

    int a[100], n;

    printf("Enter the size of the array, n = ");
    scanf("%d", &n);

    enterArray(a, n);

    consecutiveSsequence(a, n);

    return 0;
}
```

CHAPTER 8

Matrices

As described in the previous chapter, the arrays can have multiple dimensions: one-dimension, two-dimensional, etc. Matrices are two-dimensional arrays in which elements are arranged in a grid with rows and columns.

8.1 Matrices in C

In the C programming language, a matrix is a table with related data where the number of elements is known in advance. All elements of the matrix are variables of the same type. By definition, a matrix is a collection of variables of the same type placed in an array of consecutive memory locations, given a unique common name.

To access an element of the matrix, the name and the position of the element is used. The position is determined by the index of the row and the index of the column (m by n elements). Each element of the matrix is a variable that can be assigned with a value. Figure 8.1 presents an example of a two-dimensional array.

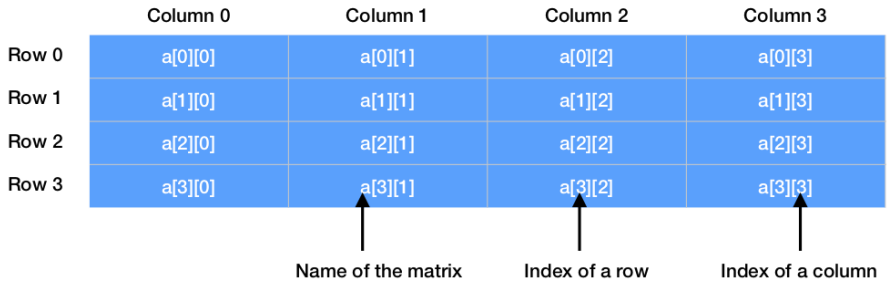

Figure 8.1: An example of a two-dimensional array.

8.2 Declaring and initialization of matrices

To declare a matrix in the C programming language, the programmer specifies the name of the matrix, the type of the elements (variables), and the number of rows and columns of the matrix. Declaring means allocating memory space. When you

declare N by M matrix, where N is the number of rows and M is the number of columns, a total of $N * M$ memory space is reserved.

The format of the statement for declaring a matrix is as follows:

```
type nameOfMatrix[NumberOfRows][NumberOfColumns];
```

where:

- **type** is the data type of the elements of the matrix. All elements must be of the same type.
- **nameOfMatrix** is the name of the matrix.
- **NumberOfRows** is the number of rows of the matrix.
- **NumberOfColumns** is the number of Columns of the matrix.

Exercises:

1. Declare: integer matrix consists of 10 rows and 10 columns; floating-point matrix consists of 3 rows and 4 columns; double-precision floating-point matrix consists of 4 rows and 3 columns.

   Solution:

   ```
   int a[10][10];
   float b[3][4];
   double c[4][3];
   ```

2. How many elements has the matrix with a declaration: *int a[10][10]*?

   Solution: 10 integer elements.

3. Which is the first element of the matrix with a declaration: *int a[10][10]*?

   Solution: The first element is a[0][0].

The initialization of a matrix can be done within the declaration. For example, the initialization can be done as follows:

```
int a[4][4] = { {0, -3, 5, 12},
                {-7, 4, 15, 3},
                {3, 0, 5, -13},
                {-2, 5, 11, 2} };


double b[2][3] = { {10.0, 10.3, 2.22}, {4.51, 23.94} };
```

When you initialize the matrix, the inner brackets are not mandatory, but some compilers can give you a warning "missing braces around initializer" if you miss them.

```
/* Valid, but not recommended. */
int a[3][4] = { 0, -3, 5, 12, -7, 4, 15, 3, 3, 0, 5, -13 };
```

If an insufficient number of elements is provided, the elements that are omitted are set to zero.

```
int a[3][3] = { {1, 2, 3}, {5}, {4, 1} }; /* Will initialize the matrix as:
                                              1 2 3
                                              5 0 0
                                              4 1 0 */

int b[3][4] = {{0}}; /* The easiest way to initialize a matrix with all 0's.
     */
```

During the initialization, the first dimension (number of rows) of the matrix can be omitted. The compiler will automatically determine it based on the number of values provided in the initialization. The second dimension (number of columns) of the matrix is mandatory and cannot be omitted.

```
/* The the first dimension (number of rows) will be automatically set to 4
    */
int a[][4] = { {0, -3, 5, 12},
               {-7, 4, 15, 3},
               {3, 0, 5, -13},
               {-2, 5, 11, 2} };


/* Invalid initialization - the second dimension cannot be omitted. */
int a[][] = { {0, -3, 5, 12},
              {-7, 4, 15, 3},
              {3, 0, 5, -13},
              {-2, 5, 11, 2} };

/* Invalid initialization - the second dimension cannot be omitted. */
int a[4][] = { {0, -3, 5, 12},
               {-7, 4, 15, 3},
               {3, 0, 5, -13},
               {-2, 5, 11, 2} };
```

Another way to initialize a matrix is by using a nested *for* loops. This is the most commonly used way to initialize a matrix. In the example below, the initialization of a matrix with elements entered from the keyboard is presented:

```
#include <stdio.h>

int main()
{
    int a[3][5], i;

    for(i = 0; i < 3; i++)
    {
        for(j = 0; j < 5; j++)
```

```
        {
            printf("a[%d][%d]=", i, j);
            scanf("%d", &a[i][j]);
        }
    }

}
```

8.3 Accessing elements and passing matrices as parameters to a function

To access an element of the matrix, the name of the matrix and the position of the element in the matrix needs to be provided. The element's position is determined by the index of the row and the index of the column. The format of the statement for accessing an element of the matrix is as follows:

```
nameOfMatrix[RowIndex][ColumnIndex];
```

where:

- **nameOfMatrix** is the name of the matrix.
- **RowIndex** is the row index.
- **ColumnIndex** is the column index.

In the C programming language, the index of the first row and column is always 0. In a matrix of N rows and M columns, the index of the last element of the row is N-1, and of the column is M-1. The index of a matrix can only be an integer value. For a matrix with a name a, N rows, and M columns, the elements will be as follows:

- a[0][0], a[0][1], ..., a[0][M-1], a[1][0], a[1][1], ..., a[N-1][M-1].

When accessing the elements of an array, C compilers do not check the boundaries of the array. It is a programmer's responsibility for the index to be in the interval of [0][0] to [N-1][M-1]. Using an index outside of the boundaries of the array causes the program to access location outside of the array.

The operator that determines the index of the array has the highest priority. Therefore the statement a[0][0]++ will increase the value of the first element of the array for 1.

C programming language allows the programmer to pass the entire matrix as a parameter to function. It also allows passing individual elements of a matrix to function.The individual elements are passed as arguments to a function similar to any other variables.

The individual elements of a matrix are passed as arguments to a function by passing the matrix name and the element's position. The format is as follows:

```
someFunction(nameOfMatrix[RowIndex][ColumnIndex])
```

To pass the entire matrix to a function, the programmer needs to pass only the matrix name as an argument. The format is as follows:

```
someFunction(nameOfMatrix)
```

When an entire matrix is passed as an argument to a function, the programmer needs to use double square brackets [][] in the function definition. The format is as follows:

```
Type_of_returned_value someFunction(type_of_matrix nameOfMatrix[][])
{
/* body of the function */
...
}
```

Programmer can pass matrix as a parameter to a function in the C programming language, but the function can not return a matrix.

Exercises:

1. Write a C program that finds the maximum element in a matrix.

Solution:

```c
#include <stdio.h>

int main(){

    int a[10][10], i, j, n, m, max;

    printf("Enter number of rows: ");
    scanf("%d", &n);
    printf("Enter number of columns: ");
    scanf("%d", &m);

    printf("Enter the matrix:\n");

    for(i = 0; i < n; i++){
        for(j = 0; j < m; j++){
            printf("a[%d][%d] = ", i, j);
            scanf("%d", &a[i][j]);
        }
    }

    max = a[0][0];

    for(i = 0; i < n; i++){
        for(j = 0; j < m; j++){
            if(a[i][j] > max){
                max = a[i][j];
            }
        }
    }
```

```
    printf("The max element is: %d\n", max);

    return 0;
}
```

2. Write a C program that finds the number of times X appears in a matrix of floating-point numbers. X is entered from the keyboard.

   Solution:

```
#include <stdio.h>

int main(){

    int i, j, n, m, count = 0;
    float a[10][10], number;

    printf("Enter number of rows: ");
    scanf("%d", &n);
    printf("Enter number of columns: ");
    scanf("%d", &m);

    printf("Enter the matrix:\n");

    for(i = 0; i < n; i++){
        for(j = 0; j < m; j++){
            printf("a[%d][%d] = ", i, j);
            scanf("%f", &a[i][j]);
        }
    }

    printf("Enter the number that you are looking for: ");
    scanf("%f", &number);

    for(i = 0; i < n; i++){
        for(j = 0; j < m; j++){
            if(a[i][j] == number){
                count++;
            }
        }
    }

    printf("The number %.2f appeared %d times.\n", number, count);

    return 0;
}
```

3. Write a C program that initializes a 10x3 matrix. The program should fill the matrix in such a way so, the first element of each row is a number starting from 1 to 10, the second element is the square of the number, and the third element is the cube of the number.

Solution:

```c
#include <stdio.h>

int power(int x, int y){
    if(y == 1){
        return x;
    }
    else{
        return x * power(x, y - 1);
    }
}

int main(){

    int a[10][10], i, j;
    int number, n = 10, m = 3;

    for(i = 0; i < n; i++){
        for(j = 0; j < 3; j++){
            a[i][j] = power(i+1, j+1);
        }
    }

    printf("The result matrix is:\n");

    for(i = 0; i < n; i++){
        for(j = 0; j < m; j++){
            printf("%d  ", a[i][j]);
        }

        printf("\n");
    }

    return 0;
}
```

4. Write a C program that computes the addition of two matrices. The matrices are inputted from the keyboard.

Solution:

```c
#include <stdio.h>

#define SIZE 10

int main(){

    int a[SIZE][SIZE], b[SIZE][SIZE], c[SIZE][SIZE];
    int i, j, n, m;

    printf("Enter number of rows: ");
    scanf("%d", &n);
    printf("Enter number of columns: ");
    scanf("%d", &m);
```

```
    printf("Enter the first matrix:\n");

    for(i = 0; i < n; i++){
        for(j = 0; j < m; j++){
            printf("a[%d][%d] = ", i, j);
            scanf("%d", &a[i][j]);
        }
    }

    printf("Enter the second matrix:\n");

    for(i = 0; i < n; i++){
        for(j = 0; j < m; j++){
            printf("b[%d][%d] = ", i, j);
            scanf("%d", &b[i][j]);

            c[i][j] = a[i][j] + b[i][j];
        }
    }

    printf("The result matrix is:\n");

    for(i = 0; i < n; i++){
        for(j = 0; j < m; j++){
            printf("%d  ", c[i][j]);
        }

        printf("\n");
    }

    return 0;
}
```

5. Write a C program that swaps two rows of a matrix. The row numbers are entered by the user.

   Solution:

```
#include <stdio.h>

#define SIZE 10

int main(){

    int a[SIZE][SIZE], i, j, n, m;
    int row1, row2, temp;

    printf("Enter number of rows: ");
    scanf("%d", &n);
    printf("Enter number of columns: ");
    scanf("%d", &m);

    printf("Enter the matrix:\n");
```

```
    for(i = 0; i < n; i++){
        for(j = 0; j < m; j++){
            printf("a[%d][%d] = ", i, j);
            scanf("%d", &a[i][j]);
        }
    }

    printf("Enter the first row: ");
    scanf("%d", &row1);

    printf("Enter the second row: ");
    scanf("%d", &row2);

    row1 -= 1;
    row2 -= 1;

    if((row1 < 0 || row2 < 0) || (row1 > n || row2 > n)){
        printf("Invalid row number.\n");
        return 0;
    }

    if(row1 == row2){
        printf("You entered the same number of row.\n");
        return 0;
    }

    for(j = 0; j < m; j++){
        temp = a[row1][j];
        a[row1][j] = a[row2][j];
        a[row2][j] = temp;
    }

    printf("The result matrix is:\n");

    for(i = 0; i < n; i++){
        for(j = 0; j < m; j++){
            printf("%d  ", a[i][j]);
        }

        printf("\n");
    }

    return 0;
}
```

6. Write a C program that transposes a matrix. The matrix is input from the keyboard.

Solution:

```
#include <stdio.h>

#define SIZE 10
```

```c
int main(){

    int a[SIZE][SIZE], b[SIZE][SIZE];
    int i, j, n, m;

    printf("Enter number of rows: ");
    scanf("%d", &n);
    printf("Enter number of columns: ");
    scanf("%d", &m);

    printf("Enter the matrix:\n");

    for(i = 0; i < n; i++){
        for(j = 0; j < m; j++){
            printf("a[%d][%d] = ", i, j);
            scanf("%d", &a[i][j]);
        }
    }

    for(i = 0; i < n; i++){
        for(j = 0; j < m; j++){
            b[j][i] = a[i][j];
        }
    }

    printf("The result matrix is:\n");

    for(i = 0; i < n; i++){
        for(j = 0; j < m; j++){
            printf("%d  ", b[i][j]);
        }

        printf("\n");
    }

    return 0;
}
```

7. Write a C program that converts all elements under the primary diagonal of a matrix to 0's.

Solution:

```c
#include <stdio.h>

#define SIZE 10

int main(){

    int a[SIZE][SIZE];
    int i, j, n;

    printf("Enter the size of the matrix: ");
    scanf("%d", &n);
```

```
    printf("Enter the matrix:\n");

    for(i = 0; i < n; i++){
        for(j = 0; j < n; j++){
            printf("a[%d][%d] = ", i, j);
            scanf("%d", &a[i][j]);
        }
    }

    for(i = 0; i < n; i++){
        for(j = 0; j < n; j++){
            if(i > j){
                a[i][j] = 0;
            }
        }
    }

    printf("The result matrix is:\n");

    for(i = 0; i < n; i++){
        for(j = 0; j < m; j++){
            printf("%d  ", a[i][j]);
        }

        printf("\n");
    }

    return 0;
}
```

8. Write a C program that finds the minimum element above the secondary diagonal of a matrix.

Solution:

