Introduction to C++ (Volume I)

This book can be accessed online at:
https://fhsu.pressbooks.pub/IntroductionToCPlusPlusVolumeI/

Introduction to C++ (Volume I)

HUSSAM GHUNAIM

Contents

Preface

HUSSAM GHUNAIM

Preface:

This book is written as an Open Education Resource (OER) to replace expensive commercial materials currently used at the Department of Computer Science at Fort Hays State University. It has two volumes corresponding to CSCI 121 and CSCI 221 courses. These courses are developed to introduce college freshmen students to Object-Oriented Programming utilizing C++.

The author tried to bridge the gap in the current programming textbooks by avoiding lengthy and, on many occasions, unnecessary details. This book's main feature is to present the discussed principles in the least wording possible while providing adequate examples and exercises to reinforce students' learning. The text is kept as simple as possible, while efforts have been made to create practical and fun examples and exercises to increase the attractiveness of the material and its suitability for the targeted audience.

The book emphasizes a step-by-step learning process with numerous practical, working code examples that students can easily experiment with. Complex topics are explained using visual aids and intuitive analogies to foster deeper understanding. The book employs a gradual introduction of concepts, building upon previously learned material in a logical progression. Visual

diagrams are used extensively to illustrate abstract concepts. Problem-solving is a central focus, with each chapter including exercises designed to challenge students to apply their knowledge and develop their analytical skills.

All the code was written utilizing Visual Studio 2022, although other compatible compilers, including online compilers, should function similarly. To aid learning, the text employs color coding for C++ keywords and data types, mirroring the Visual Studio environment. Therefore, all codes are presented as images to preserve the Visual Studio coding color scheme. A similar coding scheme is used in the main text to help first-time programmers distinguish the code keywords from the main text.

Recognizing the value of hands-on practice, students are strongly encouraged to input and test all provided code examples and exercises manually. The book is further enriched with numerous examples and exercises with their solutions provided at the end of each chapter. Additional chapter-end exercises and programming projects are also provided to solidify learning and aid in self-assessment and learning.

How to Use This Book:

It is recommended that students work through the chapters of this book sequentially, as each chapter builds upon the foundational knowledge presented in earlier sections. My hope is that this book will make learning C++ accessible, engaging, and even enjoyable for students. I believe that with a focused effort and by working through the material presented here, students will not only grasp the fundamentals of this powerful language but also develop a strong foundation for future studies and endeavors in computer science.

Embrace the challenge, be curious, and happy coding!

Acknowledgments:

I would like to thank Dr. Kwake and Dr. Zeng for invaluable feedback and support during the writing and reviewing process.

Their insights and suggestions significantly improved the clarity and accuracy of the content.

Note: All images used in this textbook were copied from the royalty-free repository Pixabay at https://pixabay.com/

CHAPTER 1

Introduction

HUSSAM GHUNAIM

Learning Objectives

At the end of this chapter, you will be able to

- Describe how computers work.
- Describe how computers run programs.
- Differentiate between High-level and Low-level programming languages.
- Explain what algorithms are.
- Explain the basic structure of C++ code.
- Differentiate between syntax, logical, and run-time errors.

HOW DO PROGRAMS WORK?

HARDWARE AND SOFTWARE

Computer systems are made of hardware and software components.

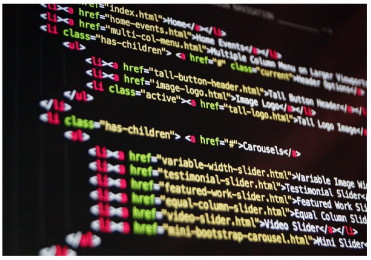


Figure 1: (a) Hardware (b) Software

Hardware

Computers have various types and purposes. Figure 2 shows some examples. A computer that is normally used by only one person at a time is called a Personal Computer PC. However, Servers and Mainframes are larger machines that can be used by many users at the same time. When a number of computers are connected so that they may share resources and information, they are called a network.

Figure 2: Computers' Types.

In general, computers have five major hardware components:

Input Devices: This group includes any device used to read data into computers, like keyboards, microphones, scanners, touchpads, and many others.

