

People's Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research
University of 8 May 1945-Guelma-
Faculty of Mathematics, Computer Science and Science of Matter
Department of Computer Science



Lab Manual

Algorithmics and Data Structures 1

Intended for first-year undergraduate students in Mathematics

Established by:

Dr. Abderrahmane KEFALI

kefali.abderrahmane@univ-guelma.dz

2023/2024

Preface

It is with great pleasure that we present this practical work notes dedicated to the module "Algorithmics and Data Structures 1". Specifically designed for first-year Mathematics students at the Department of Mathematics in the Faculty of Mathematics and Computer Science and Material Sciences at the University of 8 May 1945, Guelma. However, this document can also be useful for any non-computer science student wishing to learn the basics of algorithmics and programming.

Indeed, C programming is renowned for its simplicity, efficiency, and portability, making it a preferred choice for the development of system software, embedded applications, and many other fields. This practical work notes aims to strengthen fundamental knowledge in algorithmics and C programming. By delving into the practical work presented in this document, students will have the opportunity to explore fundamental concepts of structured programming, such as data types, control structures, loops, arrays, and much more. They will learn to translate complex problems into clear algorithms and implement them effectively in C.

This material aligns with the module syllabus and adheres to the latest pedagogical framework established for the common core in mathematics, applied mathematics, and computer science, as proposed by the national pedagogical committee of the MI field since the academic year 2018-2019. It is structured into six parts, covering the entirety of the topics covered during the semester. Each of the six practical work sheets is dedicated to a specific chapter, offering a natural progression and a comprehensive pedagogical approach. The themes covered include an introduction to algorithms, simple sequential algorithms, conditional structures, loops, arrays, strings, and user-defined types.

The first sheet serves as an introduction, presenting the development environment used during the practical work sessions and guiding students to establish their first C program. The second sheet covers simple programming concepts in C, such as basic types, input/output, assignment, etc. The third sheet focuses on conditional structures in C, providing varied exercises on different conditional structures. The fourth sheet introduces loops, structures allowing the repetition of tasks, with exercises covering the three types of loops. The fifth sheet explores arrays, multidimensional arrays, and strings, with each of these three structures studied in detail. The last practical work sheet is dedicated to user-defined types, emphasizing records and enumerations.

Each practical work sheet begins with a recap of the theoretical concepts necessary for the completion of exercises, accompanied by concrete examples to facilitate understanding. Exercises are divided into two categories: training activities designed to accompany students in the discovery of new concepts, and application exercises focused on the practical use of learned concepts to solve real-world problems.

A distinctive feature of this practical work notes lies in its detailed approach to solutions. For each exercise, I not only provide the source code but also offer a thorough description of the solution and relevant comments. This approach aims to guide students through the resolution process, fostering a deep understanding of the studied concepts. Each sheet (except the first) contains 10 application exercises and at least 2 training activities, totaling more than 60 exercises included in this document.

I sincerely hope that this material will be a valuable resource for students seeking mastery of the fundamentals of algorithmics and programming. May this document be a helpful companion throughout your academic and professional journey.

Table of Contents

Preface	i
Table of Contents	1
List of Figures	5
List of Tables	6
Lab No 1. C language Introduction	7
1) Objectives	7
2) C language overview.....	7
3) Software Required for C Programming.....	7
4) Presentation.....	8
5) Getting Started with Code::Blocks.....	8
5.1) Launching Code::Blocks.....	8
5.2) Description of the Code::Blocks 20.03 Interface	8
5.3) Creating a project.....	9
5.4) Editing the source file.....	11
5.5) Saving and opening	12
5.6) Compile and run	12
5.7) Exit Code::Blocs	12
6) Your first program in C with Code::Blocs.....	13
Lab No 2. Simple sequential algorithm	14
1) Objectives	14
2) Recap: Key Concepts in C Programming.....	14
2.1) C program structure.....	14
2.2) Constant and Variables declaration	15
2.2.1) Constant declaration.....	15
2.2.2) Variable declaration	15
2.3) Instructions.....	15
2.3.1) Assignments.....	16
2.3.2) Reading.....	16
2.3.3) Writing.....	16
2. Practice activities.....	17
2.3) Activity 1 : Declaration and assignment	17
2.4) Activity 2 : Reading instruction.....	18
3. Application exercises.....	19
3.1) Exercise 1 : Fahrenheit to Kelvin Conversion.....	19
3.2) Exercise 2 : Circle Perimeter and Area	20
3.3) Exercise 3 : Car Selling Price Calculation	20

3.4)	Exercise 4 : Basic Arithmetic Operations.....	21
3.5)	Exercise 5 : Spring Elongation Calculation.....	22
3.6)	Exercise 6 : Duration Conversion	22
3.7)	Exercise 7 : Distance Calculation.....	23
3.8)	Exercise 8 : Character Variables Swap	24
3.9)	Exercise 9 : Document Size Calculation.....	25
3.10)	Exercise 10 : Monthly Loan Repayment Calculation.....	26
Lab No 3. Conditional Structures.....		28
1)	Objectives	28
2)	Recap: Key Concepts in Conditional Structures	28
2.1)	Simple Conditional Structure.....	28
2.2)	Compound Conditional Structure	29
2.3)	Nested Conditional Structures	29
2.4)	Multiple Choice Statement (switch).....	29
2.5)	Branching Statement (goto).....	30
3)	Practice activities.....	30
3.1)	Activity 1 : simple, compound, and nested conditional statements	30
3.2)	Activity 2: Multiple-choice structure	32
3.3)	Activity 3: Branching statement	33
4)	Application exercises.....	35
4.1)	Exercise 1: Identify Minimum.....	35
4.2)	Exercise 2: Check if a point is inside a rectangle.....	36
4.3)	Exercise 3: Solve Second-degree Equation	37
4.4)	Exercise 4: Identify Character Type	38
4.5)	Exercise 5: Calculate Paper Ream Cost	39
4.6)	Exercise 6: Display Time One Second Later	40
4.7)	Exercise 7: Minimum Coins for Amount.....	41
4.8)	Exercise 8: Display Day's Name	42
4.9)	Exercise 9: Arithmetic Operations	43
4.10)	Exercise 10: Count Positive and Non-Positive Numbers	44
Lab No 4. Loops		46
1)	Objectives	46
2)	Recap: Key Concepts in Loops	46
2.1)	The While loop	46
2.2)	The do...while loop.....	47
2.3)	The for loop	47
2.4)	Nested loops.....	47
3)	Practice activities.....	48
3.1)	Activity 1: The « while » and « do...while » loops.....	48
3.2)	Activity 2 : The « for » loop	50
4)	Application exercises.....	52
4.1)	Exercise 1: Counting Positive and Negative Numbers	52

4.2)	Exercise 2: Multiplication Table.....	53
4.3)	Exercise 3: Sum of Digits.....	54
4.4)	Exercise 4: Weighted Average.....	55
4.5)	Exercise 5: Min and Max of a sequence of numbers.....	56
4.6)	Exercise 6: GCD calculation using Euclidean Algorithm.....	57
4.7)	Exercise 7: Base conversion.....	59
4.8)	Exercise 8: Fibonacci sequence.....	61
4.9)	Exercise 9: Reciprocal Powers Sum	63
4.10)	Exercise 10: Binomial Expression Expansion	64
Lab No 5. Arrays and Strings.....		67
1)	Objectives	67
2)	Recap: Key Concepts in Loops.....	67
2.1)	Arrays	67
2.1.1)	Declaration.....	67
2.1.2)	Accessing to elements of an array	68
2.1.3)	Manipulation of an array.....	68
2.2)	Multidimensional arrays.....	68
2.2.1)	Declaration.....	68
2.2.2)	Accessing to elements of a multidimensional array.....	69
2.2.3)	Manipulation of a multidimensional array	69
2.3)	Strings.....	69
2.3.1)	Declaration.....	69
2.3.2)	Accessing a character of the string.....	70
2.3.3)	Manipulation of a String	70
3)	Practice activities.....	70
3.1)	Activity 1: Arrays	70
3.2)	Activity 2 : Strings	73
4)	Application exercises.....	75
4.1)	Exercise 1: Exam Grades and Statistics.....	76
4.2)	Exercise 2: Array Value Search and Occurrences	77
4.3)	Exercise 3: Array normalization.....	80
4.4)	Exercise 4: Capturing State Changes in a Binary Array.....	81
4.5)	Exercise 5: Sieve of Eratosthenes	82
4.6)	Exercise 6: Sparse Matrix Detection	84
4.7)	Exercise 7: Image thresholding.....	85
4.8)	Exercise 8: Diagonal Permutation	86
4.9)	Exercise 9: Business Sales Analysis	88
4.10)	Exercise 10: Palindrome Checker.....	90
Lab No 6. Custom Types (Structures and enumeration).....		92
1)	Objectives	92
2)	Recap: Key Concepts in custom types.....	92
2.1)	Enumerations	92

2.1.1)	Declaration of an enumeration.....	92
2.1.2)	Manipulation of an enumeration.....	93
2.2)	Structures.....	93
2.2.1)	Declaration of a structure.....	94
2.2.2)	Accessing Structure Fields.....	95
2.2.3)	Manipulation of a structure.....	95
3)	Practice activities.....	95
3.1)	Activity 1: Enumerations.....	95
3.2)	Activity 2: Structures.....	97
4)	Application Exercises.....	99
4.1)	Exercise 1: Geometric Shape Calculator.....	99
4.2)	Exercise 2: Traffic Light Controller.....	101
	Solution:.....	101
4.3)	Exercise 3: Population Statistics.....	102
4.4)	Exercise 4: Deck of Cards.....	104
4.5)	Exercise 5: Point Distance Calculator.....	105
4.6)	Exercise 6: Age Comparison.....	106
4.7)	Exercise 7: Complex Number Operations.....	107
4.8)	Exercise 8: Invoice Calculation.....	108
4.9)	Exercise 9: Family Information.....	110
4.10)	Exercise 10: Car Park Management.....	111
References		115

List of Figures

Figure 1.1. Main interface of Code::Blocs 20.03	9
Figure 1.2. “New from template” window.....	10
Figure 1.3. Window for choosing between C and C++ languages	10
Figure 1.4. Window for entering the project title and path	11
Figure 1.5. Window for choosing the compiler.....	11
Figure 1.6. Expanding the project tree from the management area.....	12
Figure 1.7. Activating the “main.c” file.....	12
Figure 1.8. Different types of triangles	13
Figure 5.1. Example of an array	68
Figure 5.2. Example of 3×4 matrix	69
Figure 5.3. Example of the normalization of an array.....	80
Figure 5.4. Example of transitions in a binary array	81
Figure 5.5. Example of a matrix before and after diagonal permutation	87

List of Tables

Table 2.1. Basic Data Types in the C Language.....	15
Table 2.2. Input and Output Formats in the C Language.....	16
Table 5.1. Predefined functions for string manipulation.....	70

Lab No 1. C language

Introduction

1) Objectives

The objective of this initial practical work is to facilitate the students' acquaintance with the development environment designated for use throughout this semester, namely the Code::Blocks environment. In this lab, students will embark on a guided exploration to master the essentials of Code::Blocks, enabling them to create, compile, and execute their inaugural C program. This hands-on experience will not only introduce students to the Code::Blocks interface but also immerse them in the foundational commands essential for programming in C. By the conclusion of this practical work, students will have gained a robust understanding of the Code::Blocks environment, laying a solid groundwork for their programming endeavors in this semester. Let's embark on this journey into the realm of Code::Blocks, empowering students with the skills to navigate and utilize this development environment effectively.

2) C language overview

C is a versatile and widely used programming language that has stood the test of time, proving its resilience and adaptability since its creation in the early 1970s. It was developed by Dennis Ritchie at Bell Labs for the purpose of creating the Unix operating system. Over the decades, C has become one of the most influential programming languages, influencing the development of numerous other languages.

The key features of C Language are:

- Simplicity and Efficiency,
- Portability,
- Structured Programming,
- Low-Level Manipulation,
- Procedural Programming

3) Software Required for C Programming

To program in the C programming language, you need to have three types of software:

- **Text Editor:** Used for writing the source code of the C program. In theory, software like Notepad on Windows is sufficient. Ideally, it is recommended to use an intelligent text editor that automatically colorizes the code, making it much easier for you to navigate.
- **C Compiler:** This is used to transform (compile) your source code into binary (executable) code. The most well-known and widely used C compiler is **GCC** (GNU Compiler Collection). It is a cross-platform, open-source compiler, indispensable on Unix systems, created by the GNU project (for more details on this project, please refer to the

online project page: <http://www.gnu.org/>. The Windows version is called **MinGW** and can be downloaded for free from: <http://tdm-gcc.tdragon.net/download>.

- **Debugger:** A tool to help you trace errors in your program.

Before you begin, you have two options:

- **Option 1:** Obtain each of these three programs separately. This is the more complicated method, but it works. I will not detail this solution here; instead, I will discuss the simple method.
- **Option 2:** Use an all-in-one program that combines a text editor, compiler, and debugger. These all-in-one programs are called **IDE** (Integrated Development Environment).

However, there are several IDEs for the C language. Among the most well-known IDEs for Windows, we can mention:

- Microsoft Visual C++
- Borland C++ Builder
- Dev C++
- Turbo C++
- NetBeans IDE
- Code::Blocks

All these IDEs allow you to program without any issues. Some are more feature-rich, while others are a bit more intuitive to use. However, in all cases, the programs you create will be the same regardless of the IDE you use. Therefore, the choice is not as crucial as one might think.

For this course, we have chosen Code::Blocks. This will be the IDE we use in all our lab sessions.


4) Presentation

Code::Blocks is an Integrated Development Environment (IDE) mainly designed for programming in C and C++. The version of Code::Blocks that will be used during the lab sessions is version 20.03 known for its reliability and feature-rich environment.

This IDE offers a user-friendly interface and a multitude of tools, streamlining the process of code creation, debugging, and execution. Its versatility makes it an excellent choice for both beginners and seasoned developers, providing a seamless coding experience.

5) Getting Started with Code::Blocks

5.1) Launching Code::Blocks

You can launch Code::Blocks 20.03 by clicking on the desktop icon  or from the Start Menu → All Programs → Code::Blocks.

5.2) Description of the Code::Blocks 20.03 Interface

When you click on the Code::Blocks 20.03 icon, the interface described in Figure 1.1 will appear:

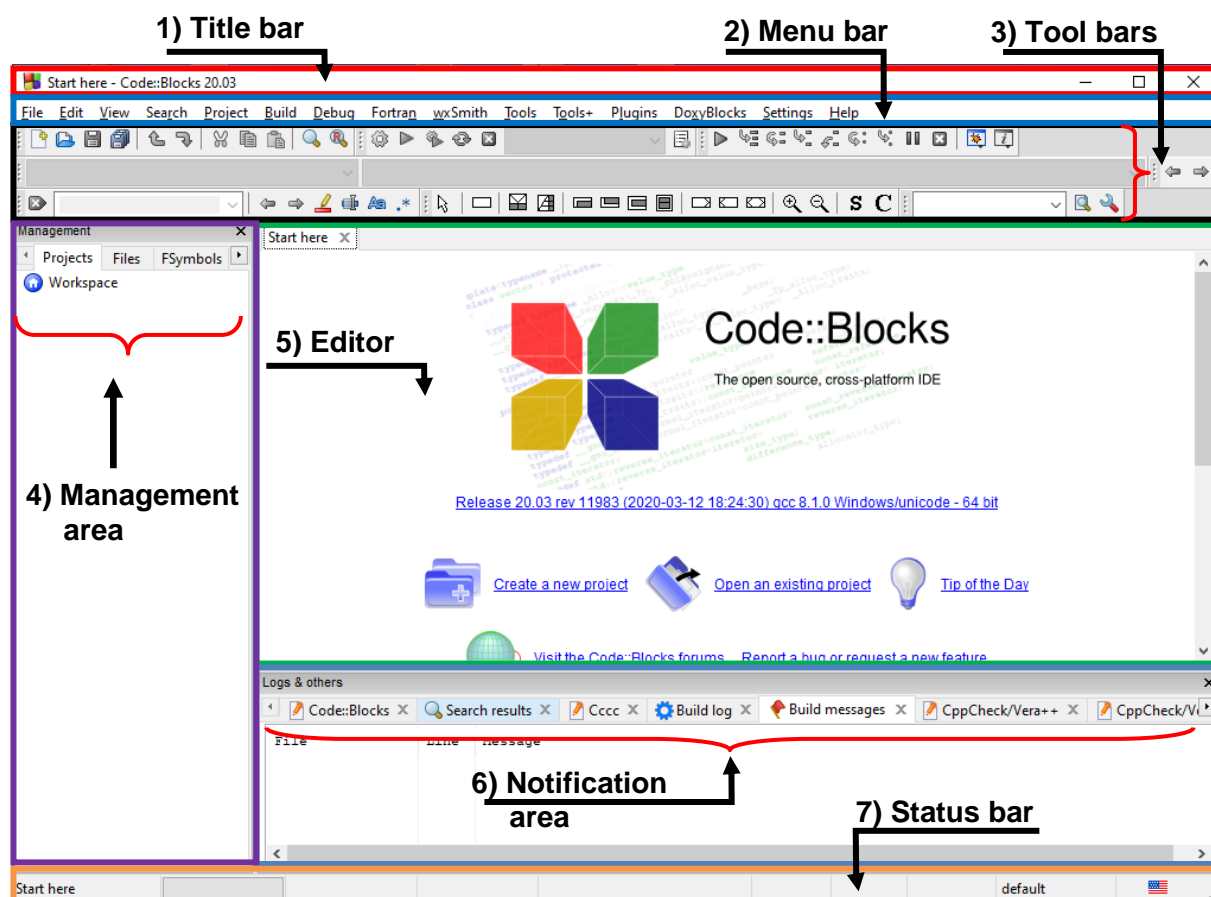



Figure 1.1. Main interface of Code::Blocs 20.03

5.3) Creating a project

To create a new project, follow these steps:

- Click on "Create a new project" in the main window, or go to the File menu → New → Project, or click the "New"  button in the standard toolbar and select "Project" from the drop-down menu.
- The "New from template" window opens and prompts you to choose a project template. Select "Console application" and click "Go". See Figure 1.2.
- The dialog box that opens serves no purpose; check the box that says "Skip this page next time" and click "Next >" to continue.
- The next window (Figure 1.3) will then ask you if you will be using C or C++. Select C, and then click the "Next >" button.

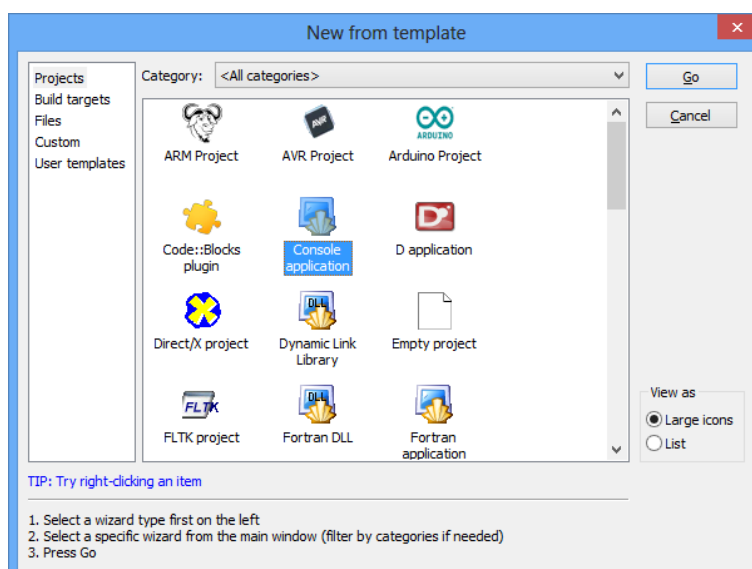


Figure 1.2. "New from template" window

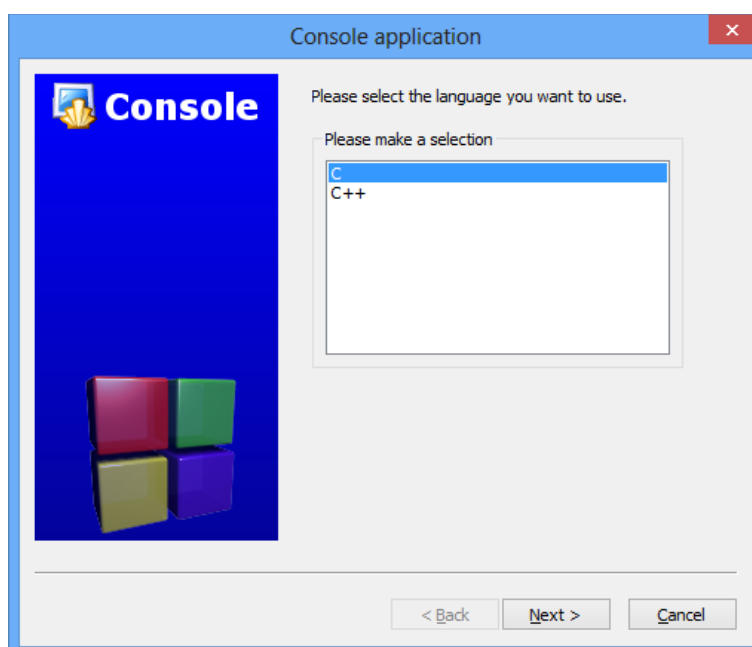



Figure 1.3. Window for choosing between C and C++ languages

- Now, you need to give your project a name and select the folder where it will be saved. Enter the name in the "Project title" field, and then click the button  to browse for the directory where the project will be saved. Confirm by clicking the "Next >" button. See Figure 1.4.
- Finally, the last window (Figure 1.5) allows you to choose the compiler (usually GNU GCC by default) and then click the "Finish" button.

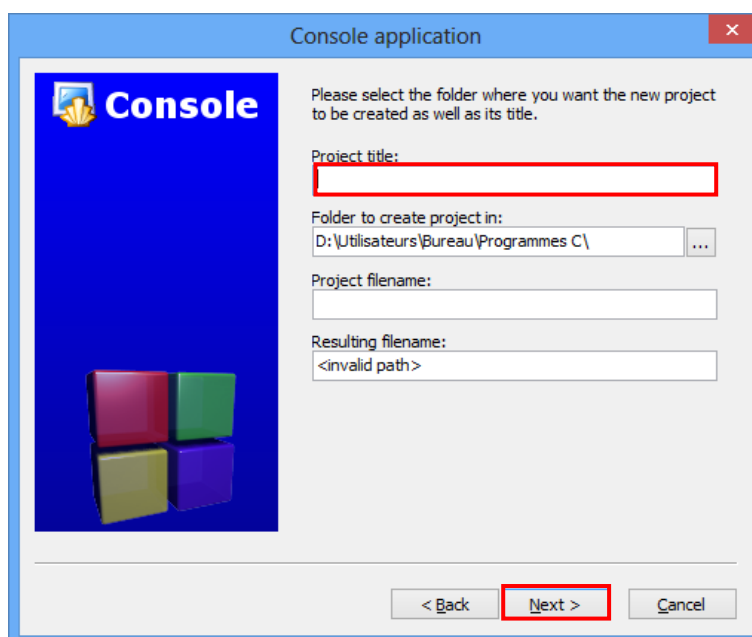


Figure 1.4. Window for entering the project title and path

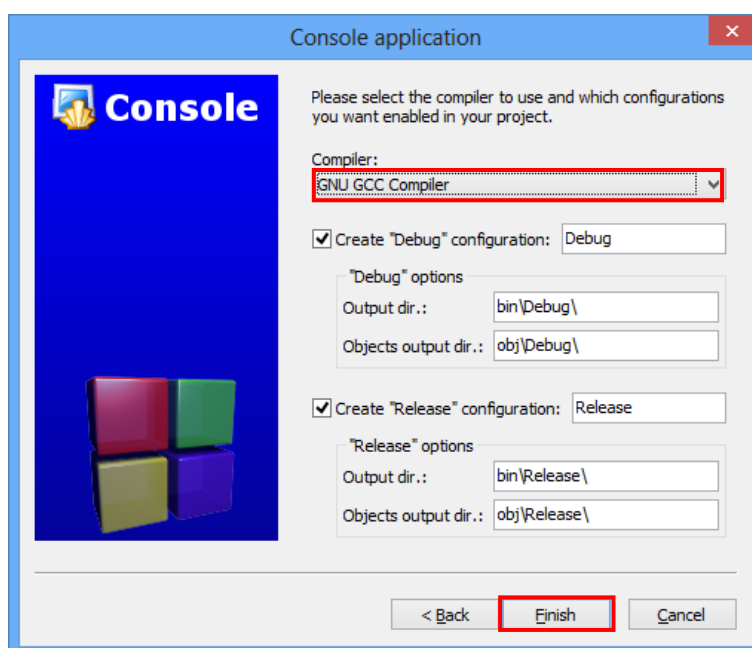



Figure 1.5. Window for choosing the compiler

- The project is now created, and it will appear in the current workspace on the left side of the interface.

