PEOPLE'S DEMOCRATIC REPUBLIC OF ALGERIA

MINISTRY OF HIGHER EDUCATION AND SCIENTIFIC RESEARCH

MOHAMED BOUDIAF UNIVERSITY - M'SILA

1985

جامعة محمد بوضياف - المسيلة

Université Mohamed Boudiaf - M'sila

Faculty of Technology
Common Base (ST)

Course handout

Computer Structure and Applications

Intended for 1st Year Bachelor's and Engineering students ST

Presented by: Dr Mourad GUESRI

Academic year 2025/2026

الجمهـــورية الجزائـــــرية الديمقـــــراطية الشعـــــبية
People's Democratic Republic of Algeria
وزارة التعليم العالي والبحث العلمي
Ministry of Higher Education and Scientific Research
جـــامعة محمد بوضـــياف – بالمســـــيلة
Mohamed BOUDIAF University – M'Sila

جامعة محمد بوضياف - المسيلة
Université Mohamed Boudiaf - M'sila

Faculty of Technology

Vice Deanship of Post-Graduation, Scientific Research and External Relations

جامعة محمد بوضياف - المسيلة
Université Mohamed Boudiaf - M'sila

كـلية التكنـــولـوجيـا

نيابة العمادة لما بعد التدرج والبحث العلمي والعلاقات الخارجية

رقم: 484/ ن.ع.ب.ع /ك.ت /2025

المسيلة في: 16 ديسمبر 2025

شهادة ادارية

المصادقة على تقارير خبرة للموافقة على مطبوعة بيداغوجية

بعد الإطلاع على تقارير لجنة الخبراء للموافقة على المطبوعة البيداغوجية للأستاذ : **قصري مراد – أستاذ محاضر قسم ب**، بالقاعدة المشتركة بكلية التكنولوجيا بجامعة محمد بوضياف بالمسيلة والتي كانت كلها ايجابية ، تمّ تقرير التّالي:

1–المصادقة على تقارير لجنة الخبراء للموافقة المطبوعة البيداغوجية والمعنونة بـــ:

Computer Structure and applications
Common Base ST– Cycle License

2– حيث تمّ تشكيل هذه اللجنة بناء على اجتماع المجلس العلمي للكلية المنعقد بتاريخ **2025/11/30** المكونة من السّادة الآتية أسماؤهم:

– **بلوطي عادل**، أستاذ محاضر "أ"، جامعة محمد بوضياف– المسيلة

– **بوجلال بلال**، أستاذ محاضر "أ"، جامعة محمد بوضياف– المسيلة

– **بوزرية حسين**، أستاذ محاضر "أ"، جامعة سعد دحلب – البليدة

وتمت الموافقة بالاجماع على هذه المطبوعة.

رئيس المجلس العلمي للكلية

جامعة محمد بوضياف - المسيلة
المجلس العلمي
كلية التكنولوجيا

د. علي خريوش

Table of Contents

LIST OF FIGURES

Page

LIST OF TABLES

LIST OF ABBREVIATIONS

CPU: Central Processing Unit

ALU: Arithmetic Logic Unit

GPU: Graphics Processing Unit

ADSL: Asymmetric Digital Subscriber Line

OS: Operating System

IDE: Integrated Development Environment

I/O: Input/Output

RTOS: Real-Time Operating System

ASCII: American Standard Code for Information Interchange

Bool: Boolean data type

GCC: GNU Compiler Collection

MSVC: Microsoft Visual C++

TCC: Tiny C Compiler

Introduction

Computer Structure and Applications is a foundational course that provides first-year technical science students with essential knowledge in computer science and programming. This course aims to establish a rigorous conceptual framework that enables students to understand the fundamental principles governing computer systems and their practical use. It begins with an introduction to computer science, its definition, and its historical evolution, highlighting the major technological advances that have shaped modern computing.

The course presents the basic principles of information representation and coding systems used for data processing. It also examines the general operating principles of a computer, emphasizing the interaction between hardware and software components. The material aspects of a computer system, including input, output, processing, and storage units, are described in a structured and systematic manner. In parallel, students are introduced to system software, particularly operating systems, as well as to application software commonly used in scientific and technical contexts.

A significant part of the course is devoted to algorithmic thinking and structured programming. Students are guided through the process of problem analysis and the design of algorithmic solutions using formal representations such as flowcharts. The structure of a program is studied in detail, along with fundamental concepts related to data types, variables, constants, and operators. Input and output operations are introduced as essential mechanisms for user–program interaction.

Furthermore, the course explores control structures, including conditional and iterative constructs, which form the basis of program logic. Practical laboratory sessions complement the theoretical content by allowing students to apply the acquired knowledge in real programming environments. Through a progressive and practice-oriented approach, this course enables students to develop analytical thinking and fundamental programming skills, thereby providing a solid foundation for more advanced studies in computer science and related disciplines.

I. Introduction to computer science

Welcome to the first chapter of our course, where we will explore what computer science is and why it matters. This chapter lays the groundwork for your understanding of how computers have evolved and how they operate today. We will begin by examining the **definition of computer science** and establishing a context for this discipline.

Next, we will look at the **historical evolution of computers**. This fascinating history spans from simple mechanical calculators to the powerful modern machines we use today. By understanding where computers come from, you can better appreciate the innovations that led to current technology.

We will also discuss **information coding systems** – the ways information is represented inside a computer. These include familiar systems like decimal (base 10), as well as binary (base 2), octal (base 8), and hexadecimal (base 16) which are widely used in computing. You will learn how to convert numbers between these systems and why such conversions are useful.

Additionally, we'll introduce **how a computer works** by breaking down the basic operations a computer performs. You'll learn the roles of key hardware components and how data moves through a computer system.

By the end of this chapter, you should have a clear understanding of what computer science encompasses, a sense of how computers developed, and a grasp of the fundamental concepts that allow computers to process information.

I.1 Definition of computer science

Computer science is the systematic study of computing systems and computation. It is not just about computers themselves, but more broadly about **information and algorithms** – how information is represented, processed, and transformed. Computer science covers both theoretical concepts (like complexity, data structures, algorithms) and practical techniques for implementing computations in hardware and software.

In simpler terms, computer science answers questions such as: What can computers do? How do we tell computers what to do? What happens inside a computer when it performs a task? It involves learning programming languages to instruct machines, but also understanding the principles that make those instructions effective and efficient.

Computer science differs from mere computer usage; it's not just how to use software, but how to **create** software and reason about computations. This field underpins the software

and systems that we use in daily life, from operating systems to mobile apps to the algorithms powering search engines and artificial intelligence.

I.2 Evolution of computer science and computers

The evolution of computing and computers is a fascinating story of progress and innovation. It spans centuries – from early ideas of computation to the modern digital age – and has radically transformed society. Some key milestones in this evolution include:

- **17th century – Mechanical Calculators:** Early mechanical calculating devices like **Blaise Pascal's Pascaline** (1642) and **Gottfried Wilhelm Leibniz's** stepped reckoner were designed to perform basic arithmetic operations, laying the foundations for later computational devices.

***Figure I.1** : A Pascaline, the mechanical calculator invented by Blaise Pascal in the 17th century, on display at the Musée des Arts et Métiers in Paris. This device could perform additions and subtractions and is one of the first examples of a computing machine.*

- **19th century – The Analytical Engine:** Visionaries like Charles Babbage conceived programmable machines. Babbage's Analytical Engine (1830s), although never completed, had concepts like a stored program and is often regarded as a precursor to modern computers. Ada Lovelace, working with Babbage, wrote what is considered the first computer algorithm for this engine.
- **Early 20th century – Electromechanical and Electronic Computers:** The first half of the 20th century saw the development of electromechanical computers and, subsequently,

fully electronic computers. **ENIAC** (1945) was one of the first electronic general-purpose computers, occupying an entire room and using thousands of vacuum tubes to perform calculations unimaginable by humans alone at the time.

- **1950s – Transistors and the Second Generation:** The invention of the transistor (1947) led to the second generation of computers in the 1950s. Transistors replaced vacuum tubes, making computers smaller, faster, more reliable, and more energy-efficient.

- **1960s – Integrated Circuits and the Third Generation:** The development of integrated circuits (microchips) in the 1960s brought about the third generation of computers. Multiple transistors and electronic components could be placed on a single silicon chip, dramatically increasing computing power and reducing cost and size.

- **1970s – Microprocessors and Personal Computers:** The 1970s introduced the first microprocessors – entire CPUs on a single chip. This led to the birth of **personal computers (PCs)**. Computers like the Apple II and Commodore PET (late 1970s) brought computing into homes and schools for the first time.

- **1980s – Graphical Interfaces and PC Revolution:** In the 1980s, IBM PCs and compatibles running MS-DOS, and later the introduction of graphical user interfaces (GUIs) like those in Apple's Macintosh (1984) and Microsoft Windows (1985), made computers more user-friendly and widespread. This decade saw explosive growth in personal and business computing.

- **1990s – The Internet Era:** The rise of the Internet in the mid-1990s revolutionized computing once again. Computers became interconnected, enabling email, the World Wide Web, and e-commerce. Computing devices continued to shrink in size and grow in power, with laptops becoming common.

- **2000s and 2010s – Mobile and Ubiquitous Computing:** The new millennium brought smartphones and tablets – powerful pocket-sized computers. Computing became truly ubiquitous, embedded in many everyday devices (IoT – Internet of Things). The 2010s also saw rapid advances in cloud computing, allowing massive data storage and processing over the Internet, and developments in artificial intelligence.

- **Today – Ongoing Innovation:** Computing continues to evolve with advancements in areas like quantum computing, machine learning, and wearable technology. Computers today range from tiny microcontrollers in appliances to giant cloud data centers powering internet services. The history of computing shows a trend towards faster, smaller, and more numerous computing devices, increasingly integrated into every aspect of life.

Understanding this evolution helps appreciate how far technology has come and provides context for why modern computers are designed the way they are.

I.3 Information coding systems

Information coding systems are methods used to represent data in the form of numbers or symbols that computers can process. Since computers operate using electrical signals (which have two states like on/off), **binary** is the fundamental language of computers. However, for human understanding and various applications, we use several numbering systems:

1. **Decimal system (base 10):** This is the everyday numbering system we use, employing ten symbols (0 through 9). Each digit's value depends on its position (powers of 10). For example, in the decimal number 365, the digits represent 3×10^2 + 6×10^1 + 5×10^0.

2. **Binary system (base 2):** Used internally by computers, it has two symbols: 0 and 1. Each binary digit (bit) represents a power of 2. For example, the binary number 1011 represents 1×2^3 + 0×2^2 + 1×2^1 + 1×2^0, which equals 8 + 0 + 2 + 1 = 11 in decimal.