Output Devices: This group includes any device used to output data from computers, like screens, printers, speakers, and many others.

CPU (Central Processing Unit): This device is responsible for running all types of programs on a computer. You can think of the CPU as the brain of a computer, figure 3 (a).

Main memory RAM (Random Access Memory): The CPU can not do anything without the help of RAM, figure 3 (b). For a CPU to be able to run a program, it must first load the program code into RAM along with all needed data.

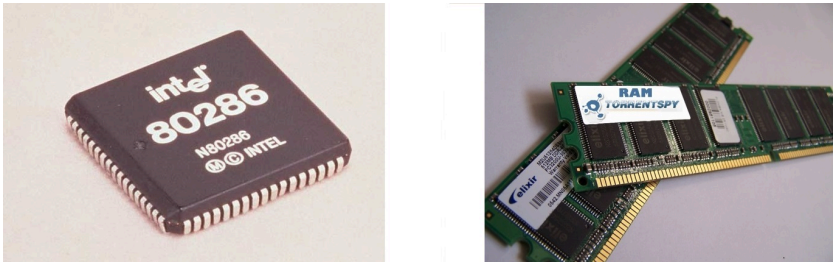


Figure 3: (a) CPU (b) RAM

Secondary Memory Devices: Although RAM is very important to run programs, it saves programs' code and data temporarily. This means that when a computer is turned off, all information in RAM will be lost. Here comes the need for secondary memory devices to save information permanently. Examples are hard drives, soft drives, flash drives, CDs, DVDs, and many others, figure 4.

Figure 4: (a) Hard Drive (b) Flash Drive (c) DVD\CD

Software

Software consists of a set of programs written to control computer operations. Broadly, software falls into two main categories: operating systems and applications. Operating systems are specifically designed to manage computer operations and oversee all applications running on the system. When you power on your computer, the first software to load into RAM and execute is the operating system. Subsequently, other applications load and run.

Notable examples of operating systems include Windows, MacOS, and Linux. Meanwhile, popular applications encompass Microsoft Office, Adobe products, internet browsers, favorite games, and more.

Software is developed using programming languages to write code that can be understood and executed by computers. Generally, programming languages can be classified into two major categories: High-Level and Low-Level programming languages.

HIGH-LEVEL AND LOW-LEVEL PROGRAMMING LANGUAGES

High-Level Programming Languages

This type of programming languages allow software developers to write computer instructions in a language similar to human languages. In Example 1 code below, you can read and guess what each line of code will do when executed by a computer. Popular examples of high-level programming languages are C++, Java, Python, and many others.

Low-Level Programming Languages

Conversely, this type of programming languages are written in a language that is closer to what computers can understand, but it is harder for people to use. Figure 5 shows an example of such languages called Assembly language. In the early days, developers had to use this type of programming languages as they were the only type available. Later, it became evident that it is better to develop high-level programming languages that are easier for everyone to use to write code.

```
push   %rbp
mov    %rsp,%rbp
sub    $0x30,%rsp
movl   $0x0,-0x4(%rbp)
movl   $0x0,-0x8(%rbp)
lea    -0x1c(%rbp),%rax
mov    %rax,%rsi
mov    $0x0,%edi
   R_X86_64_32
call   27 <0x27>
   R_X86_64_PLT32
movl   $0x1,-0xc(%rbp)
jmp    8e <0x8e>
lea    -0x20(%rbp),%rax
mov    %rax,%rsi
mov    $0x0,%edi
   R_X86_64_32
```

Figure 5: A snippet of assembly programming language code.

HOW DO PROGRAMS RUN ON COMPUTERS?

The story starts with developers writing code for a program, such as Windows Word or probably your favorite Internet browser. This code will be mainly written in a high-level programming language. But we already know that this code can not be understood and run by a computer. Therefore, we need to translate this code into a machine language that computers understand and can run. This process is really complex and has many details. For now, it is adequate enough to understand the basics of it, Figure 6 depicts the major steps in this process.