```
#include <stdio.h>

#define SIZE 10

int main(){

    int a[SIZE][SIZE];
    int i, j, n, min;

    printf("Enter the size of the matrix: ");
    scanf("%d", &n);

    printf("Enter the matrix:\n");

    for(i = 0; i < n; i++){
        for(j = 0; j < n; j++){
            printf("a[%d][%d] = ", i, j);
            scanf("%d", &a[i][j]);
        }
```

```
    }

    min = a[0][0];

    for(i = 0; i < n; i++){
        for(j = 0; j < n; j++){
            if(i + j < n - 1){
                if(a[i][j] < min){
                    min = a[i][j];
                }
            }
        }
    }

    printf("The minimum element above the secondary diagonal is %d:\n"
         , min);

    return 0;
}
```

9. Write a C program that checks whether the sum of the matrix's corner elements is equal to the sum of the elements on the primary diagonal.

Solution:

```
#include <stdio.h>

#define SIZE 10

int main(){

    int a[SIZE][SIZE];
    int i, j, n, sumCorner = 0, sumPrimary = 0;

    printf("Enter the size of the matrix: ");
    scanf("%d", &n);

    printf("Enter the matrix:\n");

    for(i = 0; i < n; i++){
        for(j = 0; j < n; j++){
            printf("a[%d][%d] = ", i, j);
            scanf("%d", &a[i][j]);

            if(i == j){
                sumPrimary += a[i][j];
            }

            if(i == 0 && j == 0){
                sumCorner += a[i][j];
            }
            else if(i == 0 && j == n - 1){
                sumCorner += a[i][j];
            }
            else if(i == n - 1 && j == 0){
```

```
                sumCorner += a[i][j];
            }
            else if(i == n - 1 && j == n - 1){
                sumCorner += a[i][j];
            }
        }
    }

    if(sumCorner == sumPrimary){
        printf("The sums are identical\n");
    }
    else{
        printf("The sums are NOT indentical\n");
    }

    return 0;
}
```

10. Write a C program that will check if a matrix is a magic square. The elements of the matrix are entered from the keyboard, and the max dimensions are 20x20. A magic square is a matrix where the sum of each row and column's elements is identical. If the matrix is a magic square, check whether the matrix is a special magic square (A matrix where the sum of the elements on the primary and secondary diagonal is equal to the sum of each row or column).

    Solution:

```
#include <stdio.h>

#define SIZE 20

int main(){

    int a[SIZE][SIZE], i, j, n, sumToCompare, sum = 0;
    int sumPrimary = 0, sumSecondary = 0;
    int isMagicSquare = 1, isSpecialMagicSquare = 1;

    printf("Enter the size of the matrix: ");
    scanf("%d", &n);

    printf("Enter the matrix:\n");

    for(i = 0; i < n; i++){
        for(j = 0; j < n; j++){
            printf("a[%d][%d] = ", i, j);
            scanf("%d", &a[i][j]);

            if(i == 0){
                sumToCompare += a[i][j];
            }

            if(i == j){
                sumPrimary += a[i][j];
            }
```

```
            if(i + j == n - 1){
                sumSecondary += a[i][j];
            }
        }
    }

    for(i = 0; i < n; i++){

        sum = 0;

        for(j = 0; j < n; j++){
            sum += a[i][j];
        }

        if(sum != sumToCompare){
            isMagicSquare = 0;
            break;
        }
    }

    if(isMagicSquare){
        printf("The matrix is a magic sqare\n");

        if(sumToCompare == sumSecondary && sumToCompare ==
             sumSecondary){
            printf("Also, the matrix is a special magic square\n");
        }
    }
    else{
        printf("The matrix is NOT a magic sqare\n");
    }

    return 0;
}
```

11. Write a program that will sort the elements of each row of a floating point number matrix in ascending order.

Solution:

```
#include <stdio.h>

#define SIZE 10

int main(){

    float a[SIZE][SIZE], temp;
    int i, j, k, n, m;

    printf("Enter number of rows: ");
    scanf("%d", &n);
    printf("Enter number of columns: ");
    scanf("%d", &m);
```

```
    printf("Enter the matrix:\n");

    for(i = 0; i < n; i++){
        for(j = 0; j < m; j++){
            printf("a[%d][%d] = ", i, j);
            scanf("%f", &a[i][j]);
        }
    }

    for(i = 0; i < n; i++){
        for(j = 0; j < m - 1; j++){
            for(k = j; k < m; k++){
                if(a[i][j] > a[i][k]){
                    temp = a[i][j];
                    a[i][j] = a[i][k];
                    a[i][k] = temp;
                }
            }
        }
    }

    printf("The result matrix is:\n");

    for(i = 0; i < n; i++){
        for(j = 0; j < m; j++){
            printf("%.2f  ", a[i][j]);
        }

        printf("\n");
    }

    return 0;
}
```

12. Write a C program that multiplies two matrices.

Solution:

```
#include<stdio.h>

int main()
{
    int i, j, n, m, k, p;
    float a[100][100], b[100][100], c[100][100];

    printf("Enter the number of rows of the 1st matrix: ");
    scanf("%d", &n);

    printf("Enter the number of columns of the 1st matrix = rows of
         the 2nd matrix: ");
    scanf("%d", &m);

    printf("Enter the number of columns of the 2nd matrix: ");
    scanf("%d", &p);
```

```
    printf("Enter the 1st matrix:\n");
    for(i = 0; i < n; i++)
        for(j = 0; j < m; j++){
            printf("a[%d][%d]=", i, j);
            scanf("%f", &a[i][j]);
        }

    printf("Enter the 2nd matrix:\n");
    for(i = 0; i < m; i++)
        for(j = 0; j < p; j++){
            printf("b[%d][%d]=", i, j);
            scanf("%f", &b[i][j]);
        }

    for(i = 0; i < n; i++)
        for(j = 0; j < p; j++){
            c[i][j] = 0;
            for(k = 0; k < m; k++)
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
        }

    printf("The new matrix is:\n");
    for(i = 0; i < n; i++){
        for(j = 0; j < p; j++)
            printf("c[%d][%d]=%.2f\t", i, j, c[i][j]);
        printf("\n");
    }

    return 0;
}
```

13. Write a C program that will swap the columns with the smallest and biggest product of elements in a matrix.

Solution:

```
#include <stdio.h>

#define SIZE 10

int main(){

    float a[SIZE][SIZE], temp, product = 1;
    float products[SIZE], min, max;
    int i, j, n, m, minI, maxI;

    printf("Enter number of rows: ");
    scanf("%d", &n);
    printf("Enter number of columns: ");
    scanf("%d", &m);

    printf("Enter the matrix:\n");

    for(i = 0; i < n; i++){
        for(j = 0; j < m; j++){
```

```
            printf("a[%d][%d] = ", i, j);
            scanf("%f", &a[i][j]);
        }
    }

    for(j = 0; j < m; j++){
        product = 1;

        for(i = 0; i < n; i++){
            product *= a[i][j];
        }

        products[j] = product;
    }

    max = min = products[0];
    maxI = minI = 0;

    for(j = 0; j < m; j++){
        if(products[j] > max){
            max = products[j];
            maxI = j;
        }
        else if(products[j] < min){
            min = products[j];
            minI = j;
        }

    }

    for(i = 0; i < n; i++){
        temp = a[i][minI];
        a[i][minI] = a[i][maxI];
        a[i][maxI] = temp;
    }

    printf("The result matrix is:\n");

    for(i = 0; i < n; i++){
        for(j = 0; j < m; j++){
            printf("%.2f  ", a[i][j]);
        }

        printf("\n");
    }

    return 0;
}
```

14. Write a C program that removes the row with the smallest sum of elements in a matrix.

Solution:

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

```c
#define SIZE 10

int main(){

    float a[SIZE][SIZE], temp, sum = 0;
    float sums[SIZE], min;
    int i, j, k, n, m, minI;

    printf("Enter number of rows: ");
    scanf("%d", &n);
    printf("Enter number of columns: ");
    scanf("%d", &m);

    printf("Enter the matrix:\n");

    for(i = 0; i < n; i++){
        for(j = 0; j < m; j++){
            printf("a[%d][%d] = ", i, j);
            scanf("%f", &a[i][j]);
        }
    }

    for(i = 0; i < n; i++){
        sum = 0;

        for(j = 0; j < m; j++){
            sum += a[i][j];
        }

        sums[i] = sum;
    }

    min = sums[0];
    minI = 0;

    for(i = 0; i < n; i++){
        if(sums[i] < min){
            min = sums[i];
            minI = i;
        }
    }

    for(j = 0; j < m; j++){
        for(i = 0; i < n; i++){
            if(i == minI){
                for(k = i; k < n - 1; k++){
                    a[k][j] = a[k+1][j];
                }
            }
        }
    }

    n--;

    printf("The result matrix is:\n");

    for(i = 0; i < n; i++){
```

```
        for(j = 0; j < m; j++){
            printf("%.2f  ", a[i][j]);
        }

        printf("\n");
    }

    return 0;
}
```

15. Write a C program that will modify a matrix so that the row with the smallest and biggest sum of elements are merged (sum) into one row.

Solution:

```
#include <stdio.h>

#define SIZE 10

int main(){

    float a[SIZE][SIZE], temp, sum = 0;
    float sums[SIZE], min, max;
    int i, j, k, n, m, minI, maxI;

    printf("Enter number of rows: ");
    scanf("%d", &n);
    printf("Enter number of columns: ");
    scanf("%d", &m);

    printf("Enter the matrix:\n");

    for(i = 0; i < n; i++){
        for(j = 0; j < m; j++){
            printf("a[%d][%d] = ", i, j);
            scanf("%f", &a[i][j]);
        }
    }

    for(i = 0; i < n; i++){
        sum = 0;

        for(j = 0; j < m; j++){
            sum += a[i][j];
        }

        sums[i] = sum;
    }

    min = max = sums[0];
    minI = maxI = 0;

    for(i = 0; i < n; i++){
        if(sums[i] > max){
            max = sums[i];
```

```
            maxI = i;
        }

        if(sums[i] < min){
            min = sums[i];
            minI = i;
        }
    }

    for(j = 0; j < m; j++){
        for(i = 0; i < n; i++){
            if(i == minI){
                a[maxI][j] += a[minI][j];
                for(k = i; k < n - 1; k++){
                    a[k][j] = a[k+1][j];
                }
            }
        }
    }

    n--;

    printf("The result matrix is:\n");

    for(i = 0; i < n; i++){
        for(j = 0; j < m; j++){
            printf("%.2f  ", a[i][j]);
        }

        printf("\n");
    }

    return 0;
}
```

CHAPTER 9

Pointers

Pointers are special types of variables that always store addresses of other memory locations, i.e., point to a memory location at the given address. The pointer contains an unsigned positive integer interpreted as a memory address where another variable is stored. Variables contain the value, i.e., direct referencing is used. The pointers contain variable addresses, i.e., indirect referencing is used. For better understanding the memory address concept, please see Figure 9.1.

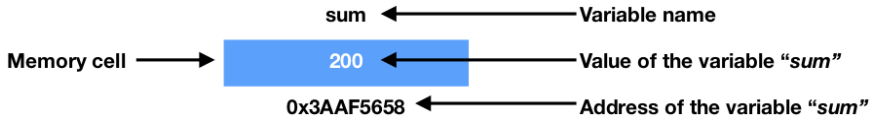

Figure 9.1: Memory address concept.

Memory space is allocated for every variable. It holds the value of that variable. This memory space has an address. For example, we live in a home. Our home has an address, which helps others to find our home. Similarly, the value of the variable is stored in a memory address, which helps the computer to find that value when is needed.

The reason why pointers are so commonly used in programming languages is rarely mentioned, but we can list several reasons for that, including:

- Complex data structures are easy to manipulate with pointers;
- Pointers enable efficient tools for accessing big memory sets;
- Pointers enable work with dynamically allocated memory.

9.1 Pointers in C

In the C programming language, the value of each pointer is a positive unsigned integer (memory address). However, the interpretation of the value stored in that memory location depends on the pointer type. Every pointer is defined with a type. The type refers to the variable that it points to. When a pointer is declared, its data type must be declared.

The format of the statement for declaring a pointer is as follows:

```
type *pointerName;
```

where:

- **type** is the data type of the variable that the pointer will point to. Pointers can be declared from any data type.
- **pointerName** is the name of the pointer.
- *** (asterisk)** is dereferencing operator. It will be explained in detail later in this chapter.

Exercise: Declare four pointers: two that points to integer variables, one that points to a floating-point variable, and one that points to a variable from type character.

Solution:

```
int *ptr, *ptr1;
float *ptrF;
char *ptrC;
```

Pointer declaration is read in the reverse order, i.e., first, we read the pointer's name, followed by the asterisk and the datatype. The asterisk is read as: is a pointer. For example, *int *ptr* is read as: *ptr is a pointer to an integer*.

The operator ***** (asterisk) or dereferencing operator is a prefix operator that returns the memory location's contents in the pointer variable. In the code, the dereferencing operator * is read as *"the content of ..."* or *"the memory location that the pointer points to ..."*.

The operator **&** (ampersand) is a prefix operator that returns the memory address where the variable is stored. The operator **&** is read as *"the address of ..."*.

The operators * and & are inverse. It means that:

- *&ptr -> *(&ptr) -> *(address of ptr) -> ptr
- &*ptr -> &(*ptr) -> &(content of ptr) -> ptr

The initialization of a pointer is done by assigning it with a valid address. Usually, the process is done by assigning the variable address that the pointer will point to. Both the pointer and the variable must be from the same datatype.

```
int a = 0; /* Declaring and initialization of an integer variable. */
int *ptr = &a; /* Declaring and initialization of the ptr to point to a */
```

A pointer can store the address of a variable of the same datatype or a special value NULL (or 0). NULL is used to mark an invalid value for the pointer.

Exercise: What will be the output from the following program?