3. **Octal system (base 8):** Uses eight symbols (0–7). Each position is a power of 8. Octal was used in some early computing contexts (like old minicomputers) because it maps neatly to binary (each octal digit represents 3 binary bits).

4. **Hexadecimal system (base 16):** Uses sixteen symbols (0–9 and A–F, where A=10, B=11, ..., F=15). Each hex digit represents a power of 16. Hex is very common in computing because it maps cleanly to binary as well (each hex digit represents 4 binary bits). It's used to represent memory addresses, color codes in web design, and other cases where a compact representation of binary data is useful. For example, the hexadecimal number 2F (which is 2×16^1 + 15×16^0) equals 47 in decimal, and corresponds to binary 0010 1111.

Conversion examples: Converting between these systems is a useful skill. For instance, to convert decimal 15 to binary, you find it equals binary 1111. To convert that to hexadecimal, you group binary bits into fours: 1111_2 is F in hex. In fact, here's a small table of numbers 1 to 16 in different systems:

Table I.1 *: example of conversion table between coding systems*

Decimal	Binary	Octal	Hexadecimal
1	1	1	1
2	10	2	2
3	11	3	3

4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10

Looking at such a table, it becomes clear how the same quantity is expressed differently in each system, even though its value remains the same. This flexibility in representation is fundamental to how computers handle data.

Complement systems: Complement systems are techniques used to represent negative numbers in binary and to simplify subtraction operations. Two common ones are:

One's Complement:

- One's complement is obtained by **flipping all the bits** in a binary number. In other words:
  - Change **0 to 1**, and
  - Change **1 to 0**.

For example:

- On 4 bits, the binary number **0101** (which is +5) becomes **1010** when its one's complement is taken.

Important Drawback:

- One's complement has **two representations for zero**:
  - **0000** (positive zero)
  - **1111** (negative zero)

This dual representation of zero is one of the reasons why **one's complement** is less commonly used today.

Two's Complement:

- Two's complement is the most widely used system for representing signed integers in modern computers.
- It is obtained by:
    1. Taking the **one's complement** (inverting all bits).
    2. Adding **1** to the result.

The **two's complement** representation avoids the issue of having two zeros, as it only has **one zero representation**. It is also much more convenient for **binary arithmetic** since both positive and negative numbers can be added without special rules.

Example:

- The binary number **0101** (which is +5) becomes **1011** after performing the two's complement operation.

Steps for Two's Complement:

1. Start with **+5**, which in binary is **0101**.
2. Take the one's complement: **1010**.
3. Add 1 to the result: **1011**.

Thus, **1011** represents **-5** in two's complement (4-bit system).

MSB (Most Significant Bit) and Its Role in Signed Binary Representations:

The **MSB** (Most Significant Bit) is the **leftmost bit** in a binary number. It represents the largest place value (the highest bit) in the number. In signed binary and complement systems, the MSB has special significance because it indicates the **sign** of the number.

In Signed Binary Representation (Using MSB for Sign):

- **MSB = 0**: The number is **positive**.
- **MSB = 1**: The number is **negative**.

For example:

- **+5** in 4-bit signed binary: **0101**
    - The MSB is **0**, so it's a positive number.
- **-5** in 4-bit signed binary: **1011**
    - The MSB is **1**, so it's a negative number.

In One's Complement and Two's Complement:

- The **MSB** (Most Significant Bit) is **used to indicate whether the number is positive or negative**:
    - **0** means **positive**.
    - **1** means **negative**.

One's Complement:

- To represent **-5** in one's complement (4-bit):
    1. **5** in binary is **0101**.
    2. The one's complement of **0101** is **1010**.
    3. **1010** represents **-5** in one's complement (4-bit).

MSB = 1 means it's a negative number.

Two's Complement:

- To represent **-5** in two's complement (4-bit):
    1. **5** in binary is **0101**.
    2. The one's complement of **0101** is **1010**.
    3. Add 1 to **1010**: **1011**.
    4. **1011** represents **-5** in two's complement (4-bit).

Again, **MSB = 1** means it's a negative number.

I.4 Basic Operating Principles of a Computer

The functioning of a computer relies on a complex combination of hardware and software. The Central Processing Unit (CPU) processes data through millions of transistors organized into logical circuits. The user interacts with the computer via input devices, and the Operating System (OS) facilitates this interaction. The execution of a program involves loading, interpreting, and executing instructions by the CPU, guided by the machine cycle. Information transfers between components occur through communication buses. Technological advances have made computers faster, more powerful, and more versatile.

I.5 Hardware Components of a Computer

The hardware part of a computer forms the physical infrastructure that enables the machine to perform complex tasks. It encompasses a variety of interconnected components working together to process, store, and manipulate information. Understanding these elements provides essential insight into the internal workings of a modern computer.

1. Central Processing Unit (CPU)

Also known as the processor, the CPU is the nerve center of any computer. It is where program instructions are executed, arithmetic operations are performed, and overall system coordination is managed. The power of a CPU is often measured in gigahertz (GHz), indicating the number of clock cycles it can complete each second.

***Figure I.2** : The processor is the central processing unit (CPU).*

2. Random Access Memory (RAM)

Random Access Memory (RAM) is another key component of the system, serving as temporary storage for data currently used by the operating system and running programs. The greater the amount of RAM, the more efficiently the computer can handle multiple complex tasks simultaneously without sacrificing performance.

Figure I.3 *: Two DDR memory modules, each with a capacity of 512 Mb*

3. Storage

Storage, in the form of hard drives, SSDs, or other technologies, ensures the permanent preservation of data. Hard drives are known for their large capacity, while SSDs offer exceptional read/write speeds. Together, these form the storage memory that retains information even when the computer is turned off.

Figure I.4 *: Storage devices: Solid-state drive (SSD) and Hard disk drive (HDD)*

4. Motherboard

The motherboard is the printed circuit board on which most components are connected. It acts as a central hub, allowing communication between the processor, memory, expansion cards, and other peripherals. It also contains the BIOS, a set of low-level routines essential for booting up the computer.

Figure I.5 *: Motherboard*

5. Graphics Cards

Graphics cards, or GPUs (Graphics Processing Units), are dedicated to processing graphics and images. Essential for graphically intensive tasks such as gaming or computer-aided design, modern graphics cards deliver outstanding performance and support advanced technologies such as ray tracing.

Figure I.6 *: Graphics Processing Unit (GPU)*

6. Input/Output Devices

Input/output devices connect the computer to the outside world. Keyboards and mice are typical input devices, while monitors, printers, and speakers are examples of output devices. USB, HDMI, and other interfaces enable connection to a wide range of external equipment.

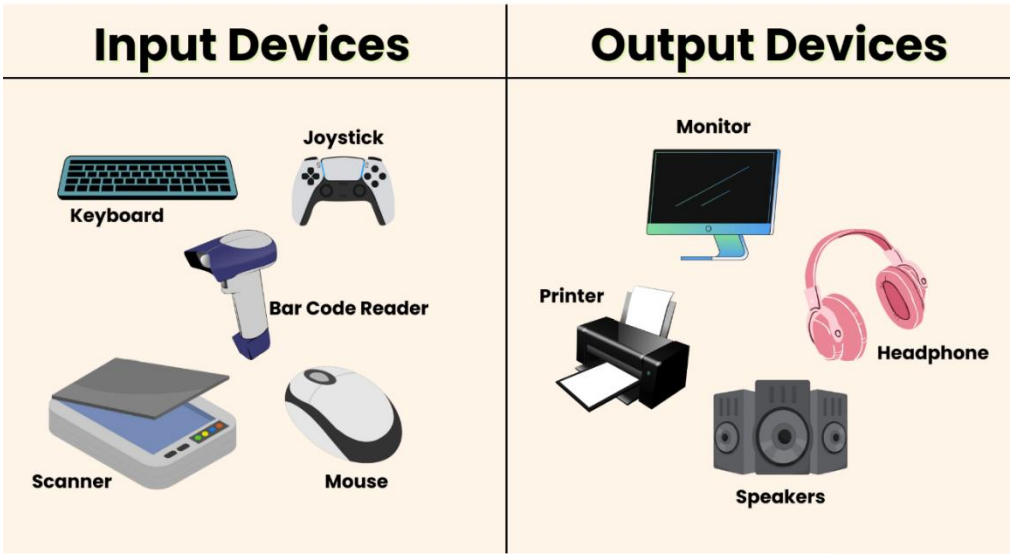

Figure I.7 *: Examples of input/output devices*

7. Communication Components

Communication components, such as network cards, facilitate connectivity with other computers or the Internet. The emergence of wireless networks has led to technologies such as Wi-Fi, providing more flexible and extensive connectivity.

Figure I.8 *: Dual Band Wireless ADSL2+ Modem Router*

8. Power Supply

The power supply provides the necessary energy for all components to operate. Power supply units convert electricity from the wall outlet into appropriate voltages for each part of the computer.

I.6 System Components

The **system part** of a computer is a collection of essential software components that facilitate its overall operation. It consists of two main elements: **operating systems** and **programming languages with application software**. These components are the key to a seamless interaction between humans and machines, enabling users to make the most of their computer hardware.

I.6.1 Basic Systems

Operating Systems

Operating Systems (OS) are the foundation of the system part. They act as an interface between the computer's hardware and software applications. The most common OS include **Windows**, **Linux**, and **macOS**. Each operating system provides a unique user experience while ensuring essential functions such as resource management, file management, security, and communication with hardware.

- **Windows**, developed by Microsoft, is ubiquitous in the world of personal computers. It is known for its user-friendly interface and broad hardware and software compatibility.
- **Linux**, based on an open-source kernel, is widely used in server and development environments. Known for its stability, security, and scalability, Linux exists in several distributions such as **Ubuntu**, **Fedora**, and **CentOS**.
- **macOS**, Apple's operating system, is exclusive to Mac computers. It is distinguished by its elegant design, ease of use, and seamless integration with other Apple products.
- In addition to these popular systems, there are other specialized OS, such as embedded systems found in electronic devices, **real-time operating systems (RTOS)** used in critical systems, and various **Unix** distributions.

Programming Languages

Programming languages are tools that allow developers to create software and interact with computers. They provide syntax and rules for writing programs, which are then translated into machine language understandable by the computer.

- Programming languages such as **C**, **C++**, and **Java** are used to develop system software, desktop applications, and embedded systems. These languages provide fine-grained control over hardware while maintaining portability across different platforms.
- **C language**, created in the early 1970s by **Dennis Ritchie**, is a structured programming language known for its efficiency and versatility. It has been widely used in education, system development, and the creation of operating systems themselves (including Unix).
- **Python**, a high-level programming language, has become popular for its simplicity and readability. It is commonly used in web development, data processing, and artificial intelligence.
- **Scripting languages** such as **JavaScript** and **PHP** are widely used in web development, enabling the creation of interactive user interfaces and dynamic websites.
- Specialized languages like **R** and **MATLAB** are used in data analysis, scientific computing, and statistics.