5.4) Editing the source file

- Expand the "Workspace" tree of projects located in the "Projects" tab of the "Management Area" in Code::Blocks.
- Click on the  sign to the left of "Sources" to display the list of files in the project as shown in Figure 1.6.
- Double-click on "main.c". It will be displayed in the central editing window with C syntax highlighting. Now, it's ready to be edited (Figure 1.7).

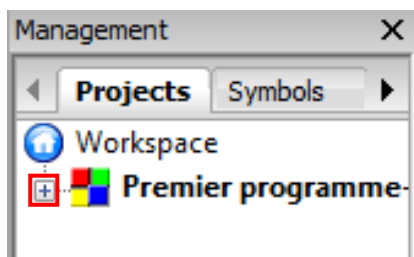


Figure 1.6. Expanding the project tree from the management area

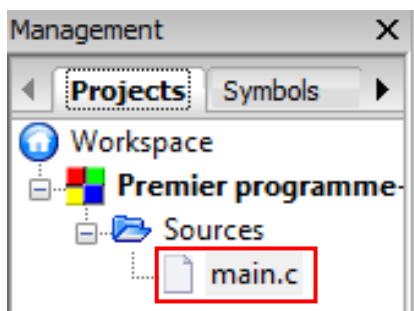







Figure 1.7. Activating the "main.c" file

5.5) Saving and opening


- Don't forget to save your program after each modification by selecting the "Save file" command from the "File" menu, clicking the  button on the "Standard" toolbar, or using the keyboard shortcut **Ctrl** + **S**.
- To open an existing project, go to the "File" menu and select the "Open..." command or use the keyboard shortcut **Ctrl** + **O**.

5.6) Compile and run

Code::Blocks has a "Build" menu and a "Compiler" toolbar  reserved for compilation and execution.

- To compile the project, simply click on the "Build" command in the "Build" menu, or click the button  in the "Compiler" toolbar, or use the keyboard shortcut **Ctrl** + **F9**.
- You can start the running by clicking either the "Run" command in the "Build" menu, or the button  in the "Compiler" toolbar, or by using the keyboard shortcut **Ctrl** + **F10**.
- You can also compile and run (in one click) by clicking the "Build and Run" command in the "Build" menu, or the button  in the "Compiler" toolbar, or by using the keyboard shortcut **F9**.

5.7) Exit Code::Blocs

You can exit Code::Blocks directly by clicking the button , or by choosing the "Quit" action from the "File" menu, or by using the keyboard shortcut **Ctrl** + **Q**.

6) Your first program in C with Code::Bloccs

Create a new project and name it "First Project-<your name>" following the same steps outlined in section 3.3. Then, open the "main.c" file by following the steps described in section 3.4. The following program will be displayed:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    printf("Hello world!\n");
    return 0;
}
```

1. Compile and run this program.
2. Modify the sentence "Hello world" to "Hello, my name is <Your name>, I am a first-year Mathematics student", then compile and run the program. What do you notice? Deduce the role of `printf`.
3. Modify the program by adding `"\n"` and then `"\t"` after "Hello," and re-run it. What do you notice? Deduce the meaning of `"\n"` and `"\t"`.
4. Now, make the necessary modifications to the program so that it displays " I am a first-year" and " Mathematics student" each on a separate line.
5. Modify the program to display:
 - a) A filled rectangle of asterisks (Figure 1.8.a)
 - b) A right-angled triangle of asterisks (Figure 1.8.b)
 - c) An isosceles triangle of asterisks (Figure 1.8.c).

*****	*	*
*****	**	***
*****	***	*****
*****	****	*****
*****	*****	*****
(a)	(b)	(c)

Figure 1.8. Different types of triangles

6. Delete all the `printf` statements in the program and replace them with: `printf("Your mark is %d/20",15)`.
7. Change 15 to 15.75. Compile and run the program. What do you notice?
8. Change `"%d"` to `"%f"`, then compile and run.
9. Deduce the role of `"%d"` and `"%f"`.

Lab No 2. Simple sequential algorithm

1) Objectives

The aim of this lab is to familiarize students with the fundamental structure and basic elements of a C language program. Through activities and practical exercises, students will acquire the skills to construct comprehensive programs in C, demonstrating proficiency in manipulating variables, constants, and basic instructions.

By the end of this lab, students should confidently execute C language programs and apply essential programming concepts. For optimal preparation, students are encouraged to refer to section number 9 of Chapter 2 titled "Translation into C language" before diving into the problem-solving aspect of this lab. This section provides detailed explanations of the concepts necessary to successfully complete the application exercises in this lab. Let's embark on this journey to establish a strong foundation in C programming!

2) Recap: Key Concepts in C Programming

Before delving into the exercises, it's essential to revisit some fundamental concepts in C programming.

2.1) C program structure

In a C program, the structure typically adheres to the following format:

```
<Library Declarations>
main()
{
    <Constant and Variable Declarations>
    <Instructions>
}
```

The `<Library Declarations>` section involves including external libraries, signaled by `#include` directives, providing additional functionalities to the program. Common inclusions, such as `#include <stdio.h>`, offer access to predefined input and output functions.

The `main()` function, denoted in lowercase, serves as the program's starting point. Within the braces indicating the scope of the program, both constant and variable declarations, as well as instructions, are enclosed. This encapsulation within braces defines the overall scope of the C program.

2.2) Constant and Variables declaration

This section is where you declare any constants and variables that will be used in the program.

2.2.1) Constant declaration

Constants are values that do not change during the program's execution, usually defined using `#define` directive according to the syntaxe:

```
#define <name_constant> <value_Constant>
```

For example, `#define PI 3.14` declares a constant named `PI` with the value 3.14.

2.2.2) Variable declaration

Variables, on the other hand, are dynamic entities that can hold varying values during program execution. Their declaration involves specifying the data type followed by the variable name as follows:

```
<type_Variable> <name_Variable>;
```

For instance, `int x;` declares an integer variable named `x`.

However, the various data types recognized in the C language are the following:

Data type	Signification	Size (Bytes)	Range of acceptable values
Char	Character	1	-128 à 127
unsigned char	Unsigned Character	1	0 à 255
short	Short Integer	2	-32768 à 32767
unsigned short	Unsigned Short Integer	2	0 à 65535
Int	Integer	4	-2147483648 à 2147483 647
unsigned int	Unsigned Integer	4	0 à 4294967295
long	Long Integer	4	-2147483648 à 2147483647
unsigned long	Unsigned long Integer	4	0 à 4294967295
Float	Floating (real)	4	3.4×10^{-38} à 3.4×10^{38}
Double	Double Floating	8	1.7×10^{-308} à 1.7×10^{308}
long double	Long Double Floating	10	3.4×10^{-4932} à 3.4×10^{4932}

Table 2.1. Basic Data Types in the C Language.

It's essential to note that both constant and variable names are identifiers and must adhere to specific rules. Identifiers are case-sensitive, can include letters, digits and underscores ('_'), but must begin with a letter or underscore.

2.3) Instructions

Within the C program structure, the section denoted as `<Instructions>` represents the heart of the program, where a sequence of instructions guides the computer on how to perform specific tasks.

In C language the basic instructions include assignments, reading, and writing. All these instructions must end with a semicolon (;).

2.3.1) Assignments

Assignment is an operation that allows assigning a value to a variable. It is symbolized by the sign '='. Its syntax is as follows:

```
<name_variable> = <value> ;
```

The right-hand part <value> can be a direct value, a constant, another variable, or an expression.

For instance, `x=7;` assigns the direct value 7 to the integer variable `x`.

2.3.2) Reading

In the C programming language, there are several functions for inputting values from the keyboard, with `scanf` being one of the most common functions for this purpose. The syntax of this function is as follows:

```
scanf("<format>", &<name_variable>);
```

Here,

- **<format>**: A format specifier string that defines the expected input format. It consists of conversion specifiers (See Table 2.2).
- **<name_variable>**: the name of the variable where the scanned values will be stored. Note that the sign of address ('&') is mandatory.

The various format specifiers are the following:

Format	Data type	Data representation
%d	Int	Signed decimal
%hd	short int	Signed decimal
%ld	long int	Signed decimal
%u	unsigned int	Unsigned decimal
%hu	unsigned short int	Unsigned decimal
%lu	unsigned long int	Unsigned decimal
%f	float	Floating-point, fixed decimal
%lf	double	Floating-point, fixed decimal
%Lf	long double	Floating-point, fixed decimal
%c	char	Character
%s		String of characters

Table 2.2. Input and Output Formats in the C Language.

For example, `scanf("%d", &x);` allows to read an integer and store it in the variable `x`.

2.3.3) Writing

The most common function to write (display) in C language is `printf` function. His syntax is as follows:

```
printf("<control string>",<expression>);
```

Such as:

- **<control string>**: The text to be displayed in addition to format specifier that defines the desired output format of the **<expression>**. The format consists of conversion specifiers as defined in Table 2.2.
It is crucial to use escape characters in the control string, such as '\n' for a newline, to ensure proper formatting and presentation of output.
- **<expression>**: Values or variables to be formatted and displayed, corresponding to the format specifiers.

For instance, `printf("The value of x is %d",x);` displays the string "The value of x is " followed by the value of the variable `x`.

2. Practice activities

The practice activities in this section serve the purpose of reinforcing and applying the concepts of C programming learned in the preceding material. They provide an opportunity for hands-on practice, allowing students to integrate and solidify his understanding of these notions by working through several scenarios.

2.3) Activity 1 : Declaration and assignment

Consider the following program:

```
#include <stdio.h>           // Use the standard input/output library
int main() {                 // The main program
    #define x 4              // Declare a constant named x with a value of 4
    int a,b,c,d,t1,t2;       // Declare 6 integer variables
    a = 5;                   // Assign the value 5 to variable a
    b = 3;                   // Assign the value 3 to variable b
    c = 1;                   // Assign the value 1 to variable c
    t1 = b * b;              // Store the result of b*b in variable t1
    t2 = x * a * c;          // Store the result of x*a*c in variable t2
    d = t1 - t2;             // Store the result of t1-t2 in variable d
    printf("The result is %d", d); // Display the result d
}
```

1. Create a new project and type the program above. Note that the sentences after `"/>"` are comments, so there is no need to write them.
2. What does this program do?
3. Replace `int` with `float` then compile and execute. What do you notice?
4. Make the necessary corrections for the program to work correctly.

Solution:

1. Creation of a new project.

2. This program begins by declaring a constant and six variables. It then assigns values, which can be either direct values or the results of expressions, to each variable. Finally, the program computes a result based on these variables and displays the final outcome using `printf`.
3. When transitioning the variable types from `int` to `float` in the program, a discrepancy becomes evident in the displayed value. Despite the actual value of the variable `d` being -11, the program erroneously shows 0. However, this inconsistency arises from the mismatch between the data types of the variables (which have been changed to `float`) and the display format specifier in the `printf` statement, (which still uses `%d`, indicating an integer). The result is a misrepresentation of the calculated value.
4. To rectify this inconsistency and accurately reflect the nature of the computation, it is essential to align the display format in the `printf` statement with the updated variable types. By employing the `%f` format specifier, the program will correctly represent the result as a floating-point number during output, ensuring coherence between variable types and display presentation.

2.4) Activity 2 : Reading instruction

The following program is supposed to calculate the result of dividing two integers entered by the user.

```
#include <stdio.h>
main() {
    int x,y,r;
    printf("Enter 2 integer numbers: ");
    scanf("%d%d",&x,&y);
    r=x/y;
    printf("%d / %d = %d",x,y,r);
}
```

1. Create a new project and type the program above.
2. Compile and run the program. Is the result correct?
3. Change the type of the variable `r` to `float`, then compile and run it. Is the issue resolved?
4. If not, replace the last `%d` in the `printf` with by `%f`, then compile and run it. Is the provided result correct?
5. As a final attempt, add `(float)` before `x` in the instruction `r=x/y`, making it `r=(float)x/y`. Compile and run it. Is the program correct now?
6. What do you conclude from what happened?

Solution:

1. Creation of a new project.
2. After compiling and running the program, no compiler errors are detected. However, upon entering values for the variables `x` and `y`, it becomes apparent that the result stored in variable `r` is inaccurate. This discrepancy stems from the fact that the

program performs integer division, truncating any decimal values and potentially leading to an imprecise outcome.

3. After converting the variable type of `r` to `float`, the displayed result becomes 0, pointing us directly to the issue discussed in the previous exercise: the incoherence between variable types and display presentation.
4. Upon replacing the last `%d` in the `printf` statement with `%f`, the result is indeed displayed as a floating-point number. However, an observation of the displayed outcome reveals that the fractional part of the number is zero. This behavior suggests that despite the change in the display format, the underlying issue persists, and it appears that an integer division is still being performed.
5. Yes, by introducing `(float)` before `x` in the instruction `r=x/y`, thus modifying it to `r=(float)x/y`, the program successfully addresses the issue.
6. From the observed behaviors and the modifications made to the program, several conclusions can be drawn:
 - **Format Specifier and Display:** The choice of format specifier in the `printf` statement is crucial for accurate representation of variable values. Mismatching the specifier with the variable type can lead to misleading or incorrect output.
 - **Integer Division:** In C, when performing division between two integers, the result is also an integer, with any fractional part truncated. This behavior can lead to imprecise results, especially when dealing with floating-point arithmetic.
 - **Casting to Float:** To ensure accurate floating-point division, it's essential to explicitly cast at least one of the operands to a float before the division operation. This casting ensures that the division involves floating-point numbers, allowing for proper representation of fractional parts.

3. Application exercises

In all the following exercises, you are not required to ensure the validity of the entered data.

3.1) Exercise 1 : Fahrenheit to Kelvin Conversion

To convert Fahrenheit degrees to Kelvins, the following formula is used:

$$K = \frac{F + 459,67}{1,8}$$

Where K is the degree in Kelvin and F is the degree in Fahrenheit.

Write a program that allows entering a temperature in Fahrenheit from the keyboard and converts it to Kelvins.

Solution:

The program consists of three steps:

- Input a temperature value in Fahrenheit.
- Utilize the given formula to convert the Fahrenheit temperature to Kelvins.
- Display the result

It is as follows:

```
#include<stdio.h>
int main() {
    float F, K;
    // Read temperature in Fahrenheit from the user
    printf("Enter temperature in Fahrenheit: ");
    scanf("%f", &F);
    // Convert Fahrenheit to Kelvins using the formula
    K = (F + 459.67) / 1.8;
    // Display the result
    printf("Temperature in Kelvin: %.2f\n", K);
    return 0;
}
```

3.2) Exercise 2 : Circle Perimeter and Area

Write a program that reads the radius R of a circle and calculates and displays its perimeter and area.

Solution:

The program first prompts the user to provide the radius value (R) and then utilizes the formulas for perimeter and area of a circle to perform the calculations.

- The perimeter (also known as the circumference) is calculated using the formula $2\pi R$.
- The area is calculated using the formula πR^2 .

Finally, the program presents the computed values for both perimeter and area.

Here is the program:

```
#include<stdio.h>
main(){
    #define pi 3.14
    float R,p,a;
    //Read the radius of the circle from the user
    printf("Enter the radius of the circle: ");
    scanf("%f",&R);
    //Calculate perimeter and area
    p=2*pi*R;
    a=pi*R*R;
    //Display the results
    printf("The perimeter is %.3f\n",p);
    printf("The area is %.3f\n",a);
    //"%f" to display 3 digits after the decimal point
}
```

3.3) Exercise 3 : Car Selling Price Calculation

The selling price of a new car includes the sum of the base price, a dealer's profit, and a sales tax. The dealer's profit percentage is 10%, and the sales tax is 9% of the base price.

Write a program allows entering the base price of a car and calculates and displays its selling price.

Solution:

The solution involves obtaining user input for the base price and then applying the specified profit and tax percentages to calculate the total selling price. The program concludes by displaying the calculated selling price as the output.

To implement this solution, the following C program is provided:

```
#include<stdio.h>
int main() {
    float basePrice, dealerProfit, salesTax, sellingPrice;
    // Read the base price of the car from the user
    printf("Enter the base price of the car: ");
    scanf("%f", &basePrice);
    // Calculate dealer's profit (10% of base price)
    dealerProfit = basePrice*10/100;
    // Calculate sales tax (9% of base price)
    salesTax = basePrice*9/100;
    // Calculate selling price
    sellingPrice = basePrice + dealerProfit + salesTax;
    // Display the result
    printf("Selling Price: %.2f\n", sellingPrice);
    return 0;
}
```

3.4) Exercise 4 : Basic Arithmetic Operations

Write a program that allows you to enter 2 integers from the keyboard and calculates and displays their sum, difference, product, quotient, and remainder of division.

Solution:

```
#include<stdio.h>
main(){
    int x,y,sum,diff,prod,quot,rem;
    //Read two integers from the user
    printf("Donner 2 nombres entiers: ");
    scanf("%d%d",&x,&y);
    //Calculate sum,difference,product,quotient,and remainder
    sum=x+y;
    diff=x-y;
    prod=x*y;
    quot=x/y;
    rem=x%y;
    //Display the results on separate lines
    printf("The sum is %d\n",sum);
    printf("The difference is %d\n",diff);
    printf("The product is %d\n",prod);
    printf("The quotient is %d\n",quot);
    printf("The remainder of division is %d\n",rem);
}
```


3.5) Exercise 5 : Spring Elongation Calculation

We aim to calculate the elongation L of a spring with stiffness K to which a mass m is attached, considering the relationship $m * g = K * L$.

Such as:

- K : stiffness of the spring,
- m : attached mass,
- L : elongation of the spring,
- g : gravitational constant equal to 9.8 m/s^2 .

Write a program that input the necessary data and calculates and displays the value of the spring's elongation

Solution:

The objective is to create a program capable of computing the elongation (L) of a spring when given the stiffness (K) and the attached mass (m).

The program achieves this by prompting the user to input the values for stiffness (K) and mass (m). Subsequently, it utilizes the provided formula to calculate the elongation (L). However, it is given that: $m * g = K * L$, and therefore $L=(m*g)/K$.

The final step involves displaying the computed elongation as the program output.

Here is the requested program:

```
#include<stdio.h>
int main() {
    #define g 9.8
    float K, m, L;
    // Prompt the user for input
    printf("Enter the stiffness of the spring (K): ");
    scanf("%f", &K);
    printf("Enter the attached mass (m): ");
    scanf("%f", &m);
    // Calculate elongation using the formula m * g = K * L
    L = (m * g) / K;
    // Display the calculated elongation
    printf("The elongation of the spring (L) is: %.2f\n", L);
    return 0;
}
```

3.6) Exercise 6 : Duration Conversion

Write a program that asks the user for a duration value expressed in seconds and displays its equivalent in hours, minutes, and seconds.

Example: 3800 seconds → 1 hour 3 minutes 20 seconds.

Solution:

In addressing this exercise, the solution hinges on employing the $/$ and $\%$ operators. Recognizing that an hour consists of 3600 seconds serves as a foundational understanding. Therefore, through integer division (dividing the total seconds by 3600), we obtain the

corresponding number of hours. For instance, in the provided example, 3800 seconds equate to precisely 1 hour (the result of $3800 / 3600$).

Subsequently, addressing the remaining seconds (200 in this case, derived from $3800 \% 3600$), we apply the rule that one minute is equivalent to 60 seconds. Achieving this conversion involves integer division ($200 / 60 = 3$), determining the number of minutes. The remainder in this division represents the remaining seconds (20).

Let's now implement this logic into a program that takes a duration value in seconds from the user and displays its equivalent in hours, minutes, and seconds

```
#include<stdio.h>
int main() {
    int totalSeconds, h, m, s;
    // Prompt the user for the duration in seconds
    printf("Enter the duration in seconds: ");
    scanf("%d", &totalSeconds);

    // Calculate hours, minutes, and seconds
    h = totalSeconds / 3600;
    m = (totalSeconds % 3600) / 60;
    s = totalSeconds % 60;
    // Display the equivalent duration
    printf("Equivalent duration:\n");
    printf("%d hours %d minutes %d seconds", h, m, s);
    return 0;
}
```

3.7) Exercise 7 : Distance Calculation

Write a program that allows reading the coordinates of two points in the plane and calculates and displays the distance between them.

Recall that the distance between two points $A(x_1, y_1)$ and $B(x_2, y_2)$ is given by the following formula:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Note: In C language, the calculation of the square root is achieved using the `sqrt` function.

Solution:

The solution involves taking user input for the coordinates of two points in the plane, specifically $A(x_1, y_1)$ and $B(x_2, y_2)$. These four coordinates are essential for calculating the distance between the two points using the distance formula. The formula involves the squared differences in x and y coordinates, and the resulting sum is then square-rooted using the `sqrt` function in the C language. This calculated distance is then displayed to the user. The program's structure adheres to the mathematical representation of the distance formula, ensuring accurate and efficient computation.

The program to perform this calculation is as follows:

```
#include<stdio.h>
#include<math.h>
main() {
    float x1,y1,x2,y2,dis;
    printf("Enter the coordinates (x,y) of point A: ");
    scanf("%f%f",&x1,&y1);
    printf("Enter the coordinates (x,y) of point B: ");
    scanf("%f%f",&x2,&y2);
    dis = sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1));
    printf("The distance between the 2 points is: %.2f",dis);
}
```

Note that the `<math.h>` header file is necessary for using the `sqrt` function, which is employed in this program to calculate the square root.

3.8) Exercise 8 : Character Variables Swap

Write a program that allows you to enter the values of 2 character variables from the keyboard and swaps their content.

Solution:

In the resolution of this exercise, which involves reading and swapping two character variables (designated as `a` and `b` in the following program), two fundamental ideas are pivotal.

The initial idea centers around the process of reading character variables using `scanf` and the format specifier `"%c"`. However, unlike numerical inputs, character inputs may be influenced by whitespace characters, including the newline character (`'\n'`), which can lead to unexpected behavior. This is because when the user enters a character and presses 'Enter', the newline character remains in the input buffer. Consequently, the next `scanf("%c", &variable);` might unintentionally capture this newline character instead of waiting for new input.

To address this issue, a space is strategically introduced before the `"%c"` in the `scanf` format string (e.g., `scanf(" %c",&variable);`). This extra space instructs `scanf` to ignore any whitespace characters, ensuring a more reliable input process.

The second idea is the mechanism employed to swap the contents of two variables (`a` and `b` in this case). To perform a correct swap without losing information, a third variable (`c` in the following program) is introduced. The purpose of the temporary variable `c` is to temporarily store the value of `a` before it is overwritten with the value of `b`.

This is necessary because if we directly assign `a = b` first, the original value of `a` would be lost, and both `a` and `b` would end up with the value of `b`. However, the sequence `c = a; a = b; b = c;` facilitates a secure exchange, preventing the loss of the original values of both `a` and `b`. The temporary variable `c` acts as a placeholder, enabling a smooth and accurate swapping process.

The program below exemplifies these adjustments, demonstrating the correct reading and swapping of two characters entered by the user:

```
#include<stdio.h>
int main() {
    char a, b, c;
    printf("Enter the first character: ");
    scanf(" %c", &a);
    //the space before %c to consume any whitespace
    printf("Enter the second character: ");
    scanf(" %c", &b);
    c = a;
    a = b;
    b = c;
    printf("After swapping, a='%c' and b='%c'\n", a, b);
    return 0;
}
```

3.9) Exercise 9 : Document Size Calculation

We want to digitally store a document consisting of two identical pages. Each page is comprised of both text and an image, with the image size fixed at 40 KB. The text on a page consists of P sentences, where each sentence concludes with a period. Each sentence is constructed from M words, and each word contains C characters, with words separated by 02 white spaces.

Create a program to calculate the size of this document in bytes, keeping in mind that each character is encoded using 1 byte.

Solution:

In this exercise, our aim is to create a program that accurately calculates the size of a digital document comprising two identical pages, each containing both text and an image.

The user is prompted to input key parameters: the number of sentences (P), words per sentence (M), and characters per word (C), shaping the structure of the textual content on each page.

The program's calculation process is methodical. It starts by computing the total character count in a sentence, considering characters within words, spaces, and the concluding period. This count serves as the foundation for subsequent calculations.

Moving forward, the program extrapolates the sentence character count to determine the overall character count on an entire page. This involves multiplying the sentence character count by the user-specified number of sentences (P), effectively encapsulating the textual content for one page.

To gauge the size of a single page in bytes, the program multiplies the total character count by the size of each character (encoded as 1 byte) and adds the fixed size of the accompanying image (40 KB, converted to bytes). Recognizing that our document consists of two identical pages, the program extends its calculations to determine the total size of the entire document in bytes.

In presenting the results, the program aims for user-friendly clarity, providing a straightforward output of the calculated document size.

The program enabling this computation is as follows:

```

#include <stdio.h>
int main() {
    //constants
    #define IMG_SIZE_KB 40
    #define CHAR_SIZE_B 1
    #define NB_SPACES 2
    // Variables
    int P,M,C,charInSentence,charInPage;
    int pageSizeBytes,documentSize;
    // User input
    printf("Enter the number of sentences (P): ");
    scanf("%d", &P);
    printf("Enter the number of words per sentence (M): ");
    scanf("%d", &M);
    printf("Enter the number of characters per word (C): ");
    scanf("%d", &C);
    // Calculate the number of characters in a sentence
    // M words with C characters each, plus M-1 spaces,
    // plus 1 period
    charInSentence = M * C + (M - 1) * NB_SPACES + 1;
    // Calculate the number of characters in a page
    charInPage = P * charInSentence;
    // Calculate the size of a page in bytes
    pageSizeBytes =charInPage*CHAR_SIZE_B+IMG_SIZE_KB*1024;
    // Calculate the document size in bytes
    documentSizeBytes=2*pageSizeBytes; //Two identical pages
    // Display the result
    printf("The size of the document is: ");
    printf("%d bytes\n", documentSizeBytes);
    return 0;
}

```

3.10) Exercise 10 : Monthly Loan Repayment Calculation

A young couple decides to build a house. The purchase of the land and the construction require a loan of E dinars. This couple wants to know how much their monthly repayment will be if the repayment is made over n years, with an interest rate of $T\%$ on the loan.

Write a program to assist the couple in determining their monthly repayment amount.

Solution:

The solution involves several steps:

- Capture the loan amount (E), the number of years (n), and the interest rate (T).
- Subsequently, it calculates the interest (I) using the formula: $I = E \times T / 100$.
- Next, it computes the annual repayment (RA) as: $(E + I) / n$,
- After that, it calculates the monthly repayment (RM) as: $RA / 12$.
- Finally, it displays the monthly repayment amount

The implementation of these steps is reflected in the following C program.