```c
#include <stdio.h>

int main(){

    int a=10, *ptr=&a;

    printf("a = %d\n", a);
    printf("*ptr = %d\n", *ptr);
    printf("ptr = %p\n", ptr);

    return 0;
}
```

Solution: The program will print on the screen:
a = 10
*ptr = 10
ptr = 0x7ffcf635da54

9.2 Operations with pointers

Pointers are variables as any others, but only a limited number of **arithmetic operations** can be performed over them, including:

- Increment or decrement of a pointer (++ or –). By performing this arithmetic operation, the pointer's value is incremented or decremented, so the pointer will point to the next or previous memory location.
- Assigning an integer value to a pointer (+ or += , - or -=). By performing this arithmetic operation, the pointer's value is increased or decreased by some fixed size N, so the pointer will point to N positions after or before the current memory location.
- Subtraction of pointers. By performing this arithmetic operation, the result of the subtraction will be the number of elements between the two addresses.

Examples with arithmetic operations with pointers:

- **ptr = ptr + 1;** or **ptr++;**. Both expressions are identical. They enable *ptr* to point to the next memory element, after the element that *ptr* previously points to.
- **ptr1 = ptr + N;**. *ptr1* points to a data element placed *N* positions after the data element to which *ptr* points to.
- **n = ptr1 - ptr;**. *n* is an integer variable that stores the number of elements between *ptr1* and *ptr*.

Pointers can only be assigned an address (value) of another pointer of the same type. Pointers of the same type must be used in **assignment operations**. If the pointers are not of the same type, the **CAST operation** should be used. Pointers of

type void *(void *)* are an exception to this rule. Void pointers or generic pointers can point to any datatype. No CAST operation is necessary to assign the value of the pointer into a VOID pointer.

Examples with CAST and assignment operations with pointers:

```
#include <stdio.h>

int main(){

    int a=10, b;
    int *ptr, *ptr1;
    double c, *ptrD;
    void *ptrV;

    ptr=&a;
    ptr1=ptr;
    ptr1=ptr+5;
    ptrV=ptr;
    ptr1=(int *)ptrV;
    ptrD=&c;
    ptrD=(double *)ptr;

    return 0;
}
```

Relational operations can also be used with pointers. Any relational expression (<, <=, ==, !=, >=, >) can be performed over them.

Examples with arithmetic operations with pointers:

- **ptr == ptr1;**. This expression checks whether *ptr* and *ptr1* points to the same memory location.
- **ptr1 < ptr;**. This expression checks whether the element that *ptr1* points to is before the memory location that *ptr* points to.

The pointer must have a value before dereferencing it. An uninitialized pointer points to a random unknown memory location. The attempt to place a value in an uninitialized location gives the error run-time error, or worse, a logical error.

```
/* Invalid - place a value in an uninitialized location gives a error */
int *ptr;
*ptr = 10;
```

In order to change the contents of variables with a pointer, the pointer must point to that variable.

```
int a, *ptr;
ptr = &a;
*ptr = 10; /* This will change the value of (initialize) a to 10. */
```

Exercise: What will be the output from the following program?

```c
#include <stdio.h>

int main(){

    int a, b, c;
    int *ptr;

    ptr = &a;
    *ptr = 10;

    ptr = &b;
    *ptr = 20;

    ptr = &c;
    *ptr = 30;

    printf("a = %d\n", a);
    printf("b = %d\n", b);
    printf("c = %d\n", c);

    return 0;
}
```

Solution: The program will print on the screen:
a = 10
b = 20
c = 30

9.3 Using pointers to functions and arrays

Pointers have a wide application in programming and can be very useful in:

- Passing arguments to functions;
- Accessing elements of arrays and passing arrays as function arguments;
- Accessing dynamically allocated memory;
- Besides pointers to variables, pointers to functions can also be defined.

Besides the standard pointers that we saw in the previous section, three other types of pointers can be defined in the C programming language. Those types are:

- **Constant pointer.** This type of pointers cannot change the address they hold, i.e., they always point to a constant memory location. Once the constant pointer is initialized to point to some variable, it is impossible to change the constant pointer to points to another variable.

  The format of the statement for declaring a constant pointer is as follows:

```c
type *const pointerName;
```

Example:

```
int *const ptr;
```

- **Pointer to constant.** This type of pointers cannot change the value of the variable they point to, i.e., they always point to a constant. A pointer to constant can change the address it holds but cannot change the value inside that address.

  The format of the statement for declaring a pointer to constant is as follows:

```
const type* pointerName;
```

  Example:

```
const int* ptr;
```

- **Constant pointer to a constant.** This type of pointers is a combination of the previous two types. It can neither change the address it points to and nor the value stored inside that address.

  The format of the statement for declaring a constant pointer to a constant is as follows:

```
const type* const pointerName;
```

  Example:

```
const int* const ptr;
```

To demonstrate which is correct and which is a wrong way to use constant pointers, pointers to constant, and constant pointers to a constant, it is presented an example with all three types of pointers in the code below. Please read the comments very carefully, so you can better understand the usage of these pointer types.

```
#include <stdio.h>

int main(){

    int a=0, b=10, c;

    int *const ptrConst = &a; /* Constant pointer */
    c = *ptrConst;
    *ptrConst = b; /* Corect: The content of the memory location that
         ptrConst points to can be changed. */
    ptrConst = &b; /* Not correct: ptrConst is a constant pointer and cannot
          point to a different locationcan. */
   /* error: assignment of read-only variable 'ptrConst' */
```

```
    const int* constPtr = &a; /* Pointer to constant */
    c = *ptrConst;
    constPtr = &b; /* Corect: ptrConst can be redirected to point to another
         location. */
    *constPtr = b; /* Not correct: the value of the variable that constPtr
        points to cannot be changed. */
   /* error: assignment of read-only location '*constPtr' */

    const int* const constPtrConst = &a; /* Constant pointer to constant */
    c = *constPtrConst;
    constPtrConst = &b; /* Not correct: constPtrConst is a constant pointer
        and cannot point to a different locationcan. */
   /* error: assignment of read-only variable 'constPtrConst' */
    *constPtrConst = b; /* Not correct: the value of the variable that
        constPtrConst points to cannot be changed. */
   /* error: assignment of read-only location '*constPtrConst' */
    return 0;
}
```

The **name of the array** is a **constant pointer** that points to the **first element of the array**. For example, if the array is declared as *int arr[100]*, then *arr* is a constant pointer *(int *const)*, and *arr* keeps the address of the first element of the array (*&arr[0]*).

The expressions ***arr[i]* and **(arr + i)* are equivalent**. They enable access to the element of the array on a position *i*. For example, if the array is declared as *int arr[10]* and a pointer is declared as *int aPtr*, then the following is true:

- aPtr = arr; is same as aPtr = &arr[0];
- arr[3] == *(arr + 3);
- arr+5 == &arr[5]
- *(arr+i) == arr[i] == i[arr]

When access to an element of an array is made by using the name and the (*arr[i]*), the compiler always internally interpreted that as **(arr+i)*. Therefore, for instance, instead of *arr[5]*, it is correct (but not recommended) to use *5[arr]*.

Arithmetic operations over pointers are beneficial when work with arrays. For instance, if the pointer that points to an array is incremented for the value of 1, it will start to point to the next element of the array.

Examples with pointers and arrays:

```
#include <stdio.h>

int main(){

    int arr[10], a, b, *aPtr, *aPtr1;

    aPtr = &arr[0]; /* The address of arr[0] is placed in aPtr. */
    aPtr = arr;  /* Same as the previous expression. */
    aPtr1 = &arr[2]; /* The address of arr[2] is placed in aPtr1. */
```

```
    aPtr1 - aPtr == 2; /* True. The number of elements between aPtr and
         aPtr1 is 2. */
    aPtr++; /* Set aPtr to points to arr[1]. */
    a=*aPtr; /* The value of arr[1] is placed in a. */
    aPtr1 = aPtr1 + 5; /* Set aPtr1 to points to arr[7]. */
    b = *(aPtr1-3); /* The value of arr[4] is placed in b. */

    return 0;
}
```

Passing arguments to a function can be done **with pointers**. This enables the function to change the contents of the argument. When a call to the function is made, the argument's address is passed with the & operator. Example:

```
void product(int *a){
*a = 10 * (*a);
}
```

and somewhere in the code, there is:

```
product(&a);
```

Arrays can also be **passed to a function** as pointers. Usually, the number of elements in the array is also passed as an argument to the function. As stated in Chapter 7, when an array is passed as an argument to a function, the programmer needs to use square brackets [] in the function definition. It is also possible the programmer to use pointer notation *, instead of square brackets []. Both ways are identical and have the same meaning since the array are always passed to the function by reference. The format when passing an array by using pointer notation is as follows:

```
Type_of_returned_value someFunction(type_of_array *nameOfArray, int size)
{
/* body of the function */
...
}
```

For example, the definitions of the following functions are identical:

```
double max(double a[], int size)
{
/* body of the function */
...
}

is same as:

double max(double *a, int size)
{
/* body of the function */
...
}
```

Passing an array to a function is performed by passing the address of the first element of the array, i.e., it is **passed to the function by reference**. Therefore, all changes in the function over the values of the array‘s elements are visible after the function ends.

Exercises:

1. Write a C function that swaps the values of two numbers.

   Solution:

```c
#include <stdio.h>

void swap(float *a, float *b){
    float temp = *a;
    *a = *b;
    *b = temp;
}

int main(){

    float a, b;

    printf("Enter the number a = ");
    scanf("%f", &a);

    printf("Enter the number b = ");
    scanf("%f", &b);

    swap(&a, &b);

    printf("The swapped numbers are a = %.2f and b = %.2f\n", a, b);
    return 0;

}
```

2. Write a program in C that calculates the sum of two numbers using pointers.

   Solution:

```c
#include <stdio.h>

int main(){

    int a, b;
    int sum;

    int *c = &sum;

    printf("Enter the first number: ");
    scanf("%d", &a);

    printf("Enter the second number: ");
```

```
    scanf("%d", &b);

    *c = a + b;

    printf("The total sum is %d\n", sum);

    return 0;
}
```

3. Write a C program that copies the elements of one array to another by using pointers.

Solution:

```
#include <stdio.h>

int main(){

    int a[100], b[100], i, n;

    printf("Enter the number of elements of the array: ");
    scanf("%d", &n);

    for(i = 0; i < n; i++){
        printf("a[%d]=", i);
        scanf("%d", (a+i));

        *(b + i) = *(a + i);
    }

    printf("The new array is:\n");
    for(i = 0; i < n; i++){
        printf("b[%d] = %d\n", i, *(b+i));
    }

    return 0;
}
```

4. Write a C program to swap the elements of two arrays using pointers.

Solution:

```
#include <stdio.h>

void enterArray(float *a, int n, char arrayName){

    int i;

    for(i = 0; i < n; i++){
        printf("%c[%d] = ", arrayName, i);
        scanf("%f", (a+i));
    }
```

```
}

void swap(float *a, float *b){
    float temp = *a;
    *a = *b;
    *b = temp;
}

void swapElements(float *a, float *b, int n){
    int i;

    for(i = 0; i < n; i++){
        swap((a+i), (b+i));
    }
}

void printArray(float *a, int n, char arrayName){

    int i;

    for(i = 0; i < n; i++){
        printf("%c[%d] = %.2f\n", arrayName, i, *(a+i));
    }

}

int main(){

    float a[100], b[100];
    int n;

    printf("Enter the number of elements of the array: ");
    scanf("%d", &n);

    printf("Enter the first array:\n");
    enterArray(a, n, 'a');

    printf("Enter the second array:\n");
    enterArray(b, n, 'b');

    swapElements(a, b, n);

    printf("The first array is:\n");
    printArray(a, n, 'a');

    printf("The second array is:\n");
    printArray(b, n, 'b');

    return 0;
}
```

5. Write a C program to find the smallest element of an array using a function and pointers notation.

   Solution:

```c
#include <stdio.h>

void enterArray(float *a, int n){

    int i;

    for(i = 0; i < n; i++){
        printf("a[%d] = ", i);
        scanf("%f", (a+i));
    }

}

void findSmallest(float *a, int n, float *min){
    int i, j;

    *min = *a;

    for(i = 1; i < n; i++){
        if(*(a+i) < *min){
            *min = *(a+i);
        }
    }
}

int main(){

    float a[100], min;
    int n;

    printf("Enter the number of elements of the array: ");
    scanf("%d", &n);

    printf("Enter the array:\n");
    enterArray(a, n);

    findSmallest(a, n, &min);

    printf("The smallest element is %.2f\n", min);

    return 0;
}
```

6. Write a C program to sort an array in descending order using pointers.

Solution:

```c
#include <stdio.h>

void enterArray(float *a, int n){

    int i;

    for(i = 0; i < n; i++){
```

```
        printf("a[%d] = ", i);
        scanf("%f", (a+i));
    }

}

void swap(float *a, float *b){
    float temp = *a;
    *a = *b;
    *b = temp;
}

void sortArray(float *a, int n){
    int i, j;

    for(i = 0; i < n - 1; i++){
        for(j = i; j < n; j++){
            if(*(a+i) > *(a+j)){
                swap((a+i), (a+j));
            }
        }
    }
}

void printArray(float *a, int n){

    int i;

    for(i = 0; i < n; i++){
        printf("a[%d] = %.2f\n", i, *(a+i));
    }

}

int main(){

    float a[100];
    int n;

    printf("Enter the number of elements of the array: ");
    scanf("%d", &n);

    printf("Enter the array:\n");
    enterArray(a, n);

    sortArray(a, n);

    printf("The sorted array is:\n");
    printArray(a, n);

    return 0;
}
```

7. Write a C function that will accept one array of real numbers and the numbers of elements of the array. The function should modify the array so that all numbers with more than 2 divisors are removed (Divisors 1 and the number

itself are excluded).

- For determining the number of divisors define a function int *numberOfDivisors*(int);
- Solve the problem using pointers.

Solution:

```c
#include <stdio.h>

void enterArray(int *a, int n){

    int i;

    for(i = 0; i < n; i++){
        printf("a[%d] = ", i);
        scanf("%d", (a+i));
    }

}

int numberOfDivisors(int n){
    int total = 0;
    int j = 2;

    while(j < n){
        total += !(n % j++);
    }

    return total;
}

void modifyArray(int *a, int *n){

    int i = 0, j;

    while(i < *n){
        if(numberOfDivisors(*(a+i)) > 2){
            for(j = i; j < *n-1; j++){
                *(a+j) = *(a+j+1);
            }
            *n -= 1;
        }
        else{
            i++;
        }
    }