I.6.2 Programming Languages and Application Software

Application software forms the upper layer of the system part. These are specific programs designed to meet particular user needs. Applications cover a wide range of domains, from productivity and entertainment to specialized professional tasks.

- **Office suites** such as **Microsoft Office** and **LibreOffice** include applications like Word, Excel, and PowerPoint for creating documents, spreadsheets, and presentations.
- **Web browsers** such as **Google Chrome**, **Mozilla Firefox**, and **Safari** allow users to access the Internet and explore online content.
- **Graphic design software** such as **Adobe Photoshop** and **Photo Express** provide advanced tools for creating and editing images.
- **Integrated Development Environments (IDEs)** such as **Visual Studio** and **Eclipse** facilitate software creation, debugging, and deployment for developers.
- **Security software**, including antivirus programs and firewalls, is essential to protect computers from online threats and malware.
- **Entertainment applications**, ranging from video games to streaming platforms, represent a major segment of application software, offering diverse leisure experiences.

- Exercises

Exercise 1

Convert the decimal number $(17)_{10}$ to binary, octal and hexadecimal.

Solution:

1. **To Binary:**
   Divide 17 by 2 and record the remainders:
   - 17 ÷ 2 = 8, remainder 1
   - 8 ÷ 2 = 4, remainder 0
   - 4 ÷ 2 = 2, remainder 0
   - 2 ÷ 2 = 1, remainder 0
   - 1 ÷ 2 = 0, remainder 1

   So, 17 in binary is **10001**.

2. **To Octal:**
   To convert **17** to octal, divide by 8:

   - 17 ÷ 8 = 2 remainder 1
   - 2 ÷ 8 = 0 remainder 2

   Reading the remainders from bottom to top, we get **21**.

   So, 17 in octal is **21**.

3. **To Hexadecimal:**
   To convert **17** to hexadecimal, divide by 16:

   - 17 ÷ 16 = 1 remainder 1
   - 1 ÷ 16 = 0 remainder 1

   Reading the remainders from bottom to top, we get **11**.

   So, $\mathbf{(17)_{10} = (11)_{16}}$.

Exercise 2

Convert the binary number $(111011)_2$ to octal and hexadecimal.

Solution:

1. **To Octal:**
   Group the binary number into groups of 3:
   - 111011 → 111 011
   - Now, convert each group to octal:
     - 111 = 7
     - 011 = 3

   So, 111011 in octal is **73**.

2. **To Hexadecimal:**
   Group the binary number into groups of 4:
   - 111011 → Add leading zeros to the last group: 0011 1011
   - Now, convert each group to hexadecimal:
     - 1011= B
     - 0011 = 3

   So, 111011 in hexadecimal is **B3**.

Exercise 3

Convert the numbers $(100011)_2$, $(70)_8$ and $(F1)_{16}$ into decimal.

Solution:

1. **Binary $(100011)_2$ to decimal:**
   - $(100011)_2 = (1 \times 2^5) + (0 \times 2^4) + (0 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)$
   - $= 32 + 0 + 0 + 0 + 2 + 1$
   - = **35** (decimal)
2. **Octal $(70)_8$ to decimal:**
   - $(70)_8 = (7 \times 8^1) + (0 \times 8^0)$
   - $= 56 + 0$
   - = **56** (decimal)
3. **Hexadecimal $(F1)_{16}$ to decimal:**
   - $(F1)_{16} = (F \times 16^1) + (1 \times 16^0)$

- F = 15 in decimal
- = (15 × 16) + (1 × 1)
- = 240 + 1
- = **241** (decimal)

Exercise 4

On a 6-bit system, convert the number $(-7)_{10}$ into: signed binary and two's complement

Solution:

- **In Signed Binary (6-bit)**:

- $(7)_2$ = 000111 (binary).
- $(-7)_2$ = **1**00111 (MSB is **1**, indicating negative).

- **In Two's Complement (6-bit)**:

- Convert **7** to binary: **000111**.
- Invert the bits: **111000**.

II. Concepts of algorithms and programs

In this second part of our course, we will delve into the fundamentals of programming and algorithmic problem solving. This chapter is an essential step in understanding the basic concepts necessary for creating functional and efficient computer programs.

We will begin our exploration by defining the concept of an **algorithm**, the cornerstone of all computer programming. By understanding the ins and outs of algorithmic design, you will be equipped to approach computing problems in a systematic and logical manner.

To visually represent our algorithms, we will study the use of **flowcharts** – diagrams that clearly and concisely illustrate the flow of instruction execution. This graphical representation will facilitate the understanding and communication of algorithms between programmers.

As we continue, we will analyze the **structure of a computer program**, examining its essential components. Understanding this structure is crucial for writing well-organized and maintainable programs.

Once we have established the basics of programming, we will address the **approach and analysis of a problem** – a preliminary step that is essential to creating an effective algorithm. We will also explore data structures in detail, discussing constants, variables, and the different types of data used in programming.

Next, we will dive into the world of **operators**, which are essential tools for manipulating and processing data in a program. We will review assignment, relational, logical, and arithmetic operators, as well as the rules of precedence that govern their use.

Finally, we will study **input/output operations** and **control structures**, such as conditional and repetitive structures, which allow you to control the flow of a program's execution based on conditions and to repeat certain actions as needed.

This chapter is a crucial step in your programming learning journey, laying the necessary foundations for creating functional and efficient programs.

II.1 Concept of an algorithm

The concept of an **algorithm** is at the very heart of computer science and programming. An algorithm is a sequence of well-defined, ordered instructions designed to solve a specific problem or accomplish a particular task. It is essentially a *recipe* or a *step-by-step plan* for how to achieve a given result from a set of inputs. An algorithm must be **clear**, **precise**, and **finite** – it should have unambiguous steps and eventually terminate after a limited number of steps.

To illustrate, consider an everyday example: a recipe for baking a cake is like an algorithm. It takes certain ingredients (inputs), and through a series of steps (the procedure), produces a cake (output). Each step in the recipe must be clear and doable, and the recipe has a definite end.

Important properties of algorithms include:

- **Correctness:** The algorithm should produce the correct output for all valid inputs.
- **Efficiency:** The algorithm should make optimal use of resources – typically time (speed of execution) and space (memory usage). We often discuss efficiency in terms of time complexity (roughly, how the running time grows with input size) and space complexity.
- **Clarity:** It should be easy to understand (at least for someone versed in the context). Clarity helps in verifying correctness and in modifying the algorithm later if needed.

Algorithms can be expressed in many forms: in natural language, pseudocode, flowcharts, or programming languages. During the design phase, we often use **pseudocode** (an informal, high-level description of the algorithm) or flowcharts before writing actual code.

There are many categories of algorithms, such as:
- **Sorting algorithms** (e.g., bubble sort, quicksort) – for ordering data.
- **Searching algorithms** (e.g., binary search) – for finding elements in data.
- **Graph algorithms** (e.g., finding shortest paths) – for working with networks of nodes.
- **Mathematical algorithms** (e.g., algorithms for prime numbers, encryption, etc.).

The choice of algorithm can greatly affect the performance of software. Hence, understanding and comparing algorithms is a big part of computer science.

For example, here is a simple algorithm (in pseudocode) for calculating the sum of two numbers:

Example:

```
Display "Enter the first number: "
Read a
Display "Enter the second number: "
Read b
sum ← a + b
Display "The sum of ", a, " and ", b, " is: ", sum
End
```

This pseudocode outlines the steps: prompt the user for two numbers (a and b), calculate their sum, and then display the result. It's clear, finite, and if followed, it will correctly compute the desired result.

Complement: In programming, once an algorithm is designed, the next step is to translate that algorithm into a specific programming language so that it can be executed by a computer. The programming language provides the syntax and structures necessary to express the algorithm in a form the machine understands (after compiling or interpreting). A good grasp of algorithms and their properties will help you write programs that are not only correct but also efficient and maintainable.

II.2 Flowchart representation

A **flowchart** is a diagrammatic method of representing algorithms or processes. It uses various symbols to denote different types of steps and arrows to show the flow of execution order. Flowcharts are helpful because they provide a visual overview of the algorithm, which can make understanding the logic easier, especially for complex processes.

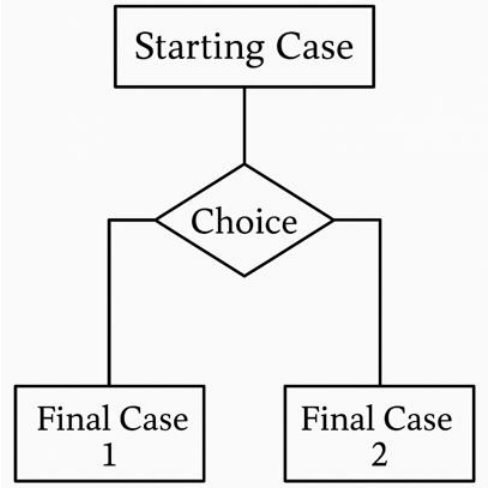

Figure II.1 *: Programming organisation flowchart*

1. Principles of Flowchart Representation

- **Start/End:** Typically represented by an oval or pill-shaped symbol. Every flowchart begins with a Start (where the algorithm begins) and ends with an End (where the algorithm terminates).
- **Process :** Represented by a **rectangle**. A process symbol indicates an operation or action step (e.g., a calculation like `sum = a + b`, or an instruction like “initialize counter = 0”).
- **Input/Output:** Represented by a **parallelogram**. This symbol is used whenever data is input (e.g., reading a value from the user) or output (displaying a result).

- **Decision :** Represented by a **diamond shape**. A decision symbol poses a question (a condition check). Based on the answer (Yes/No or True/False), the flowchart branches into different paths. Each outgoing arrow from a decision is typically labeled with the outcome of the decision (e.g., one arrow for "Yes" and another for "No").

- **Flow lines and arrows :** These connect the symbols and indicate the direction of the algorithm's flow (the order in which steps are executed). Arrows guide the reader from the Start through various operations and decisions to the End.

- **Connector :** Often a small **circle** or a label, used if the flowchart jumps to another point (to avoid crossing lines or if the diagram is split into parts).