```
#include <stdio.h>
int main() {
    // Variables
    float E, T, I, RA, RM;
    int n;
    // User input
    printf("Enter the loan amount in dinars: ");
    scanf("%f", &E);
    printf("Enter the annual interest rate (%): ");
    scanf("%f", &T);
    printf("Enter the loan duration in years: ");
    scanf("%d", &n);
    // Calculate the value of the interest (I)
    I = E * T / 100;
    // Calculate the annual repayment (RA)
    RA = (E + I) / n;
    // Calculate the monthly repayment (RM)
    RM = RA / 12;
    // Display the Result
    printf("The monthly repayment is: %.2f dinars\n", RM);
    return 0;
}
```

Lab No 3. Conditional Structures

1) Objectives

Embark on a journey of discovery with this practical work designed to delve into the realm of conditional structures in the C programming language. Through a curated set of activities and exercises, our primary goal is to impart a profound understanding of the diverse forms of conditional structures and their practical applications.

Upon the completion of this practical work, students will possess the skills to adeptly navigate problems involving multiple situations. Mastery of the studied conditional instructions will empower them to craft solutions that are not only accurate but also adhere to best practices in programming.

This practical work goes beyond the syntax and mechanics of conditional structures; it aims to cultivate a problem-solving mindset, an indispensable skill in the dynamic landscape of programming. Get ready to unravel the intricacies of conditional structures, and let's transform theoretical knowledge into actionable proficiency.

2) Recap: Key Concepts in Conditional Structures

Before diving into the exercises, let's revisit some key concepts related to conditional structures in C programming.

Conditional structures allow the execution of different blocks of code based on the result of tests. Tests can take two forms: simple and alternative.

2.1) Simple Conditional Structure

This structure executes a block of instructions only if a logical expression (condition) is true. The syntax is as follows:

```
if(<condition>
    <block of instructions>;
```

The `<condition>` refers to a boolean expression or a logical statement that evaluates to either `true` or `false`.

Example:

```
int a;
scanf("%d", &a);
if (a > 0)
    printf("positive number");
```

Remarks:

- The C language does not have a boolean type or **TRUE** and **FALSE** values. The **FALSE** logical value is expressed by the value 0, and the **TRUE** value by any other value.
- The **<block of instructions>** must be enclosed in curly braces "{...}" if it consists of more than one instruction.

2.2) Compound Conditional Structure

This structure executes one block of instructions if a logical expression (condition) is true and another block if the expression is false. The syntax is as follows:

```
if(<condition>
    <block of instructions1>;
else <block of instructions2>;
```

Example:

```
int a;
scanf("%d", &a);
if (a >= 0)
    printf("positive number");
else
    printf("negative or zero number");
```

2.3) Nested Conditional Structures

Nested conditional structures involve placing one conditional statement inside another. This allows for more complex decision-making. The nesting of multiple tests is done in the C language as follows:

```
if(condition1) <block of instructions1>;
else if(condition2) <block of instructions2>;
    else if(condition3) <block of instructions3>;
        .....
            else <block of instructions_n>;
```

2.4) Multiple Choice Statement (switch)

The multi-choice statement allows the execution of instructions based on the evaluation of the value of an expression acting as a selector. Its syntax is as follows:

```
switch(<expression>) {
    case <value>:    <block of instructions1>
                    break;
    case <value2>:  <block of instructions2>
                    break;
    .....
    case <value_n>: <block of instructions n>
                    break;
    default:       <block of instructions n+1>
}
}
```


If the value of the expression is equal to one of the values, the corresponding block of instructions is executed. Otherwise, the block of other instructions corresponding to **default** is executed.

Remarks:

- The value of the **<expression>** can only be an integer or a character.
- The **break** statement is necessary to exit the **switch** structure. If it is absent at the end of the instructions block for **<value_i>**, execution continues with the instructions for the next value.
- The **default** statement is optional.
- It is not necessary to group the instructions in curly braces “{ . . . }” in this structure.

2.5) Branching Statement (goto)

Branching statements allow the transfer of control to a labeled statement. This is achieved in the C language using the **goto** statement. The general syntax for the **goto** statement is:

```
<Label_name>:
    instruction i ;
    .....
    goto <Label_name>;
```

Here, **<Label_name>** is a user-defined identifier followed by a colon, and it marks the target statement where control will be transferred.

3) Practice activities

The activities presented in this section are tailored to offer valuable training and hands-on practice, allowing you to reinforce your understanding of conditional structures in a practical context.

3.1) Activity 1 : simple, compound, and nested conditional statements

Consider the following program:

```
#include <stdio.h>
int main() {
    int n;
    printf("Enter an integer number: ");
    scanf("%d", &n);
    if(n%2==0) printf("The number %d is divisible by 2",n);
    if(n%2!=0) printf("The number %d is not divisible by 2",n);
    return 0;
}
```

1. Create a new project and type the program above.
2. Compile and run it. What does this program do?
3. Modify the program to perform the same task but with a single comparison.
4. Make the necessary adjustments to the program to test whether the entered number is divisible by 2 and 3, only by 2, only by 3, or by neither of them.

Solution:

1. Creation of a new project.
2. The provided C program prompts the user to enter an integer, checks if it is divisible by 2, and then prints a corresponding message.
3. To perform the same task with a single comparison, we should use the compound test structure (**if - else**). Here's the modified program:

```
#include <stdio.h>
int main(){
    int n;
    printf("Enter an integer number: ");
    scanf("%d",&n);
    if(n%2==0)
        printf("The number %d is divisible by 2",n);
    else printf("Le nombre %d is not divisible by 2",n);
    return 0;
}
```

It's important to note a crucial distinction between the two versions of the program. In the original code, despite its apparent similarity to the modified version, a noteworthy inefficiency exists. The initial program unconditionally performs two separate tests, irrespective of the outcome of the first condition. In contrast, the modified version employs a single **if-else** statement, ensuring that only one condition is evaluated. If the first condition (**n%2==0**) is true, the corresponding block of code executes, bypassing the **else** block. This enhancement enhances the code's efficiency, particularly when conditions are mutually exclusive, by eliminating redundant evaluations.

4. To examine various divisibility conditions based on the entered number, the program employs nested conditional structures. The code is as follows:

```
#include <stdio.h>
int main() {
    int n;
    // Input
    printf("Enter an integer number: ");
    scanf("%d", &n);
    // Test for divisibility by 2 and 3
    if (n % 2 == 0 && n % 3 == 0)
        printf("The number %d is divisible by both 2 and 3", n);
    // Test for divisibility by only 2
    else if (n % 2 == 0)
        printf("The number %d is divisible only by 2", n);
    // Test for divisibility by only 3
    else if (n % 3 == 0)
        printf("The number %d is divisible only by 3", n);
    // Not divisible by 2 or 3
    else printf("The number %d is neither divisible by 2 nor by 3", n);
    return 0;
}
```

3.2) Activity 2: Multiple-choice structure

We want to write a program that displays, for a selected season, the list of months it contains. The program starts by displaying, in a menu, all the seasons of the year. The user is then prompted to select one of the seasons. Finally, the program displays the months of that season.

Recall that there are four seasons (spring, summer, autumn, and winter), each containing three months, and that March is the first month of spring.

The menu to be displayed is in the following format:

```
List of seasons:
  1: Spring
  2: Summer
  3: Autumn
  4: Winter
Enter your choice:
```

If the user, for example, enters 3, the program displays:

The months are: September, October, November

The following code, once completed, is supposed to perform the requested task:

```
#include <stdio.h>
int main() {
    int s;
    printf("List of seasons:\n");
    printf("\t1: Spring\n");
    .....
    .....
    switch(s) {
        case 1: .....
                break;
        .....
        default: .....
    }
    return 0;
}
```

1. Create a new project and type the code provided above.
2. Complete the program to make it perform the desired task, then compile and run it.

Solution:

1. Creation of a new project.

2. Here is the expected program. It is designed to display a menu featuring all the seasons of the year. It then prompts the user to enter a season number and utilizes a `switch` statement to display the list of months associated with the selected season.

```
#include <stdio.h>
int main() {
    int s;
    // Display the menu of seasons
    printf("List of seasons:\n");
    printf("\t1: Spring\n");
    printf("\t2: Summer\n");
    printf("\t3: Autumn\n");
    printf("\t4: Winter\n");
    // Prompt user to enter his choice
    printf("Enter your choice: ");
    scanf("%d", &s);
    // Complete the switch statement to display the months
    //based on the user's choice
    switch (s) {
        case 1:
            printf("The months: March, April, May\n");
            break;
        case 2:
            printf("The months: June, July, August\n");
            break;
        case 3:
            printf("The months: September, October, November");
            break;
        case 4:
            printf("The months: December, January, February ");
            break;
        default:
            printf("Incorrect season number ");
    }
    return 0;
}
```

3.3) Activity 3: Branching statement

Here is an incomplete C program that uses the `goto` branching instruction. Your task is to fill in the missing parts to create a program that allows users to input a sequence of positive integers. The program should continue accepting numbers until a negative or zero value is entered. Finally, it should calculate and display the sum of the entered numbers.

```

#include <stdio.h>
int main() {
    int n,sum;
    sum=0;
    .....
    printf("Enter a positive integer. ");
    printf("Enter a negative or zero value to end:\n");
    scanf("%d",&n);
    if(n > 0){
        .....
        .....
    }
    printf("The sum is %d",sum);
    return 0;
}

```

Solution:

In order to complete the requested task, we must first assign a label (**input** in the following program) to the instruction that we want to jump back, which is the input instruction. After each input, we test if the entered number is positive. If it is the case, we add it to the sum and we use the **goto** statement to jump to the labeled instruction in order to input another number, creating a loop-like behavior. This allows the program to repeatedly prompt the user for positive integers until a negative or zero value is entered. The completed program looks like this:

```

#include <stdio.h>
int main() {
    int n,sum;
    sum=0;
    // Assign a label to the input instruction
input:
    printf("Enter a positive integer. ");
    printf("Enter a negative or zero value to end:\n");
    scanf("%d",&n);
    // Check if the entered value is positive
    if(n > 0){
        // Add the positive number to the sum
        sum=sum+n;
        // Use goto to jump back to the labeled instruction
        goto input;
    }
    //Display the sum once a negative or zero value is entered
    printf("The sum is %d",sum);
    return 0;
}

```

4) Application exercises

The application exercises in this section are designed to provide practical applications and reinforce your understanding of the concepts covered in the previous sections.

4.1) Exercise 1: Identify Minimum

Write a program that prompts the user to enter 3 integer numbers and then identifies the minimum among them.

Solution:

To determine the minimum among three integers, we start by prompting the user to input these values. Given the exercise's scope, there are no input-error cases. Following the input phase, the next step involves comparing the three integers to identify the smallest value. In this context, the minimum is defined as the number that is either smaller than or equal to the other two. Finally, it remains only to display the result.

We present here two distinct programs that achieve this purpose. The first program utilizes simple conditional structures (**if** statement), while the second employs compound conditional structures (**if-else** statement).

First program:

```
#include <stdio.h>
int main() {
    int n1,n2,n3,min;
    //Prompt the user to enter three integers
    printf("Enter the first integer: ");
    scanf("%d", &n1);
    printf("Enter the second integer: ");
    scanf("%d", &n2);
    printf("Enter the third integer: ");
    scanf("%d", &n3);
    //Determine the minimum among the three integers
    min = n1;
    if (n2 < min) {
        min = n2;
    }
    if (n3 < min) {
        min = n3;
    }
    //Display the result
    printf("The minimum is: %d\n", min);
    return 0;
}
```

Second program:

```
#include <stdio.h>
int main() {
    int n1,n2,n3,min;
    // Prompt the user to enter three integers
    printf("Enter the first integer: ");
```

```

scanf("%d", &n1);
printf("Enter the second integer: ");
scanf("%d", &n2);
printf("Enter the third integer: ");
scanf("%d", &n3);
//Determine the minimum among the three integers
if (n1 <= n2 && n1 <= n3) {
    min = n1;
}
else if (n2 <= n1 && n2 <= n3) {
    min = n2;
}
else {
    min = n3;
}
//Display the result
printf("The minimum is: %d\n", min);
return 0;
}

```

4.2) Exercise 2: Check if a point is inside a rectangle

Establish a program that reads the coordinates (x, y) of a point and determines if this point is inside a rectangle defined by the coordinates of its top-left point (x_1, y_1) and its bottom-right point (x_2, y_2) .

Solution:

To solve this problem, we need to check whether the given point (x, y) lies inside the rectangle defined by its top-left point (x_1, y_1) and bottom-right point (x_2, y_2) . We can determine this by comparing the x and y coordinates of the given point with the corresponding coordinates of the rectangle.

If the x -coordinate of the point is greater than x_1 and less than x_2 , and the y -coordinate of the point is greater than y_1 and less than y_2 , then the point is inside the rectangle.

The program is the following:

```

#include <stdio.h>
int main() {
    //Read coordinates of the point
    int x,y,x1,y1,x2,y2;
    printf("Enter the coordinates of the point (x, y): ");
    scanf("%d%d", &x, &y);
    //Read coordinates of the top-left point of the rectangle
    printf("Enter the coordinates of the top-left point of the rectangle
(x1, y1): ");
    scanf("%d%d", &x1, &y1);
    //Read coordinates of the bottom-right point of the rectangle
    printf("Enter the coordinates of the bottom-right point of the
rectangle (x2, y2): ");
    scanf("%d%d", &x2, &y2);
}

```

```

//Check if the point is inside the rectangle
if (x > x1 && x < x2 && y > y1 && y < y2) {
    //Print the result
    printf("The point is inside the rectangle.\n");
} else {
    printf("The point is outside the rectangle.\n");
}
return 0;
}

```

4.3) Exercise 3: Solve Second-degree Equation

Write the program to solve a second-degree equation of the form:

$$a x^2 + b x + c = 0.$$

Solution :

The resolution of a second-degree equation, represented in the form $a x^2 + b x + c = 0$, involves determining the roots of the equation.

However, to solve a second-degree equation, the coefficients a , b , and c are first obtained from the user. The program then verifies that the equation is indeed second-degree (where a is nonzero). Subsequently, the discriminant ($\Delta = b^2 - 4 a c$) is calculated to discern the nature of the roots.

If the discriminant is positive, the equation has two real solutions. If it is zero, there is one real solution, which indicates a repeated root. In the case of a negative discriminant, the equation has not a real solution. Finally, the program displays the results.

Here is the requested program:

```

#include <stdio.h>
#include <math.h>
int main() {
    int a, b, c, delta;
    float x1, x2;
    // Read coefficients a, b, and c
    printf("Enter the coefficients a, b, c of the equation: ");
    scanf("%d%d%d", &a, &b, &c);
    // Check if the equation is not a second-degree
    if (a == 0) {
        printf("First degree equation\n");
        //Calculate and print the single solution
        x1 = (float)-c / b;
        printf("It has one solution: %.2f", x1);
    }
    else {
        printf("Second degree equation\n");
        // Calculate the discriminant
        delta = b * b - 4 * a * c;
        // Check the value of the discriminant
        if (delta < 0)

```



```

        printf("It has no real solutions");
    else if (delta == 0) {
        //Calculate and print the double solution
        x1 = (float)-b / (2 * a);
        printf("It has a double solution: %.2f", x1);
    }
    else {
        // Calculate and print the two distinct solutions
        x1 = (float)(-b - sqrt(delta)) / (2 * a);
        x2 = (float)(-b + sqrt(delta)) / (2 * a);
        printf("It has two solutions: %.2f and %.2f",x1, x2);
    }
}
return 0;
}

```

Note that the `<math.h>` header file is necessary for using the `sqrt` function, which is employed in this program to calculate the square root.

4.4) Exercise 4: Identify Character Type

Write a program that reads a character from the keyboard and determines whether it is a letter, a digit, or a symbol.

Solution:

To determine whether a character entered by the user is a letter, a digit, or a symbol, we can leverage the ASCII values. Letters in the ASCII table fall within specific ranges, as do digits and symbols. By comparing the ASCII value of the entered character with these ranges, we can make the classification.

The program implementing this classification is as follows:

```

#include <stdio.h>
int main() {
    char ch; // User-input character
    // Read a character from the keyboard
    printf("Enter a character: ");
    scanf(" %c", &ch);
    //Note the space before %c to consume any whitespace
    //Check if the entered character is letter, digit, or symbol
    if ((ch >= 'A' && ch <= 'Z') || (ch >= 'a' && ch <= 'z')){
        printf("The entered character is a letter");
    } else if (ch >= '0' && ch <= '9') {
        printf("The entered character is a digit");
    } else {
        printf("The entered character is a symbol");
    }
    return 0;
}

```

4.5) Exercise 5: Calculate Paper Ream Cost

Write a program that allows a stationery to calculate the total cost of an order of paper reams. The unit price for a ream is 340 DA for a quantity exceeding 50 units or if the customer has made previous purchases from the seller. When the customer has no prior purchases, the unit price for a ream is 370 DA for a quantity between 20 and 50, and 400 DA for a quantity less than 20. The program should input the order quantity and the number of previous purchases, then calculate and display the total amount to pay.

The program should account for all possible cases, including input errors.

Solution:

The requested program aims to facilitate a stationery in calculating the total cost of a customer's order of paper reams based on specific pricing conditions. The unit price per ream varies depending on both the order quantity and whether the customer has made previous purchases.

The solution begins by prompting the user to enter essential information, specifically the order quantity and the number of previous purchases. To ensure accuracy, the program rigorously validates the entered values, promptly signaling an input error if necessary. Following this, the program systematically applies specified pricing conditions, checking the order quantity against defined thresholds and considering whether the customer has made prior purchases. Based on these conditions, the program calculates the unit price and subsequently computes the total amount to be paid for the paper reams. The final step involves displaying the calculated total amount in Algerian Dinar (DA).

This solution is implemented by the following C program.