}

void printArray(int *a, int n){

    int i;

    for(i = 0; i < n; i++){
        printf("a[%d] = %d\n", i, *(a+i));
    }
```

```
}

int main(){

    int a[100], n;

    printf("Enter the number of elements of the array: ");
    scanf("%d", &n);

    printf("Enter the array:\n");
    enterArray(a, n);

    modifyArray(a, &n);

    printf("The modified array is:\n");
    printArray(a, n);

    return 0;
}
```

8. Write a C function that accepts one array of numbers as input parameters and the number of elements of the array. The function will remove all elements in an array smaller than the largest negative number of the array. Use a function for determining the largest negative number of the array. Write a *main*() function to test your code.
*__Note__: Use pointers notation.

Solution:

```
#include <stdio.h>

void enterArray(float *a, int n){

    int i;

    for(i = 0; i < n; i++){
        printf("a[%d] = ", i);
        scanf("%f", (a+i));
    }

}

float findLargestNegative(float *a, int n){

    int i;
    float maxNegative = 1;

    for(i = 0; i < n; i++){

        if(*(a+i) < 0){
            if(maxNegative == 1){
                maxNegative = *(a+i);
            }
```

```
            if(*(a+i) > maxNegative){
                maxNegative = *(a+i);
            }
        }
    }

    return maxNegative;

}

void modifyArray(float *a, int *n){

    int i = 0, j;

    float largestNegative = findLargestNegative(a, *n);

    if(largestNegative == 1){
        printf("The array does not have any negative numbers.\n\n");
    }
    else{
        while(i < *n){
            if(*(a+i) < largestNegative){
                for(j = i; j < *n-1; j++){
                    *(a+j) = *(a+j+1);
                }
                *n -= 1;
            }
            else{
                i++;
            }
        }
    }
}

void printArray(float *a, int n){

    int i;

    for(i = 0; i < n; i++){
        printf("a[%d] = %.2f\n", i, *(a+i));
    }

}

int main(){

    float a[100];
    int n;

    printf("Enter the number of elements of the array: ");
    scanf("%d", &n);

    printf("Enter the array:\n");
    enterArray(a, n);

    modifyArray(a, &n);
```

```
    printf("The modified array is:\n");
    printArray(a, n);

    return 0;
}
```

9. Write a C function that accepts one array of numbers and the number of elements of the array. The function will calculate the sum of each two neighbouring elements from the array so each new element is the result of the addition. This procedure is repeated until one number left. The function returns this number. Write a main() function to test the previously defined function.
*__Note__: Do not use [x] (squared brackets). Use only pointer notation (*).

Solution:

```
#include <stdio.h>

void enterArray(float *a, int n){

    int i;

    for(i = 0; i < n; i++){
        printf("a[%d] = ", i);
        scanf("%f", (a+i));
    }

}

void printArray(float *a, int n){

    int i;

    for(i = 0; i < n; i++){
        printf("%.1f  ", *(a+i));
    }

}

float rearrangeArray(float *a, int n){

    int i;

    while(n > 1){
        for(i = 0; i < n-1; i++){
            *(a+i) += *(a+i+1);
        }
        n--;
        printArray(a, n);
        printf("\n");
    }

    return *a;
```

```
}

int main(){

    float a[100], total;
    int n;

    printf("Enter the number of elements of the array: ");
    scanf("%d", &n);

    printf("Enter the array:\n");
    enterArray(a, n);

    total = rearrangeArray(a, n);

    printf("The total is %.2f\n", total);

    return 0;
}
```

CHAPTER 10

Strings

Strings are sequences (arrays) of characters. Many programming languages have e regularly defined data type for strings. It means that all details about the use of arrays are implemented in the compiler and the executive environment. Typical examples include Basic, Turbo Pascal, Scheme/Lisp, Java, C#, etc.

10.1 Strings in C

In the C programming language, there is no standard data type defined for strings. There is an agreement for strings to be stored in arrays of chars. According to the convention, the string's end is marked with a NULL terminator (a symbol with an ASCII code 0).

To declare a string, an array of characters should be declared, or a pointer to be created.

The format of the statement for declaring a string is as follows:

```
char nameOfString[NumberOfCharacters];

OR

char *nameOfString;
```

Working with arrays of characters in C is, in fact, working with pointers to NULL-terminated arrays of chars. The standard library *string.h* provides a large number of functions for working with arrays of characters.

Below, examples of declaration and initialization of strings are given.

```
char c[100];
char *c1;
char c2[5] = "ABC";
char *c3 = c2;
char *c4 = "Test string";
char s[] = "test";
```

In the last example, *char s[] = "test"*, the array will be initialized with five elements, although only four characters are in the string. The compiler will

automatically add one extra slot for the NULL terminator and add it at the end of the string. Figure 10.1 shows the memory allocation for this example.

Figure 10.1: Memory allocation when a string is initialized

The null terminator is automatically removed by the compiler when the string is printed on the screen. Thus, all operations related to the null terminator are hidden from the programmer with a high abstraction level.

Let's suppose that the null terminator is not presented at the end of the string. In that case, the printf() function will print all symbols until the symbol for null terminator is found even outside of the array's range, or the program will end with an error (segment fault etc.).

Let's suppose that we define two textual arrays as follows:

```
char s1[10] = "test";
char *s2 = "string";
```

When we work with the above arrays of characters, the statement s2 = s1 doesn't enable copying the arrays. The reason is that both s2 and s1 are pointers to the first element of the array. Hence only the address from s1 is copied into s2. But in a case we print them, the output on the screen will be identical "test" for both of them. The reason is that after the execution of s2 = s1, both of them points to the same location, i.e. the location of s1.

Example of initialization of array of characters without null terminator:

```
#include <stdio.h>

int main()
{
    char s[10];
    s[0] = 't';
    s[1] = 'e';
    s[2] = 's';
    s[3] = 't';
    printf("%s\n", s);

    return 0;
}
```

In the example above, the initialization is performed by entering the arrays elements character by character. In that case, the null terminator will be not added at the end of the string. Hence, the output of the *printf()* function will be similar to:

```
test%z- ml3
```

10.2 Entering and printing strings

Entering strings can be done in several ways. The first way is by using the ***scanf()*** function. The format of the statement for entering string by using the *scanf()* function is as follow:

```
scanf("%s", str);
```

The address operator & is omitted because the name of the string is already the address of the first character. The *scanf()* function jumps over blank spaces. It takes all symbols and puts them in *str* until it found a blank space. *scanf()* always adds \0 (null terminator) at the end of the array. An array of characters entered with *scanf()* will never contain blank spaces. It is not recommended to use *%s* in *scanf()*, *fscanf()*, and *sscanf()* because of security reasons. The reason is that the compiler did not check whether there is enough space to store the string in the user-defined array of character. It is always good practice to specify the maximum number of characters that the array can accept ex. to use %9s.

Example

```
#include <stdio.h>

int main()
{
    char s[10];
    scanf("%s",s); /* bad */
    scanf("%9s",s); /* good */
    printf("%s",s);

    return 0;
}
```

Another way to enter a string is by using the ***getchar()*** function. The code below is an example of filling an array of characters. It reads characters (including blank spaces) until the user presses the ENTER key or enters the maximum number of characters, i.e., nine characters.

```
#include <stdio.h>
#define N 10

int main() {
    char c, s[N];
    int i;
    for(i=0; i<(N-1)&&(c=getchar())!='\n';i++)
        s[i]=c;

    s[i]='\0';

    return 0;
}
```

The ***gets()*** function allows entering string by reading a line entered from the keyboard in a buffer (memory space) that *s* points to until it found a \n (new line) or

the EOF (end of file) symbol. The function replaces those symbols with the null terminator \0. It also reads the blank spaces.The general structure of the function is as follows:

```
char *gets(char *s);
```

The function *gets()* does not check if the memory is sufficient to store all characters entered from the keyboard. Hence, the programmer must use the function carefully because of security reasons.
Example

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
char s[100];

int main() {
    gets(s);
    printf("buffer = %s\n", s);

    return 0;
}
```

Printing strings can be done in several ways. The first way is by using the ***printf()*** function. The format of the statement for entering string by using the *printf()* function is as follow:

```
printf("%s",str);
```

Another way to enter a string is by using the ***puts()*** function. The \n (new line) character is automatically added at the end of the string when printed with *puts()* function.
Example

```
#include <stdio.h>

int main() {
    char s[]="Test string";
    puts(s); /* adds \n after s */


    return 0;
}
```

Both functions, printf() and puts(), do not print the null terminator \0.

10.3 String manipulation - *strng.h* library

The *string.h* library contains prototypes of various functions for string manipulation. Table 10.1 describes the commonly used functions from the *string.h* library.

Function	**Description**
strlen(s)	Returns the length of the string *s*.
strcmp(s1,s2)	Compares strings *s1* and *s2*.
strncmp(s1,s2,n)	Compares at most the first *n* characters of strings *s1* and *s2*.
strcpy(s1,s2)	Copies the content of *s2* into *s1*.
strncpy(s1,s2)	Copies up to *n* characters from string *s2* into *s1*.
strcat(s1,s2)	Appends string *s2* to the end of *s1*.
strncat(s1,s2)	Appends up to *n* characters from string *s2* to the end of *s1*.
strchr(s,c)	Finds the first occurrence of the character *c* in the string *s*.
strrchr(s,c)	Finds the last occurrence of the character *c* in the string *s*.
strstr(s1,s2)	Finds the first occurrence of the string *s2* in the string *s1*.
strpbrk(s1,s2)	Finds the first character in the string *s1* that matches any character specified in *s2*.
strspn(s1,s2)	Returns the length of the first segment of *s1*, which consists entirely of characters in *s2*.
strcspn(s1,s2)	Returns the length of the first segment of *s1*, which consists entirely of characters NOT in *s2*.
strtok(s,c)	Breaks string *s* into a series of tokens separated by the delimiter *c*.
memchr(s,c,n)	Finds for the first occurrence of the character *c* in the first *n* characters of the string *s*.
memcmp(s1,s2,n)	Compares the first *n* characters of strings *s1* and *s2*.
memcpy(s1,s2,n)	Copies *n* characters from string *s2* to *s1*.
memset(s,c,n)	Copies the character *c* to the first *n* positions of the string *s*.

Table 10.1: List of functions from *strng.h* library

Implementation and full description of the commonly used functions from *string.h* library, with some simple examples, is given in Appendix A.1 of this book.

10.4 Testing and mapping characters - *ctype.h* library

The *ctype.h* library contains prototypes of various functions for testing and mapping characters. Table 10.2 describes the commonly used functions from the *ctype.h* library.

Function	Description
isalpha(c)	Checks whether *c* is an alphabetic character.
isdigit(c)	Checks whether *c* is a decimal digit character.
isalnum(c)	Checks whether *c* is an alphanumeric character.
iscntrl(c)	Checks whether *c* is a control character.
ispunct(c)	Checks whether *c* is a punctuation character.
isspace(c)	Checks whether *c* is a white-space character.
isprint(c)	Checks whether *c* is a printable character.
isxdigit(c)	Checks whether *c* is a hexadecimal digit character.
islower(c)	Checks whether *c* is a lowercase letter character.
isupper(c)	Checks whether *c* is a uppercase letter character.
tolower(c)	Converts uppercase letter to lowercase.
toupper(c)	Converts lowercase letters to uppercase.

Table 10.2: List of functions from *ctype.h* library

Implementation and full description of the commonly used functions from *ctype.h* library, with some simple examples, is given in Appendix A.2 of this book.

Exercises:

1. Write a C program that merges two strings.

Solution:

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

int main(){

    char str1[100], str2[100];
    char result[200];

    printf("Enter the first string: ");
    gets(str1);

    printf("Enter the second string: ");
    gets(str2);

    strcat(result, str1);
```

```
    strcat(result, str2);

    printf("The result string is: %s\n", result);

    return 0;
}
```

2. Write a C program that counts the number of dots (.), commas (,) and blank spaces in a string entered by the user.

   Solution:

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

int countCharacters(char *str){
    int i = 0, count = 0;

    while (str[i] != '\0'){
        count += str[i] == ',' || str[i] == '.' || isspace(str[i]);
        i++;
    }

    return count;
}

int main(){

    char str[100];

    printf("Enter the string: ");
    gets(str);

    printf("The total count is %d\n", countCharacters(str));

    return 0;
}
```

3. Write a C program that checks if a string entered by the user is a palindrome.

   Solution:

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

int isPalindrome(char *str){
    int i = 0, check = 1;

    int length = strlen(str);

    for(i = 0; i < length/2; i++){
        if(str[i] != str[length - i - 1]){
```

```
            check = 0;
            break;
        }
    }

    return check;
}

int main(){

    char str[100];

    printf("Enter the string: ");
    gets(str);

    if(isPalindrome(str)){
        printf("The entered string is a palindrome.\n");
    }
    else{
        printf("The entered string is NOT a palindrome.\n");
    }

    return 0;
}
```

4. Write a C program that checks if the password entered by the user is correct. The password is correct if the length is over 8 characters, and consists of at least one lower case letter, one number, one special character and one upper case letter.

   Solution:

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

int checkPassword(char *str){
    int i = 0, length;
    int hasLower = 0, hasUpper = 0;
    int hasNumber = 0, hasSpecial = 0;

    length = strlen(str);

    if(length <= 8){
        return 0;
    }

    while(str[i] != '\0'){
        if(isalnum(str[i])){
            if(isdigit(str[i])){
                hasNumber = 1;
            }
            else{
                if(isupper(str[i])){
                    hasUpper = 1;
```

```
                }
                else{
                    hasLower = 1;
                }
            }
        }
        else{
            hasSpecial = 1;
        }
        i++;
    }

    return hasLower && hasUpper && hasNumber && hasSpecial;
}

int main(){

    char str[100];

    printf("Enter the password: ");
    gets(str);

    if(checkPassword(str)){
        printf("The entered password is correct.\n");
    }
    else{
        printf("The entered string is NOT correct.\n");
    }

    return 0;
}
```

5. Write a C function that accepts one array of characters as an input parameter. The function will sort the characters of the string alphabetically. Write a *main()* function to test the previously defined function.