2. Example of a Flowchart

Suppose we want to represent a simple decision process – checking if a number is positive or negative – in a flowchart:

- Start (begin the algorithm)
- Input a number N (ask the user to provide a value for N)
- Decision: Is N >= 0? (the diamond decision node)
- If **Yes** (true): output "Number is positive."
- If **No** (false): output "Number is negative."
- End (terminate the algorithm)

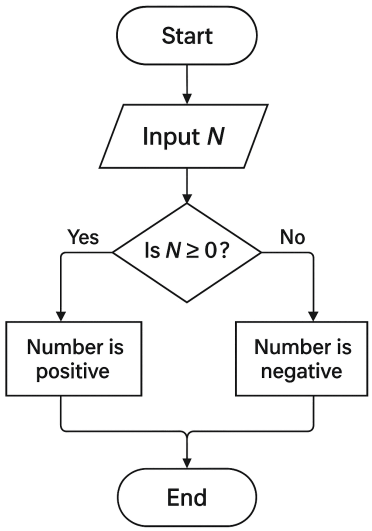

***Figure II.2** : A flowchart that checks whether the number entered by the user is positive or negative.*

3. Advantages of Flowchart Representation

- **Clarity:** Flowcharts provide a clear and intuitive visualization of an algorithm's flow. This can make it easier to spot logical errors or understand the overall structure at a glance.
- **Communication:** A flowchart can serve as a common reference for team members. It's often easier to communicate complex logic with a diagram rather than through text or code, especially when collaborating with those who may not be familiar with the programming language syntax.
- **Planning:** Designing a flowchart encourages *structured planning before coding*. By laying out the logic visually first, programmers can think through the steps required without getting bogged down in code details. This often leads to cleaner, more organized code later.

4. Limitations of Flowchart Representation

- **Complexity for Large Algorithms:** For very complex algorithms or systems, flowcharts can become huge and difficult to follow on paper. A very large flowchart may be more cumbersome than helpful, as it might not fit on a single page or screen and could become overly complicated.
- **Maintenance:** Updating flowcharts can be time-consuming. If an algorithm changes, the flowchart needs to be redrawn or modified. In fast-paced development, some people skip flowcharts because maintaining the diagrams can lag behind code changes.
- **Not Executable:** Unlike code, a flowchart can't be executed by a computer. It's a planning tool. One must still translate the flowchart into actual code. This extra step is necessary, though usually worthwhile for understanding.

Flowcharts are best used for planning and explanation. In practice, small to medium-sized problems can benefit greatly from flowchart visualization, whereas extremely large or dynamic systems might use other design tools or may jump from pseudocode directly to coding.

Here is a simple diagram that calculates the sum of two numbers.

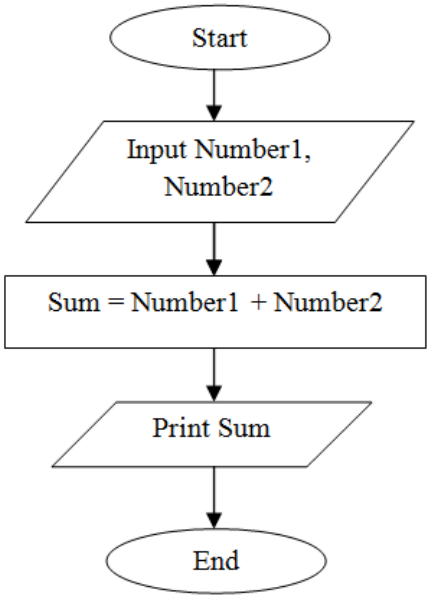

***Figure II.3** : A flowchart that calculates the sum of two numbers*

II.3 Structure of a program

1. Introduction to the C language

The **C language** is a structured, general-purpose programming language that was created in the early 1970s by Dennis Ritchie at Bell Labs. C was originally developed to rewrite the Unix operating system and has since become one of the most influential programming languages in history. It is renowned for its efficiency, low-level hardware access, and powerful capabilities, while still providing high-level abstractions.

Key characteristics of C include:

- **Procedural Paradigm:** C is procedural, meaning it focuses on procedures or routines (functions) to operate on data. It is not object-oriented (like Java or Python), though it influenced those languages.
- **Structured Programming:** C encourages structured programming (using loops, conditionals, functions) which leads to clear and maintainable code.

- **Static Typing:** Every variable in C has a type (int, float, char, etc.) that is determined at compile-time. This helps catch certain errors early.

- **Manual Memory Management:** C provides direct access to memory through pointers. This allows for powerful operations and efficient programs, but also means the programmer must manage memory (allocate/free) carefully to avoid errors.

- **Portability:** C code can be compiled on many types of machines with minimal changes. This made C popular for system/software development across different platforms. Many operating systems (like various Unix flavors, Windows kernel parts, etc.) and performance-critical applications are written in C.

- **Influence:** C has influenced countless other languages. C++ was developed as an extension of C. Languages like C#, Java, and many others inherited C's syntax style (curly braces, control structures). Learning C provides a strong foundation that makes it easier to learn these other languages.

Despite its age, C remains extremely relevant. It is widely used in systems programming (like operating systems, embedded systems, device drivers), game development, and anywhere performance is critical. In academic settings, C is often taught to provide insight into how computers manage memory and process data at a low level. By learning C, beginners gain a deeper understanding of how software interacts with hardware.

The different compilers used for the C language: Unlike some languages which have a single official implementation, C has many compilers. A *compiler* is a tool that translates C source code (human-readable) into executable machine code (binary). Popular C compilers include:

- **GCC (GNU Compiler Collection):** A widely-used open-source compiler available on Linux, Windows (via MinGW or Cygwin), and Mac.

- **Microsoft Visual C++ (MSVC):** Part of Microsoft's Visual Studio, commonly used on Windows for C/C++ development.

- **Clang/LLVM:** A modern compiler that aims to be highly modular and produce useful error messages, often used as an alternative to GCC.

- **Tiny C Compiler (TCC):** A small, fast compiler useful for quick compilation tasks.

- **Borland/Embarcadero C++ Builder:** Historically used on Windows (Borland Turbo C was popular decades ago in educational settings).

For this course, we will use the **Dev-C++ 4.9.9.2** integrated development environment (IDE). Dev-C++ is an IDE that comes packaged with the MinGW version of GCC for Windows. It provides a user-friendly interface to write C (and C++) programs, compile them, and run them. Using an IDE like Dev-C++ can simplify the process of writing code, as it often includes features like syntax highlighting, auto-indentation, and debugging tools.

Here is a useful website for learning C programming from A to Z: **learn-c.org** – an interactive tutorial that covers C basics step by step.

Here is also a comprehensive video that explains the C language step by step: A free full **C Programming Tutorial for Beginners** is available on YouTube (for example, the *freeCodeCamp.org* channel offers a multi-hour C programming course).

2. Structure of a C program

A C program consists of several parts, each with a specific role in the program's development and execution process. Here are the main elements of a C program:

- **Documentation/Comment Header (Program Header):** While not required, it is good practice for a C program to start with a comment block that contains important information about the program, such as its name, purpose, author, and date of creation. This is analogous to the "program header" in Pascal (which used the `program` keyword to name a program). In C, there is no special keyword for the program name; instead, we use comments for human-readable documentation.

Example :

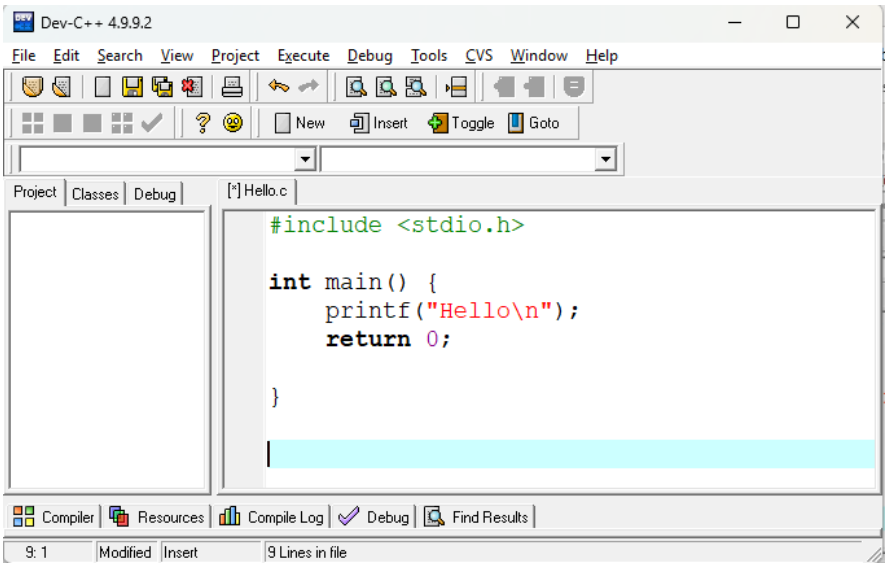

***Figure II.4** : C program that displays the message Hello*

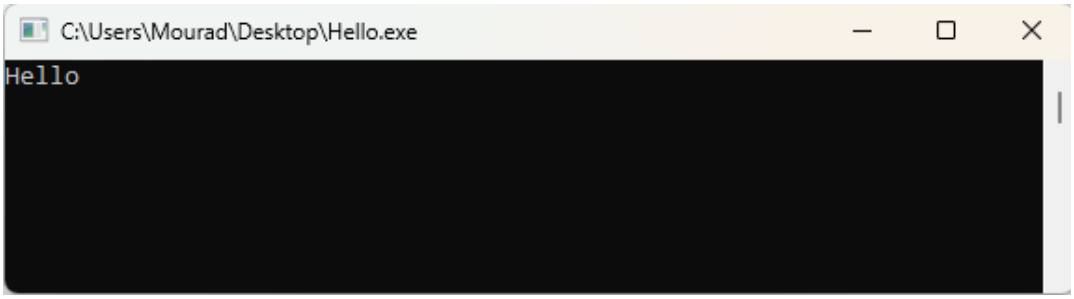

Figure II.5 *:* The output display

- **Preprocessor Directives and Global Declarations:** These are usually placed at the top of the source file, after any initial comments. Preprocessor directives (lines beginning with #) are instructions to the compiler's preprocessor. Commonly, you will see `#include` directives here to include necessary header files (libraries). You might also see `#define` directives to define constants or macros. Additionally, if you have global constants or global variables, they would be declared here (outside of any function).

Example (Includes and Declarations in C):

```
#include <stdio.h>     // Include standard Input/Output library
#include <stdbool.h>   // Include Boolean type (for using bool)

#define PI 3.14159     // Define a constant macro for pi
// Global variable example (generally to be avoided unless needed):
int globalCount = 0;
```

In the above example, we included the stdio library (needed for printf/scanf) and stdbool (which allows use of `bool` for Boolean values). We also defined a constant `PI`. In C, you can define constants using `#define` or by using the `const` keyword (e.g., `const double PI = 3.14159;` at the global scope). We try to keep global variables to a minimum for better program structure, but it's shown for completeness.

- **`main` Function (Program Body):** Every C program must have a `main` function. This is the starting point of execution for the program. When you run the compiled program, the operating system calls the `main` function to begin. The `main` function is typically of type `int` and returns an integer value (by convention, returning 0 signifies that the program executed successfully). Inside `main`, we write the code that performs the tasks we want, possibly calling other functions we wrote.