```
#include <stdio.h>
int main() {
    int qtt, nbPurch;
    float unitPrice, totalAmount;
    //Input order quantity and number of previous purchases
    printf("Enter the order quantity: ");
    scanf("%d",&qtt);
    printf("Enter the number of previous purchases: ");
    scanf("%d",&nbPurch);
    if(qtt<=0 || nbPurch<0)printf("Input error");
    else{
        //Determine the unit price
        if (qtt > 50 || nbPurch > 0) {
            unitPrice = 340;
        }
        else if (qtt >= 20 && qtt <= 50) {
            unitPrice = 370;
        }
        else {
            unitPrice = 400;
        }
        // Calculate the total amount to pay
        totalAmount = qtt * unitPrice;
    }
}
```

```

        // Display the total amount to pay
        printf("Total amount to pay: %.2f DA",totalAmount);
    }
    return 0;
}

```

4.6) Exercise 6: Display Time One Second Later

Write a program that reads a time in the format (hour:minutes:seconds) and displays the time one second later.

For example, if the user enters 21:32:8, the program should display: "One second from now, it will be 21:32:9".

The program should account for all possible cases, including input errors.

Solution:

The requested program is designed to read a time in the format (hour:minutes:seconds) and then display the time one second later. To achieve this, the solution first prompts the user to input a time, validates the input for correctness, and then increments the seconds component by one. If the seconds component exceeds 59, it resets to 0, incrementing the minutes component. Similarly, if the minutes component surpasses 59, it resets to 0, incrementing the hours component. If the hours component reaches 24, it resets to 0, indicating midnight at the start of a new day. The final result, representing the time one second later, is then displayed to the user.

The described steps are encapsulated in the following program:

```

#include <stdio.h>
#include <stdlib.h>
int main() {
    //Declare variables to store hours, minutes, and seconds
    int h, m, s;
    //input a time in the format hour:minutes:seconds
    printf("Enter a time (hour:minutes:seconds): ");
    scanf("%d:%d:%d", &h, &m, &s);
    //Validate the input for correctness
    if (h < 0 || h > 23 || m < 0 || m > 59 || s < 0 || s > 59)
        printf("Invalid time");
    else {
        //Increment the seconds component by one
        s = s + 1;
        //Adjust seconds if necessary
        if (s > 59) {
            s = 0;
            m = m + 1;
            //Adjust minutes if necessary
            if (m > 59) {
                m = 0;
                h = h + 1;
                // Reset hours to 0 if it exceeds 23
                if (h > 23)

```

```

        h = 0;
    }
}
// Display the time one second later
printf("After 1 second,it will be %d:%d:%d\n",h,m,s);
}
return 0;
}

```

Remark:

Usually, `scanf` validates input using either the Enter key or spaces as separators. However, it is also possible to specify a different character as a separator by incorporating that character into the format string. In our scenario, the colons (':') within the format string "%d:%d:%d" serve as separators, facilitating the matching and extraction of hours, minutes, and seconds as the user inputs the time through the keyboard.

4.7) Exercise 7: Minimum Coins for Amount

Write a program that asks for an amount of money between 1 and 100 DA and then displays the minimum number of coins (50, 10, 5, and 1) required to make up that amount.

Solution:

To accomplish the requested task, the solution begins by prompting the user to input an amount within the range of 1 to 100 DA. To ensure the validity of the entered amount, the program performs a validation check. If the input is valid, it employs a systematic approach to determine the optimal distribution of coins.

The program iteratively utilizes the div operator (/) to determine the number of each denomination and the mod operator (%) to update the remaining amount, until the target amount is reached, keeping track of the count for each denomination.

The final result, representing the minimum number of each coin required, is then displayed to the user.

The program accomplishing this task is as follows:

```

#include <stdio.h>
int main() {
    //Declare variables
    int amount,nb50,nb10,nb5,nb1,r;
    //Input an amount of money less than 100 DA
    printf("Enter an amount of money less than 100 DA: ");
    scanf("%d", &amount);
    // Validate the input within the specified range
    if (amount <= 0 || amount >= 100)
        printf("Invalid input");
    else {
        //Calculate the number of each coin denomination
        nb50 = amount / 50;
        r = amount % 50;
        nb10 = r / 10;
        r = r % 10;
    }
}

```

```

    nb5 = r / 5;
    nb1 = r % 5;
    //Display the results
    printf("We have:\n");
    printf("\t%d pieces of 50 DA\n", nb50);
    printf("\t%d pieces of 10 DA\n", nb10);
    printf("\t%d pieces of 5 DA\n", nb5);
    printf("\t%d pieces of 1 DA", nb1);
}
return 0;
}

```

4.8) Exercise 8: Display Day's Name

Write a program that allows to enter a day of the week (a number between 1 and 7) and display the corresponding day's name. For example, Saturday corresponds to the number 1.

Utilize a **switch** statement to efficiently handle the different cases, associating each valid input with the corresponding day's name.

Solution:

The solution adheres to the directive to use a **switch** statement instead of nested **if-else** structures. The process begins by prompting the user to input a day number. Subsequently, a **switch** statement is employed, with the day number serving as the selector. Each case in the **switch** statement corresponds to a day of the week, and for each valid case (1 to 7), a **printf** instruction displays the respective day in words. Any other values outside this range are considered incorrect, and the program provides an error message indicating input error.

Here is the program:

```

#include<stdio.h>
int main(){
    //Declare variable to store the user-inputted day number
    int day;
    //Enter a day of the week (a number between 1 and 7)
    printf("Enter a day number: ");
    scanf("%d",&day);
    //Use switch statement to determine the day's name
    //based on the entered number
    switch(day){
        case 1: printf("Saturday");
                break;
        case 2: printf("Sunday");
                break;
        case 3: printf("Monday");
                break;
        case 4: printf("Tuesday");
                break;
        case 5: printf("Wednesday");
                break;
        case 6: printf("Thursday");

```

```

        break;
    case 7: printf("Friday");
        break;
    //Display an error message for invalid input
    default:printf("Incorrect day number");
}
return 0;
}

```

4.9) Exercise 9: Arithmetic Operations

Write a program that, from a menu, allows you to perform addition, subtraction, multiplication, or division of two numbers based on the user's choice. The two numbers and the operation to be performed should be entered by the user.

Use **switch** statement.

Solution:

The program is designed to function as a simple calculator, offering the user a menu to choose between addition, subtraction, multiplication, or division of two numbers. The solution employs a **switch** statement to efficiently handle the user's choice. The process begins by prompting the user to input two numbers and select the desired operation from the menu. A **switch** statement then interprets the user's choice as the selector, executing the corresponding operation and displaying the result. Special attention is given to the division operation, as the program checks for division by zero to prevent errors.

Here is the implementation:

```

#include<stdio.h>
int main() {
    float a, b, r;
    char op;
    //Prompt the user to enter two numbers
    printf("Enter two numbers: ");
    scanf("%f%f", &a, &b);
    // Display the menu for operation selection
    printf("Press + for addition\n");
    printf("Press - for subtraction\n");
    printf("Press * for multiplication\n");
    printf("Press / for division\n");
    printf("Your choice: ");
    // Added space before %c to skip whitespace characters
    scanf(" %c", &op);
    // Use switch statement to perform the selected operation
    switch(op) {
        case '+':
            r = a + b;
            printf("%.2f + %.2f = %.2f\n", a, b, r);
            break;
        case '-':
            r = a - b;

```

```

        printf("%.2f - %.2f = %.2f\n", a,b, r);
        break;
    case '*':
        r = a * b;
        printf("%.2f * %.2f = %.2f\n", a, b, r);
        break;
    case '/':
        // Check for division by zero
        if(b != 0) {
            r = a / b;
            printf("%.2f / %.2f = %.2f\n", a, b, r);
        } else {
            printf("Unable to divide");
        }
        break;
    default:
        printf("Unknown Operator");
}
return 0;
}

```

4.10) Exercise 10: Count Positive and Non-Positive Numbers

Write a C program that utilizes `goto` statement to prompt the user to enter 10 integer numbers. The program should then count and display the number of positive values and the number of non-positive values (including zero).

Solution:

The solution involves using a `goto` statement to establish a loop for input validation and the counting of positive and non-positive values within a series of integers. The solution utilizes three counters: one for the loop, another for positive values, and the third for non-positive values.

The program initializes the counters for the loop, positive, and non-positive values to zero and then enters a loop labeled as "input". In this loop, the user is prompted to input an integer number. A conditional structure is employed to determine whether the entered number is positive or non-positive (including zero). The loop counter is subsequently incremented, indicating the processing of another value. The program checks if the predefined number of values has not been reached yet, allowing it to jump back to the label and reiterate the process.

After processing all input numbers, the program displays the final count of positive and non-positive values.

```

#include<stdio.h>
int main() {
    //Declare variables
    int n,i,nbPos, nbNeg;
    //Initialize counters
    i = 0;
    nbPos = 0;

```

```
nbNeg = 0;
//Loop for input validation and counting
input:
    //Prompt the user to enter an integer number
    printf("Enter an integer number: ");
    scanf("%d", &n);
    //Check if the entered number is positive
    if (n > 0)
        nbPos = nbPos + 1; //Increment positive counter
    else
        nbNeg = nbNeg + 1; //Increment non-positive counter
    i = i + 1; //Increment loop counter
    // Check if 10 numbers have been entered
    if (i < 10)
        //Jump back to the label for next iteration
        goto input;
//Display the result
printf("Number of positive values: %d\n", nbPos);
printf("Number of non-positive values: %d", nbNeg);
return 0;
}
```


Lab No 4. Loops

1) Objectives

In this comprehensive lab, our primary aim is to provide the students with a profound understanding of iterative structures in the C programming language. Through a series of examples, activities and practical exercises, we intend to demystify the intricacies of different loop types, equipping the students with the knowledge and skills necessary to tackle a diverse range of programming challenges.

Upon completion of this lab, the students should be able to solve any problem involving repetitions and use all the studied loops correctly.

This lab aims not only to teach the students the syntax and mechanics of loop structures but also to instill a problem-solving mindset, preparing them for the dynamic landscape of programming where loop mastery is a fundamental skill. Embrace the journey into the world of C programming loops, and let's transform theoretical knowledge into practical proficiency!

2) Recap: Key Concepts in Loops

Before delving into the exercises, let's review some fundamental concepts related to loops in C programming.

Loops provide a powerful mechanism for executing a block of code repeatedly, and they come in different forms, each tailored for specific scenarios. The main types of loops include: **while**, **do...while**, and **for** loops.

2.1) The While loop

The **while** loop executes a block of instructions as long as a specified condition is met. Its basic syntax is as follows:

```
while (<Condition>)  
    <block of instructions>;
```

As for If statement, the **<condition>** is a boolean expression or a logical statement that evaluates to either **true** or **false**.

In this loop, the condition is tested before entering the loop. Therefore, the block of instructions that forms the body of the loop may never be executed; this happens when the condition is false from the beginning.

Example:

The following piece of code displays the integers from 1 to 10:

```
int i = 1;  
while (i <= 10) {  
    printf("%d ", i);  
    i++;  
}
```

2.2) The do...while loop

The **do...while** loop shares similarities with the **while** loop in that it iterates the execution of a block of instructions as long as a condition is true. However, it differs from the **while** loop in that the condition check is performed after the code is executed. Consequently, the block of code is guaranteed to execute at least once. The syntax of **do...while** loop is as follows:

```
do{
    <block of instructions>;
}while(<condition>;
```

Example:

The following piece of code displays the integers from 1 to 10 using **do...while** loop:

```
int i = 1;
do {
    printf("%d ", i);
    i++;
} while (i <= 10);
```

2.3) The for loop

The **for** loop provides a concise way to iterate over a range of values. Its syntax includes an initialization statement, a condition, and an iteration statement:

```
for (<initialization>; <condition>; <iteration>)
    <block of instructions>;
```

Such as:

- **<initialization>** is an initialization expression. It initializes the loop control variable and is typically in the form: **<counter> = <initial value>**.
- **<condition>** is a boolean expression that is evaluated before each iteration. If the condition is true, the loop continues; otherwise, it exits.
- **<iteration>**) is a progression expression. It typically updates the loop control variable by incrementing or decrementing it.

Example:

The piece of code allowing to display integers from 1 to 10 using **for** loop is the following:

```
int i;
for (i = 1; i <= 10; i++) {
    printf("%d ", i);
}
```

2.4) Nested loops

Nested loops involve placing one loop structure inside another. This concept introduces increased complexity and flexibility. For example, using nested loops can be effective when dealing with multi-dimensional arrays or when solving problems that involve hierarchical structures. The flexibility offered by these loop structures empowers programmers to efficiently handle a wide range of scenarios.

Example:

The following nested loops prints all pairs of (i, j) where i takes values from 1 to 3, and for each value of i , j takes values from 1 to 4.

```
for (i = 1; i <= 3; i++)
  for (j = 1; j <= 4; j++){
    printf("(%d,%d)\n", i,j);
  }
```

3) Practice activities

This section is dedicated to hands-on practice with loop structures in C programming. Each activity includes a code snippet, accompanied by instructions to test, analyze, and potentially modify the code. Through these activities, students will reinforce their understanding of loops, practice debugging, and gain proficiency in utilizing different loop types to solve varied problems. The activities are designed to challenge students progressively, encouraging them to actively engage with the provided code and apply loop concepts in practical scenarios.

3.1) Activity 1: The « while » and « do...while » loops

Consider the following program:

```
#include <stdio.h>
int main() {
    float a,b,c;
    printf("Please enter two real numbers : ");
    scanf("%f%f",&a,&b);
    while (b == 0) {
        printf("Provide a non-zero value for the divisor: ");
        scanf("%f", &b);
    }
    c = a / b;
    printf("%f / %f = %f", a, b, c);
}
```

1. Create a new project and type the code above.
2. Compile and run the program. What does this program do?
3. What will happen if we give a zero value to the variable **b**?
4. What will happen if we give a non-zero value to the variable **b** from the beginning?
5. Remove the `scanf("%f",&b)` statement, then compile and run the program. What happened?
6. What is the difference between using `while` and using `if` in this example?
7. Rewrite the code using a `do...while` loop and explain the difference with `while`.

Solution:

1. Creation of a new project.

2. The provided program is designed to take two real numbers as input, perform division (a / b), and display the result. However, there is a check in place using a while loop to ensure that the divisor b is not zero. If the user enters zero as the divisor, the program prompts him to provide a non-zero value until a valid input is received. Once a non-zero value is obtained, the division is executed, and the result is displayed.
3. If a zero value is provided for the variable b , the program will prompt the user to provide a non-zero value for the divisor, and it will continue to do so until the user enters a non-zero value.
4. If we give a non-zero value to the variable b from the beginning, the program will proceed directly to perform the division and display the result without repeating the input of a non-zero value for the divisor as the condition for entering the loop is not met.
5. If the `scanf("%f", &b)` statement is removed, the program will not take input for the variable b inside the loop. Consequently, if the initial value entered by the user for variable b is zero, the program will get trapped in an infinite loop. This occurs because the condition `while (b == 0)` remains true, continuously prompting the user for a non-zero value for b without making progress.
6. In this example, if we use an `if` statement for checking the divisor's value, the program would prompt the user for input once. If the entered value for b is zero, the program would continue with the calculation using this zero value, potentially leading to incorrect results without giving the user a chance to correct the input. On the other hand, when employing a `while` loop, the program ensures iterative input validation by repeatedly prompting the user for input until a non-zero value is provided for variable b . This approach allows for multiple attempts to input a valid value, ensuring accurate calculations.

7. The code using the do...while loop

```
#include <stdio.h>
int main() {
    float a,b,c;
    printf("Please enter a real number: ");
    scanf("%f",&a);
    do{
        printf("Please enter another non-zero real number: ");
        scanf("%f", &b);
    }while(b == 0);
    c = a / b;
    printf("%f / %f = %f", a, b, c);
}
```

The main difference between the `while` and `do...while` loops in this example lies in when the condition is checked. In the `while` loop, the condition (`b==0`) is checked before entering the loop, so if the condition is false initially, the loop won't execute at all. In contrast, the `do...while` loop checks the condition after the loop body, ensuring that the body executes at least once before checking the condition.

3.2) Activity 2 : The « for » loop

Consider the following program:

```
#include <stdio.h>
int main() {
    int i,n;
    printf("Enter a positive integer : ");
    scanf("%d", &n);
    for (i = 1; i <= n; i++) {
        if(i%2==0)printf("%d is an even number\n",i);
        else printf("%d is an odd number\n",i);
    }
}
```

1. Create a new project and type the code above.
2. Compile and run the program. What does this program do?
3. Add the necessary instructions for it to also calculate and display the number of values in each class.
4. Add the instruction `i=i+1` at the end of the loop. What do you notice?
5. Change the last instruction to `i=i-1`. What happened?
6. Rewrite the program using a `while` loop.

Solution:

1. Creation of a new project.
2. This program prompts the user to enter a positive integer `n`. Then, for each number from 1 to `n`, it displays whether it is even or odd. The program uses a `for` loop to iterate 1 to `n`.
3. To modify the program to calculate and display the number of values in each class (even and odd), we introduce two other variables, one for counting even numbers (`nbEven`) and another for counting odd numbers (`nbOdd`). These variables are initialized to zero before entering to the loop. After that, we increment the respective counters based on the classification of each number. Here's the modified code:

```
#include <stdio.h>
int main() {
    int i, n, nbEven, nbOdd;
    printf("Enter a positive integer: ");
    scanf("%d", &n);
    nbEven = 0;
    nbOdd = 0;
    for (i = 1; i <= n; i++) {
        if (i % 2 == 0) {
            printf("%d is an even number\n", i);
            nbEven++;
        }
    }
}
```

```

        else {
            printf("%d is an odd number\n", i);
            nbOdd++;
        }
    }
    printf("There are %d even numbers\n", nbEven);
    printf("There are %d odd numbers\n", nbOdd);
}

```

4. When adding the instruction `i=i+1` at the end of the loop, we notice that the program displays only odd numbers. In fact, this instruction increments the value of `i` by 1 after each iteration. However, the loop header already contains `i++`, which also increments `i` by 1 at the beginning of each iteration. As a result, there are two increments for each iteration, causing `i` to take on only odd values.
5. If the last instruction is changed to `i=i-1`, it results in an infinite loop where the value of `i` is continually decremented, negating the increment operation in the loop header (`i++`). As a result, the loop condition (`i <= n`) may always be true, leading to an infinite loop. The program is trapped, and it may not progress as expected.
6. The given `for` loop program can be rewritten using a `while` loop assuring the same functionality. In this modified version, the condition (`i <= n`) determines whether the loop will continue to execute. The increment operation `i++` is now placed inside the loop. This adjustment ensures that the loop will iterate as long as `i` remains less than or equal to `n`.

Here is the modified program using a `while` loop:

```

#include <stdio.h>
int main() {
    int i, n, nbEvens, nbOdds;
    printf("Enter a positive integer: ");
    scanf("%d", &n);
    nbEvens = 0;
    nbOdds = 0;
    i = 1;
    while (i <= n) {
        if (i % 2 == 0) {
            printf("%d is an even number\n", i);
            nbEvens++;
        }
        else {
            printf("%d is an odd number\n", i);
            nbOdds++;
        }
        i++;
    }
    printf("There are %d even numbers\n", nbEvens);
    printf("There are %d odd numbers\n", nbOdds);
}

```

4) Application exercises

Application exercises this section provide scenarios where you can apply the programming concepts you've learned. These exercises are designed to challenge you to create comprehensive solutions, encouraging practical problem-solving and the integration of various programming skills.

4.1) Exercise 1: Counting Positive and Negative Numbers

Write a program that allows the user to input a series of non-zero numbers. The program should continuously prompt the user to enter a number until he input zero. While taking input, the program should count and display the number of positive and negative numbers in the series. The input should stop when the user enters zero.

Solution:

The requested program is designed to interact with the user to input a series of numbers. To achieve this, we employ a loop to continuously prompt the user for input until he enters zero. Two counters, `nbPos` and `nbNeg`, are initialized to zero before entering the loop. During this process, these counters are updated based on whether the entered number is positive or negative. The loop terminates when the user inputs zero, and we finally display the final counts of positive and negative numbers.

However, we've implemented two programs to achieve this goal. The first utilizes a `while` loop, while the second employs a `do...while` loop. Below, you'll find both programs:

First program:

```
#include <stdio.h>
int main() {
    //Declare variables
    int x, nbPos, nbNeg;
    //Initialize counters
    nbPos = 0;
    nbNeg = 0;
    // Get input from the user
    printf("Enter a series of numbers (enter 0 to stop):\n");
    printf("Enter a number: ");
    scanf("%d", &x);
    // Continue reading numbers until 0 is entered
    while (x!=0) {
        //Check if the entered number is positive or negative
        if (x > 0) nbPos=nbPos+1;
        else nbNeg=nbNeg+1;
        printf("Enter a number: ");
        scanf("%d", &x);
    }
    // Display the counts
    printf("Number of positive numbers: %d\n", nbPos);
    printf("Number of negative numbers: %d\n", nbNeg);
    return 0;
}
```

Second program:

```
#include <stdio.h>
int main() {
    //Declare variables
    int x, nbPos, nbNeg;
    //Initialize counters
    nbPos = 0;
    nbNeg = 0;
    //Get input from the user
    printf("Enter a series of numbers (enter 0 to stop):\n");
    do{
        printf("Enter a number: ");
        scanf("%d", &x);
        //Check if the entered number is positive or negative,
        if (x > 0) nbPos=nbPos+1;
        else if (x<0) nbNeg=nbNeg+1;
        //Continue reading numbers until 0 is entered
    }while (x!=0);
    // Display the counts
    printf("Number of positive numbers: %d\n", nbPos);
    printf("Number of negative numbers: %d\n", nbNeg);
    return 0;
}
```

4.2) Exercise 2: Multiplication Table

Create a program that prompts the user to input an integer between 1 and 9. The program should then display the multiplication table for the entered number. If the user inputs a number outside this range, the program should prompt the user for input until a valid number is provided.

Solution:

In this exercise we aim to generate the multiplication table for a user-input integer within the range of 1 to 9. To achieve this, the solution utilizes a **do...while** loop to prompt the user for an integer between 1 and 9 continuously until a valid input is provided. This loop structure guarantees that the program won't proceed until a suitable number is entered.

Once a valid input is obtained, the program proceeds to calculate and display the multiplication table for the chosen integer. This is achieved through a **for** loop that iterates from 1 to 10, calculating the product of the user's integer and the loop variable (representing the multiplicand). The result is then printed for each iteration, presenting a clear and organized multiplication table.

The implemented solution is reflected in the following program.

```
#include <stdio.h>
int main() {
    //Declare variables
    int nb,i,res;
    // Prompt user for input within the specified range
```



```

do {
    printf("Enter an integer between 1 and 9: ");
    scanf("%d", &nb);
    //Continue input the number until a number within
    //the range is entered
} while (nb < 1 || nb > 9);
// Generate and display the multiplication table
printf("Multiplication table for %d:\n", nb);
for (i = 1; i <= 10; ++i) {
    // Calculate the result of multiplication
    res = nb * i;
    // Display the multiplication equation and result
    printf("%d * %d = %d\n", nb, i, res);
}
return 0;
}

```

4.3) Exercise 3: Sum of Digits

Write a program that asks the user to enter a positive integer n , then calculates and displays the sum of its digits.

Execution example:

Enter a positive integer: 148

The sum of the digits in 148 is 13.

Solution:

The following program prompts the user to enter a positive integer n and then calculates and displays the sum of its digits. The solution involves extracting individual digits from the entered number and accumulating their sum. The extraction of digits is achieved using a loop (specifically, a **while** loop, as the number of repetitions is not known in advance). In each iteration, the program extracts the last digit using the modulo operator (%), and integer division (/) is utilized to discard the last digit, effectively progressing to the next one.

The program ensures that the input is a positive integer and displays an error message if a non-positive integer is provided.