   Solution:

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

void sortString(char *str){
    int i, j;
    int length = strlen(str);

    char temp;

    for(i = 0; i < length - 1; i++){
        for(j = i; j < length; j++){
            if(str[i] > str[j]){
                temp = str[i];
                str[i] = str[j];
                str[j] = temp;
            }
        }
```

```
    }
}

int main(){

    char str[100];

    printf("Enter the string: ");
    gets(str);

    sortString(str);

    printf("The sorted string is: %s\n", str);

    return 0;
}
```

6. Write a C function that accepts two arrays of characters as an input parameter. The function should remove all characters from the first array that appear in the second array.

Solution:

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

int inArray(char ch, char *str){
    int i = 0, length = strlen(str);

    while(str[i] != '\0'){
        if(str[i++] == ch){
            return 1;
        }
    }

    return 0;
}

void removeCharacters(char *str1, char *str2){

    int i = 0, j, length = strlen(str1);

    while(i < length){
        if(inArray(str1[i], str2)){
            for(j = i; j < length-1; j++){
                str1[j] = str1[j+1];
            }
            length -= 1;
        }
        else{
            i++;
        }
    }

    str1[length] = 0;
```

```
}

int main(){

    char str1[100], str2[100];
    char result[200];

    printf("Enter the first string: ");
    gets(str1);

    printf("Enter the second string: ");
    gets(str2);

    removeCharacters(str1, str2);

    printf("The result string is: %s\n", str1);

    return 0;
}
```

7. Write a function that returns the ratio of lowercase to uppercase letters in an array of chars that is accepted as an input parameter. After the function is finished, the array of chars should have the first letter in uppercase, and all the others in lowercase. Write a *main()* function that gets a text from the keyboard, uses the previous function and prints the resulting ratio and the resulting text.
   Example:
   Input: thiS IS test STring
   Result: This is test string, and the returned value (ratio) is 2.2.

   Solution:

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

float getRatio(char *str){

    int i = 0, length = strlen(str);
    int lower = 0, upper = 0;

    while(str[i] != '\0'){
        upper += isupper(str[i]);
        lower += islower(str[i]);

        str[i] = !i ? toupper(str[i]) : tolower(str[i]);
        i++;
    }

    return upper > 0 ? lower/upper : 1;

}

int main(){
```

```
    char str[100];

    printf("Enter the first string: ");
    gets(str);

    float ratio = getRatio(str);

    printf("The ratio of lowercase to uppercase characters %.1f\n",
        ratio);
    printf("The modified string is %s\n", str);

    return 0;
}
```

8. Write a C function that accepts one array of characters as an input parameter. The function will change all uppercase characters to lowercase characters, and it will remove all special characters (!@#$ . . .) and blank spaces. Also, the function will return the number of blank spaces that were removed. Write *main()* function to implement this function.

   Solution:

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

int modifyString(char *str){

    int i = 0, j = 0, length = strlen(str);
    int spaces = 0;

    char newString[length];

    while(str[i] != '\0'){
        if(isalnum(str[i])){
            newString[j] = islower(str[i]) ? toupper(str[i]) : tolower
                (str[i]);
            j++;
        }
        else if(isspace(str[i])){
            spaces++;
        }

        i++;
    }

    newString[j] = 0;
    strcpy(str, newString);

    return spaces;
}

int main(){
```

```
    char str[100];

    printf("Enter the first string: ");
    gets(str);

    int spaces = modifyString(str);

    printf("The modified string is %s\n", str);
    printf("Total blank spaces removed: %d\n", spaces);

    return 0;
}
```

9. Write a C function that removes the duplicate successive characters (case insensitive) in one array of characters. Also, the function returns the number of removed characters. Write a *main()* function that gets a text from the keyboard, uses the previous function, and prints the resulting text.
Example:
Input: ThiSs is Programming 11..
Result: This is Programing 1.

Solution:

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

void removeCharacters(char *str){

    int i = 0, j, length = strlen(str);

    while(i < length - 1){
        if(tolower(str[i]) == tolower(str[i+1])){
            for(j = i; j < length-1; j++){
                str[j] = str[j+1];
            }
            length -= 1;
        }
        else{
            i++;
        }
    }

    str[length] = 0;

}

int main(){

    char str[100];

    printf("Enter the first string: ");
    gets(str);
```

```
    removeCharacters(str);

    printf("The result string is: %s\n", str);

    return 0;
}
```

10. Write a C function that accepts one array of characters as an input parameter. The function should replace all non-alphanumeric characters with the character *, and it will return the number of characters that were replaced from the original. Write a *main()* function that will use this function.
Example:
Input: Some@# Exaˆmp&le Te#xt 123%456
Result: Some** Exa*mp*le Te*xt 123*456, and the returned value is 6.

Solution:

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

int modifyString(char *str){

    int i = 0, length = strlen(str);
    int count = 0;

    while(str[i] != '\0'){
        if(!isalnum(str[i])){
            str[i] = '*';
            count++;
        }
        i++;
    }

    return count;

}

int main(){

    char str[100];

    printf("Enter the first string: ");
    gets(str);

    int totalReplaced = modifyString(str);

    printf("The result string is: %s, and the total of replaced
        characters is %d\n", str, totalReplaced);

    return 0;
}
```

11. Write a C function that accepts one array of characters as an input parameter. The function should format and print the phone number “0038970123456” in this format:
“+389 (0) 70 123-456”
Write *main()* function where the user enters the phone number, and the function should print the formatted telephone number. If the entered number is not valid, the program will warn the user with this message “Invalid phone number”.

Solution:

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

// 0038970123456
// 0123456789

void printFormattedPhoneNumber(char *str){

    int i, length = strlen(str);

    if(length != 13){
        printf("Invalid phone number.\n");
    }
    else{
        char formattedPhone[20] = "+";
        char tempStr[100] = "";

        for(i = 2; i < length; i++){
            switch(i){
                case 5:
                    strcpy(tempStr, " (0) ");
                    strcat(formattedPhone, tempStr);
                    break;
                case 7:
                    strcpy(tempStr, " ");
                    strcat(formattedPhone, tempStr);
                    break;
                case 10:
                    strcpy(tempStr, "-");
                    strcat(formattedPhone, tempStr);
                    break;

            }

            strncat(formattedPhone, (str+i), 1);

        }
        printf("The formatted phone number is: %s\n", formattedPhone);
    }
}

int main(){
```

```
    char str[100];

    printf("Enter the phone number: ");
    gets(str);

    printFormattedPhoneNumber(str);

    return 0;
}
```

12. Write a C program that performs coding of a text message. The coding is performed as follows: it starts from the left and takes one letter from each side of the string. The program should stop when the middle letter is reached.
Example:
Input: Test string
Result: Tgensitr ts

Solution:

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

// test string
// tg

void codeString(char *str){

    int i, j = 0;
    int length = strlen(str);

    char newString[length];

    for(i = 0; i < length/2; i++){
        newString[j++] = str[i];
        newString[j++] = str[length - i - 1];
    }

    if(length % 2){
        newString[j++] = str[length/2];
    }

    newString[j] = 0;

    printf("%s\n", newString);

    strcpy(str, newString);

}

int main(){

    char str[100];

    printf("Enter the string: ");
```

```
    gets(str);

    codeString(str);

    // printf("The coded string is %s\n", str);

    return 0;
}
```

CHAPTER 11

Files

A file represents a sequence of bytes that are used to store a large volume of data. The operations described in the previous chapters produce results stored only in the computer's temporary memory. Usually, the program users need these results to be fetched again, which means they need to be written on permanent storage. Using files is a common way for permanent storage of data.

11.1 Working with files in C

C programming language provides predefined functions for file management. The operations that can be performed on a files are as follows:

- Creation of a file.
- Opening a file.
- Reading from file.
- Writing to file.
- Moving to a specific location in a file.
- Closing a file.

The first step in the process of working with files in C is to declare a pointer of type FILE. This pointer will help in communication between the file and the program. The format of the statement for declaring a file pointer is as follows:

```
FILE *nameOfFilePointer;
```

where:

- **nameOfFilePointer** is the name of the file pointer.

Creating a file pointer enables using of the file management functions.

Table 11.1 describes the commonly used functions for file management in C. These functions are part of the *stdio.h* library.

Function	Description
fopen()	Creates a new file or opens an existing file.
fscanf()	Reads a set of data from a file.
fprintf()	Writes a set of data to a file.
getc()	Reads a character from a file.
putc()	Writes a character to a file.
getw()	Reads an integer from a file.
putw()	Writes an integer to a file.
fclose()	Closes a file.
fseek()	Sets the file pointer to a specified location.
rewind()	Sets the file pointer at the beginning of a file.
ftell()	Returns the position of the file pointer.

Table 11.1: List of functions for file management

The function *fopen()* is used to open a file. If the file can not be found in the system, the function will create a new file and then open it. The syntax of *fopen()* function is as follows:

```
fopen("FileName", "Mode");
```

where:

- **FileName** is the name of the file.
- **Mode** is the access mode.

The second argument of the *fopen()* function specifies the access mode. A file, in the C programming language, can be opened for reading or writing. Figure 11.2 describes the different types of file access modes in C.

Mode	Description
r	Open a file for reading.
w	Open a file for writing. Content is overwritten.
a	Open a file for appending. The new content is added to the end of the file.
r+	Open a file for reading and writing.
w+	Open a file for reading and writing.
a+	Open a file for reading and appending.

Table 11.2: List of file access modes

Based on the selected access mode during file opening, we can perform certain operations on the file, i.e., reading or writing, using the functions for that purpose

described in Table 11.1. Both **r+** and **w+** open a file for reading and writing. The difference between these two modes is that in case the file doesn't exist, w+ will create a new file and open it, while r+ will not create a new file. In the same manner, if the file exists, w+ will open the file and delete the existing content, while r+ will not delete the existing content. Similarly to r+, **r** also doesn't create a new file if the file doesn't exist.

After finishing with all file operations, the file needs to be close. It will release any memory that the compiler reserve for the file and prevent accidental damage. The *fclose()* function closes the open file. The function returns 0 if the file is closed successfully or **EOF** if there is an error during the closing. The EOF (End Of File) is a constant defined in the *stdio.h* library. The syntax of *fclose()* function is as follows:

```
fclose("FileName");
```

where:

- **FileName** is the name of the file.

Exercises:

1. Write a simple C Program that opens a file, write in it, and close the file.

   Solution:

```c
#include <stdio.h>

int main(){

    char userInput[1000];

    FILE *file;

    file = fopen("ex1.txt", "w");

    if (file == NULL) {
        printf("Error!");
        return -1;
    }

    printf("Enter the text:\n");
    gets(userInput);

    fprintf(file, "%s", userInput);
    fclose(file);

    return 0;

}
```

2. Write a C program that compares two files. The program should return "Files are same" if the files are identical or the position of the first difference if the

files are not the same.

Solution:

```c
#include <stdio.h>

int compareFiles(FILE *file1, FILE *file2){
    char ch1 = getc(file1);
    char ch2 = getc(file2);

    while (ch1 != EOF && ch2 != EOF){

        printf("CH1: %c\n", ch1);
        printf("CH2: %c\n", ch1);

        if (ch1 != ch2){
            return 0;
        }

        ch1 = getc(file1);
        ch2 = getc(file2);
    }

    return 1;
}

int main(){

    FILE *file1, *file2;

    file1 = fopen("ex2_1.txt", "r");
    file2 = fopen("ex2_2.txt", "r");

    if (file1 == NULL || file2 == NULL) {
        printf("Error!");
        return -1;
    }

    if (compareFiles(file1, file2)){
        printf("The files are identical.\n");
    }
    else{
        printf("The files are NOT identical.\n");
    }

    fclose(file1);
    fclose(file2);

    return 0;

}
```

3. Write a C program that counts the number of words in a given text file. A word consists of alphanumeric characters and is separated by the other word with at least one space.

Solution:

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

int main(){

    char c;
    int nrWords = 0, inWord = 0;
    FILE *file;

    if((file = fopen("ex3.txt", "r")) == NULL){
        fprintf(stderr,"File cannot be opened");
        return -1;
    }

    while((c = fgetc(file)) != EOF){
        if (isalnum(c)){
            if(!inWord)
                inWord = 1;
        }
        else if(inWord){
            inWord = 0;
            nrWords++;
        }
    }

    if(inWord)
        nrWords++;

    printf("Total %d words\n", nrWords);
    return 0;

}
```

4. Write a C program that calculates the ratio of numbers to letters in a textual file. The user enters the name of the file.

Solution:

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

int main(){

    char c;
    int totalNumbers = 0, totalLetters = 0;
    FILE *file;
```

```
    if((file = fopen("ex4.txt", "r")) == NULL){
        fprintf(stderr, "File cannot be opened");
        return -1;
    }

    while((c = fgetc(file)) != EOF){
        totalNumbers += isdigit(c);
        totalLetters += isalpha(c);
    }

    printf("The ratio of numbers to letters is %.2f\n", (float)
         totalNumbers/totalLetters);

    return 0;

}
```

5. Write a C program that will print all words composed of at least three letters from a given file. The user enters the name of the file.

Solution:

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

int main(){

    char c;
    int inWord = 0;
    int count = 0, charsCount = 0;

    FILE *file;

    if((file = fopen("ex5.txt", "r")) == NULL){
        fprintf(stderr,"File cannot be opened");
        return -1;
    }

    while((c = fgetc(file)) != EOF){
        if (isalnum(c)){
            if(!inWord){
                charsCount = 0;
                inWord = 1;
            }

            charsCount++;
        }
        else if(inWord){
            inWord = 0;

            if(charsCount > 3){
                count++;
            }
        }
    }
```

```
    printf("Total %d words with more than 3 letters\n", count);

    return 0;

}
```

6. Write a C program that will count the number of paragraphs with more than five words. A paragraph is a group of words ending with a new line.

Solution:

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

int main(){

    char ch, str[100];

    int inWord = 0, nrWords = 0;
    int i, count = 0;

    FILE *file;

    if((file = fopen("ex6.txt", "r")) == NULL){
        fprintf(stderr,"File cannot be opened");
        return -1;
    }

    while(fgets(str, 100, file) != NULL){

        i = 0;
        ch = str[0];

        nrWords = 0;
        inWord = 0;

        while (ch != '\n'){
            if (isalnum(ch)){
                if (!inWord)
                    inWord = 1;
            }
            else if(inWord){
                inWord = 0;
                nrWords++;
            }

            i++;
            ch = str[i];
        }

        if(inWord)
            nrWords++;

        if(nrWords > 5)
```

```
            count++;
        }

        printf("Total %d paragraphs with more than 5 words\n", count);

        return 0;

    }
```

CHAPTER 12

Structures

A structure is a composite group of variables of different data types grouped in a list under a single name. It allows different variables in the structure that can be accessed via a single pointer. Similarly to arrays, the structures reference a contiguous memory block.