The structure of `main` often looks like:

```
int main(void) {
    // variable declarations
    // ... (code statements)
    return 0;
}
```

or if command-line arguments are needed:

```
int main(int argc, char *argv[]) {
    // ... (use argc, argv if needed)
    return 0;
}
```

Inside `main`, you will typically: - Declare and initialize local variables. - Call functions (including library functions like `printf` or `scanf`). - Implement the algorithm (logic) of your program. - Eventually return an integer (0 for success).

Example (A simple program body in C):

```
int main(void) {
    int x = 10;
    int y = 20;
    int sum = x + y;
    printf("The sum of %d and %d is: %d\n", x, y, sum);
    return 0;
}
```

In this example, within `main` we declared two integers `x` and `y`, computed their sum, and then used `printf` to output the result. Finally, `return 0;` ends the program, returning control to the operating system.

- **Other Functions:** Beyond `main`, a C program can (and often should) have additional functions. These functions allow you to break the program into smaller, reusable pieces. For instance, you might have a function to calculate factorial of a number, another to check if a number is prime, etc. Functions in C are defined outside of `main` (usually either above `main` or below, with a prototype above `main`). A function definition includes a return type, a name, and parameters in parentheses, followed by a body enclosed in braces.

Example:

```
// Function prototype (declaration)
int add(int a, int b);

int main(void) {
    int result = add(5, 7);
    printf("Result: %d\n", result);
    return 0;
}

// Function definition
int add(int a, int b) {
    return a + b;
}
```

Here, we declared a function add that takes two integers and returns their sum. We then used it in `main`. Organizing code into functions helps make programs modular and easier to test.

- **End of program:** In C, the end of the `main` function (the closing brace} and the return statement) signifies the end of the program's execution. There is no special keyword like "end." as in Pascal. The source file might end after the closing brace of `main` or after other function definitions. Essentially, when the `main` function returns or exits (for example, via `return 0;`), the program terminates. If `main` finishes without an explicit return in a modern C compiler, it will implicitly return 0.

Example (End of program):

```
return 0;
}
```

This is simply the last line of the main function. Once reached, the program will stop running.

Putting it all together: A complete simple C program might look like this:

```
#include <stdio.h>

int main(void) {
    // This program adds two numbers and prints the result.
    int a, b;
    printf("Enter first number: ");
```

```
    scanf("%d", &a);
    printf("Enter second number: ");
    scanf("%d", &b);
    int sum = a + b;
    printf("The sum of %d and %d is %d.\n", a, b, sum);
    return 0;
}
```

This demonstrates the typical structure: includes at the top, then `main` with declarations, input, processing, and output.

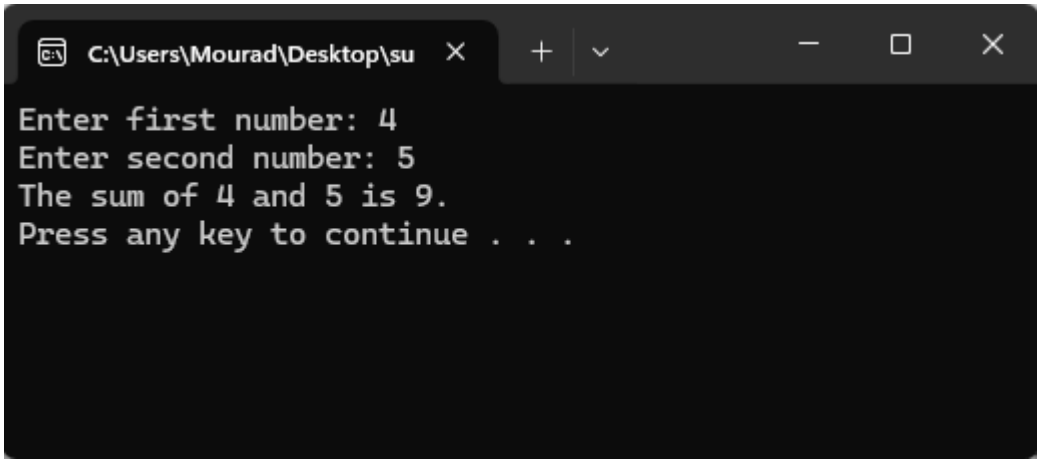

***Figure II.6** : the display of the C program that calculates the sum of two numbers entered by the user (as described above)*

II.4 Approach and Analysis of a Problem

When faced with a computing problem, it's important to take a methodical approach before jumping into coding. **Solving a problem using the C language** generally follows a series of steps:

1. **Understanding the Problem:** Make sure you clearly understand what is being asked. Identify the inputs (what data is given or will be provided), the required outputs (what results need to be produced), and any specific details or constraints of the problem. At this stage, it can be helpful to restate the problem in your own words or draw examples of input/output. For instance, if the task is to find the largest number in a list, ensure you know what a "list" might be in terms of input, and what exactly should happen if all numbers are equal, etc.

2. **Designing the Solution (Analysis & Algorithm Design):** Once the problem is understood, plan how to solve it. This could involve writing pseudocode, drawing a flowchart, or simply jotting down the steps in plain language. Break the problem down into smaller sub-problems if needed. Determine what data structures might be appropriate (e.g., will you use an array to store that "list" of numbers?) and what algorithms or formulae are relevant. Continuing the example, you might outline an algorithm: "assume first number is largest, then iterate through each number and update the largest when a bigger number is found."

3. **Implementation in C:** After designing a clear algorithm, you translate this solution into actual C code. This involves declaring the right variables, using appropriate control structures (loops, conditionals) as dictated by your algorithm, and calling any necessary library functions. While coding, you must also handle details like data types (ensuring, for example, that you use `float` or `double` if dealing with non-integers), memory considerations, and input/output formatting. As you implement, you might find the need to adjust your initial solution to handle edge cases or errors (like checking if the list is empty, in our example).

4. **Verification and Debugging:** After writing the code, test it with various inputs to verify that it works as expected. Start with simple test cases (including edge cases) and then try more complex or random inputs. If the output is not correct, use debugging techniques to find where the issue lies. This might involve adding temporary `printf` statements to check values of variables at certain steps, or using a debugger tool to step through the program. The goal is to ensure the program not only works for one scenario but for all valid scenarios and that it fails gracefully (or with clear messages) for invalid scenarios if applicable.

Throughout these steps, especially during design and implementation, keep the principles of good programming in mind: clarity, simplicity, and structure. It's often beneficial to document your code (with comments) while implementing, to explain non-obvious parts of your logic.

By following this structured approach to problem solving, you reduce the chance of getting stuck or writing incorrect programs. It transforms programming from a trial-and-error task into a more predictable and manageable process.

1. Understanding the Problem

Make sure to **read the problem carefully** and determine exactly what is required. Identify the knowns (inputs) and unknowns (outputs). For example, if the task is "compute the factorial of a number n", the input is the number n, and the output is n! (n factorial). Constraints might be

that n is a non-negative integer, and perhaps that n is not too large (to avoid overflow or long computation time).

Techniques for understanding and defining the problem include:
- Writing down what the input format is and what output format is expected.
- Drawing examples: “If n = 5, output should be 120.” This helps verify understanding.
- Noting special cases: “What if n = 0? What if n = 1? Factorial of 0 is defined as 1, so handle that.”
- Ensuring you understand any domain-specific terms or conditions in the problem statement.

If any part of the problem is unclear, it’s important to clarify those aspects (by asking the teacher, or if self-study, researching or re-reading the problem).

2. Designing the Solution

Once the problem is well understood, it is time to design a solution. This may involve researching known algorithms or figuring out a new logical procedure for the task at hand. You should plan the overall organization of the program at this stage:

- Decide what major steps are needed (you can often outline these in comments or pseudocode form).
- Determine if you can reuse or adapt any known algorithms.
- Choose appropriate data structures: for instance, do you need an array, a struct, etc., to hold your data?
- Consider the use of functions to structure the solution into sub-tasks (e.g., a separate function to calculate a factorial, a separate function to handle input/output, etc., which will be called from main).

Breaking the solution into **step-by-step pseudocode** is very helpful. Continuing the factorial example, pseudocode might be:

```
function factorial(n):
    if n < 2:
        return 1
    else:
        result = 1
        for i from 2 to n:
            result = result * i
        return result
```

This pseudocode designs a clear solution for factorial. It will later be easy to translate into C.

3. Implementation in C

Now, translate your designed solution into C code. This step involves coding and is often where details matter a lot:

- **Declare Variables:** Decide on the correct types. For factorial, `int n;` might suffice for small n, but factorial grows fast – maybe use a larger type (like `long long` in C for 64-bit integer) or even libraries for big integers if n can be large. In a beginner context, we might assume n isn't huge, and use `unsigned long long` for example. Also declare a variable for result if needed.

- **Control Structures:** Use loops (`for`, `while`) and conditionals (`if`/`else`) as per your algorithm. Each pseudocode step should translate fairly directly to one or a few lines of C. For example, the pseudocode loop "for i from 2 to n" becomes a `for` loop in C.

- **Functions:** If you planned to break the problem into functions, define those functions in C with appropriate return types and parameters. Using functions can make your `main` cleaner and the code more modular.

- **Handle Input/Output:** Use `scanf` to get inputs (making sure to use & before variables, and correct format specifiers) and `printf` to output results in the required format.

- **Include Necessary Headers:** Don't forget to include headers like `<stdio.h>` (for I/O) or others if you use functions from them (like `<math.h>` if you used math functions, etc.).

During implementation, it's common to compile and test in small pieces if possible (especially for more complex programs). For instance, after writing the factorial function, you might test it independently with a few values before integrating it into the rest of the program.

4. Verification and Debugging

After writing the program, thoroughly test it:

- **Sample Tests:** Start with simple, known cases. If computing factorial, test n=0 (should output 1), n=1 (1), n=5 (120). If any of these are wrong, there's a bug.

- **Edge Cases:** Test boundaries. If there are limits, test them. For factorial, maybe test n=10 or n=15 to ensure it's still correct and see if there's any integer overflow issues (depending on type). If negative input is not allowed but possible a user enters, test how your program behaves (ideally, it should handle or at least inform about invalid input).

- **Unusual Situations:** For general problems, consider things like: empty input, maximum/minimum values, etc., as applicable.

If the program output doesn't match expected results, use debugging strategies:
- Add `printf` statements to print intermediate values (e.g., inside a loop, print the loop counter and current result to see where it might go wrong).

- Check that all loops terminate correctly and all conditions are written correctly (a common bug might be using = instead of == for comparison, etc.).

- Ensure that all variables are properly initialized before use.

- Use a debugger tool if available to step through your program line by line and inspect variable values.

Repeat the edit-compile-run cycle as needed until the program produces correct results for all test cases.