```

#include <stdio.h>
int main() {
    //Declare variables
    int n, i, sum, digit;
    // Prompt user for a positive integer
    printf("Enter a positive integer: ");
    scanf("%d", &n);
    // Check if the entered number is negative
    if (n < 0)
        printf("Input error");
    else {
        // Initialize sum to 0
        sum = 0;

```

```

// Extract digits and calculate sum using a while loop
while (n > 0) {
    // Extract the last digit
    digit = n % 10;
    // Add the digit to the sum
    sum = sum + digit;
    // Remove the last digit from the number
    n = n / 10;
}
// Display the sum of the digits
printf("The sum of the digits is %d", sum);
}
return 0;
}

```

4.4) Exercise 4: Weighted Average

Write a program to help students calculate the weighted average of their first-semester courses. The user should be prompted to enter the number of courses they want to consider. For each course, the program should request a valid coefficient and the mark. The program should then calculate and display the weighted average using the provided coefficients and marks based on the formula:

$$\text{weighted average} = \frac{\sum \text{mark} \times \text{coefficient}}{\sum \text{coefficient}}$$

Solution:

To facilitate students in computing their weighted average, we have devised a solution centered around loops. Initially, the user inputs the number of courses they wish to consider. The program ensures that the entered number is both positive and non-zero. Subsequently, the program prompts the user to input coefficients and corresponding marks for each course. Employing a `do...while` loop ensures continuous prompting until valid values are entered. The program multiplies these values, adding the result to the weighted sum. Simultaneously, each coefficient is added to a sum of coefficients. This process iterates until the specified number of courses is reached, utilizing a `for` loop for this purpose. The weighted average is then calculated by dividing the weighted sum by the sum of coefficients.

Here is the program:

```

#include <stdio.h>
int main() {
    // Declare variables
    float coeff, mark, weighSum, sumCoeff, weighAvg;
    int i, nbCourses;
    //Get the number of courses
    printf("Enter the number of courses: ");
    scanf("%d", &nbCourses);
    // Validate that the entered number of courses
    //is positive and non-zero
    if (nbCourses <= 0)
        printf("input Error");
}

```

```

else{
    //Initialize the sums
    weighSum = 0;
    sumCoeff = 0;
    // Input coeffs and marks for each course
    for (i = 0; i < nbCourses; i++) {
        // Input mark
        do {
            printf("Enter mark for course %d: ", i + 1);
            scanf("%f", &mark);
            //Continue input mark while the entered mark
            //is not valid
        }while (mark < 0 || mark>20);
        // Input coefficient
        do {
            printf("Enter coefficient for course %d: ",i+ 1);
            scanf("%f", &coeff);
            //Continue input coeff while the entered coeff
            //is not valide
        } while (coeff <= 0);
        // Update the weighted sum and sum of coeffs
        weighSum = weighSum+(coeff * mark);
        sumCoeff = sumCoeff+coeff;
    }
    // Calculate and display the weighted average
    weighAvg = weighSum / sumCoeff;
    printf("The weighted average is: %.2f\n", weighAvg);
}
return 0;
}

```

4.5) Exercise 5: Min and Max of a sequence of numbers

Write a program that reads a sequence of real numbers ending with 0, and calculates and displays the minimum and maximum of the entered numbers.

Solution:

The solution to this exercise involves utilizing two variables (**min** and **max**) to keep track of the minimum and maximum values. Initially, the user is prompted to input the first value of the sequence, considering it as both the minimum and maximum until further input. If the first value entered by the user is zero, the program displays a message indicating that there are no valid values to calculate the minimum and maximum. Subsequently, a **while** loop is employed to continuously input the subsequent numbers in the sequence. As the user inputs real numbers, the program checks each number against the current minimum and maximum. If a number is smaller than the current minimum or larger than the current maximum, the corresponding variables are updated accordingly. This iterative process continues until the user inputs 0, indicating the end of the sequence. After the input is complete, the program displays the calculated minimum and maximum values based on the entered sequence of real numbers.

The program for this task is as follows:

```
#include <stdio.h>
int main() {
    // Declare variables
    float number, min, max;
    // Prompt user to enter the first real number
    printf("Enter a real number (enter 0 to end): ");
    scanf("%f", &number);
    // Check if the first value is zero
    if (number == 0)
        printf("No valid values entered");
    else{
        //Initialize min and max with the first entered number
        min = number;
        max = number;
        // Continue reading real numbers until 0 is entered
        while (number != 0) {
            // Update min if the current number is smaller
            if (number < min)
                min = number;
            // Update max if the current number is larger
            else if (number > max)
                max = number;
            // Prompt user for the next real number
            printf("Enter a real number (enter 0 to end): ");
            scanf("%f", &number);
        }
        // Display the calculated minimum and maximum values
        printf("Minimum: %.2f\n", min);
        printf("Maximum: %.2f\n", max);
    }
    return 0;
}
```

4.6) Exercise 6: GCD calculation using Euclidean Algorithm

The Greatest Common Divisor (GCD) of two integers is their largest common divisor. We want to use the Euclidean algorithm to calculate the GCD of two positive integers (the GCD of two integers is equal to the GCD of their absolute values, so we restrict ourselves to positive integers). The algorithm is based on the following observations:

- The GCD of two numbers is not changed if we replace the larger of the two by their difference.
In other words, for $a > b$, $\text{GCD}(a, b) = \text{GCD}(a - b, b)$.
- The GCD of any number and 0 is always the number itself. In other words, $\text{GCD}(a, 0) = a$.

Exemple : Calculate the GCD of 35 and 20.

$$35 - 20 = 15 \quad \Rightarrow \quad \text{GCD}(20, 15).$$

$$20 - 15 = 5 \quad \Rightarrow \quad \text{GCD}(15, 5).$$

$$15 - 5 = 10 \quad \Rightarrow \quad \text{GCD}(10, 5).$$

$$10 - 5 = 5 \Rightarrow \quad \text{GCD}(5, 5).$$

$$5 - 5 = 0 \quad \Rightarrow \quad \text{GCD}(5, 0) = 5.$$

So GCD(35, 20) = 5.

Write a program that asks the user to provide two positive integers, then uses the Euclidean algorithm to calculate their GCD.

Solution:

The following program is supposed to calculate the Greatest Common Divisor (GCD) of two integer numbers entered by the user using the Euclidean algorithm.

The program initiates by prompting the user to input two positive integers, validating the input to ensure positivity. It then employs the Euclidean algorithm in a **while** loop, iteratively replacing the larger of the two numbers with their difference until one of them becomes 0. This process effectively calculates the GCD of the original integers. Once the loop terminates, the non-zero number represents the GCD, which is subsequently displayed to the user.

```
#include <stdio.h>
int main() {
    // Declare variables
    int a, b, gcd;
    // Prompt user to enter two integer numbers
    printf("Enter 2 integer numbers: ");
    scanf("%d %d", &a, &b);
    // Check for input errors (negative numbers)
    if (a < 0 || b < 0)
        printf("Input error");
    else {
        // Check if either of the numbers is 0
        if (a == 0 || b == 0)
            gcd = 0;
        else {
            // Calculate GCD using the Euclidean algorithm
            while (a != b) {
                if (a > b)
                    a = a - b;
                else
                    b = b - a;
            }
            gcd = a;
        }
        // Display the calculated GCD
        printf("The GCD is %d", gcd);
    }
    return 0;
}
```

4.7) Exercise 7: Base conversion

Write a program that converts a number from decimal format to binary format (base 2), and then to octal format (base 8).

Modify the program to convert the input to any base provided by the user.

Solution:

a) Binary conversion:

To convert a decimal number to binary, the process begins by prompting the user to enter a positive integer. Subsequently, the program rigorously verifies the positivity of the input, issuing an error message if a negative number is detected. Employing a `while` loop, the program executes binary conversion by iteratively dividing the number by 2. During each iteration, the remainder is calculated using the modulo operator (%), representing the binary digit for that iteration. These remainders are methodically accumulated in reverse order, with each iteration contributing to the construction of the binary representation. The final output is then displayed as the equivalent binary representation. The following program embodies the solution:

```
#include <stdio.h>
int main() {
    //Declare variables
    int n, digit, n2, weight;
    //Prompt user to enter a positive integer
    printf("Enter a positive integer: ");
    scanf("%d", &n);
    //Check for input errors (negative numbers)
    if (n < 0)
        printf("Input error");
    else {
        //Variable to store the binary equivalent (initially 0)
        n2 = 0;
        //Weight of each digit in the result (initially 1)
        weight = 1;
        //Binary conversion using a while loop
        while (n != 0) {
            //Calculate the remainder (binary digit)
            digit = n % 2;
            //Update the number by dividing it by 2
            n = n / 2;
            //Accumulate the binary representation
            n2 = n2 + digit * weight;
            //Update the weight for the next digit
            weight = weight * 10;
        }
        //Display the equivalent binary representation
        printf("The equivalent in binary is: %d", n2);
    }
    return 0;
}
```

b) Octal conversion:

The conversion into octal is performed in a manner similar to the procedure used for binary conversion. The only distinction lies in the base used for division and accumulation of remainders. While the binary conversion involves dividing the decimal number by 2, for octal conversion, the decimal number is divided by 8 during each iteration of the while loop. The program enabling this conversion is as follows:

```
#include <stdio.h>
int main() {
    //Declare variables
    int n, digit, n8, weight;
    //Prompt user to enter a positive integer
    printf("Enter a positive integer: ");
    scanf("%d", &n);
    //Check for input errors (negative numbers)
    if (n < 0)
        printf("Input error");
    else {
        //Variable to store the octal equivalent (initially 0)
        n8 = 0;
        //Weight of each digit in the result (initially 1)
        weight = 1;
        //Binary conversion using a while loop
        while (n != 0) {
            //Calculate the remainder
            digit = n % 8;
            //Update the number by dividing it by 8
            n = n / 8;
            //Accumulate the octal representation
            n8 = n8 + digit * weight;
            //Update the weight for the next digit
            weight = weight * 10;
        }
        //Display the equivalent octal representation
        printf("The equivalent in binary is: %d", n8);
    }
    return 0;
}
```

c) Conversion to a base B:

Building upon the previous programs, we have crafted a versatile solution capable of converting a number to any desired base. This program prompts the user to input a positive integer and specify the intended base for conversion. Similar to the earlier implementations, the process involves iterative divisions, with the key distinction being that the divisions are performed based on the user-specified base (denoted as B). Here is the program:

```

#include <stdio.h>
int main() {
    //Declare variables
    int n, digit, nB, weight, B;
    //Prompt user to enter a positive integer
    printf("Enter a positive integer: ");
    scanf("%d", &n);
    //Prompt user to enter the base for conversion
    printf("Enter the base for conversion: ");
    scanf("%d", &B);
    // Check for input errors (negative numbers or invalid bases)
    if (n < 0 || B < 2)
        printf("Input error");
    else {
        //Variable to store the result (initially 0)
        nB = 0;
        //Weight of each digit in the result
        weight = 1;
        //Base conversion using a while loop
        while (n != 0) {
            //Calculate the remainder (base B digit)
            digit = n % B;
            //Update the number by dividing it by the base
            n = n / B;
            //Accumulate the base B representation
            nB = nB + digit * weight;
            //Update the weight for the next digit
            weight = weight * 10;
        }
        // Display the equivalent representation in the base B
        printf("The equivalent in base %d is: %d", B, nB);
    }
    return 0;
}

```

4.8) Exercise 8: Fibonacci sequence

The Fibonacci sequence is defined as follows :

- $F_1 = 1$
- $F_2 = 1$
- $F_i = F_{i-1} + F_{i-2}$ for $i > 2$.

Write a program to calculate the n^{th} term of the Fibonacci sequence for a positive integer n given by the user.

Solution:

The solution to this exercise involves creating a program that calculates the n^{th} term of the Fibonacci sequence for a positive integer n , as specified by the user.

The program employs three variables to manage the Fibonacci sequence calculation: **Fi** represents the current value of the Fibonacci sequence, while **Fi_2** and **Fi_1** store the two preceding terms. The program begins by prompting the user to input the desired term **n** of the Fibonacci sequence.

Following the user's input, the program ensures the validity of the entered value by checking if it is a positive integer. Subsequently, it initializes the values of **Fi_2** and **Fi_1** with the pre-defined initial terms of the Fibonacci sequence (both set to 1).

To iteratively calculate the Fibonacci sequence up to the desired term, the program employs a **for loop** that starts from 3. This starting point is chosen because the first two terms of the Fibonacci sequence are already pre-defined as 1. In each iteration of the loop, the **Fi** is calculated as the sum of the current values of **Fi_2** and **Fi_1**. The values of **Fi_2** and **Fi_1** are then updated to prepare for the next iteration. This process continues until the loop reaches the specified term **n**.

The final result, representing the n^{th} term of the Fibonacci sequence, is displayed to the user.

Here's the corresponding program in C:

```
#include <stdio.h>
int main() {
    //Declare variables
    int n, i, Fi_1, Fi_2, Fi;
    //Enter a positive integer greater than zero
    printf("Enter a positive integer greater than zero: ");
    scanf("%d", &n);
    //Check for input errors (non-positive integers)
    if (n <= 0)
        printf("Input error");
    else {
        //Handle special cases for n = 1 and n = 2
        if (n == 1 || n == 2)
            Fi = 1;
        else {
            //Initialize variables for Fibonacci sequence
            Fi_1 = 1;
            Fi_2 = 1;
            Fi = 0;
            //Calculate Fibonacci sequence up to n
            for (i = 3; i <= n; i++) {
                //Update current term (Fi) based on
                //previous two terms
                Fi = Fi_1 + Fi_2;
                //Update previous two terms for
                //the next iteration
                Fi_2 = Fi_1;
                Fi_1 = Fi;
            }
        }
        //Display the calculated Fibonacci term
    }
}
```

```

        printf("Fi(%d) = %d", n, Fi);
    }
    return 0;
}

```

4.9) Exercise 9: Reciprocal Powers Sum

Create a program that prompts the user to input a positive non-zero integer n and another positive integer p . Calculate and display the sum of the reciprocal powers of the first n natural numbers raised to the power p using the formula:

$$S = \sum_{i=1}^n \frac{1}{i^p} = \frac{1}{1^p} + \frac{1}{2^p} + \dots + \frac{1}{n^p}$$

For instance, if $n = 4$ and $p = 5$, the sum S would be:

$$S = \frac{1}{1^5} + \frac{1}{2^5} + \frac{1}{3^5} + \frac{1}{4^5}$$

Ensure that the entered values are valid (positive integers) and handle any input errors appropriately.

Note: Implement the power calculation within your program without relying on predefined functions.

Solution:

The program is designed to calculate and display the sum of reciprocal powers of the first n natural numbers raised to the power p . The solution to this intricate problem involves the adept use of nested loops. Here's the systematic breakdown of the resolution process:

The process initiates by prompting the user to input two positive integers, n and p , ensuring the validity of their values. Subsequent to this, the sum (denoted as s) is initialized to 0. Following the initialization, the program utilizes a `for` loop (the outer one) to iteratively calculate the sum based on the provided formula. This main loop is responsible for traversing through the terms from 1 to n . Inside this primary loop, an inner loop is employed to calculate the power of each term without using predefined functions, thereby achieving the power to the p operation. The result of each power operation is then accumulated in the sum variable (s), which holds the overall sum of reciprocal powers. This accumulation process continues until the specified number of terms is reached. Finally, the calculated sum is displayed to the user.

Here is the program:

```

#include <stdio.h>
int main() {
    //Declare variables
    int n, p, powRes, i, j;
    float S;
    //Prompt user for input
    printf("Enter the number of terms (n): ");
    scanf("%d", &n);
    printf("Enter the power (p): ");
    scanf("%d", &p);
}

```

```

//Initialize the sum
S=0;
//Validate input
if (n <= 0 || p <= 0)
    printf("Input error");
else {
    //Calculate sum of reciprocal powers using nested loop
    for (i = 1; i <= n; i++) {
        powRes = 1;
        // Nested loop to calculate power
        for (j = 1; j <= p; j++) {
            powRes= powRes * i;
        }
        // Accumulate the result in the sum variable
        S = S+ (float)1 /powRes;
    }
    // Display the result
    printf("The sum of reciprocal powers is: %f",S);
}
return 0;
}

```

4.10) Exercise 10: Binomial Expression Expansion

In mathematics, the binomial formula is a well-known expression that allows for the expansion of a binomial expression raised to a positive integer power. The general form of the binomial formula is given by

$$(a + b)^n = \sum_{k=0}^n \binom{n}{k} a^{n-k} b^k$$

Where $\binom{n}{k}$ represents the binomial coefficient, calculated as:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

This formula provides a systematic way to expand expressions like $(a + b)^n$ into a sum of terms. The binomial coefficients represent the coefficients of each term in the expansion, and they follow the pattern of Pascal's Triangle.

Write a program that prompts the user to enter two numbers a and b , as well as a positive integer n . The program should then calculate and display the expanded form of the binomial expression $(a + b)^n$ using the binomial formula. Ensure that the entered values are valid, and handle any potential input errors.

Solution:

The solution to this problem involves utilizing nested loops to systematically calculate the binomial coefficients and powers for the expanded form of the binomial expression $(a + b)^n$.

The process begins by prompting the user to input three values: a , b , and n , ensuring their validity. The program then initializes the result variable and employs nested loops. The outer

loop controls the iteration through the terms of the binomial expansion (n terms), while the inner loops calculate the binomial coefficients and powers for each term. The results are accumulated, and the expanded form of the binomial expression is displayed to the user.

The program facilitating this expansion is as follows:

```
#include <stdio.h>
int main() {
    //Declare variables
    int a, b, n, i, k, nFact, kFact, fact, aPow, bPow;
    int coeff, Res;
    //Prompt user for input
    printf("Enter the values for a, b, and n: ");
    scanf("%d %d %d", &a, &b, &n);
    //Validate input
    if (n < 0)
        printf("Invalid input for n");
    else{
        //Display the expanded form of the binomial expression
        printf("Expanded form of (%d + %d)^%d: ", a, b, n);
        for (k = 0; k <= n; k++) {
            //Calculate n!
            nFact=1;
            for(i=1;i<=n;i++)
                nFact=nFact*i;
            //Calculate k!
            kFact=1;
            for(i=1;i<=k;i++)
                kFact=kFact*i;
            //Calculate (n-k)!
            fact=1;
            for(i=1;i<=n-k;i++)
                fact=fact*i;
            //Calculate binomial coefficient
            coeff=nFact/(kFact*fact);
            //Calculate a^(n-k)
            aPow=1;
            for(i=1;i<=n-k;i++)
                aPow=aPow*a;
            //Calculate b^k
            bPow=1;
            for(i=1;i<=k;i++)
                bPow=bPow*b;
            //Calculate the term result
            Res = coeff * aPow * bPow;
            //Display the term
            printf("%d", Res);
            // Display the term's sign if not the last term
            if(i <= n) {
                printf(" + ");
            }
        }
    }
}
```

```
        }  
    }  
}  
return 0;  
}
```

Lab No 5. Arrays and Strings

1) Objectives

The primary objective of this lab is to foster a deep and practical understanding of utilizing arrays and character strings in the context of C programming. Arrays, recognized as fundamental data structures, empower programmers to efficiently store and manage multiple values of the same type within a singular variable. The distinctive elements within an array are easily accessible through their respective indices, providing a systematic approach to organized data storage.

In the absence of a dedicated String type in the C language, character strings are ingeniously represented as arrays of characters, culminating with the special character '\0'. This lab serves as a comprehensive exploration into the intricacies of arrays and strings, offering students a robust foundation in handling structured data.

By the conclusion of this lab, students are expected to not only master the declaration and initialization of arrays and strings but also to adeptly manipulate these data structures, whether they are one or two-dimensional. Through practical exercises and hands-on activities, this lab aims to instill proficiency in students, enabling them to navigate and harness the power of arrays and character strings effectively in their C programming endeavors.

2) Recap: Key Concepts in Loops

In this comprehensive recap, we explore fundamental concepts that form the backbone of C programming: arrays, multidimensional arrays, and strings. These elements play a pivotal role in data manipulation and are essential for any programmer to grasp. Throughout this recap, we will delve into the definition, declaration, accessing of elements, and the fundamental manipulation of these data structures. Let's explore these concepts in detail.

2.1) Arrays

An array is a data structure that stores under a single name a collection of finite elements of the same type, each identified by an index or a key.

2.1.1) Declaration

The declaration of an array in c language follows the following syntax:

```
<type_elements> <name_Array>[<size>;
```

Such as: `<type_elements>` represents the type of elements composing the array, `<name_Array>` is an identifier designating the array, and `<size>` is the number of maximum usable elements.

In C language, we can initialize the array during its declaration by putting, within curly braces, the list of initialization values.

Examples:

```
int t[5]; //declares an array containing 5 integer elements
```

```
int t[5]={1,2,3,4,5}; //declares an array containing 5 integer elements and initializes it with the values from 1 to 5.
```

2.1.2) Accessing to elements of an array

It is possible to access an element of the array by specifying the index (the position) corresponding to that element within square brackets. Note that arrays are indexed starting from 0, not 1. Thus, `t[0]` indicates the first element of the array `t`. The fifth element is `t[4]`, and so on, as indicated by Figure 5.1.

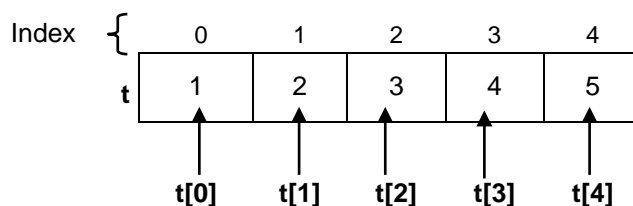


Figure 5.1. Example of an array

2.1.3) Manipulation of an array

The elements of an array are manipulated individually and possess all the properties of a variable of the same type. They can be read, written, assigned a value, etc.

Generally, manipulating an array involves using a loop, specifically the `for` loop, to iterate through array elements for operations such as accessing, reading, updating, or processing each element.

Example:

To read all elements of the array `t`, we use :

```
for (i=0; i<5; i++)
    scanf("%d", &t[i]);
```

2.2) Multidimensional arrays

Multidimensional arrays offer a structured way to organize data in more complex scenarios where elements are arranged in multiple dimensions. A multidimensional array is an array whose elements can themselves be arrays, which can, in turn, contain other arrays, and so on.

In this course, we will focus on two dimensions arrays commonly referred to as matrices.

2.2.1) Declaration

Declaring a multidimensional array involves specifying the type, name, and dimensions. For example, a two-dimensional array in C is declared as follows:

```
<type_elements> <name_Array>[<nbRows>][<nbColumns>];
```

Here, `<type_elements>` represents the type of elements composing the matrix, `<name_Array>` is an identifier for the matrix, `<nbRows>` is the number of rows, and `<nbColumns>` is the number of columns.

Similar to one-dimensional arrays, multidimensional arrays can be initialized during declaration. Initialization involves providing a list of values enclosed within nested curly braces.

Examples:

```
int M[3][4]; // declares an integer matrix of 3 rows and 4 columns
```

```
int matrix[3][4] = {{1,2,3,4},{5,6,7,8},{9,10,11,12}}; //initializes an integer matrix of 3 rows and 4 columns with specific values.
```

2.2.2) Accessing to elements of a multidimensional array

Accessing elements in a multidimensional array requires specifying the indices for both the row and column within double square brackets.

However, `M[1][2]` refers to the element of row 1 and column 2 as illustrated in Figure 5.2.

		0	1	2	3		
M	{	0	1	2	3	4	
		1	5	6	7	8	↖ <code>M[1][2]</code>
		2	9	10	11	12	

Figure 5.2. Example of 3x4 matrix

2.2.3) Manipulation of a multidimensional array

Manipulating multidimensional arrays involves nested loops to traverse elements efficiently. For a two-dimensional array, a nested `for` loop is commonly used, iterating over both rows and columns.

Example:

The following nested loops allow displaying all the elements of the matrix `M`:

```
for(i=0;i<3;i++)
    for(j=0;j<4;j++)
        printf("%d",M[i][j]);
```

2.3) Strings

Strings are sequences of characters that play a crucial role in handling text data. Unlike some other programming languages, C does not have a dedicated data type for strings. Instead, they are represented as arrays of characters, terminated by a null character `'\0'`.

2.3.1) Declaration

In C, strings can be declared and initialized in various ways, with the most common method being the use of a character array.

Strings can also be initialized during declaration:

Examples:

```
char name[20]; //declares a character array with a capacity for 19 characters
```



```
char wilaya[] = "Guelma"; //declares and initializes the string wilaya with a
specific value.
```

2.3.2) Accessing a character of the string

Accessing a character within a string is done similarly to accessing elements in a regular array. For example `wilaya[0]` refers to the first character of the string `wilaya` which is 'G'.

2.3.3) Manipulation of a String

Manipulating strings involve various string manipulation functions provided by the C Standard Library (`<string.h>`). Common operations include reading, writing, finding the length of a string, concatenating two strings, comparing strings, and extracting substrings.

Reading a string can be achieved using the `scanf` function with the format code `%s`. However, C provides other functions specifically dedicated to reading strings. One such function is `gets`, with the syntax:

```
gets (<name_String>);
```

Similarly, writing a string may be accomplished by the `printf` function with the format code `%s`. C also offers other functions dedicated to writing strings, such as the `puts` function, with the syntax:

```
puts (<name_String>);
```

Several other string manipulation functions are available, summarized in the following table:

Fonctions	Description
<code>strcat()</code>	Concatenate two strings
<code>strcmp()</code>	Compare two strings
<code>strcpy()</code>	Copy one string into another
<code>strlen()</code>	Return the length of a string

Table 5.1. Predefined functions for string manipulation

3) Practice activities

This section offers practice exercises to enhance your understanding of arrays and strings in C programming. Test, analyze, and potentially modify the provided code snippets to reinforce your skills. These activities aim to challenge you progressively, fostering active engagement and practical application of arrays and strings concepts.

3.1) Activity 1: Arrays

Consider the following program:

```
#include <stdio.h>

int main() {
    #define n 4
    int T[n]; int i;
    printf("Please fill the array\n");
    for (i = 0; i < n; i++) {
        printf("T[%d] = ", i);
        scanf("%d", &T[i]);
    }
    printf("The array T contains the values :\n");
    for (i = 0; i < n; i++) {
        printf("%d\n", T[i]);
    }
    return 0;
}
```

1. Create a new project and type the above code.
2. Compile and run, what does this program do?
3. What is the nature of the variable `T`?
4. What is the difference between a variable of type `int` and an array of the same type?
5. How can you access each element in an array?
6. What will happen if we try to access an index beyond the size of the array?
7. Replace the expression « `i < n` » in both loops with « `i < 10` » and execute the code. Is there a problem? Explain what happens.
8. Change the program to calculate the sum of the elements in the array.
9. Make the necessary modifications for the program to declare a matrix with 4 rows and 3 columns (a two-dimensional array), read its elements, and calculate their sum.

Solution:

1. Creation of a new project.
2. The provided C program prompts the user to input 4 values and then display them.
3. The variable `T` in the provided C program is an array. Specifically, it is an array of integers (`int`) used to store multiples values.
4. The primary difference between a variable of type `int` and an array of the same type lies in their nature and usage. A variable of type `int` is a singular storage location that holds a single integer value, identified by a name. On the other hand, an array of type `int`, is a collection of multiple storage locations, each capable of holding an integer value. However, while a variable is suitable for representing a single numeric entity, an array is designed to handle scenarios where multiple related values need to be stored and processed together.

5. We can access each element using its index, which depends on the order of elements in the array. The first element has an index of 0, the second has an index of 1, and so on, up to the n^{th} element, which has an index of $(n-1)$.
6. Students are very likely to answer that the program will produce an error; we need to explain to them that this is not true. In the C language, if we exceed the last index $(n-1)$, we will simply access the memory area beyond the space allocated for the array. This could potentially modify other variables if that memory area belongs to the currently running program or result in a segmentation fault if that memory area does not belong to the program (the same thing happens when you use a negative index, but in that case, you access the memory area before the allocated space for the array). The following example should help address this question.
7. If we replace the expression « $i < n$ » in both loops with « $i < 10$ » and execute the code, it might lead to issues depending on the actual size of the array T and how it was declared. Expect to see different behaviors in each example. The problems we are likely to encounter are as follows:
 - Segmentation fault during execution.
 - Infinite loop because the value of i unintentionally changes when accessing indices greater than $(n - 1)$.
 - Values can be read, but during display, you get different values or symbols.
 - and so on.

In all cases, no issues are reported during compilation; memory management is the responsibility of the programmer.

8. To compute the sum of the elements in the array, we enhance the program by introducing a dedicated variable for accumulating the sum, and adding a loop accomplishing this accumulation. Before the accumulation loop, this variable is initialized to zero. Within the loop, each element is iteratively added to the sum, ensuring a cumulative total. Below is the refined version of the program:

```
#include <stdio.h>
int main() {
    #define n 4
    int T[n], i, s;
    //Input values into the array
    printf("Please fill the array\n");
    for (i = 0; i < n; i++) {
        printf("T[%d] = ", i);
        scanf("%d", &T[i]);
    }
    //Initialize the sum variable to zero
    s = 0;
    //Calculate the sum of the elements
    for (i = 0; i < n; i++)
        s = s + T[i];
    // Display the sum
    printf("The sum of the array is : %d", s);
    return 0;
}
```

9. To tailor the program for a 4x3 matrix, involving two dimensions (rows and columns), crucial adjustments are required in the array declaration, input process, and sum calculation. This entails the addition of a second constant (representing the columns), as the initial constant (**n**) signifies the number of rows. Moreover, the introduction of an additional variable (**j**), becomes essential to traverse the columns of the matrix during input and sum calculation. The utilization of nested loops is pivotal for handling the matrix elements. Here's the modified program:

```
#include <stdio.h>
int main() {
    //Constants for the number of rows (n) and columns (m)
    #define n 4
    #define m 3
    //Declare a 2D array A with n rows and m columns
    int A[n][m];
    //Declare variables i (row index),j (column index),s (sum)
    int i, j, s;
    //Prompt the user to fill the matrix
    printf("Please fill the matrix\n");
    //Input values into the matrix using nested loops
    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++) {
            printf("Row %d, Column %d: ", i, j);
            scanf("%d", &A[i][j]);
        }
    }
    //Initialize the sum variable to zero
    s = 0;
    //Calculate the sum of elements using nested loops
    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++) {
            s = s + A[i][j];
        }
    }
    // Display the sum of the matrix
    printf("The sum of the matrix is: %d", s);
    return 0;
}
```

3.2) Activity 2 : Strings

Consider the following program:

```
#include <stdio.h>

int main() {
    #define n 10
    char s[n]; int i;
    printf("Enter a string of characters : ");
    gets(s);
    i = 0;
    while (s[i] != '\0') {
        i = i + 1;
    }
    printf("The string \"%s\" contains %d characters", s, i);
    return 0;
}
```

1. Create a new project and type the above code.
2. Compile and run, what does this program do?
3. What does the format "%s" mean in the « printf » statement?
4. Considering the previous code:
 - a) What is the maximum length of the string that can be entered without encountering problems?
 - b) Test the code with a string of length 10. Is there an issue? Explain.
 - c) Test the code with a string length greater than 10. What do you observe?
5. Modify the code to count the number of spaces in the string entered by the user.

Solution:

1. Creation of a new project.
2. This C program prompts the user to enter a string of characters and then calculates and displays the length (number of characters) of the entered string.
3. In the `printf` statement, the format specifier "%s" is used to indicate that a string of characters should be inserted at that position in the formatted output. It is similar to "%d" for integers or "%f" for floating-point numbers
4. Considering the previous code:
 - a) The maximum length of the string that can be entered without encountering problems is 9 characters. Students are very likely to answer 10. We need to explain to them that the last character is reserved for the special character '\0'. The following questions should help explain the phenomenon.
 - b) When tested with a string of length 10, the program displays a value different from 10 for the length, with improperly encoded characters at the end of the string. The problem arises because the program struggles to recognize the end of the string when the end character '\0' is not properly encoded.
 - c) Testing the code with a string length greater than 10 using the `gets` function leads to buffer overflow issues. Thus, we should observe a behavior similar to the previous

question (incorrect value for the length, improperly encoded characters at the end of the string).

5. To modify the code to count the number of spaces in the string entered by the user, we should introduce a counter variable called `nbSpaces` and iterate through the characters of the string to check for spaces. The counter is first initialized to zero. Then we use a `while` loop to iterate, and at each iteration an `if` statement checks whether the current character is a space (' '). If a space is encountered, the `nbSpaces` is incremented. After iterating through the entire string, the program prints the count of spaces it contains. Here's the modified code:

```
#include <stdio.h>
int main() {
    #define n 10
    char s[n];
    int i, nbSpaces;
    //Prompt the user to enter a string
    printf("Enter a string of characters: ");
    //Read the string from the user
    gets(s);
    //Initialize variables
    i = 0;
    nbSpaces = 0;
    //Iterate through each character in the string
    while (s[i] != '\0') {
        //Check if the current character is a space
        if (s[i] == ' ') {
            //Increment the space count
            nbSpaces = nbSpaces + 1;
        }
        //Move to the next character
        i = i + 1;
    }
    //Display the count of spaces in the entered string
    printf("The entered string contains %d spaces", nbSpaces);
    return 0;
}
```

In this implementation, a `while` loop is employed for traversing the string due to the absence of prior knowledge about the length of the string entered by the user. Although the maximum allowable length is set at 10 characters, the actual input may be shorter. Therefore, the `while` loop iterates dynamically until it encounters the null character `'\0'`, signifying the end of the string.

4) Application exercises

Embark on a deeper exploration of your C programming skills with the following application exercises. These exercises are designed to challenge and reinforce your understanding of arrays, strings, and their practical applications.

4.1) Exercise 1: Exam Grades and Statistics

Write a program that enables a teacher to enter the exam marks of a group of 30 students, stores them in an array, calculates the group's average, and determines the minimum and maximum scores. Additionally, the program should count the number of scores equal to or higher than the group's average.

Solution :

To address this problem, the solution employs loops to handle both the input of marks and the subsequent calculation of statistical measures from the array. Dedicated variables, including `min` and `max` to monitor the minimum and maximum marks, `sum` and `avg` to manage the cumulative sum of scores and their average, and `nbHighAvg` to track the count of marks equal to or surpassing the group's average, are instrumental in this process.

The initial loop, a `for` loop, guides the user in entering individual marks for each student and store them in the array. Following this, a second loop dynamically calculate the `sum` of the marks while adjusting the `min` and `max` variables. Prior to entering this loop, `sum`, `min` and `max` are initialized with the first element of the array. This second loop iterates through the array starting from the second mark (because the first one has already been visited), updating `sum`, `min` and `max` based on the encountered elements.

Upon concluding the second loop, the average (`avg`) is computed as usual. A third loop is then deployed to determine the number of scores equal to or exceeding the group's average (`avg`). Here, the counter `nbHighAvg` is initialized to zero before the loop. During each iteration the current mark is compared to `avg` and the counter `nbHighAvg` is updated accordingly. The concluding step involves displaying the calculated statistics.

The requested program is the following:

```
#include <stdio.h>
int main() {
    //Define the size of the array
    #define n 30
    //Declare an array of size n to store marks
    int T[n];
    //Variables for statistics and counters
    float min, max, sum, avg;
    int i, nbHighAvg;
    //Input scores for each student
    printf("Please enter the exam scores for 30 students:\n");
    //Loop to read marks into the array
    for (i = 0; i < n; i++) {
        printf("Enter score for student %d: ", i + 1);
        scanf("%f", &T[i]);
    }
    //Initialize variables for statistics
    //with the first mark in the array
    sum = T[0];
    min = T[0];
    max = T[0];
```

```

//Loop to calculate sum, min, and max
for (i = 1; i < n; i++) {
    //Add the current mark to the sum
    sum = sum + T[i];
    //Check if the current element is smaller
    //than the current min, update min
    if (T[i] < min) min = T[i];
    //Check if the current element is larger
    //than the current max, update max
    if (T[i] > max) max = T[i];
}
//Calculate average
avg =sum / n;
//Count scores higher than or equal to average
nbHighAvg = 0;
for (i = 0; i < n; i++) {
    if (T[i] >= avg) {
        nbHighAvg = nbHighAvg + 1;
    }
}
// Display calculated statistics
printf("Statistics:\n");
printf("Minimum score: %.2f\n", min);
printf("Maximum score: %.2f\n", max);
printf("Average score: %.2f\n", avg);
printf("Number of scores >= average: %d\n", nbHighAvg);
return 0;
}

```

4.2) Exercise 2: Array Value Search and Occurrences

- Write a program that asks the user to fill an array T of size n , and then provide a value x . The program should then indicate whether x belongs to T .
- Modify the program to display the number of occurrences of x , and the index of its first and last occurrence if it belongs to T .

Solution :

In this exercise, two programs are expected.

- For the first part of the exercise, the solution entails reading an array from the user, akin to the approach employed in the previous exercise, utilizing a `for` loop. Given that the exercise does not specify the size of the array, a size of $n=10$ is proposed. Following this, the program prompts the user for a value, denoted as x . Subsequently, a `while` loop is introduced to search through the array and determine the presence of x . To facilitate this, a Boolean variable (called `found`) is declared to ascertain whether the value x exists in the array. Initially, `found` is set to `False`. The while loop iterates as long as the array is not terminated and the value is not yet found. In each iteration, the program checks if the current value matches the sought-after x . If affirmative, the search concludes, and `found` transitions to `True`. If not, the program proceeds to the next

element and continues this process until the value is located or the array is traversed without finding **x**. After exiting the loop, the program displays whether **x** is found or not based on the value of the Boolean variable **found**. In C, the boolean variable is declared as an **int** since the boolean type does not exist. Below is the implementation of the program:

```
#include <stdio.h>
int main() {
    //Define the size of the array
    #define n 10
    //Declare the variables
    int T[n];
    int i, x, found;
    //Prompt the user to fill the array
    printf("Please fill the array\n");
    //Loop to input values into the array
    for (i = 0; i < n; i++) {
        printf("T[%d] = ", i);
        scanf("%d", &T[i]);
    }
    //Prompt the user to enter a value to search for
    printf("Enter a value: ");
    scanf("%d", &x);
    //Initialize the 'found' variable to 0 (False)
    found = 0;
    //Loop to search for the value in the array
    for (i = 0; i < n; i++) {
        //Set 'found' to 1 (True) if the value is found
        if (T[i] == x) {
            found = 1;
        }
    }
    // Display whether the value is found or not
    if (found==1) {
        printf("The value %d is in the array", x);
    } else {
        printf("The value %d is not in the array", x);
    }
    return 0;
}
```

- b. In the second part of the exercise, we diverge from using a Boolean variable, as the objective is not only to determine the existence of a value in the array and stop the search as soon as it is found. Instead, the goal is to compute the number of occurrences of a value **x** and identify the indices of its first and last occurrences in the array. In this modified program, additional variables are introduced, including a counter (**count**) to tally the number of occurrences, and two indices (**firstIndex** and **lastIndex**) to store the positions of the first and last occurrences of **x**.

As the program iterates through the array, it uses a `for` loop and increments the counter for each occurrence of `x` and updates the indices accordingly. If `x` is found at a given index, the program checks whether this index is the first occurrence (`firstIndex`), and if so, it records the index. Subsequently, as the program continues its traversal, it updates the `lastIndex` with each subsequent occurrence of `x`. After completing the loop, the program then displays the count of occurrences along with the indices of the first and last occurrences of `x` in the array.

The modified program enabling this logic is as follows:

```
#include <stdio.h>
int main() {
    #define n 10
    int T[n];
    int i, x, count, firstIndex, lastIndex;
    //Prompt user to fill the array
    printf("Please fill the array\n");
    for (i = 0; i < n; i++) {
        printf("T[%d] = ", i);
        scanf("%d", &T[i]);
    }
    //Prompt user to enter a value to search for
    printf("Enter a value: ");
    scanf("%d", &x);
    //Initialize the occurrences counters
    count = 0;
    //Initialize the variables for tracking indices
    firstIndex = -1;
    lastIndex = -1;
    //Iterate through the array to find occurrences
    for (i = 0; i < n; i++) {
        if (T[i] == x) {
            //If it's the first occurrence, record the index
            if (count == 0) {
                firstIndex = i;
            }
            //Update the index for the last occurrence
            lastIndex = i;
            //Increment the count of occurrences
            count = count + 1;
        }
    }
    // Display results based on occurrences
    if (count > 0) {
        printf("The value %d appears %d times\n", x, count);
        printf("Its first index is %d\n", firstIndex);
        printf("Its last index is %d\n", lastIndex);
    } else {
        printf("The value %d is not in the array\n", x);
    }
}
```

```

    }
    return 0;
}

```

4.3) Exercise 3: Array normalization

Write a program to input an array of 10 non-zero real elements, normalize it, and display the normalized array.

Normalization of an array involves dividing all its elements by the largest element, ensuring that all numbers fall within the range of 0 to 1.

Example :

In the array described in the figure below, the maximum value is 8.

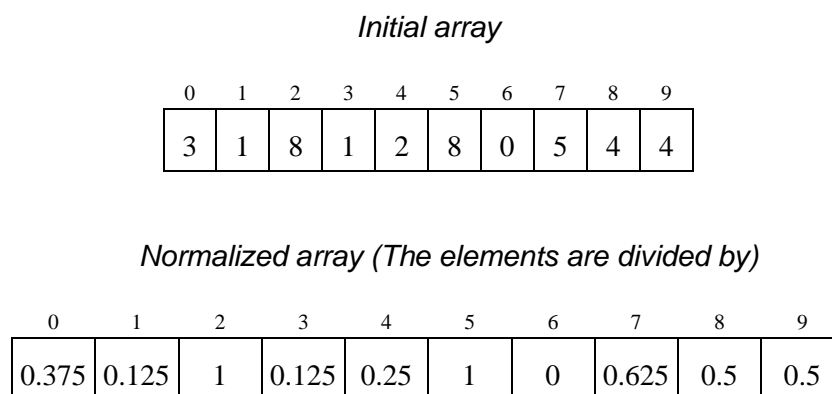


Figure 5.3. Example of the normalization of an array

Solution:

To address the requested task, the solution involves a three-step process. Firstly, the program employs a `for` loop to prompt the user for input, allowing entry of ten non-zero real numbers that are stored in an array. Moving to the second step, the program identifies the maximum element within the array using another loop, similar to the approach used in exercise 1. In the third step, the program iterates through the array, dividing each element by the identified maximum. This process ensures normalization, scaling all elements proportionally to fit within the range of 0 to 1. Finally, the normalized array is displayed, providing a clear representation of the results.

Below is the implementation of the program:

```

#include <stdio.h>
int main() {
    #define n 10
    //Declare variables
    float T[n];
    float max;
    int i;
    //Step 1: Input ten non-zero real numbers into the array
    printf("Enter ten non-zero real numbers:\n");
    for (i = 0; i < n; i++) {
        printf("T[%d] = ", i);
        scanf("%f", &T[i]);
    }
}

```

```

}
//Step 2: Find the maximum element in the array
//Initialize max with the first element of the array
max = T[0];
for (i = 1; i < n; i++) {
    //Update max if a larger element is found
    if (T[i] > max) {
        max = T[i];
    }
}
//Step 3: Normalize the array
for (i = 0; i < n; i++) {
    //Divide each element by the maximum
    T[i] = T[i] / max;
}
//Display the normalized array
printf("Normalized Array:\n");
for (i = 0; i < n; i++) {
    printf("%.3f ", T[i]);
}
return 0;
}

```

4.4) Exercise 4: Capturing State Changes in a Binary Array

Consider an array of 20 binary values (0 or 1) representing the states of a physical object at 20 different moments. We define a transition as the change of state between two consecutive moments, in other words, the shift from 0 to 1 or from 1 to 0.

Write a program to fill the array and calculate the number of transitions in the array. The program should ensure that the array is filled only with 0s and 1s.

Example:

The number of transitions in the following array is: 5

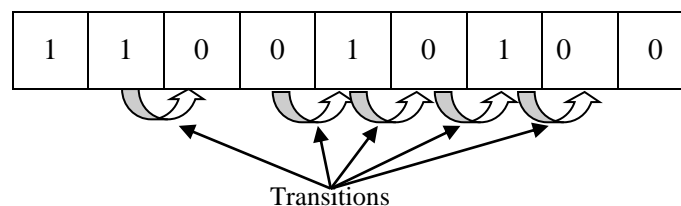


Figure 5.4. Example of transitions in a binary array

Solution:

The current problem involves capturing the states of a physical object over time in an array of binary values (0 or 1) and detecting transitions, defined as shifts between consecutive moments (from 0 to 1 or 1 to 0).

To address this challenge, the solution begins by prompting the user to input binary states and storing them in the designated array using a conventional `for` loop. A distinct aspect of this exercise is ensuring that only valid binary inputs are accepted. This is achieved by

implementing a `do...while` loop within each iteration for reading each element of the array. Following the completion of input collection, the program calculates the transition count via a second `for` loop. An integer counter (`nbTrans`) is employed to track the number of transitions, initialized to zero before the loop. In each iteration, the current state is compared with the previous one. If they are different, the counter is incremented. Note that the last loop iterates starting from the second element and not from the first, as the first element lacks a previous one for comparison. Finally, the program outputs the total number of detected transitions.

Here is the program:

```
#include <stdio.h>
int main() {
    #define n 20
    //Declare variables
    int t[n], i, nbTrans;
    //Prompt user to fill the array with binary states
    printf("Please fill the array:\n");
    for (i = 0; i < n; i++) {
        //Ensure only 0 or 1 is entered
        do {
            printf("Enter the state (0 or 1) at instant %d: ", i);
            scanf("%d", &t[i]);
        } while (t[i] != 0 && t[i] != 1);
    }
    nbTrans = 0;
    // Count transitions by comparing current state with the
previous one
    for (i = 1; i < n; i++) {
        if (t[i] != t[i - 1]) {
            nbTrans = nbTrans + 1;
        }
    }
    // Display the number of transitions
    printf("The number of transitions is: %d", nbTrans);
    return 0;
}
```

4.5) Exercise 5: Sieve of Eratosthenes

We want to create a program to find all prime numbers between 1 and a certain maximum value N using the Sieve of Eratosthenes method.

Eratosthenes was an ancient Greek scholar who led the great Library of Alexandria two and a half centuries before Christ. He became famous for the sieve method that bears his name, which provides a list of all prime numbers below a given value.

The principle of the sieve is straightforward. Given an integer N , we initially create a list of integers from 1 to N . The process involves marking (sifting) all numbers that are not prime in the following manner:

- Eliminate 1, as 1 is not a prime number.

- Move to the next number, which is 2, and eliminate all its multiples.
- Move to the next number (3) and eliminate all its multiples.
- Continue this process for all numbers up to \sqrt{N} .

Thus, the sieve, the structure containing the sequence of integers, is usually represented by an array. However, the array elements can be simply booleans, as only the presence or absence of the number in the sieve is significant.

Solution:

To implement the Sieve of Eratosthenes we have first to fix the value of N as a constant ($N = 100$ in the following program). The solution employs an array of boolean values representing the sieve. Initially, all elements are set to `true` to indicate that every number is potentially prime. The algorithm then iterates through the array, starting from the first prime number (2), and marks its multiples as non-prime. This process continues until the square root of N is reached.

The algorithm involves nested loops. The outer loop iterates through the numbers from 2 to the square root of N . For each number, the inner loop marks its multiples in the array as `false`. After completing the sieve, the algorithm scans the array, and the numbers with `true` values are identified as prime.

The program implementing the sieve of Eratosthenes is as follows:

```
#include <stdio.h>
#include <math.h>
int main() {
    //Set the maximum value
    #define N 100
    int isPrime[N+1];
    int i, j;
    //Initialize the Sieve (array)
    //To simplify, assume the number 0 and 1 as not prime
    isPrime[0] = 0;
    isPrime[1] = 0;
    //Assume all other numbers are prime initially
    for (i = 2; i <= N; i++) {
        isPrime[i] = 1;
    }
    // Apply the Sieve of Eratosthenes
    for (i = 2; i <= sqrt(N); i++) {
        //If the current number is prime,
        //marks its multiples in the array as false.
        if (isPrime[i]==1) {
            for (j = i+1; j <= N; j++) {
                if(j%i==0){
                    isPrime[j] = 0;
                }
            }
        }
    }
}
```

```

//Display prime numbers
printf("Prime numbers up to %d are:\n", N);
for (i = 0; i < N; i++) {
    if (isPrime[i] == 1) {
        printf("%d ", i);
    }
}
return 0;
}

```

4.6) Exercise 6: Sparse Matrix Detection

Consider a scenario where you are overseeing a logistics company with 5 different transportation routes and 4 types of vehicles. Create a program to read a matrix representing the usage of each vehicle on specific routes. The matrix, having 5 rows (routes) and 4 columns (vehicle types), should be examined to determine if it qualifies as sparse.

In this context, a matrix is deemed sparse if over two-thirds ($2/3$) of its elements are zero. This analysis can provide insights into routes where specific vehicle types may not be frequently utilized, aiding in strategic decision-making for resource allocation.

Solution:

To determine whether a given matrix is sparse, the solution begins by prompting the user to input the matrix elements. This process involves the use of nested loops for efficient traversal of the matrix. Subsequently, the solution utilizes nested loops again to traverse the matrix, counting the number of zero elements. A counter variable is initialized to zero before the loop, and during each iteration, the current element's value is compared to zero. The counter is updated based on this comparison.

The program then compares this counter with the predefined threshold for sparsity, set at two-thirds of the total number of elements in the matrix. Finally, the program outputs the result, indicating whether the matrix is considered sparse based on the comparison results.

The program enabling this processing is as follows:

```

#include <stdio.h>
int main() {
    //Define the number of rows and columns
    #define n 5
    #define m 4
    //Declare the matrix A and the other variables
    int A[n][m];
    int i,j,nbZeros;
    // Prompt the user to input the matrix elements
    printf("Please enter the matrix elements:\n");
    // Nested loops to input matrix elements
    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++) {
            printf("A[%d][%d] = ", i, j);
            scanf("%d", &A[i][j]);
        }
    }
}

```