12.1 Structures in C

In the C programming language, a structure is a group of data elements grouped under one name. These data elements, known as members, can have different types and lengths. Structures can be declared using the **struct** keyword. The statement for declaring a structure has the following syntax:

```
struct [structureName] {
    memberType1 memberName1;
    memberType2 memberName2;
    memberType3 memberName3;
    ...
 } [structureVariables];
```

where:

- **structureName** is the name of the structure. Structure name is optional but recommended to be used.
- **memberType** is the datatype of the member element.
- **memberName** is the name of the member element.
- **structureVariables** declares variables from structure data type. Declaration is also optional in this part.

Examples of structure declaration:

```
struct animal {
    char name[50];
    int age;
    float weight;
```

```
 } cat, dog;
```

The above structure type is called animal and defines three members: name, age and weight, each of a different datatype. In the end, this declaration directly creates two variables of this newly defined type: cat and dog. Another way to create variables from this newly defined type is as follows:

```
struct animal {
    char name[50];
    int age;
    float weight;
 };

 ........

struct animal cat,dog;
```

Note that once the animal structure is declared, it is used just like any other datatype. It is important to make difference between the structure type name (animal) and a variable of this type (dog and cat). Many variables (such as dog and cat) can be declared from a single structure type (animal).

12.2 Accessing structure members and initialization

Structure members can be accessed and assigned values using the member access operator (.), also called period. Structure members have no meaning individually without the structure. The member access operator (.) is inserted between the structure variable name and the member name. The syntax of the statement for accessing structure members is as follows:

```
structureVariable.memberName
```

where:

- **structureVariable** is the structure variable name.
- **memberName** is the name of the member element.

For **example**, to access the member elements in the **animal** structure defined in the previous chapter, the syntax will be as follows:

```
cat.name      dog.name
cat.age       dog.age
cat.weight    dog.weight
```

Similar to a variable of any other datatype, the structure variable can also be **initialized** within the declaration. For example, the initialization can be done as follows:

```
struct animal {
    char name[50];
    int age;
    float weight;
 };

struct animal cat = { "Tom" , 7, 2.5 };   //initialization
```

Also, it is possible to initialize each structure member separately. For example, the initialization can be done as follows:

```
struct animal cat;
cat.name = "Tom";   //initialization of each structure member separately
cat.age = 7;
cat.weight = 2.5;
```

12.3 Array of structures and nested structures

C programming language allows declaring an array of structure variables. Each element of such a defined array representing a *struct* variable. For example, we can create an array of *animal* structure declared in section 12.1 as follows:

```
struct animal cat[10];   //array of structures.
```

Nested structure is when one structure is inside another. Multiple structures can be nested together, meaning the structure has one or many structures as member variables. C programming language allows nesting of structures. In the example below, the animalColor variable, which is from struc type color (structure), is a member element of the structure animal.

```
 struct animalColor {
    char head[50];
    char body[50];
    char tail[50];
 };

struct animal {
    char name[50];
    int age;
    float weight;
    struct animalColor color;
 } cat, dog;
```

Another way to declare the above nested structure is as follows:

```
struct animal {
    char name[50];
    int age;
    float weight;
```

```
    struct animalColor {
        char head[20];
        char body[20];
        char tail[20];
    } color;
} cat, dog;
```

The following example shows how to access the elements of nested structure color and how to initialize it.

```
cat.color.body = "Black";
```

12.4 Pointers to structures and structures to function

Like any other datatypes, structures can be pointed to by their own type of pointers. The format of the statement for creating a pointer to a structure is as follows:

```
struct structureName *pointerName;
```

where:

- **structureName** is the name of the structure.
- **pointerName** is the name of the pointer.
- *** (asterisk)** is dereferencing operator.

The pointer to a structure stores the address of a structure variable. The address of the structure variable is assigned by using the '&' operator. The format of the statement for assigning an address is as follows:

```
pointerName = &structureVariable;
```

where:

- **structureVariable** is the name of the structure variable.

The arrow (->) operator is used to access the structure member elements using a pointer. The format of the statement for accessing the structure member elements using a pointer is as follows:

```
pointerName -> memberName;
```

where:

- **memberName** is the name of the member element.

The **example** below demonstrates a simple program for using a pointer to structure.

```c
#include <stdio.h>

struct animal {
    char name[50];
    int age;
    float weight;
 };

int main() {
    struct animal cat = { "Tom" , 7, 2.5 };
    struct animal *catPtr;

    catPtr = &cat;
    printf( "The name of the cat is : %s\n", catPtr -> name);

    return 0;
}
```

Structure variables are passed as arguments to a function similar to the other predefined datatypes. In the example below, the *cat* structure variable is passed to the function *print*.

```c
void print(struct animal cat);
```

The return value of the function can be a structure. In the example below, the readData function returns a structure.

```c
struct animal readData() {
  struct animal dog;

  printf("Enter name: ");
  scanf ("%s", dog.name);

  printf("Enter age: ");
  scanf("%d", &dog.age);

  printf("Enter weight: ");
  scanf("%f", &dog.weight);

  return dog;
}
```

It is also possible to **pass structures by reference** similar to the other predefined datatypes. In the example below, the *cat* structure variable is passed to the function *print* by reference.

```c
void print(struct animal *cat);
```

Exercises:

1. Write a C program that stores information about students using structure. The structure keeps information about the student name (array of characters), ID number (integer) and marks (float).

   Solution:

```c
#include <stdio.h>

struct student{
    char firstName[50];
    int roll;
    float marks;
};

int main(){

    int i;
    student s[10];

    printf("Enter information of students:\n");

    for (i = 0; i < 5; ++i) {
        s[i].roll = i + 1;

        printf("\nFor student number %d:\n", s[i].roll);
        printf("Enter first name: ");
        scanf("%s", s[i].firstName);

        printf("Enter marks: ");
        scanf("%f", &s[i].marks);
    }

    printf("Displaying Information:\n\n");

    for (i = 0; i < 5; ++i) {
        printf("Student number: %d\n", i + 1);
        printf("First name: ");
        puts(s[i].firstName);

        printf("Marks: %.1f", s[i].marks);
        printf("\n");
    }
    return 0;
}
```

2. Write a C function that calculates the sum of two complex numbers. The function should accept two structures and return a structure. Write a main() function to test the previously defined function.

   Solution:

```c
#include <stdio.h>

typedef struct complex {
    float real;
    float imag;
} complex;

complex add(complex n1, complex n2) {
    complex temp;
    temp.real = n1.real + n2.real;
    temp.imag = n1.imag + n2.imag;
    return (temp);
}

int main() {
    complex n1, n2, result;

    printf("For 1st complex number \n");
    printf("Enter the real and imaginary parts: ");

    scanf("%f %f", &n1.real, &n1.imag);
    printf("\nFor 2nd complex number \n");
    printf("Enter the real and imaginary parts: ");

    scanf("%f %f", &n2.real, &n2.imag);

    result = add(n1, n2);

    printf("Sum = %.1f + %.1fi\n", result.real, result.imag);

    return 0;
}
```

3. Write a C program that keeps information and performs analysis for a class of 15 students. The structure consists of information about Student ID, Name, mid-term scores (2 mid-term per semester), total score, and final grade. Create an interactive menu for:
   1. Add a new student
   2. Delete student
   3. Update student info
   4. Print all students
   5. Calculate the final grade from student's points.
   6. Show student who gets the max total score
   7. Show student who gets the min total score
   8. Find student by ID
   9. Sort students by total scores

   Note: You need to create an array of structures.

   Solution:

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

```
struct student{
    int id, midTerm1, midTerm2;
    int totalScore, finalScore;
    char name[100];
};

struct student findStudent(struct student *s, int n, int id){
    int i;

    for(i = 0; i < n; i++){
        if(s[i].id == id){
            return s[i];
        }
    }

    printf("Student with ID=%d cannot be found\n", id);

    struct student temp;

    temp.id = 0;

    return temp;
}

void updateStudent(struct student *s, int n, int id){

    int found = 0;
    int i, j;

    for(i = 0; i < n; i++){
        if(s[i].id == id){
            found = 1;

            printf("Enter the name: ");
            gets(s[i].name);

            printf("Enter student ID: ");
            scanf("%d", &s[i].id);

            printf("Enter MidTerm1 points: ");
            scanf("%d", &s[i].midTerm1);

            printf("Enter MidTerm2 points: ");
            scanf("%d", &s[i].midTerm2);

            s[i].totalScore = s[i].midTerm1 + s[i].midTerm2;
        }
    }

    if(found){
        printf("Student with ID=%d is updated\n", id);
    }
    else{
        printf("Student with ID=%d cannot be found\n", id);
    }

}
```

```
void removeStudent(struct student *s, int *n, int id){

    int i, j;
    int found = 0;

    for(i = 0; i < *n; i++){
        if(s[i].id == id){
            found = 1;
            for(j = i; j < *n-1; j++){
                s[j] = s[j+1];
            }
            *n = *n - 1;
        }
    }

    if(found){
        printf("Student with ID=%d is removed\n", id);
    }
    else{
        printf("Student with ID=%d cannot be found\n", id);
    }

}

void print(struct student s){
    printf("Name: %s, 1st mid-term: %d, 2nd mid-term %d. Total score:
        %d\n", s.name, s.midTerm1, s.midTerm2, s.totalScore);
}

int calculateFinalGrade(struct student s){

    double points = s.totalScore/2.0;

    if(points <= 50){
        return 5;
    }
    else if(points > 50 && points <= 60){
        return 6;
    }
    else if(points > 60 && points <= 70){
        return 7;
    }
    else if(points > 70 && points <= 80){
        return 8;
    }
    else if(points > 80 && points <= 90){
        return 9;
    }
    else if(points > 90 && points <= 100){
        return 10;
    }
    else{
        return 0;
    }

}

struct student findMax(struct student *s, int n){
```

```
    int i = 0;
    struct student max = s[i];

    for(i = 1; i < n; i++){
        if(s[i].totalScore > max.totalScore){
            max = s[i];
        }
    }

    return max;
}

struct student findMin(struct student *s, int n){
    int i = 0;
    struct student min = s[i];

    for(i = 1; i < n; i++){
        if(s[i].totalScore < min.totalScore){
            min = s[i];
        }
    }

    return min;
}

void sortStudents(struct student *s, int n){
    int i, j;
    struct student temp;

    for(i = 0; i < n-1; i++){
        for(j = 1; j < n; j++){
            if(s[i].totalScore < s[j].totalScore){
                temp = s[i];
                s[i] = s[j];
                s[j] = temp;
            }
        }
    }
}

int main(){

    int i = 0, n = 0, selection = 0;

    int id, found = 0;

    char name[100];

    struct student s[100];
    struct student temp;

    while(selection != 10){
        printf("\nSelect action:\n");
        printf("\t1: Add a new student\n");
        printf("\t2: Delete a student\n");
        printf("\t3: Update student info\n");
        printf("\t4: Print all students\n");
        printf("\t5: Calculate the final grade from student points\n")
```

```
        ;
    printf("\t6: Show student who gets the max total score\n");
    printf("\t7: Show student who gets the min total score\n");
    printf("\t8: Find student by ID\n");
    printf("\t9: Sort students by total scores\n");
    printf("\t10: Exit menu.\n");

    scanf("%d", &selection);

    switch (selection){
        case 1:
            printf("Enter the info about the student:\n\n");
            printf("Enter the name: ");
            gets(s[n].name);

            printf("Enter student ID: ");
            scanf("%d", &s[n].id);

            printf("Enter MidTerm1 points: ");
            scanf("%d", &s[n].midTerm1);

            printf("Enter MidTerm2 points: ");
            scanf("%d", &s[n].midTerm2);

            s[n].totalScore = s[n].midTerm1 + s[n].midTerm2;

            n++;

            break;
        case 2:

            printf("Enter the studnet's ID: ");
            scanf("%d", &id);

            removeStudent(s, &n, id);

            break;
        case 3:

            printf("Enter the studnet's ID: ");
            scanf("%d", &id);

            updateStudent(s, n, id);

            break;
        case 4:
            for(i = 0; i < n; i++){
                print(s[i]);
            }
            break;
        case 5:
            printf("Enter the studnet's ID: ");
            scanf("%d", &id);

            temp = findStudent(s, n, id);

            if(temp.id){
                int grade = calculateFinalGrade(temp);
```

```
                    if(grade){
                        temp.finalScore = grade;
                        printf("The final grade is: %d\n", grade);
                    }

                }
                break;
            case 6:
                print(findMax(s, n));
                break;
            case 7:
                print(findMin(s, n));
                break;
            case 8:

                printf("Enter the studnet's ID: ");
                scanf("%d", &id);

                temp = findStudent(s, n, id);

                if(temp.id){
                    print(temp);
                }

                break;
            case 9:
                sortStudents(s, n);
                break;
            case 10:

                break;

        }
    }

    return 0;
}
```

Appendices

APPENDIX A

Commonly used C library functions

This appendix provides implementation and a full description of the commonly used C library functions, with some simple examples.

A.1 Implementation of the functions from *string.h* library

- **size_t strlen(const char *s)**

The function returns the length of the string *s*. The null terminator is not counted. *size_t* is an unsigned integer datatype defined in *string.h* library.

Example with *strlen(s)*:

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

int main() {
   char str[50] = "Hello World!";
   int len;

   len = strlen(str);
   printf("Length of %s is %d\n", str, len);

   return 0;
}
```

The output of the above code will be:

```
Length of Hello World! is 12
```

- **int strcmp(const char *s1, const char *s2)**

The function compares strings *s1* and *s2*. The function returns 0 if the arrays are

equal, 1 if *s1* is greater than *s2* and -1 if *s1* is less than *s2*.

Example with *strcmp(s1,s2)*:

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

int main() {
   char s1[10] = "abba", s2[10] = "aba";

   if(strcmp(s1,s2) < 0) {
      printf("s1 is less than s2");
   } else if(strcmp(s1,s2) > 0) {
      printf("s1 is greater than s2");
   } else {
      printf("s1 is equal to s2");
   }

   return 0;
}
```

The output of the above code will be:

```
s1 is greater than s2
```

- **int strncmp(const char *s1, const char *s2, size_t n)**

Similarly to previous function, this function compares at most the first *n* characters of *s1* and *s2*. The function returns 0 if the arrays are equal, 1 if *s1* is greater than *s2* and -1 if *s1* is less than *s2*.