Remember, finding and fixing bugs (debugging) is a normal part of programming. Even experienced programmers make mistakes – what matters is systematically narrowing down the cause of a problem and correcting it.

By following these steps in problem solving and analysis, you'll develop programs that are correct, efficient, and well-structured.

II.5 Data Structure

In programming, **data structure** refers to the way data is stored, organized, and managed in a program. At this introductory stage, we focus on fundamental data structures that are built into the language (simple variables and their types), rather than complex structures (like linked lists or trees, which come later).

1. Constants

A **constant** is a data value that cannot be altered during the execution of the program. It's a fixed value. Constants are useful for representing values that appear in a program but should not change, because they have a special meaning or are used in multiple places. For example, the value of π (pi) is constant (3.14159…), or if a program always needs an array of size 100, that size can be a constant.

In C, there are a couple of ways to define constants:

- **Literal constants:** Just writing numbers or characters directly in the code. e.g., `10` is an integer literal constant, `'A'` is a character literal constant, `3.14159` is a floating-point literal constant.

- **`#define` macro:** As shown earlier, you can use the preprocessor to define a constant. For example,

  ```
  #define MAX_STUDENTS 30
  ```

  After this line, anywhere in the code `MAX_STUDENTS` appears, the compiler will substitute the number 30. This is a simple textual substitution, but by convention we use it to define constants (the name is typically written in uppercase to denote a constant).

- **`const` keyword:** You can declare a variable with the `const` qualifier, which means once you initialize it, you cannot modify it. For example:

  ```
  const float PI = 3.14159f;
  ```

  Now `PI` behaves like a variable of type float, but any attempt to assign a new value to `PI` will result in a compile-time error. This method is often preferable to `#define` in modern C because it obeys scope rules and type checking.

Using constants makes programs easier to read and maintain. For instance, if you use 30 in many places and later want to change it to 40, having a `#define MAX_STUDENTS 30` means you change just that line, rather than every occurrence of 30 (which might be error-prone if some 30s mean something else).

2. Variables

A **variable** is a named storage location that can hold a value which may change during the execution of the program. In C (as in Pascal and other languages), you must declare each variable before using it, specifying its data type.

When declaring a variable in C, the syntax is:
`<type> <variable_name>;`
You can also initialize it in the same statement:
`<type> <variable_name> = <initial_value>;`

Some rules and tips for variables in C:

- Variable names should start with a letter or underscore, followed by letters, digits, or underscores. They are case-sensitive (`sum` vs `Sum` are different).

- Choose meaningful names for clarity (e.g., `total_price` rather than `tp`).

- You cannot use C reserved keywords as variable names (for example, you can't name a variable `int` or `while`).

- It's good practice to initialize variables when you declare them, to avoid indeterminate values (in C, if you don't initialize, variables of automatic storage have undefined values).

Example:

```
int age;           // declaring an integer variable named age
age = 20;          // assigning a value to age
float price = 9.99; // declaring a float variable price and initializing it
char letter = 'A'; // declaring a char variable letter with initial value 'A'
```

In memory, each variable occupies space depending on its type (for instance, typically 4 bytes for an `int` on many systems, 4 bytes for a `float`, 1 byte for a `char`, etc., although these can vary). The program can change the content of that memory location by assigning new values to the variable.

Using variables allows programs to store intermediate results and state. For instance, if you're summing numbers in a loop, you would use a variable to accumulate the total.

3. Data Types

Data types specify what kind of data a variable can hold and how much space it occupies. They also determine what operations are allowed on that data. C provides several basic data types:

- **Integer types:** For whole numbers (no fractional part). Common integer types in standard C:
- `int` – typically 32-bit (can hold values roughly in the range –2 billion to +2 billion). The exact size can depend on the system, but it's usually the "natural" word size of the machine (4 bytes on 32-bit and 64-bit systems).
- `short` – typically 16-bit (range about –32,768 to 32,767).
- `long` – at least 32-bit, often 64-bit on modern systems.
- `long long` – at least 64-bit, for very large integers.
- You also have unsigned versions (`unsigned int`, `unsigned long`, etc.) which only represent non-negative values but can roughly double the positive range since they don't need to account for negatives.

- **Floating-point types:** For real numbers (numbers with fractional parts).
- `float` – typically 32-bit single precision floating point (around 7 decimal digits of precision).
- `double` – typically 64-bit double precision (around 15 decimal digits of precision).
- `long double` – possibly 80-bit or more (depending on the compiler) for extended precision.
- **Character type:**
- `char` – typically 1 byte. Can hold a single character (like 'A', '5', '?' etc.) or a small integer. In C, `char` is actually an integer type under the hood (usually an 8-bit signed or unsigned value). Characters are stored as numbers using ASCII or Unicode encoding (ASCII codes for 'A' is 65, etc.).
- **_Bool type:**
- `_Bool` – a built-in type in C99 that can hold values 0 or 1 (for false/true).
- `bool` – not a distinct type by itself, but if you include `<stdbool.h>`, it defines `bool` as an alias of `_Bool` and also defines the constants `true` (1) and `false` (0) for convenience.
- If `<stdbool.h>` is not used, one can still use integers to represent truth values (convention: 0 = false, nonzero = true), but using `bool` makes the intention clearer.
- **Void type:** `void` is a special type that basically means "no type". You see this with functions that return nothing (`void functionName(...)`) or take no arguments (`int main(void)` means main takes no parameters). You cannot declare a normal variable of type void.
- **Derived types:** C allows creating more complex types from these basics: *arrays* (sequences of elements of a given type), *pointers* (which hold memory addresses), *structures* (combining multiple variables into one compound type), *unions*, *enums* (enumeration of constants), etc. These are more advanced and will be introduced in subsequent lessons.

Here are some examples of declaring variables of various types and what they represent:

```
int quantity = 5;          // integer number (for count of items, etc.)
float temperature = 36.6; // floating-point number (e.g., for Celsius tempera
ture)
char grade = 'A';          // single character (perhaps a grade in a test)
bool isValid = true;       // boolean value (requires #include <stdbool.h>)
```

And an example showing an array (which is a basic data structure made up of elements of one type):

```
int numbers[5];          // an array of 5 integers (index 0 through 4)
numbers[0] = 10;         // assign first element
numbers[1] = 20;
```

Each data type has limits. If you try to store a number beyond the range of the type, you get overflow (which leads to undefined or wrap-around behavior in C). For instance, adding 1 to the maximum int might wrap to a negative number without warning. So choosing appropriate types and being mindful of their range is important.

In summary, understanding data types allows you to choose the right type for the right job: use `int` for countable quantities, `float`/`double` for measurable quantities or those requiring precision, `char` for characters, and so on. This ensures that your program uses memory efficiently and that operations on variables behave as expected.

Note: (The example above underlines that in C, unlike some languages, strings are not a built-in type but are handled as arrays of `char`. For a beginner course, detailed discussion of strings might be minimal at first, focusing just that a "string" can be represented as an array of characters ending with a null terminator `'\0'`, and perhaps using the `char name[50]` approach as shown before.)

II.6 Operators

In **C**, as in other languages, operators are symbols that instruct the program to perform specific operations on one or more operands (values or variables). C has a rich set of operators. We will cover the main categories:

- **Assignment Operator**
- **Arithmetic Operators**
- **Relational (Comparison) Operators**
- **Logical Operators**
- **Other operators** (like increment/decrement, etc., though those might be introduced later)

II.6.1. Assignment Operator

The assignment operator in C is the single equals sign =. This operator assigns the value on its right to the variable on its left.

Example:

```
int x;
x = 10;     // Assign 10 to x
```

After this operation, x contains the value 10. You can also chain assignments or assign from expressions:

```
int a, b;
a = 5;
b = a;      // Now b gets the value of a, which is 5
a = a + 2; // a is now 7 (previous value 5 plus 2)
```

In C, assignment itself produces a value (the value assigned), so you will sometimes see things like:

```
int c, d;
c = d = 0; // assigns 0 to d, and then assigns that result (0) to c, so both
are 0
```

This works because `d = 0` yields 0, which then is assigned to `c`. However, while this is valid, beginners should be careful with complex assignment expressions and perhaps avoid chaining until comfortable.

It is crucial to distinguish the assignment operator = from the equality comparison operator ==. Using one in place of the other is a common mistake. For instance:

```
if (x = 1) { ... }  // This actually assigns 1 to x and then checks if x (whi
ch is now 1) is nonzero.
```

The above is a bug (it will always be true because x becomes 1). It should be `if (x == 1)`. Always use == when comparing.

C also provides *compound assignment* operators that combine arithmetic with assignment for convenience: - `x += 5;` (equivalent to `x = x + 5;`) - `y -= 2;` (equivalent to `y = y - 2;`) - `z *= 3;` (`z = z * 3;`) - `n /= 10;` (`n = n / 10;`) - `m %= 4;` (`m = m % 4;` – remainder after division by 4)

These can make code more concise.

II.6.2. Relational Operators

Relational (or comparison) operators compare two values and yield a boolean result (true/false, in C essentially 1 or 0). They are used in conditions (like in `if` statements, loops, etc.) to decide the flow of control.

The primary relational operators in C are:

- == : Equal to. (Checks if the left side is equal to the right side.)
- != : Not equal to. (True if the two sides are different.)
- < : Less than.
- <= : Less than or equal to.
- > : Greater than.
- >= : Greater than or equal to.

Example:

```
int a = 5, b = 8;
bool result1 = (a == b);  // result1 is false (0) because 5 is not equal to 8
bool result2 = (a < b);     // result2 is true (1) because 5 is less than 8
bool result3 = (a != 0);    // result3 is true because 5 is not equal to 0
```

Relational operators have lower precedence than arithmetic operators, but higher than assignment. It's still a good practice to use parentheses in complex expressions to make the intention clear.

II.6.3. Logical Operators

Logical operators are used to combine or invert boolean expressions:

- && (Logical AND): True if and only if both operands are true.
- || (Logical OR): True if at least one of the operands is true.
- ! (Logical NOT): Unary operator that yields true if the operand is false, and vice versa (basically it inverts a boolean value).

These operators allow you to form compound conditions. They operate on truth values (in C, 0 is false, non-zero is true).

Example:

```
bool x = true;
bool y = false;
bool res1 = x && y;    // false, because y is false (AND requires both true)
bool res2 = x || y; //true, because x is true (OR requires at least one true)
bool res3 = !x;        // false, because x is true and NOT true yields false
bool res4 = !(5 < 3);  // true, because (5<3) is false, and NOT false is true
.
```

It's important to note that C's logical operators use **short-circuit evaluation**. This means in an expression like `condition1 && condition2`, if `condition1` is false, `condition2` won't even be evaluated (because the whole result is already determined false). Similarly, for `condition1 || condition2`, if `condition1` is true, `condition2` is not evaluated (since the OR is already true). This can be useful, for instance:

```
if (ptr != NULL && ptr->value > 0) { ... }
```

Here, if `ptr` is NULL, the second part (`ptr->value > 0`) won't execute, preventing a crash. This idiom is common in C.