```

//initialize the zero counter
nbZeros = 0;
//Count the number of zero elements
for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
        //Check if the entered element is zero
        if (A[i][j] == 0) {
            nbZeros++;
        }
    }
}
// Output whether the matrix is sparse or not
if (nbZeros > (2 * n * m) / 3) {
    printf("The matrix is sparse");
} else {
    printf("The matrix is not sparse");
}
return 0;
}

```

4.7) Exercise 7: Image thresholding

You are tasked with implementing a program for image thresholding, a fundamental technique in image processing. In this context, the image is represented as a matrix, where each element signifies the gray level of a pixel within the range of 0 to 255.

Image thresholding involves setting matrix elements (pixels) to 0, representing black, if their value is less than a defined threshold. Conversely, if their value is equal to or above the threshold, they are set to 255, representing white. The threshold for this exercise is determined as the average of all gray levels in the image.

The program should read a matrix corresponding to an image, calculate the threshold, apply thresholding, and finally display the resulting image (matrix). For simplicity, it is assumed that the values entered by the user are correct, and thus, there is no need to ensure their validity.

Solution:

To implement image thresholding based on the average gray level, the solution involves several steps. First, the program reads a matrix representing the image from the user using nested `for` loops. Subsequently, it calculates the average gray level by traversing the matrix and summing up all the gray levels. Thus, Other nested loops are employed to traverse the matrix and calculate the sum of its elements. The sum is initialized to zero before entering the loops. Once the loops are complete, the threshold is determined as the average of the gray levels.

Once the threshold is established, the program iterates through the matrix again using nested loops. For each pixel, an `if` statement checks the gray level of the pixel (the value of the element). If it is less than the threshold, it is set to 0 (black); otherwise, it is set to 255 (white). Finally, the resulting image matrix is displayed.

Below is the corresponding program in C:

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



```

int main() {
    //Constants for matrix size
    #define n 3 // Number of rows
    #define m 3 // Number of columns
    // Matrix representing the image
    int A[n][m];
    //Other variables
    int i,j,sum,th;
    //Read the matrix elements from the user
    printf("Enter the matrix elements (gray levels):\n");
    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++) {
            printf("Pixel (%d,%d): ",i,j);
            scanf("%d", &A[i][j]);
        }
    }
    //Calculate the sum of gray levels
    sum = 0;
    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
            sum=sum + A[i][j];
    //Calculate the threshold as the average gray level
    th = sum / (n * m);
    //Apply thresholding to the image matrix
    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++) {
            //Set pixel to black (0) or white (255)
            //based on the threshold
            if(A[i][j] < th)
                A[i][j] = 0;
            else A[i][j] = 255;
        }
    }
    //Display the resulting image matrix
    printf("Resulting Image Matrix:\n");
    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++) {
            printf("%d ", A[i][j]);
        }
        printf("\n");
    }
    return 0;
}

```

4.8) Exercise 8: Diagonal Permutation

Write a program to input a square matrix of size $N \times N$ and swap its two diagonals. Finally, display the resulting matrix.

Example:

The following figure illustrates an example of such permutation:

		Original matrix						Matrix after permutation					
		0	1	2	3	4	5	0	1	2	3	4	5
0	7	3	1	5	3	5	5	3	1	5	3	7	
1	9	11	25	9	4	0	9	4	25	9	11	0	
2	8	8	0	1	6	1	8	8	1	0	6	1	
3	4	7	1	0	3	4	4	7	0	1	3	4	
4	27	15	4	6	5	2	27	5	4	6	15	2	
5	9	2	4	5	27	3	3	2	4	5	27	9	

Figure 5.5. Example of a matrix before and after diagonal permutation

Solution :

To implement the diagonal permutation of a square matrix, the solution involves user input for the initial matrix. Nested loops are employed to traverse the matrix, facilitating the entry of values. After obtaining the matrix, the program proceeds with swapping its two diagonals.

The algorithm for the diagonal permutation is executed using a loop that iterates through each row. Within each row, the algorithm swaps the elements that correspond to both diagonals. In a square matrix, the main diagonal consists of elements where the row index (i) equals the column index (i). The secondary diagonal consists of elements where the sum of the row and column indices equals one less than the matrix size ($N - 1$). For each row (i), the algorithm exchanges the element at position (i, i) with the element at position ($i, N-1-i$) by using a temporary variable.

Finally, the resulting matrix, after the diagonal permutation, is displayed to the user.

The permutation processing is implemented by the following program:

```
#include <stdio.h>
main() {
    //Set the size of the square matrix
    #define N 5
    //Declare the square matrix and the other variables
    int M[N][N];
    int i,j,x;
    //Input the initial matrix
    printf("Please enter the matrix:\n");
    for(i=0;i<N;i++){
        for(j=0;j<N;j++){
            printf("Element [%d,%d]: ",i,j);
            scanf("%d",&M[i][j]);
        }
    }
    //Swap the diagonals
```

```

for(i=0;i<N;i++){
    x=M[i][i];
    M[i][i]=M[i][N-1-i];
    M[i][N-1-i]=x;
}
//Display the resulting matrix after diagonal permutation
printf("The matrix after diagonal permutaion:\n");
for(i=0;i<N;i++){
    for(j=0;j<N;j++){
        printf("%d\t",M[i][j]);
    }
    printf("\n");
}
return 0;
}

```

4.9) Exercise 9: Business Sales Analysis

Envision a dynamic business landscape with 15 strategically located branches offering nine distinct product categories. Tracking daily sales for each product category at every branch is critical for informed decision-making.

Your mission is to develop a program to efficiently input daily sales data for all product categories across all 15 branches. To manage this complex dataset, the most fitting data structure is a matrix. Here, the matrix's rows correspond to individual branches, while its columns represent various product categories. Consequently, the matrix entry $[i, j]$ encapsulates the daily sales of product category j for branch i .

Following the input phase, the program will undertake key calculations:

- **Total Sales Breakdown:** Compute and showcase the total sales for each branch and product category. Notably, these totals will be strategically stored, with the last column containing the total sales for each branch, and the last row encapsulating the total sales for each product category.
- **Performance Peaks:** Uncover and print valuable insights by identifying the branch and product category with the highest total sales. This provides a snapshot of the most successful facets of the business.

Solution:

To address the challenge of efficiently managing and analyzing daily sales data for a business with 15 branches and nine product categories, the proposed solution employs a matrix-based approach. Here, rows correspond to individual branches, and columns signify distinct product categories. It's worth noting that during the declaration, an additional row and column are added, resulting in a matrix of size $16 * 10$. These extra rows and columns are dedicated to storing the total sales.

The initial phase of the program involves user input, where nested `for` loops prompt for daily sales figures for each product category at every branch. This data is systematically organized in the matrix, providing a structured representation of the business's sales landscape.

Following data input, the program proceeds with crucial calculations. To compute the total sales for each branch, the program utilizes a `for` loop to iterate through each row (branch), summing up daily sales and storing the sum in the last entry of the row. Similarly, the program iterates through each column (product category), summing up daily sales and storing the result in the last entry of the column.

To identify the branch and product category with the highest total sales, the solution involves iterating through the last row of the matrix (total sales for each product category) and through the last column (total sales for each branch). This is achieved using the same logic as the maximum determination in Exercise 1. The final step is to print all calculated values.

The program is the following:

```
#include <stdio.h>
int main() {
    //The maximum number of branches and product categories
    #define nbBran 4
    #define nbCat 3
    //Declare a matrix to store daily sales and results
    int A[nbBran + 1][nbCat + 1];
    int i,j;
    //Variables for branch and category with the highest total
    int maxBra, maxCat;
    //Initialize matrix elements to zero
    for (i = 0; i <= nbBran; i++) {
        for (j = 0; j <= nbCat; j++) {
            A[i][j] = 0;
        }
    }
    //Fill in daily sales for each branch and category
    for (i = 0; i < nbBran; i++) {
        for (j = 0; j < nbCat; j++) {
            printf("Sales for Branch %d,Category %d: ",i+1,j+ 1);
            scanf("%d", &A[i][j]);
        }
    }
    //Calculate and store total sales for each branch
    for (i = 0; i < nbBran; i++) {
        A[i][nbCat] = 0;
        for (j = 0; j < nbCat; j++) {
            A[i][nbCat] = A[i][j]+A[i][nbCat];
        }
    }
    //Calculate and store total sales for each category
    for (j = 0; j < nbCat; j++) {
        A[nbBran][j] = 0;
        for (i = 0; i < nbBran; i++) {
            A[nbBran][j] = A[i][j]+A[nbBran][j];
        }
    }
}
```

```

//Identify the branch and category with the highest total
maxBra = 0;
maxCat = 0;
for (i = 1; i < nbBran; i++) {
    if (A[i][nbCat] > A[maxBra][nbCat]) {
        maxBra = i;
    }
}
for (j = 1; j < nbCat; j++) {
    if (A[nbBran][j] > A[nbBran][maxCat]) {
        maxCat = j;
    }
}
// Display the calculated values
printf("\nTotal sales for each branch:\n");
for (i = 0; i < nbBran; i++) {
    printf("Branch %d: %d\n", i + 1, A[i][nbCat]);
}
printf("\nTotal sales for each product category:\n");
for (j = 0; j < nbCat; j++) {
    printf("Category %d: %d\n", j + 1, A[nbBran][j]);
}
printf("\nBranch with the highest total: %d\n",maxBra+1);
printf("Product with the highest total: %d",maxCat+1);
return 0;
}

```

4.10) Exercise 10: Palindrome Checker

A word is called a "palindrome" if it reads the same from left to right or from right to left.

Write a program that asks the user to provide a word of maximum 10 characters and then determines whether the word is a palindrome.

You are not allowed to use predefined functions for this challenge.

Example:

"radar", "level", "redder", "racecar" are palindromes.

Solution:

This exercise involves creating a program to determine whether a given word is a palindrome. The solution starts by prompting the user to input a word with a maximum length of 10 characters. This input is obtained similarly to any other values and doesn't require a loop.

The program then checks whether the provided word is a palindrome by comparing characters from both ends of the word towards the center. Before this check, the program calculates the actual length of the entered word using a **while** loop, as demonstrated in Activity 2. Subsequently, another **while** loop is used to iterate through the characters of the word, starting from the first character and the last character located at the position (length-1) and moving towards the center.

If, at any point, the characters do not match, the word is not a palindrome, and the loop is interrupted. On the other hand, if the loop completes without finding a mismatch, the word is confirmed to be a palindrome. To achieve this, a Boolean variable is used, initialized to **true** before the loop. This variable controls the loop, switching to **false** in the case of a mismatch, leading to an exit from the loop.

Here is the requested program:

```
#include <stdio.h>
#include <string.h>
int main() {
    #define n 10 //Maximum length of the word
    //Declare variables
    char W[n]; //The word
    int i,j,length,isPalindrome;
    //Prompt user for input
    printf("Enter a word (maximum length %d): ", n);
    gets(W);
    //Calculate the actual length of the entered word
    i = 0;
    while(i < n && W[i]!='\0')
        i++;
    length = i;
    //Initialize the flag for palindrome check
    isPalindrome = 1;
    //Check if the word is a palindrome
    i = 0;
    while (i < length / 2 && isPalindrome==1) {
        if (W[i] != W[length - 1 - i]) {
            isPalindrome = 0;
        }
        i = i + 1;
    }
    //Display the result
    if (isPalindrome == 1) {
        printf("%s is a palindrome", W);
    } else {
        printf("%s is not a palindrome", W);
    }
    return 0;
}
```

Lab No 6. Custom Types (Structures and enumeration)

1) Objectives

This lab focuses on custom types, particularly structures and enumerations, in the C programming language. Students will engage in hands-on activities to develop a strong understanding of these concepts. The emphasis is on practical learning, allowing students to become proficient in creating, initiating, and managing these user-defined types. The goal is to help students grasp the syntax and meaning of custom type declarations, enabling them to build personalized data structures for various programming needs.

Through practical exercises, students will seamlessly integrate structures and enumerations into their programs. Exploring structures will teach students how to consolidate diverse data elements into a unified type for improved data organization. Simultaneously, the usage of enumerations will be explored to provide a clear representation of named constant values, contributing to enhanced code readability.

The ultimate objective of the lab is to equip students with both practical skills in working with custom types and strategic insights to identify optimal use cases for structures and enumerations in real-world programming scenarios.

2) Recap: Key Concepts in custom types

Beyond the predefined types, the C language provides the flexibility to define custom types, also known as user-defined types, using the `typedef` keyword. This capability allows programmers to create their own types tailored to specific needs.

Example:

```
typedef int number;  
//defines a new custom type called number whose values are those of the predefined int  
type.
```

In this lab, we delve into two primary types of custom types: **enumerations** and **structures**.

2.1) Enumerations

Enumerations offer a convenient way to define named constant values in a program. By assigning names to integral constants, enumerations enhance code clarity and maintainability.

2.1.1) Declaration of an enumeration

The declaration of an enumeration involves defining a set of named constants enclosed within curly braces. Each constant is assigned an integer value, starting from zero by default,

and subsequent values increment by one. The `typedef` keyword allows creating a custom type for the enumeration, facilitating its use in the program.

Example:

```
typedef enum {
    Red,
    Green,
    Blue
} Colors;

//Defines an enumeration named Colors with constants Red, Green, and Blue.
```

2.1.2) Manipulation of an enumeration

Manipulating enumerations in C involves various operations, including reading, writing, assigning, comparing, and utilizing these symbolic constants within the program.

Enumerations in C are read and written using standard input functions (`scanf` and `printf`). The format code `%d` is used to read enumeration values, treating them as integer variables.

Enumerations can also be easily assigned and compared like any other integral data type.

Assigning values to enumeration variables is straightforward. Simply use the assignment operator (`=`) with the desired enumeration constant.

For comparisons, enumerations can be compared using standard relational operators (`==`, `!=`, `<`, `>`, `<=`, `>=`). This allows for conditional operations based on enumeration values.

Examples:

```
Colors color1; //Declare an enumeration variable.
scanf("%d", &color1); //Read an enumeration value.
color1 = Blue; // Assign an enumeration value.
printf("Selected color: %d", color1); // Display the enumeration value.
if (color1 == Red) {
    .....
}
else {
    .....
}
```

2.2) Structures

A structure in the C language is a data structure that enables the aggregation of data, whether of the same or different types, into a single entity associated with a particular object. This composite record consists of individual elements known as **fields**, where each field represents a distinct piece of data.

2.2.1) Declaration of a structure

In C, defining a structure involves using the `struct` keyword in two-step process. There are two ways to declare a structure. The first way, without using `typedef`.

The syntax of declaration is as follows:

```
struct <name_Structure>{
    <type_Fields1> <name_Fields1>;
    <type_Fields2> <name_Fields2>;
    ...
    <type_Fieldsn> <name_Fieldsn>;
};
```

Where: `<name_Structure>` is the name of the defined structure, `<type_Fieldsi>` is the type of the i^{th} field of the structure, and `<name_Fieldsi>` is the name of the i^{th} field of the structure.

This way of declaration requires using `struct` along with the structure name for variable declarations.

```
struct <name_Template> <name_Variable>;
```

Example:

```
struct Person{
    char firstName[30];
    char lastName[30];
    int age;
};
struct Person pers;
// Declaration of a structure variable named "pers"
```

The second way, with `typedef`, allows creating an alias for the structure, making variable declarations cleaner and more readable. The syntax for defining a structure type in C is as follows:

```
typedef struct {
    <type_Field1> <name_Field1>;
    <type_Field2> <name_Field2>;
    ...
    <type_Fieldn> <name_Fieldn>;
} <name_Type>;
```

Where: `<name_Type>` is the name of the defined type.

In this second method, you can directly declare a variable using the alias without the need for `struct`:

```
<name_Type> <name_Variable>;
```

Example:

```
typedef struct {
    char firstName[30];
    char lastName[30];
    int age;
} Person;
Person pers;

// Declaration of a structure variable named "pers"
```

2.2.2) Accessing Structure Fields

To access the fields within a structure, the dot (.) operator is employed. Thus, field access follows the following syntax:

```
<name_Structure>.<name_Field>
```

Example:

```
Pers.age; //access to the field age of the structure pers
```

2.2.3) Manipulation of a structure

Manipulating a structure in the C language involves various operations, facilitating the interaction with data encapsulated within the structure. These operations encompass reading, writing, modifying, and utilizing the structure's fields.

In fact, the only possible instruction to manipulate a variable of record type (in its entirety) without accessing its fields is assignment. However, the fields of a record can be manipulated individually like any other variable of a similar type. They can be read, written, assigned values, and used in conditions, loops, etc.

Example:

```
Person pers1,pers2;
scanf("%s",&pers1.lastName); //read the 'lastName' field
pers1.age=19; //assign the value 19 to the 'age' field
printf("%s",firstName); //print the 'firstName' field
pers2=pers1; //Assign the values of 'pers1' to 'pers2'
```

3) Practice activities

In this section, you will apply your understanding of custom types, specifically structures and enumerations, through a series of hands-on practice activities that will enhance your proficiency in declaring, initializing, and manipulating these user-defined types in the C programming language.

3.1) Activity 1: Enumerations

Imagine you are developing a straightforward program to manage different days of the week. Enumerations can be a helpful way to represent the days in a structured manner. The following program, when completed, is designed to deliver appropriate messages based on the current day entered by the user, utilizing a switch statement. For instance, it might output messages like "It's the weekend, relax," "Weekend, do not forget the Jumu'ah prayer," or "It's a working day".

```

#include <stdio.h>
int main() {
    //Declare the enumeration type for Days
    enum Days {
        Saturday, Sunday, Monday, Tuesday, Wednesday, Thursday, Friday
    };
    //Declare a variable of type Days
    enum Days today;
    //Input the current day
    .....
    //Switch statement to provide messages based on the day
    switch (.....) {
        case .....:
            .....
            break;
        .....
        default:
            .....
            break;
    }
    return 0;
}

```

1. Fill in the above program to complete the enum-based day management system.
2. Test the program by changing the value of `today` and observing the different outputs based on the day.

Solution:

1. To fulfill the specified task, we need to incorporate the reading instruction to prompt the user for input regarding the current day. Additionally, we must augment the `switch` statement with the selector (the `today` variable), enumerate all the cases within the `switch`, and define the corresponding processing for each case. In this scenario, the processing involves displaying an informative message based on the current day entered by the user. The comprehensive program is outlined below:

```

#include <stdio.h>
int main() {
    //Declare the enumeration type for Days
    enum Days {
        Saturday, Sunday, Monday, Tuesday, Wednesday, Thursday, Friday
    };
    //Declare a variable of type Days
    enum Days today;
    //Input the current day
    printf("Enter the current day (0:Saturday, ..., 6:Friday):");
    scanf("%d", &today);
    //Switch statement to provide messages
    switch (today) {
        case Saturday:

```

```
        printf("It's the weekend, relax!\n");
        break;
    case Sunday:
        printf("Start your week with enthusiasm!");
        break;
    case Monday:
        printf("It's a Working day");
        break;
    case Tuesday:
        printf("It's Working day");
        break;
    case Wednesday:
        printf("It's Working day");
        break;
    case Thursday:
        printf("It's a Working day");
        break;
    case Friday:
        printf("Weekend, do not forget the Jumu'ah prayer");
        break;
    default:
        printf("Input error");
        break;
}
return 0;
}
```

2. When testing the program with various values for the `today` variable, the system responds by displaying one of four messages, each corresponding to a specific day of the week entered by the user. This functionality allows users to observe the program's dynamic behavior, showcasing its ability to provide tailored messages based on different inputs.

3.2) Activity 2: Structures

The following program is supposed to calculate the sum of two durations entered by the user. Duration is considered here as a structure type composed of three fields: `h`, `m`, and `s` of type `int`, describing respectively: hours, minutes, and seconds.

```

#include <stdio.h>
int main() {
    struct Duration{
        int h,m,s;
    };
    struct Duration d1;
    struct Duration d2;
    struct Duration ds;
    printf("Please enter the first duration: \n");
    printf("Hours: ");scanf("%d",&d1.h);
    printf("Minutes: ");scanf("%d",&d1.m);
    printf("Seconds: ");scanf("%d",&d1.s);
    printf("Please enter the second duration: \n");
    printf("Hours: ");scanf("%d",&d2.h);
    printf("Minutes: ");scanf("%d",&d2.m);
    printf("Seconds: ");scanf("%d",&d2.s);
    .....
    .....
    printf("The sum is: %dh %dm %ds",ds.h,ds.m,ds.s);
    return 0;
}

```

1. Create a new project and type the code above.
2. Complete this code to perform the desired task.
3. How can you avoid the repetition of the `struct` keyword when declaring variables `d1`, `d2`, and `ds`?

Solution:

1. Creation of a new project.
2. To achieve the intended objective, the chosen approach involves converting both durations into seconds, computing their sum, and then converting the total seconds back into hours, minutes, and seconds. These conversions utilize integer division (/) and modulo operations (%). The program declares three integer variables: one for the first duration in seconds, another for the second duration in seconds, and the last one for the sum of durations in seconds.

Here is the provided program:

```

#include <stdio.h>
int main() {
    // Structure definition for Duration
    struct Duration {
        int h, m, s;
    };
    //Declare variables for two durations and the result
    struct Duration d1, d2, ds;
    //Input for the first duration
    printf("Please enter the first duration:\n");
    printf("Hours: ");
    scanf("%d", &d1.h);

```

```

printf("Minutes: ");
scanf("%d", &d1.m);
printf("Seconds: ");
scanf("%d", &d1.s);
//Input for the second duration
printf("Please enter the second duration:\n");
printf("Hours: ");
scanf("%d", &d2.h);
printf("Minutes: ");
scanf("%d", &d2.m);
printf("Seconds: ");
scanf("%d", &d2.s);
//Convert durations to seconds
int secondsD1 = d1.h * 3600 + d1.m * 60 + d1.s;
int secondsD2 = d2.h * 3600 + d2.m * 60 + d2.s;
//Calculate the sum in seconds
int sumInSeconds = secondsD1 + secondsD2;
//Convert the sum in seconds to hours,minutes,and seconds
ds.h = sumInSeconds / 3600;
ds.m = (sumInSeconds % 3600) / 60;
ds.s = sumInSeconds % 60;
//Display the sum
printf("The sum is: %dh %dm %ds\n", ds.h, ds.m, ds.s);
return 0;
}

```

3. To avoid the repetition of the struct keyword when declaring variables `d1`, `d2`, and `d4`, we can use the `typedef` keyword to create an alias for the structure `Duration`. Here's the modification:

```

typedef struct{
    int h,m,s;
}Duration;
Duration d1;
Duration d2;
Duration ds;
.....

```

4) Application Exercises

These exercises of this section are designed to reinforce your skills in utilizing custom types effectively. Work through each exercise, applying the concepts learned in the previous sections, and enhance your problem-solving abilities in real-world scenarios.

4.1) Exercise 1: Geometric Shape Calculator

Design an enumeration named `Shape` to represent geometric shapes such as circle, square, and triangle. Write a program that allows the user to choose a shape, then calculates and prints the area of the selected shape.

You can use simple `switch` statements for this.

Solution:

The problem addressed in this exercise is the computation of the area of a geometric shape selected by the user. We assume here the availability of only three types of shapes: circle, square, and triangle.

To tackle this problem, the solution utilizes an enumeration named `Shape` to represent three geometric shapes: `CIRCLE`, `SQUARE`, and `TRIANGLE`. The program initiates by prompting the user to input a number corresponding to its desired shape (0 for Circle, 1 for Square, 2 for Triangle). A `switch` statement is then used to handle each case, where the program prompts the user for relevant dimensions (e.g., radius, side length, base, and height) and calculates the area accordingly. The area calculation is performed within each `switch` case, storing the result in a variable named `area`. Subsequently, the program prints the calculated area with a message specifying the area of the selected shape. In the event of an invalid choice, the program displays an error message.

The program implementing this functionality is presented as follows:

```
#include <stdio.h>
int main() {
    //Enumeration for geometric shapes
    enum Shape {
        CIRCLE,
        SQUARE,
        TRIANGLE
    };
    enum Shape choice;
    float radius,base,height,side,area;
    // Prompt the user to choose a shape
    printf("Choose a shape (0:Circle,1:Square,2:Triangle): ");
    scanf("%d",&choice);
    switch (choice) {
        case CIRCLE: {
            printf("Enter the radius of the circle: ");
            scanf("%f", &radius);
            //Area of a circle:  $\pi * r * r$ 
            area = 3.14159 * radius * radius;
            break;
        }
        case SQUARE: {
            printf("Enter the side length of the square: ");
            float side;
            scanf("%f", &side);
            // Area of a square: side * side
            area = side * side;
            break;
        }
        case TRIANGLE: {
            printf("Enter the base and height of triangle: ");
            scanf("%f%f", &base, &height);
```