Example with *strncmp(s1, 2, n)*:

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

int main() {
   char s1[10] = "abba", s2[10] = "aba";

   if(strncmp(s1, s2, 2) < 0) {
      printf("s1 is less than s2");
   } else if(strncmp(s1, s2, 2) > 0) {
      printf("s1 is greater than s2");
   } else {
      printf("s1 is equal to s2");
   }

   return 0;
}
```

The output of the above code will be:

```
s1 is equal to s2
```

- **char* strcpy(char* s1, const char* s2)**

The function copies the string pointed by *s2* (including the null terminator) to the *s1* and does not check if there is enough space to store the content of *s2*. The function returns the copied string.

Example with *strcpy(s1, s2)*:

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

int main() {
   char s1[10] = "Hello", s2[10];

   strcpy(s2, s1);
   printf("%s", s2);

   return 0;
}
```

The output of the above code will be:

```
Hello
```

- **char *strncpy(char *s1, const char *s2, size_t n)**

Similarly to previous function, this function copies up *n* characters of *s2* to the *s1*. The function returns the copied string.

Example with *strncpy(s1, s2, n)*:

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

int main() {
   char s1[10] = "Hello", s2[10];

   memset(s2, '\0', sizeof(s2)); //Add null terminator at the end of s2.
   strncpy(s2, s1, 2);
   printf("%s", s2);

   return 0;
}
```

The output of the above code will be:

```
He
```

- **char *strcat(char *s1, const char *s2)**

The function appends string *s2* to the end of *s1*. The function returns the resulting string.

Example with *strcat(s1, s2)*:

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

int main() {
   char s1[50] = "Hello ", s2[50] = "world!";

   strcat(s1, s2);
   printf("%s", s1);

   return 0;
}
```

The output of the above code will be:

```
Hello world!
```

- **char *strncat(char *s1, const char *s2, n)**

Similarly to previous function, this function appends up *n* characters of *s2* to the end of *s1*. The function returns the resulting string.

Example with *strncat(s1, s2, n)*:

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

int main() {
   char s1[50] = "Hello ", s2[50] = "world!";

   strncat(s1, s2, 3);

   printf("%s", s1);

   return 0;
}
```

The output of the above code will be:

```
Hello wor
```

- **char *strchr(const char *s, int c)**

The function searches for the first occurrence of the character *c* in the string *s*. The function returns a pointer to the first occurrence of the character *c* in the string *s*, or (null) if the character is not found.

Example with *strchr(s, c)*:

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

int main() {
   char s[20] = "Hello world!";
   char c = 'l';

   printf("%s", strchr(s, c));

   return 0;
}
```

The output of the above code will be:

```
llo world!
```

- **char *strrchr(const char *s, int c)**

The function searches for the last occurrence of the character *c* in the string *s*. The function returns a pointer to the last occurrence of the character *c* in the string *s*, or (null) if the character is not found.

Example with *strchr(s, c)*:

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

int main() {
   char s[20] = "Hello world!";
   char c = 'l';

   printf("%s", strrchr(s, c));

   return 0;
}
```

The output of the above code will be:

```
ld!
```

- **char *strstr(const char *s1, const char *s2)**

The function searches for the first occurrence of the string *s2* in the string *s1*. The function returns a pointer to the first occurrence of the string *s2* in the string *s1*, or (null) if the character is not found.

Example with *strstr(s1, s2)*:

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

int main() {
   char s1[20] = "Hello world!";
   char s2[20] = "wor";

   printf("%s", strstr(s1, s2));

   return 0;
}
```

The output of the above code will be:

```
world!
```

- **char *strpbrk(const char *s1, const char *s2)**

The function finds the first character in the string *s1* that matches any character from *s2*. The function returns a pointer to the character in *s1* that matches one of the characters from *s2*, or (null) if the character is not found.

Example with *strpbrk(s1, s2)*:

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

int main() {
   char s1[20] = "Hello world!";
   char s2[20] = "wor";

   printf("%s", strpbrk(s1, s2));

   return 0;
}
```

The output of the above code will be:

```
o world!
```

- **size_t strspn(const char *s1, const char *s2)**

The function returns the length of the first segment of *s1*, which consists entirely of characters in *s2*.

Example with *strspn(s1, s2)*:

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

int main() {
   char s1[20] = "Hello world!";
   char s2[20] = "Hel";
   int len;

   len = strspn(s1, s2);
   printf("The length of the first matching segment is %d", len);

   return 0;
}
```

The output of the above code will be:

```
The length of the first matching segment is 4
```

- **size_t strcspn(const char *s1, const char *s2)**

The function returns the length of the first segment of *s1*, which consists entirely of characters NOT in *s2*.

Example with *strcspn(s1, s2)*:

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

int main() {
   char s1[20] = "Hello world!";
   char s2[20] = "wow";
   int len;

   len = strcspn(s1, s2);
   printf("The first matching character is at position %d", len);

   return 0;
}
```

The output of the above code will be:

```
The first matching character is at position 4
```

A.2 Implementation of the functions from *ctype.h* library

- **int isalpha(int c)**

The function checks whether *c* is an alphabetic character. It returns non-zero value (true) if *c* is an alphabetic character, else it returns 0 (false).

Example with *isalpha(c)*:

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

int main() {
   int c = 'a';

   if(isalpha(c) ) {
      printf("%c is an alphabetic character\n", c );
   } else {
      printf("%c is not an alphabetic character\n", c);
   }
   return 0;
}
```

The output of the above code will be:

```
a is an alphabetic character
```

- **int isdigit(int c)**

The function checks whether *c* is a decimal digit (0 1 2 3 4 5 6 7 8 9). It returns non-zero value (true) if *c* is a decimal digit, else it returns 0 (false).

Example with *isdigit(c)*:

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

int main() {
   int c = '3';

   if(isdigit(c) ) {
      printf("%c is a decimal digit\n", c );
   } else {
      printf("%c is not a decimal digit\n", c);
   }
   return 0;
}
```

The output of the above code will be:

```
3 is a decimal digit
```

- **int isalnum(int c)**

The function checks whether *c* is an alphanumeric character. It returns non-zero value (true) if *c* is a letter or a digit, else it returns 0 (false).

Example with *isalnum(c)*:

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

int main() {
   int c = '3';

   if(isdigit(c) ) {
      printf("%c is an alphanumeric character\n", c );
   } else {
      printf("%c is not an alphanumeric character\n", c);
   }
   return 0;
}
```

The output of the above code will be:

```
3 is an alphanumeric character
```

- **int ispunct(int c)**

The function checks whether *c* is a punctuation character. A punctuation character is any character that has graphical representation, and that is not alphanumeric. It returns non-zero value (true) if *c* is a punctuation character, else it returns 0 (false).

Example with *ispunct(c)*:

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

int main() {
   int c = '?';

   if(ispunct(c) ) {
      printf("%c is a punctuation character\n", c );
   } else {
      printf("%c is not a punctuation character\n", c);
   }
   return 0;
}
```

The output of the above code will be:

```
? is a punctuation character
```

- **int isspace(char c)**

The function checks whether *c* is a white-space character. In C, standard white-space characters are: space (' '); tab ('\t'); newline ('\n'); vertical tab ('\v'); feed ('\f') and return ('\r'). The function returns non-zero value (true) if *c* is a white-space character, else it returns 0 (false).

Example with *isspace(c)*:

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

int main() {
   char c = 'A';

   if(isspace(c) ) {
      printf("%c is a white-space character\n", c );
   } else {
      printf("%c is not a white-space character\n", c);
   }
   return 0;
}
```

The output of the above code will be:

```
A is not a white-space character
```

- **int isprint(int c)**

The function checks whether *c* is a printable character. Printable characters are all characters except the control characters. The function returns non-zero value (true) if *c* is a printable character, else it returns 0 (false).

Example with *isprint(c)*:

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

int main() {
   int c = 'A';

   if(isprint(c) ) {
      printf("%c is a printable character\n", c );
   } else {
      printf("%c is not a printable character\n", c);
   }
   return 0;
}
```

The output of the above code will be:

```
A is a printable character
```

- **int islower(int c)**

The function checks whether *c* is a lowercase letter. The function returns non-zero value (true) if *c* is a lowercase letter, else it returns 0 (false).

Example with *islower(c)*:

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

int main() {
   int c = 'z';

   if(islower(c) ) {
      printf("%c is a lowercase letter\n", c );
   } else {
      printf("%c is not a lowercase letter\n", c);
   }
   return 0;
}
```

The output of the above code will be:

```
z is a lowercase letter
```

- **int isupper(int c)**

The function checks whether *c* is an uppercase letter. The function returns non-zero value (true) if *c* is an uppercase letter, else it returns 0 (false).

Example with *isupper(c)*:

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

int main() {
   int c = 'Q';

   if(isupper(c) ) {
      printf("%c is an uppercase  letter\n", c );
   } else {
      printf("%c is not an uppercase  letter\n", c);
   }
   return 0;
}
```

The output of the above code will be:

```
Q is an uppercase  letter
```

- **int tolower(int c)**

The function converts uppercase letter *c* to lowercase. It returns a lowercase equivalent to *c*, if such value exists, else c remains unchanged.

Example with *tolower(c)*:

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

int main() {
   int c = 'Q';

   if(isupper(c) ) {
      printf("The lowercase equivalent of %c is %c\n", c, tolower(c));
   } else {
      printf("%c is not an uppercase  letter\n", c);
   }
   return 0;
}
```

The output of the above code will be:

```
The lowercase equivalent of Q is q
```

- **int toupper(int c)**

The function converts lowercase letter *c* to uppercase. It returns a uppercase equivalent to *c*, if such value exists, else c remains unchanged.

Example with *toupper(c)*:

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

int main() {
   int c = 'a';

   if(islower(c) ) {
      printf("The uppercase equivalent of %c is %c\n", c, toupper(c));
   } else {
      printf("%c is not a lowercase  letter\n", c);
   }
   return 0;
}
```

The output of the above code will be:

```
The uppercase equivalent of a is A
```

A.3 Functions for numeric conversions and random number generation - *stdlib.h* library

The *stdlib.h* C standard library contains prototypes of various functions for numeric conversions and random number generation. Table A.1 describes the commonly used functions from the *stdlib.h* library.

Function	Description
atoi(s)	Converts the string *s* to an integer number.
atof(s)	Converts the string *s* to a floating-point number (double).
atol(s)	Converts the string *s* to a long integer number.
strtod(s, &p)	Converts the string *s* to a floating-point number and returns a pointer *&p* to the first character after the number.
strtol(s, &p, base)	Converts the string *s* to a long integer number according to the given *base* and returns a pointer *&p* to the first character after the number.
strtoul(s, &p, base)	Converts the string *s* to an unsigned long integer number according to the given *base* and returns a pointer *&p* to the first character after the number.
rand()	Returns an integer value between 0 and *RAND_MAX*.
srand(a)	Sets the starting point for the random number generator used by the function *rand()*.

Table A.1: List of functions from *stdlib.h* library

- **int atoi(const char *s)**

The function converts the string *s* to an integer number. It returns the converted number as an *int* type. If the conversion could not be performed, the function returns zero.

- **double atof(const char *s)**

The function converts the string *s* to a floating-point number. It returns the converted number as an *double* type. If the conversion could not be performed, the function returns zero.

- **long int atol(const char *s)**

The function converts the string *s* to a long integer number. It returns the converted number as an *long int* type. If the conversion could not be performed, the function returns zero.

Example with *atoi(s); atof(s) and atol(s)*:

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

int main () {
   char s[10] = "229463";
   char s1[10] = "hello";

   printf("String value = %s, Floating-point value = %f\n", s, atof(s));
   printf("String value = %s, Integer value = %d\n", s, atoi(s));
   printf("String value = %s, Long integer value = %ld\n", s1, atol(s1));

   return(0);
}
```

The output of the above code will be:

```
String value = 229463, Floating-point value = 229463.000000
String value = 229463, Integer value = 229463
String value = hello, Long integer value = 0
```

- **double strtod(const char *, char **p)**

The function converts the string *s* to a floating-point number and returns a pointer ***p* to the first character after the number if ***p* is not NULL. If the conversion could not be performed, the function returns zero.

- **long int strtol(const char *s, char **p, int base)**

The function converts the string *s* to a long integer number according to the given *base* and returns a pointer ***p* to the first character after the number. The *base* must be between 2 and 36 inclusive, or 0. If the conversion could not be performed, the function returns zero.

- **unsigned long int strtol(const char *s, char **p, int base)**

The function converts the string *s* to an unsigned long integer number according to the given *base* and returns a pointer ***p* to the first character after the number. The *base* must be between 2 and 36 inclusive, or 0. If the conversion could not be performed, the function returns zero.

Example with *strtod(s, &p); strtol(s, &p, base) and strtoul(s, &p, base)*:

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

int main () {
   char s[30] = "223.131234 Test double";
```

```
    char s1[40] = "1176 Test long integer";
    char s2[40] = "1176 Test unsigned long integer";
    char *p;

    printf("The number (double) is: %f\n", strtod(s, &p));
    printf("String part is: %s\n\n", p);
    printf("The number (long integer) is: %ld\n", strtol(s1, &p, 10));
    printf("String part is: %s\n\n", p);
    printf("The number (unsigned long integer) is: %lu\n", strtoul(s2, &p,
        10));
    printf("String part is: %s", p);

    return(0);
}
```

The output of the above code will be:

```
The number (double) is: 223.131234
String part is:  Test double
The number (long integer) is: 1176
String part is:  Test long integer
The number (unsigned long integer) is: 1176
String part is:  Test unsigned long integer
```

- **int rand(void)**

The function returns an integer value between 0 and RAND_MAX. RAND_MAX is a constant whose default value may vary, but it is always at least 32767.

- **void srand(unsigned int a)**

Sets the starting point for the random number generator used by the function *rand()*. The function does not return any value.

Example with *rand() and srand(a)*:

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

int main () {
   int i;

   srand(time(NULL)); /* Intializes random number generator */

   /* Print 10 random numbers between 0 and 100 */
   for( i = 0 ; i < 10 ; i++ )
      printf("%d\n", rand() % 100);

   return(0);
}
```

One possible output of the above code will be:

```
90
79
13
98
4
41
3
70
68
55
```

A.4 Commonly used functions from *math.h* library

The *math.h* C standard library contains prototypes of various mathematical functions. All functions in this library take ***double*** as an input parameter and return ***double*** as a result. Table A.2 describes the commonly used functions from the *ctype.h* library.