4. Arithmetic Operators

Arithmetic operators are used for performing mathematical calculations. C supports:

- `+` (addition)
- `-` (subtraction or negation if unary)
- `*` (multiplication)
- `/` (division)
- `%` (modulo, remainder of integer division)

For example:

```
int a = 14, b = 4;
int sum = a + b;        // sum = 18
int diff = a - b;       // diff = 10
int prod = a * b;       // prod = 56
int quot = a / b;   // quot = 3 (integer division: fractional part truncated)
int rem  = a % b;       // rem = 2 (14 divided by 4 is 3 remainder 2)
```

Note that for `%` both operands must be integers (you cannot modulo a float in C directly).

Division between integers deserves special attention: it **truncates** toward 0. In the example above, 14/4 gave 3, not 3.5, because these were ints. If we want a floating point result, at least one operand must be float/double:

```
double result = (double)a / b; // result = 3.5 (casting a to double makes the division floating-point)
```

C follows standard algebraic precedence rules for these operators: multiplication, division, and modulo have higher precedence than addition and subtraction. All arithmetic operators evaluate left to right (they're left-associative). So `x + y * z` is interpreted as `x + (y * z)`, not `(x + y) * z`.

To alter the normal precedence, use parentheses:

```
int value = (a + b) * c; // forces addition first, then multiplication
```

C also has unary + and - (for example, `-x` to negate the value of x).

Additionally, C provides increment and decrement operators (which we can mention briefly): - ++x (pre-increment) / x++ (post-increment): Increase x by 1.
- `--x` / `x--`: Decrease x by 1.
These have side effects and a specific difference between prefix and postfix form, which can be explained as needed.

5. Priorities in Operations

As mentioned, operations in C have a defined order of precedence. A simplified summary from highest to lower precedence:

1. Parentheses `()` – can be used to explicitly dictate evaluation order. Anything inside parentheses is evaluated first.
2. Unary operators – like `!` (logical NOT), unary `-` (negation), ++/`--` (when used as prefix), etc.
3. Multiplicative – *, /, %
4. Additive – +, -
5. Relational – <, <=, >, >=
6. Equality – ==, `!=` (they are slightly lower than relational in precedence)
7. Logical AND – &&
8. Logical OR – ||
9. Assignment – =, +=, `-=`, etc.

If in doubt about how an expression will be grouped, it's always safer (and often clearer) to use parentheses to make the grouping explicit.

Example of precedence:

```
int x = 5, y = 10, z = 2;
int result = x + y / z;
```

According to precedence, this will do y / z first (10/2 = 5) then add x: result becomes 5 + 5 = 10. If we wanted to add x and y first, we'd need to write:

```
int result2 = (x + y) / z;  // (5+10)=15, /2 gives 7 (since these are ints, 15/2 truncates to 7)
```

Another example with logical and relational:

```
int a = 4;
int b = 5;
bool res = a < 10 && b > a + 2;
```

Here, a + 2 is evaluated first (giving 6), then the comparisons: a < 10 (true) and b > 6 (false, since 5 is not > 6). Then the && combines them (true && false = false). So res would be false.

Example illustrating parentheses:

```
int res3 = (x + y) * z; // parentheses cause addition first
```

This ensures a different order than default.

By mastering operators and their precedence, you can write correct and efficient expressions in C. When expressions become complex, break them into smaller parts or use parentheses to clarify, which also helps prevent bugs.

Example Summary:

```
int x = 3, y = 4, z = 5;
int result = (x + y) * z;   // (3+4)=7, 7*5 = 35
result = x + (y * z);       // 3 + (4*5) = 23
result = x + y * z; //without parentheses, * has priority, same as above (23)
bool cond = x < y || x > z; // (3<4) is true, (3>5) is false, true || false = true.
```

II.7 Input/Output (I/O) Operations

Input/Output operations allow a program to interact with the outside world – typically with the user via keyboard/screen, or with files on disk. In **C**, I/O is not part of the core language (unlike Pascal's built-in `readln`/`writeln`). Instead, C relies on standard library functions (mainly from `<stdio.h>`) to perform I/O.

1. Data Input

To read data from the user (standard input, usually the keyboard) or from a file, C provides various functions. In a beginner context, the primary function for console input is `scanf`. For more advanced input (like reading entire lines or strings safely) functions like `fgets` can be used.

- **`scanf` function:** This reads formatted input from `stdin` (standard input). It requires a *format string* that specifies the type of data expected, and pointers to variables where the read values should be stored. For example:

  ```
  int age;
  scanf("%d", &age);
  ```

  The `"%d"` in the format string tells `scanf` to expect an integer. The `&age` passes the address of the age variable so that `scanf` can fill that memory with the input value. Common format specifiers include `%d` for int, `%f` for float, `%lf` for double, `%c` for char, `%s` for string (a character array, `scanf` stops at whitespace for `%s`).

It's important to include the & (address-of) for all variables except strings (arrays naturally decay to pointers). Forgetting & leads to a bug because `scanf` will then attempt to write at an address equal to the value, rather than the address of the variable (often causing a crash).

- **`fgets` function:** This reads an entire line (including spaces) into a string (character array). For example:

  ```
  char line[100];
  fgets(line, sizeof(line), stdin);
  ```

  This will read up to 99 characters (leaving space for the null terminator) or until a newline is encountered, whichever comes first. It's safer for string input because it guards against overflow (you provide the buffer size). `fgets` will include the newline character if it fits, and add a `'\0'` at the end.

- **`getchar` function:** This reads a single character from input. You might use this in a loop to process input char by char, or simply to pause (like reading an Enter key press).

At this stage, `scanf` is usually used for its simplicity in reading basic data types.

Example – reading multiple inputs:

```
int day, year;
char monthName[20];
printf("Enter day, month, year: ");
scanf("%d %19s %d", &day, monthName, &year);
```

If the user enters: `15 March 2024`, then after this day will be 15, `monthName` will be `"March"` (as a string), and `year` will be 2024. We used `%19s` to limit the string input to at most 19 characters to avoid overflow in monthName (which has space for 20 including null terminator).

When reading numeric input, any whitespace (spaces, newlines, tabs) in the format string means `scanf` will skip any whitespace in the input. So `%d %d` can read two numbers separated by spaces or newlines.

One thing to note: `scanf` leaves the newline in the input buffer for strings, and reading characters after numbers can be tricky due to leftover newline. That's an advanced point (we often use `getchar()` to consume leftover newline if needed, or use `fgets` to read lines and parse them).

2. Data Output

To display data to the user or write it to a file, the C standard library provides output functions. The most commonly used for console output is `printf`.

- **`printf` function:** This prints formatted output to `stdout` (standard output, usually the console). It uses a format string with placeholders (format specifiers) for variables. For example:

  ```
  int age = 20;
  printf("You are %d years old.\n", age);
  ```

  This will output: `You are 20 years old.` (followed by a newline because of `\n` in the format string). The `%d` is replaced by the value of age. The variables to print are provided as additional arguments after the format string. Common format specifiers:
- `%d` or `%i` for int
- `%ld` for long, `%lld` for long long
- `%u` for unsigned int
- `%f` for float, `%lf` for double (both will format as floating-point decimal)
- `%c` for char (it will print the character corresponding to the char code)

- %s for a string (expects a `char *` pointing to a null-terminated char array)
- There are others for formatting in different ways (hex %x, etc., not needed just yet).

We also include special characters in the format string like `\n` (newline), `\t` (tab), etc., which are not printed literally but have control effects.

Example – printing multiple values:

```
int a = 5;
int b = 7;
printf("a = %d, b = %d, a+b = %d\n", a, b, a+b);
```

Possible output: `a = 5, b = 7, a+b = 12`. The format string had three %d placeholders, so we provided three integers to match them in order.

Another example, demonstrating different types:

```
char name[] = "Alice";
int age = 30;
float height = 1.68;
printf("Name: %s, Age: %d, Height: %.2f meters\n", name, age, height);
```

Output: `Name: Alice, Age: 30, Height: 1.68 meters`. Here, `%.2f` prints the float with 2 digits after the decimal point.

- **putchar and puts:** For simple cases, `putchar(c)` prints a single character, and `puts(string)` prints a string followed by a newline. For instance, `puts("Hello")` is like doing `printf("Hello\n")`. These are simpler but less flexible than `printf`.

In summary, printing in C is about constructing the output string with placeholders, and `printf` does the work of combining text with variable values.

Important note: When using %s in `printf`, ensure the string is properly null-terminated. `printf` will print characters until it finds a `'\0'`. Also ensure the type of the variable matches the format specifier (for example, using %d for a float can lead to unpredictable output).

II.8 Control Structures

Control structures allow you to control the flow of a program's execution based on conditions or to repeat certain operations. There are two main categories of control structures:

1. **Conditional control structures** – for making decisions (if some condition is true, do this, otherwise do that, etc.).
2. **Repetitive control structures (loops)** – for repeating a block of instructions multiple times (either a fixed number of times, or until a condition is met).

II.8.1 Conditional Control Structures

The most common conditional structure in C is the **if statement**. C also provides a multi-way decision with **switch**, but we'll focus on `if` here (similar in spirit to Pascal's if-then-else).

`if` statement: It allows the program to execute a block of code only if a specified condition is true. Optionally, you can provide an `else` part to execute an alternate block if the condition is false.

The syntax in C is:

```
if (condition) {
    // Instructions to execute if the condition is true
} else {
    // Instructions to execute if the condition is false
}
```

- The condition is placed in parentheses and typically involves relational operators (==, <, etc.) and/or logical operators. - The braces {} enclose the blocks of code for each case. (In C, braces are required if you have multiple statements in the block. If there's only one statement, you can omit braces, but it's often clearer to use them anyway.) - The `else` part is optional. You can have an `if` without an else if you only need to do something when the condition is true and do nothing otherwise.

Example with if-else:

```
int number;
printf("Enter an integer: ");
scanf("%d", &number);
if (number % 2 == 0) {
    printf("The number is even.\n");
} else {
```

```
    printf("The number is odd.\n");
}
```

In this example, if the number is divisible by 2 (remainder 0), the first printf executes; if not, the second printf executes.

You can also chain multiple conditions with `else if` for checking multiple mutually exclusive conditions:

```
int score = /* some value */;
if (score >= 90) {
    printf("Grade: A\n");
} else if (score >= 80) {
    printf("Grade: B\n");
} else if (score >= 70) {
    printf("Grade: C\n");
} else {
    printf("Grade: F\n");
}
```

This checks conditions in order. Once one condition is true, its block executes and the rest are skipped.