```

        // Area of a triangle: 0.5 * base * height
        area = 0.5 * base * height;
        break;
    }
    default:
        printf("Invalid choice.\n");
}
printf("The area of the selected shape is: %.2f\n", area);
return 0;
}

```

4.2) Exercise 2: Traffic Light Controller

In this exercise, you are requested to simulate a simple traffic light controller using enumerations. The traffic light can be in one of three states: **RED**, **YELLOW**, or **GREEN**. The objective is to create a program that prompts the user to input the current state of a traffic light, and based on that input, it cycles through these states, mimicking the behavior of a real-world traffic light.

Solution:

This exercise involves creating a program to simulate a traffic light controller using C enumerations. The traffic light is modeled with three states: **Red**, **Yellow**, and **Green**. The solution employs an enumeration named **TrafficLight** to represent these states. A variable, **currentLight**, is initialized to indicate the current state of the traffic light. The program then enters a loop to cycle through the traffic light sequence: **Red** → **Green** → **Yellow** → **Red**. During each iteration, the program displays the current state of the traffic light. To mimic real-world timing, the **sleep** function is used to introduce delays between state changes, making the simulation more realistic. The program repeats this sequence a specified number of times, providing a visual representation of a functioning traffic light.

Here is the program:

```

#include <stdio.h>
#include <unistd.h> //Required for sleep function
int main() {
    //Enumeration for Traffic Light States
    enum TrafficLight {
        Red, Yellow, Green
    };
    //Variable to represent the current state of traffic light
    enum TrafficLight currentLight;
    //Input the current state of the traffic light
    printf("Enter the current state of the traffic light ");
    printf("(0 for Red, 1 for Yellow, 2 for Green): ");
    scanf("%d", &currentLight);
    //Validate the user's input
    if (currentLight < 0 || currentLight > 2) {
        printf("Invalid input");
    }
    else{

```



```

//Simulate the traffic light sequence in a loop
for (int i = 0; i < 10; ++i) {
    //Display the current state of the traffic light
    //and Update the state based on the current state
    printf("Traffic Light is ");
    switch (currentLight) {
        case Red:
            printf("Red\n");
            currentLight = Green;
            break;
        case Yellow:
            printf("Yellow\n");
            currentLight = Red;
            break;
        case Green:
            printf("Green\n");
            currentLight = Yellow;
            break;
    }
    //Simulate a delay
    sleep(2); //Sleep for 2 seconds
}
return 0;
}

```

4.3) Exercise 3: Population Statistics

Consider a population of 100 persons, and each person can fall into one of the following categories: student, worker, unemployed, or retired.

Design a C program that uses an enumeration named **Category** to represent these categories. Create an array to store the category of each person, and allow the user to input the category for each of the 100 persons. After entering the data, display statistics about the distribution of categories, including the count of persons in each category. Ensure to handle input validation to ensure that the user enters a valid category.

Solution :

The solution involves the use of an enumeration named **Category** to represent the four categories: student, worker, unemployed, and retired. An array is established to store the category assigned to each of the 100 persons. The program utilizes a **for** loop to prompt the user to input the category for each person. Within each iteration, a **do...while** loop ensures input validation, guaranteeing that the user enters a valid category.

Following the data input phase, the program proceeds to compute the count of individuals in each category. This is achieved by utilizing four integer variables as counters, initialized to zero. A subsequent **for** loop traverses the array, comparing the category of each person and incrementing the appropriate counter based on the current category. The comparison is facilitated by a **switch** statement.

The program, structured as described, is as follows:

```
#include <stdio.h>
int main() {
    #define n 100
    //Enumeration for person categories
    enum Category {
        Student,
        Worker,
        Unemployed,
        Retired
    };
    //Array to store the category of each person
    enum Category T[n];
    int nbStudents,nbWorkers,nbUnemployed,nbRetired;
    //Enter the category for each person
    printf("Enter the category for each person ");
    printf("0:Student,1:Worker,2:Unemployed,3:Retired):\n");
    for (int i = 0; i < n; ++i) {
        //Input validation
        do {
            printf("Person %d: ", i + 1);
            scanf("%d",&T[i]);
        } while (T[i] < Student || T[i] > Retired);
    }
    //Count of persons in each category
    nbStudents=0;nbWorkers=0;nbUnemployed=0;nbRetired=0;
    for (int i = 0; i < n; ++i) {
        switch (T[i]) {
            case Student:
                nbStudents=nbStudents+1;
                break;
            case Worker:
                nbWorkers=nbWorkers+1;
                break;
            case Unemployed:
                nbUnemployed=nbUnemployed+1;
                break;
            case Retired:
                nbRetired=nbUnemployed+1;
                break;
        }
    }
    //Display statistics
    printf("\nStatistics:\n");
    printf("- Students: %d persons\n",nbStudents);
    printf("- Workers: %d persons\n", nbWorkers);
    printf("- Unemployed: %d persons\n", nbUnemployed);
    printf("- Retired: %d persons", nbRetired);
    return 0;
}
```

4.4) Exercise 4: Deck of Cards

We want to create a deck of 32 cards. The deck is represented by an array of 32 distinct elements, each of the type `Card`. This type is characterized by a suit and a number.

In the cards, there are four suits: clubs, diamonds, spades, and hearts. Similarly, there are 8 numbers: seven, eight, nine, ten, jack, queen, king, and ace.

Write a program to:

- Define the types for describing a suit, a number, and a card.
- Construct the deck of 32 cards, where each card is the result of the combination of a suit and a number.
- Display the deck of cards

Solution:

The solution involves the creation of a deck of 32 cards, each represented by a combination of a suit and a number. To accomplish this, enumerations are utilized to define types for suits (`enum Suit`) and numbers (`enum Number`), and the card is declared as a structure (`struct Card`). The deck is implemented as an array of cards (`struct Card D[32]`).

The deck initialization process involves the use of two nested loops, systematically combining each suit with each number to populate the array. Additionally, the program declares arrays of strings for suits and numbers, directly incorporating string representations within the code.

In the final step, the program traverses the deck using a loop to display the entire set of cards. For each card, it prints the corresponding number and suit names.

Here is the program:

```
#include <stdio.h>
int main() {
    //Enumeration for card suits
    enum Suit {
        CLUBS,
        DIAMONDS,
        SPADES,
        HEARTS
    };
    //Enumeration for card numbers
    enum Number {
        SEVEN,
        EIGHT,
        NINE,
        TEN,
        JACK,
        QUEEN,
        KING,
        ACE
    };
    //Structure representing a card
    struct Card {
```

```

        enum Suit suit;
        enum Number number;
    };
    //Create an array to represent the deck of 32 cards
    struct Card D[32];
    //Other variables
    int i;
    enum Suit suit;
    enum Number number;
    //Arrays of strings for suits and numbers
    char suits[][10] = {"Clubs", "Diamonds", "Spades", "Hearts"};
    char    num[][10]= {"Seven", "Eight", "Nine", "Ten", "Jack", "Queen",
"King", "Ace"};
    //Initialize the deck: combine each suit with each number
    i = 0;
    for (suit = CLUBS; suit <= HEARTS; ++suit) {
        for (number = SEVEN; number <= ACE; ++number) {
            D[i].suit = suit;
            D[i].number = number;
            i++;
        }
    }
    // Display the deck of cards
    printf("Deck of Cards:\n");
    for (i = 0; i < 32; ++i) {
        printf("%s of %s\n", num[D[i].number], suits[D[i].suit]);
    }
    return 0;
}

```

4.5) Exercise 5: Point Distance Calculator

Consider a point in space as an object characterized by three values: x , y , and z .

Write a program that allows to:

- Define the most suitable data structure to describe a point.
- Enter two points from the keyboard and calculate the distance between them.

Solution:

The solution involves creating a program that deals with points in space, each characterized by three values: x , y , and z . To represent a point, a structure named `Point` is introduced, with three members (`x`, `y`, and `z`) to store the coordinates of a point in space.

The program prompts the user to input two points from the keyboard. It then calculates and display the distance between these two points in 3D space using the Euclidean distance formula. The Euclidean distance between two points (x_1, y_1, z_1) and (x_2, y_2, z_2) is given by the formula:

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

The program allowing this calculation is as follows:

```
#include <stdio.h>
#include <math.h>
int main() {
    //Structure representing a point in space
    typedef struct {
        float x;
        float y;
        float z;
    }Point;
    //Declare two points and the distance
    Point p1, p2;
    float dis;
    //Prompt the user to enter coordinates for two points
    printf("Enter coordinates for Point 1 (x y z): ");
    scanf("%f %f %f", &p1.x, &p1.y, &p1.z);
    printf("Enter coordinates for Point 2 (x y z): ");
    scanf("%f %f %f", &p2.x, &p2.y, &p2.z);
    //Calculate and display the distance between the 2 points
    dis=sqrt(pow(p2.x-p1.x,2)+pow(p2.y-p1.y,2)+pow(p2.z-p1.z,2));
    printf("Distance between the 2 Points: %.2f\n", dis);
    return 0;
}
```

4.6) Exercise 6: Age Comparison

Write a program that allows entering information about two individuals, then displays which person is older between the two and the age difference. Each person is defined by their name and age.

Solution:

The solution involves the creation of a simple program to compare the ages of two individuals based on user input. The program defines a structure named **Person** to represent each individual, with attributes for the **name** and **age**. It prompts the user to input information for two persons, including their names and ages. Subsequently, it compares the ages and calculates the age difference. Finally, it displays the results, indicating which person is older and providing the age difference.

The program implementing this solution is the following:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    //Define a structure named Person
    typedef struct {
        char name[20];
        int age;
    } Person;
    //Declare variables to store information about two persons
    Person pers1, pers2;
```

```

char p;
//Input information for the first person
printf("Please enter information for the 1st person:\n");
printf("\tName: ");
gets(pers1.name);
printf("\tAge: ");
scanf("%d", &pers1.age);
scanf("%c", &p);
//Input information for the second person
printf("Please enter information for the 2nd person:\n");
printf("\tName: ");
gets(pers2.name);
printf("\tAge: ");
scanf("%d", &pers2.age);
//Compare ages and display the results
if (pers1.age > pers2.age) {
    printf("%s is older.\n", pers1.name);
    printf("The age difference is %d",pers1.age-pers2.age);
} else if (pers2.age > pers1.age) {
    printf("%s is older.\n", pers2.name);
    printf("The age difference is %d",pers2.age-pers1.age);
} else {
    printf("Both persons have the same age.\n");
}
return 0;
}

```

4.7) Exercise 7: Complex Number Operations

A complex number Z is represented by its real part a and imaginary part b in the form $Z = a + ib$.

Develop a program using structures that reads two complex numbers $Z1$ and $Z2$, and calculate and display their modulus, sum, and conjugate, defined as follows:

- The modulus of a complex number Z is defined by: $|Z| = \sqrt{a^2 + b^2}$
- The conjugate of a complex number $Z = a + ib$ is the complex number $Z' = a - ib$.
- The sum of two complex numbers $Z1 = a1 + ib1$ and $Z2 = a2 + ib2$ is the complex number $Z = (a1 + a2) + i(b1 + b2)$.

Solution:

The program presented here focuses on the manipulation of complex numbers, each characterized by its real part a and imaginary part b in the form $Z = a + ib$.

Utilizing a structure named **Complex**, the program prompts the user to input the real and imaginary parts of two complex numbers, **z1** and **z2**. Following this input phase, the program proceeds to execute the prescribed operations using the specified formulas from the exercise.

Subsequent to these computations, the program displays the results to the user.

Here is the program:

```
#include <stdio.h>
#include <math.h>
int main() {
    //Structure representing a complex number
    struct Complex {
        float a;
        float b;
    };
    //Declare complex numbers using structures
    struct Complex Z1, Z2;
    //Enter real and imaginary parts of Z1
    printf("Enter real and imaginary parts of Z1 (a b): ");
    scanf("%f%f", &Z1.a, &Z1.b);
    //Enter real and imaginary parts of Z2
    printf("Enter real and imaginary parts of Z2 (a b): ");
    scanf("%f%f", &Z2.a, &Z2.b);
    //Calculate and display modulus of Z1
    float modZ1 = sqrt(Z1.a * Z1.a + Z1.b * Z1.b);
    printf("Modulus of Z1: %.2f\n", modZ1);
    //Calculate and display conjugate of Z1
    struct Complex conjZ1;
    conjZ1.a = Z1.a;
    conjZ1.b = -Z1.b;
    printf("Conjugate of Z1: %.2f+i%.2f\n", conjZ1.a, conjZ1.b);
    //Calculate and display modulus of Z2
    float modZ2 = sqrt(Z2.a * Z2.a + Z2.b * Z2.b);
    printf("Modulus of Z2: %.2f\n", modZ2);
    //Calculate and display conjugate of Z2
    struct Complex conjZ2;
    conjZ2.a = Z2.a;
    conjZ2.b = -Z2.b;
    printf("Conjugate of Z2: %.2f+i%.2f\n", conjZ2.a, conjZ2.b);
    //Calculate and display the sum of Z1 and Z2
    struct Complex sum;
    sum.a = Z1.a + Z2.a;
    sum.b = Z1.b + Z2.b;
    printf("Sum of Z1 and Z2: %.2f + i %.2f\n", sum.a, sum.b);
    return 0;
}
```

4.8) Exercise 8: Invoice Calculation

Suppose that an invoice is an array of 10 elements. Each element of this invoice is an item with fields `unitPrice` and `quantity`.

Write a program that declares an invoice, reads invoice data from the keyboard, and calculates and displays the total amount of the invoice.

Solution:

This program is designed to handle invoice data, considering an invoice as an array of 10 elements, where each element represents an item with attributes such as **unitPrice** and **quantity**.

The solution begins by defining a structure called **InvoiceItem** to encapsulate the details of each item, incorporating fields for the cost per unit (**unitPrice**) and the quantity of units (**quantity**). Subsequently, an array named **invoice** is declared to represent the entire invoice, consisting of 10 elements of the **InvoiceItem** structure. The program then enters a loop, prompting the user to input specific data for each item in the invoice, namely the unit price and quantity. Following the data input phase, another loop iterates through the items in the invoice, calculating the total amount by multiplying the unit price with the quantity for each item and accumulating the results. Finally, the program displays the calculated total amount of the invoice.

The aforementioned program is as follows:

```
#include <stdio.h>
int main() {
    #define n 3
    //Define the structure for an invoice item
    typedef struct {
        float unitPrice;
        int quantity;
    }InvoiceItem;
    //Declare an array to represent the invoice
    InvoiceItem invoice[n];
    //Other variables
    int i;
    float total;
    // Read invoice data from the keyboard
    for (i = 0; i < n; ++i) {
        printf("Enter details for item %d:\n", i + 1);
        //Input unit price
        printf("Unit Price: ");
        scanf("%f", &invoice[i].unitPrice);
        //Input quantity
        printf("Quantity: ");
        scanf("%d", &invoice[i].quantity);
    }
    //Calculate and display the total amount of the invoice
    total = 0;
    for (int i = 0; i < n; ++i) {
        total= total+invoice[i].unitPrice*invoice[i].quantity;
    }
    printf("Total Amount of the Invoice: %.2f\n", total);
    return 0;
}
```


4.9) Exercise 9: Family Information

Consider the type **Person** as a structure composed of two fields: **lastName** and **firstName**. Also, consider the type **Family**, allowing to describe a real-world family, characterized by the following fields:

- **father**: of type **Person**. Represents the father of the family.
- **mother**: of type **Person**. Represents the mother of the family.
- **nbChildren**: of type Integer. Indicates the number of children in the family.
- **children**: an array of children. Represents the list of children in the family.

Write a program to:

- Define the types **Person** and **Family**.
- Read information about a list of families and then display them.

Solution:

This exercise manages information about families by defining two structure types: **Person** and **Family**. The **Person** structure represents an individual, while the **Family** structure is employed to depict a real-world family.

The accompanying program follows a straightforward structure, commencing with the declaration of an array of **Family** structures to store information about a list of families. After defining the types and declaring variables, the program utilizes a loop to prompt the user for information about each family. For each family, various details, including the last name and first name of each family member, the number of children, and the names of the children, are input. Subsequently, another loop is employed to display the entered family information.

The program is as follows:

```
#include <stdio.h>
#include <string.h>
int main() {
    //Define a constant for the number of families
    #define n 3
    //Define a structure for a person
    typedef struct {
        char lastName[20], firstName[20];
    }Person;
    // Define a structure for a family
    typedef struct {
        Person father, mother;
        int nbChildren;
        Person children[7];
        //Assuming the number of children does not exceed 7
    } Family;
    // Declare an array of Family structures
    Family T[n];
    int i, j;
    // Input information for each family
    for (i = 0; i < n; i++) {
        printf("Enter information for family %d:\n", i + 1);
```

```

//Input father's information
printf("\tFather:\n");
printf("\t\tLast Name: ");
gets(T[i].father.lastName);
printf("\t\tFirst Name: ");
gets(T[i].father.firstName);
//Input mother's information
printf("\tMother:\n");
printf("\t\tLast Name: ");
gets(T[i].mother.lastName);
printf("\t\tFirst Name: ");
gets(T[i].mother.firstName);
//Input the number of children
printf("\tNumber of Children: ");
scanf("%d", &T[i].nbChildren);
getchar(); // Clear the buffer
// Input information for each child
printf("\tList of Children:\n");
for (j = 0; j < T[i].nbChildren; j++) {
    printf("\t\tChild %d: ", j + 1);
    //Copy the last name of the father for the child
    strcpy(T[i].children[j].lastName,T[i].father.lastName);
    gets(T[i].children[j].firstName);
}
}
// Display the entered families
printf("The entered families are:\n");
for (i = 0; i < n; i++) {
    printf("Family %d:\n", i + 1);
    printf("\tFather: ");
    printf("%s ",T[i].father.lastName);
    printf("%s\n",T[i].father.firstName);
    printf("\tMother: ");
    printf("%s ",T[i].mother.lastName);
    printf("%s\n",T[i].mother.firstName);
    printf("\tNumber of Children: %d\n", T[i].nbChildren);
    //Display names of each child
    for (j = 0; j < T[i].nbChildren; j++) {
        printf("\t\t%s\n", T[i].children[j].firstName);
    }
}
return 0;
}

```

4.10) Exercise 10: Car Park Management

Consider a car park consisting of N cars, each represented by its: Registration number, Brand, Maximum speed, and Color.

In order to model the problem, we consider the car park as an array of records, where each record represents a car.

- a) Write a program that allows to enter information for all the cars in the park and then display the registration number and calculate the number of cars registered in Guelma.
- b) Modify the program to calculate and display the number of cars registered by Wilaya (province).

Solution:

The exercise comprises two parts, each involving the creation of a program that facilitates user input for a set of cars in a park and subsequently presents pertinent statistics. In both programs, a shared data structure named `Car` is utilized to encapsulate details about individual cars, and an array of cars named `park` is employed to store all the car details.

- a) The first program concentrates on presenting the number of cars registered in a specific location, in this case, Guelma. To achieve this, an integer variable is employed as a counter to track this information. The program begins with the declaration section, followed by a loop that prompts the user to enter information about all the cars in the park. Subsequently, the counter is initialized to zero, and a second loop is utilized to traverse the array for counting the relevant cars. For each car, the last two digits of its registration number are extracted using the modulo operator (%), representing the wilaya's code. If the code equals 24 (the code for Guelma), the counter is incremented, and the registration number of the car is displayed. After exiting the loop, the program displays the counter contents. Here is the program:

```
#include <stdio.h>
#include <string.h>
//Define constants
int main() {
    #define n 4
    //Define structure for a car
    typedef struct {
        int RegNb;
        char Brand[30];
        char Color[30];
        float MaxSpeed;
    }Car;
    //Define an array of cars (Park)
    Car Park[n];
    //Other variables
    int i, num, nbGuelma;
    //Input information for each car in the park
    for (i = 0; i < n; ++i) {
        printf("Enter information for car %d\n", i + 1);
        printf("Registration Number: ");
        scanf("%d", &Park[i].RegNb);
        printf("Brand: ");
        scanf(" %s", &Park[i].Brand);
        printf("Max Speed: ");
        scanf("%f", &Park[i].MaxSpeed);
    }
}
```

```

        printf("Color: ");
        scanf(" %s",&Park[i].Color);
    }
    //Count and display cars registered in Guelma
    nbGuelma = 0;
    for (i = 0; i < n; ++i) {
        num = Park[i].RegNb % 100;
        if (num == 24) {
            printf("%d\n",Park[i].RegNb);
            nbGuelma=nbGuelma+1;
        }
    }
    // Display the number of cars registered in Guelma
    printf("Number of cars registered Guelma: %d", nbGuelma);
    return 0;
}

```

- b) For the second part of the exercise, which involves computing the number of cars registered in each wilaya, we have modified the previous program by incorporating an array of integers to act as counters for all wilayas. Following the input phase, all elements of the array are initialized to zero using a loop. Subsequently, another loop is employed to traverse the park, extract the wilaya's code, and increment the corresponding element in the counters array. Finally, we display all elements of the counters array.

The modified program is as follows:

```

#include <stdio.h>
#include <string.h>
//Define constants
int main() {
    #define n 4
    #define nbWilayas 58
    //Define structure for a car
    typedef struct {
        int RegNb;
        char Brand[30];
        char Color[30];
        float MaxSpeed;
    }Car;
    //Define an array of cars (Park)
    Car Park[n];
    int W[nbWilayas];
    //Other variables
    int i, num;
    //Input information for each car in the park
    for (i = 0; i < n; ++i) {
        printf("Please enter information for car %d\n", i + 1);
        printf("Registration Number: ");
        scanf("%d",&Park[i].RegNb);
        printf("Brand: ");
    }
}

```

```
        scanf(" %s",&Park[i].Brand);
        printf("Max Speed: ");
        scanf("%f", &Park[i].MaxSpeed);
        printf("Color: ");
        scanf(" %s",&Park[i].Color);
    }
    //Count and display cars registered in Guelma
    for(i=0;i<nbWilayas;i++)
        W[i] = 0;
    for (i = 0; i < n; ++i) {
        num = Park[i].RegNb % 100;
        W[num-1]= W[num-1] + 1;
    }
    //Display the number of cars registered per wilaya
    printf("The number of cars registered per wilaya:\n");
    for(i=0;i<nbWilayas;i++)
        printf("Wilaya %d: %d cars", i+1, W[i]);
    return 0;
}
```

References

- [1] N. Flasque, H. Kassel, F. Lepoivre, B. Velikson, "Exercices et problèmes d'algorithmique", Dunod, 2010.
- [2] C. Delannoy, "Programmer en langage C", Eyrolles, 1996.
- [3] R. Malgouyres, R. Zrour, F. Feschet, "Initiation à l'algorithmique et à la programmation en C - Cours avec 129 exercices corrigés", DUNOD, 2nd edition, 2015.
- [4] M. Amad, "Algorithmique et Structures de Données", Support de Cours et Travaux Dirigés, 1st and 2nd year License, Abderrahmane Mira University of Bejaia, 2016.
- [5] T Slimani, "Programmation et structures de données avancées en langage C: cours et exercices corrigés", Lulu.com Edition, 2014.
- [6] L. Baba Hamed, S, Hocine, "Algorithmique et structure de données statiques: cours et exercices avec solutions", University Publications Office, Algiers, 2006.
- [7] Joyce Farrell, "Programming logic and design: comprehensive version", 8th Edition. Cengage Learning, 2015.
- [8] M. Belaid, "Algorithmique & Programmation en Pascal, Cours, Exercices, Travaux Pratiques, Corrigés", Eurl Pages Bleues Internationales, Bouira - Algeria, 2008.
- [9] B. Bessaa, "Exercices corrigés d'Algorithmique", The LMD Booklets, Blue Pages, Algiers, 2018.
- [10] D. Bouchicha, "Initiation à l'algorithmique et à la programmation en Pascal", Rached Edition, Sidi Bel Abbes - Algeria, 1st edition, 2019.
- [11] T. H. Cormen, "Algorithms Notions de base", DUNOD, 2013.
- [12] J. Tisseau, "Initiation à l'algorithmique", University Press - National School of Engineers Brest, 2009.
- [13] D. Zegour, "Apprendre et enseigner l'algorithmique (Tome 1): Cours et annexes", European University Editions. 2013.