Function	Description
sqrt(a)	Returns the square root of *a*.
pow(a, n)	Returns the result of *a* to the power *n*.
fabs(a)	Returns the absolute value of *a*.
ceil(a)	Returns the smallest integer value greater than or equal to *a*.
floor(a)	Returns the largest integer value less than or equal to *a*.
fmod(a, b)	Returns the remainder of the dividing of *a* by *b*.
exp(n)	Returns the result of exponential *e* to the power *n*.
log(a)	Returns the natural logarithm of *a* (base-e).
log10(a)	Returns the common logarithm of *a* (base-10).
sin(a)	Returns the sine of angle *a*. The angle *a* is in radians.
cos(a)	Returns the cosine of angle *a*. The angle *a* is in radians.
tan(a)	Returns the tangent of angle *a*. The angle *a* is in radians.
sinh(a)	Returns the hyperbolic sine of *a*.
cosh(a)	Returns the hyperbolic cosine of *a*.
tanh(a)	Returns the hyperbolic tangent of *a*.
asin(a)	Returns the arc sine of *a*. The result is in radians.
acos(a)	Returns the arc cosine of *a*. The result is in radians.
atan(a)	Returns the arc tangent of *a*. The result is in radians.

Table A.2: List of functions from *math.h* library

Example with functions from *math.h* library:

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

int main() {
   double a = 2.2, n = 2.0, b = 0.9;

   printf("The square root of %f is %f \n", a, sqrt(a));
   printf("%f to the power of %f is %f \n", a, n, pow(a, n));
   printf("The absolute value of %f is %f \n", a, fabs(a));
   printf("The smallest integer value greater than or equal to %f is %f \n",
         a, ceil(a));
   printf("The the largest integer value less than or equal to %f is %f \n",
         a, floor(a));
   printf("The reminder of %f divided to %f is %f \n", a, n, fmod(a, n));
   printf("e to the power of %f is %f \n", a, exp(a));
   printf("The natural logarithm of %f is %f \n", a, log(a));
   printf("The common logarithm of %f is %f \n", a, log10(a));
   printf("The sine of %f radians is %f \n", a, sin(a));
   printf("The cosine of %f radians is %f \n", a, cos(a));
   printf("The tangent of %f radians is %f \n", a, tan(a));
   printf("The hyperbolic sine of %f is %f \n", a, sinh(a));
   printf("The hyperbolic cosine of %f radians is %f \n", a, cosh(a));
   printf("The hyperbolic tangent of %f radians is %f \n", a, tanh(a));
   printf("The arc sine of %f is %f radians\n", b, asin(b));
   printf("The arc cosine of %f is %f radians\n", b, acos(b));
   printf("The arc tangent of %f is %f radians\n", b, atan(b));

   return 0;
}
```

The output of the above code will be:

```
The square root of 2.200000 is 1.483240
2.200000 to the power of 2.000000 is 4.84000
The absolute value of 2.200000 is 2.200000
The smallest integer value greater than or equal to 2.200000 is 3.000000
The the largest integer value less than or equal to 2.200000 is 2.000000
The reminder of 2.200000 divided to 2.000000 is 0.200000
e to the power of 2.200000 is 9.025013
The natural logarithm of 2.200000 is 0.788457
The common logarithm of 2.200000 is 0.342423
The sine of 2.200000 radians is 0.808496
The cosine of 2.200000 radians is -0.588501
The tangent of 2.200000 radians is -1.373823
The hyperbolic sine of 2.200000 is 4.457105
The hyperbolic cosine of 2.200000 radians is 4.567908
The hyperbolic tangent of 2.200000 radians is 0.975743
The arc sine of 0.900000 is 1.119770 radians
The arc cosine of 0.900000 is 0.451027 radians
The arc tangent of 0.900000 is 0.732815 radians
```

A.5 Functions for manipulating date and time - *time.h* library

The *time.h* library contains prototypes of various functions for manipulating date and time. Table A.3 describes the commonly used datatypes (line 1 to 3) and functions (line 4 to 11) from the *time.h* library.

Datatype(1-3) Function(4-11)	**Description**
time_t	Datatype capable of storing the calendar time.
clock_t	Datatype capable of storing the processor time.
struct tm	Structure used to store the time and date.
clock()	Returns the processor clock time used since the start of the program in *clock_t* format.
time(*t)	Calculates the current calendar time and encodes it in *time_t* format.
mktime(*t)	Returns a value corresponding to the calendar time passed as structure *struct tm*.
*asctime(*t)	Returns a pointer to a string containing the date and time information in a human-readable format.
*ctime(*t)	Returns a string representing the local date and time information in a human-readable format.
difftime(t1, t2)	Returns the difference of two times in seconds.
*gmtime(*t)	Returns a pointer to a *tm struct* structure with the time information, expressed in GMT timezone.
*localtime(*t)	Returns a pointer to a *tm struct* structure with the time information, expressed in local timezone.

Table A.3: List of datatypes and functions from *time.h* library

The structure *tm struct* has the following definition:

```
struct tm {
   int tm_sec;         /* seconds */
   int tm_min;         /* minutes */
   int tm_hour;        /* hours */
   int tm_mday;        /* day of the month */
   int tm_mon;         /* month */
   int tm_year;        /* year */
   int tm_wday;        /* day of the week */
   int tm_yday;        /* day in the year, 0 to 365  */
   int tm_isdst;       /* daylight saving time */
};
```

- **clock_t clock(void)**

The function returns the number of clock ticks elapsed since the start of the program.

- **time_t time(time_t *seconds)**

The function returns the time since the Epoch (January 1, 1970, 00:00:00 UTC), measured in seconds.

- **time_t mktime(struct tm *t)**

The function converts the structure pointed to by *t* into a time_t value according to the local time zone.

- **char *asctime(const struct tm *t)**

The function returns a pointer to a string containing the date and time information in a human-readable format. The information is in the following format: **Www Mmm dd hh:mm:ss yyyy**, where *Www* is the weekday name, *Mmm* is the month name, *dd* is the day of the month, *hh:mm:ss* is the time (*hh* hour, *mm* minutes, *ss* seconds) and *yyyy* is the year.

- **char *ctime(const time_t *t)**

The function returns a string representing the local date and time information in a human-readable format. The information is in the following format: **Www Mmm dd hh:mm:ss yyyy**, where *Www* is the weekday name, *Mmm* is the month name, *dd* is the day of the month, *hh:mm:ss* is the time (*hh* hour, *mm* minutes, *ss* seconds) and *yyyy* is the year.

- **double difftime(time_t t1, time_t t2)**

The function returns the difference of two times, t1 and t2, in seconds. The two times are specified in calendar time.

- **struct tm *gmtime(const time_t *t)**

The function returns a pointer to a *tm struct* structure with the time information field with the values representing the corresponding time, expressed in Coordinated Universal Time (UTC) or Greenwich Mean Time (GMT) timezone. The definition of the *tm struct* structure is given in the listing above.

- **struct tm *localtime(const time_t *t)**

The function returns a pointer to a *tm struct* structure with the time information field with the values representing the corresponding time, expressed in local timezone. The definition of the *tm struct* structure is given in the listing above.

Example with functions from *time.h* library:

```
#include <stdio.h>
#include <time.h>
int main () {
    time_t t, start_t, end_t;
    clock_t start_c, end_c;
    struct tm *local_time, *gm_time;

    start_c = clock();
    time(&start_t);

    time(&t);
    local_time = localtime(&t);
        printf("Current local time: %2d:%02d\n", local_time->tm_hour,
            local_time->tm_min);

    time(&t);
    gm_time = gmtime( &t );
        printf("Current GM day and time: %s\n", asctime(gm_time));

    time(&end_t);
        printf("Program execution took: %f seconds\n", difftime(end_t,
            start_t));

    end_t = clock();
        printf("Total CPU time: %f seconds\n", ((double)(end_c - start_c) /
            CLOCKS_PER_SEC));

   return(0);
}
```

The output of the above code will be:

```
Current local time: 10:46
Current GM day and time: Mon May  3 10:46:08 2021
Program execution took: 0.00 seconds
Total CPU time: 0.000926 seconds
```

APPENDIX B

Using command-line arguments in C

C programming language allows user to pass some values from the command line to the programs when they are executed. These values are called command line arguments. The command line arguments in the *main()* function are a simple way of passing information in the program (names of files, options, etc.). They are specified in the *main()* function as follows:

```
#include <stdio.h>

int main(int argc, char **argv) {

    printf("Hello, world!\n");

    return 0;
}
```

where:

- **argc** is an integer number that refers to the number of arguments passed to the program.
- ****argv** is a pointer array that points to each argument passed to the program. The first element of the array is the name of the program. The command line arguments are passed separated by whitespace. If the argument has whitespace, then such argument is passed by putting it inside double quotes "" or single quotes ”.

Example:

```
#include <stdio.h>

int main(int argc, char **argv) {

    int i;
    for(i = 0; i < argc; i++) {
        printf("%s\n", argv[i]);
    }
```

```
    return 0;
}
```

In the above example, the loop is iterated for every argument in the command line. When the above example is compiled and executed from the command line, it produces the following result:

```
> gcc hello.c -o hello
> Hello world! This is "a test."
Hello
world!
This
is
a test.
```

During compiling of the example with the parameters given above, *argc=5* and *argv[0]="Hello"*, *argv[1]="world!"*, *argv[2]="This"*, *argv[3]="is"*, *argv[4]="a test."*.

APPENDIX C

C keywords

In the C programming language, there are 32 keywords. Keywords are predefined, reserved words and cannot be used as a variable name. Keywords must be written in lowercase. The list of reserved keywords in C is given in Table C.1.

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

Table C.1: List of keywords in C

A short description of the C keywords is given below. The syntax and application of the keywords are discussed in the respective chapters of this book.

- **auto** - Keyword used to declares automatic variables.
- **break and continue** - Keywords used to change the standard execution of the loop. Break is also used to terminate the switch statement (Chapter 4 and 5).
- **if, else, switch, case and default** - Keywords used to make decisions in selection structures (see Chapter 4).
- **int, double, float and char** - Keywords used to declare variables from a specific datatype (see Chapter 2).
- **const** - Keyword used to declare constant (see Chapter 2).
- **for, while and do** - Keywords used to create loop (see Chapter 5).
- **goto** - Keyword used to transfer control of the program to the specified label (see Chapter 5).

References

1. Kernighan Brian W., and Ritchie Dennis. *The C Programming Language, second edition*. Pearson, 1988.

2. Perry Greg, and Dean Miller. *C Programming: Absolute Beginner's Guide, third edition*. Que, 2013.

3. Peter Prinz, and Tony Crawford. *C in a Nutshell: The Definitive Reference, second edition*. O'Reilly Media, 2016.

4. Reema Thareja. *Computer Fundamentals And Programming In C, second edition*. Oxford University Press, 2016.

5. K. N. King. *C Programming: A Modern Approach, second edition*. W. W. Norton & Company, 2008.

6. Herbert Schildt. *Teach Yourself C, third edition*. McGraw-Hill Osborne Media, 1997.

7. Mike McGrath. *C Programming in easy steps, fifth edition*. In Easy Steps Limited, 2018.

8. David Griffiths, and Dawn Griffiths. *Head First C: A Brain-Friendly Guide, first edition*. O'Reilly Media, 2012.

Short excerpts from the reviewers

Prof. Cvetko Andreeski, PhD

The book "Introduction to programming by examples" is a textbook intended for teaching the content of the course "Introduction to programming". The book teaches the C programming language, a representative of the third generation of languages for structural and procedural programming.

The textbook consists of 12 chapters that are written in order for students to master the knowledge of structural programming in the easiest way. Students have the opportunity to learn all aspects of working in the C programming language, in an easy and fun way through examples that are selected to provide students with a complete knowledge of working with the programming language. In addition to the knowledge that students acquire for working with the programming language, they learn how to make an optimal code and how to solve real problems in a way that the solution is obtained in an optimal time. A special aspect is dedicated to creating codes that are easy to follow as well as codes in which appropriate comments are inserted that will direct the programmer to the segments of the code.

When creating codes for more complex problems, an important aspect of the operation is the knowledge of debugging the code operation, as well as monitoring the values of the variables used in the cod. A special section is devoted in the textbook for removing errors and monitoring the values of variables. With this knowledge, students can make a complete analysis of the operation of segments of codes that are essential to the operation of the code.

Finally, examples are given for working with command line parameters and working with pointers, variables such as string and files. With this knowledge, students will be able to create programs for solving complex problems that will result in obtaining a solution not only by displaying the screen, but also by generating and modifying files that can be stored in the secondary memory. The textbook is also a good basis for acquiring prerequisite knowledge for further study of object-oriented programming languages in programming subjects.

Assoc. Prof. Aneta Velkoska, PhD

The textbook "Introduction to programming by examples" includes the content provided by the curriculum of the course "Introduction to programming" studied in the first year at the University of Information Science and Technology "St. Paul the Apostle" - Ohrid. The contents presented in this textbook are divided into 12 chapters and 3 appendices, with standard contents relevant to introduction to programming. The authors strived to present the textbook content precisely and clearly, so that it would be more accessible to both students and other interested readers.

This manuscript contains the most important programming methods in the C programming language, which are necessary when introducing students to the field of programming. Therefore, I recommend that this manuscript be published as a textbook for the course Introduction to programming.

Copyright

!!

CIP - Cataloging of the publication
National and University Library "St. Kliment Ohridski ", Skopje

004.438C:004.42(075.8)

Introduction to Programming by Examples / Atanas Hristov et al. : University of Information Science and Technology "St. Paul the Apostle", 2022.
- 206 pages : 12 figures; 25 cm

Access method (URL):
https://uist.edu.mk/wp-content/uploads/2022/04/introduction-to-programming-by-examples.pdf
- Title taken from the screen. Description of the source on 29.03.2022.
- Bibliography: p. 201. Also contains: Appendix

ISBN 978-608-66225-1-0

1.Hristov, Atanas [author] 2.Tuntev, Naum [author] 3.Marina, Ninoslav[author]
a) Computer programming - University textbooks

COBISS.MK-ID 56827653