`if` without `else`:
You can use a solo if when you only need an action on true:

```
if (count == 0) {
    printf("Count is zero!\n");
}
```

If `count` isn't zero, nothing happens and the program continues past the if.

Also note, if your `if` or `else` block has a single statement, you can write it without braces:

```
if (x < 0)
    x = -x;
```

But be careful with this, especially with nested ifs, and it's usually safer to include braces to avoid logical errors when modifying code later.

Nested if statements:
You can put ifs inside if-blocks. Indentation becomes important for readability:

```
if (age > 0) {
    if (age < 18) {
```

```
        printf("You are a minor.\n");
    } else {
        printf("You are an adult.\n");
    }
} else {
    printf("Age must be positive.\n");
}
```

Here we first check age > 0, then inside that, another if checks <18.

To avoid confusion with multiple if/else, especially in the absence of braces, remember that an `else` matches the nearest preceding unmatched `if`. This is known as the "dangling else" problem. Always use braces or proper indentation to clarify.

There is also a simpler variant: an `if` without an `else` when no alternate action is needed.

Example summary in pseudocode (C syntax):

```
if (condition) {
    // do something
}
// (optional) else-if chain:
else if (other_condition) {
    // do something else
}
else {
    // default action if none of the above conditions were true
}
```

II.8.2 Repetitive Control Structures

Repetitive control structures, or loops, allow a set of instructions to be executed multiple times. C provides several types of loops: `for`, `while`, and `do...while`. Each is useful in different scenarios, but they can all be used to similar effect; choosing one is often about clarity and style.

- **`for` loop:** Typically used when the number of iterations is known in advance or can be determined by a counter. It has the form:

  ```
  for (initialization; condition; update) {
      // instructions to repeat
  }
  ```

- The *initialization* step runs once at the beginning (e.g., `int i = 1;`).

- The *condition* is checked before each iteration; if it's true, the loop body executes, if false, the loop ends.
- After each iteration of the loop body, the *update* step executes (e.g., `i++` to increment a counter).

Example (`for` loop):

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

This will print numbers 1 through 10. The loop runs 10 times: i starts at 1, and as long as i <= 10, it prints i then increments i.

- **`while` loop:** Used when you want to repeat something *while a condition remains true*. It checks the condition at the start of each iteration.

  ```
  while (condition) {
      // instructions to repeat
  }
  ```

  If the condition is false initially, the loop body may not execute at all.

Example (`while` loop):

```
int count = 5;
while (count > 0) {
    printf("Countdown: %d\n", count);
    count--;
}
```

This will print count from 5 down to 1. When count becomes 0, the condition `count > 0` fails and the loop ends.

- **`do...while` loop:** Similar to `while`, but the condition is checked *after* each iteration. This guarantees that the loop body runs at least once.

  ```
  do {
      // instructions
  } while (condition);
  ```

  Note the semicolon at the end of a do...while loop. The loop executes the body, then checks the condition; if true, it loops back, if false, it exits.

Example (`do...while`):

```
int x;
do {
    printf("Enter a positive number: ");
    scanf("%d", &x);
} while (x <= 0);
```

This loop will keep prompting the user until they enter a positive number. Even if the user enters a positive number the first time, the loop body did run that one time (which is desired in this case to get input).

Choosing loops:

- Use `for` loops when you have a definite loop count or want to iterate over a sequence of values systematically (like from 1 to N). The `for` loop keeps loop control (initialization, condition, update) in one place, which can be convenient. - Use `while` when you want to loop until a condition is false, and you might not know how many times exactly (the loop could even be infinite if the condition never falsifies). - Use `do...while` when you need to ensure the loop runs at least once (for example, menu display at least once, then repeat if choice invalid, etc.)

Break and Continue (loop control):

Within loops, you can use `break;` to exit the loop immediately (often used when a search condition is met or an error occurs).

`continue;` can be used to skip the rest of the loop body and go immediately to the next iteration (checking the condition again for `while`, or doing the update step then condition in a `for`). Use these sparingly as they can sometimes make logic harder to follow.

Example using break:

```
for (int i = 1; i <= 100; ++i) {
    if (i * i > 2000) {
        break; // exit loop if square of i exceeds 2000
    }
    printf("%d ", i);
}
```

Example using continue:

```
for (int i = 1; i <= 10; ++i) {
    if (i % 2 == 0) {
        continue; // skip even numbers
```

```
    }
    printf("%d ", i); // this will print only odd numbers 1 3 5 7 9
}
```

Putting it all together in a simple loop example:

```
// Print the first 5 multiples of 3
int n = 1;
while (n <= 5) {
    printf("%d ", 3 * n);
    n++;
}
printf("\n");
```

This uses a `while` loop to output: `3 6 9 12 15`.

By understanding and using these control structures, you can direct your program's logic: making decisions with `if/else` and repeating actions with loops (`for`, `while`, `do...while`). These are fundamental tools for algorithm implementation.

Exercices:

Exercise 1: Write an algorithm and then a C program that finds the maximum of three numbers.

Algorithm:

```
1. Start
2. Input three numbers: a, b, c
3. If a > b and a > c then
   Print a
4. Else if b > a and b > c then
   Print b
5. Else
   Print c
6. End
```

C Program:

```
#include <stdio.h>

int main() {
    int a, b, c;

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

    if (a > b && a > c)
        printf("The maximum number is: %d\n", a);
    else if (b > a && b > c)
        printf("The maximum number is: %d\n", b);
    else
        printf("The maximum number is: %d\n", c);

    return 0;
}
```

Exercise 2: Develop an algorithm and implement a C program to check if a number is Prime.

Algorithm:

```
1. Start
2. Input a number n
3. If n < 2 then
   Print "Not Prime"
4. For i := 2 to sqrt(n) do
```

```
   If n mod i = 0 then
      Print "Not Prime"
      Stop
5. If no divisor is found then
   Print "Prime"
6. End
```

C Program:

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

int main() {
    int n, i;
    int is_prime = 1; // Assume the number is prime initially

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

    if (n < 2) {
        printf("Not Prime\n");
    } else {
        for (i = 2; i <= sqrt(n); i++) {
            if (n % i == 0) {
                is_prime = 0;
                break;
            }
        }

        if (is_prime)
            printf("Prime\n");
        else
            printf("Not Prime\n");
    }

    return 0;
}
```

Exercise 3: Create an algorithm and write a C program to compute the Factorial of a Number.

Algorithm:

```
1. Start
2. Input a number n
3. Set factorial := 1
4. For i := 1 to n do
   factorial := factorial * i
```

```
5. Print factorial
6. End
```

C Program:

```
#include <stdio.h>

int main() {
    int n, i;
    long long factorial = 1;

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

    if (n < 0)
        printf("Factorial is not defined for negative numbers.\n");
    else {
        for (i = 1; i <= n; i++) {
            factorial *= i;
        }
        printf("Factorial of %d is: %lld\n", n, factorial);
    }

    return 0;
}
```

Exercise 4: Write an algorithm and code a C program to reverse a number.

Algorithm:

```
1. Start
2. Input a number n
3. Set reversed := 0
4. While n > 0 do
   Set remainder := n mod 10
   Set reversed := reversed * 10 + remainder
   Set n := n div 10
5. Print reversed
6. End
```

C Program:

```
#include <stdio.h>

int main() {
    int n, reversed = 0, remainder;
```

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

    while (n != 0) {
        remainder = n % 10;
        reversed = reversed * 10 + remainder;
        n /= 10;
    }

    printf("Reversed number is: %d\n", reversed);

    return 0;
}
```

Exercise 5: Construct an algorithm and develop a C program to find the sum of digits of a number

Algorithm:

```
1. Start
2. Input a number n
3. Set sum := 0
4. While n > 0 do
   Set sum := sum + n mod 10
   Set n := n div 10
5. Print sum
6. End
```

C Program:

```
#include <stdio.h>

int main() {
    int n, sum = 0;

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

    while (n != 0) {
        sum += n % 10;
        n /= 10;
    }

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

    return 0;
}
```

References

1. Abelson, H., & Sussman, G. J. (1996). *Structure and interpretation of computer programs* (p. 688). The MIT Press.
2. Balagurusamy, E. (2009). *Fundamentals of computers* (4th ed.). McGraw-Hill Education.
3. Dahl, O. J., Dijkstra, E. W., & Hoare, C. A. R. (Eds.). (1972). *Structured programming*. Academic Press Ltd..
4. Drouillon, F. (2025). *Langage C - Maîtriser la programmation procédurale (avec exercices pratiques)* (3e éd.). ENI. (Informations détaillées sur l'édition 2025 non sourcées, l'éditeur est supposé ENI d'après une recherche connexe).
5. Gill, N. S. (2016). *Essentials of computer and network technology*. Khanna Book Publishing.
6. Goel, A. (2010). *Computer fundamentals*. Pearson Education India.
7. Hetland, M. L. (2005). *Beginning Python: from novice to professional*. Berkeley, CA: Apress.
8. Hopcroft, J. E., Ullman, J. D., & Aho, A. V. (1983). *Data structures and algorithms* (Vol. 175). Boston, MA, USA:: Addison-wesley.
9. Kernighan, B. W., & Ritchie, D. M. (1988). The C Programming Language Second.
10. Knuth, D. E. (1997). *The art of computer programming, Vol. 1: Fundamental algorithms* (3rd ed.). Addison-Wesley.
11. Kochan, S. G. (1993). *Programming in Ansi C*. Prentice-Hall, Inc..
12. Léry, J.-M. (2024). *Le langage C : Apprendre à programmer - Avec plus de 250 exemples et exercices corrigés*. Ellipses.
13. Nassi, I., & Shneiderman, B. (1973). Flowchart techniques for structured programming. *ACM Sigplan Notices*, *8*(8), 12-26.
14. Oualline, S. (1997). *Practical C programming*. " O'Reilly Media, Inc.".
15. Rajaraman, V., & Adabala, N. (2025). *Fundamentals of computers*. PHI Learning Pvt. Ltd..
16. Schneier, B. (2007). *Applied cryptography: protocols, algorithms, and source code in C*. john wiley & sons.
17. Sem, M., & Hours, N. M. Fundamentals of Computers. *Language*, *4*, 3.
18. Szuhay, J. (2022). *Learn C Programming: A beginner's guide to learning the most powerful and general-purpose programming language with ease*. Packt Publishing Ltd.
19. Thareja, R. (2012). *Computer fundamentals & programming in c*. Oxford University Press.
20. Van Rossum, G., & Drake, F. L. (2003). Python language reference manual.
21. Xinogalos, S. (2013, March). Using flowchart-based programming environments for simplifying programming and software engineering processes. In *2013 IEEE Global Engineering Education Conference (EDUCON)* (pp. 1313-1322). IEEE.
22. Zobel, J. (1997). *Writing for computer science*.