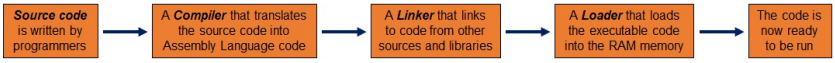


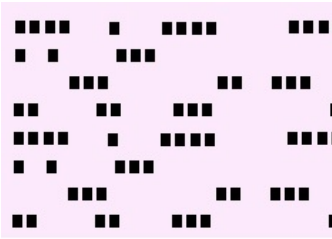
Figure 6: The preparation steps needed to run code on computers.

The process of preparing programs to run on computers starts with a compiler that translates the source code of a high-level language into an intermediate code, usually called Object Code. Then comes the job of a linker that links our program object code with other needed code written and compiled by other developers, usually called libraries. Lastly, a loader loads all necessary code into RAM, which makes the program ready to run.

HISTORY OF PROGRAMMING LANGUAGES

The early start of computers can be traced back to the eighteenth century. At that time, only mechanical calculators existed, see figure 7(a). In a later era, punched cards were used as a storage medium to store programmed instructions, figure 7 (b). Some notable names in this field are Charles Babbage, who designed the first programmable computer which also known as the Analytical Engine. Ada Lovelace wrote the first program for Babbage's machine.

Figure 7: (a) Mechanical calculators



(b) Punched cards

High-level programming languages that were capable of handling complex operations did not come into existence until the 1950s. These languages were designed for specific fields. For example, COBOL specialized in business applications, FORTRAN in scientific computing, and BASIC in educational applications.

Programming languages are a dynamic field and are constantly evolving. New languages are developed to meet the challenges and needs of various industries. The **C** programming language was developed in the 1970s to support both low-level and high-level features for writing operating systems and other system software. Object-Oriented Programming (OOP) languages such as Smalltalk and C++ were developed in the 1970s and 1980s to allow programmers to write more complex software applications. The 1990s saw the birth of a new programming paradigm known as scripting languages. Some examples are Perl and Python. Rust, Julia, and Swift are another set of languages that are designed to address specific needs such as system programming, scientific computing, and mobile applications.

WRITING PROGRAMS TO SOLVE PROBLEMS

It is important to note that every program is developed to solve a problem or to meet a need or requirement. Solutions to problems are usually called algorithms. **An algorithm can generally be defined as a series of steps that must be carried out to solve the problem at hand.** Algorithm writing is usually the first phase in the software writing lifecycle, which includes analysis, design, implementation, and testing.

ALGORITHMS

The best way to explore how algorithms are used to solve problems is through examples.

Example 01

Let's start with an example that everyone is familiar with, which is calculating the average of a set of numbers. Before checking the provided answer and as a practice, write in detail the series of steps that you would need to find the average. As you already know, computers need detailed instructions to do anything useful.

Answer:

1. Read all the given numbers
2. Add all the given numbers and find their total
3. Divide the total by how many numbers we have
4. Print out the average

The following section shows how algorithm steps can be translated into code.

CONVERTING ALGORITHMS INTO PROGRAMS

This section is not about introducing C++ coding; rather, it is just a simple example to show how algorithms can be converted into code. Throughout this book, sometimes line numbers are provided for easier reference, they are not part of the code. The first 4 lines are the standard way to start C++ code. More on that in Chapter 2. Lines 6, 7, 8, and 9 are used to read three integers (whole numbers) from the keyboard. These lines are to execute the first step in our algorithm described in the previous section. Then, line 10 finds the total of these entered numbers, that is, step 2 in the algorithm. After that, line 11 finds the average, step 3. Finally, line 12 prints out the result, hence, step 4.

```
1   #include <iostream>
2   using namespace std;
3
4   int main()
5   {
6       int number01, number02, number03;
7       double total, average;
8       cout << " Enter three numbers to calculate their average ." << endl;
9       cin >> number01 >> number02 >> number03;
10      total = number01 + number02 + number03;
11      average = total / 3;
12      cout << " The average = " << average << endl;
13
14      return 0;
15  }
```

Code_Snippet 1: Code for example 1.

COMMENTS STYLES

One essential skill that is usually overlooked by students and first-time programmers is writing meaningful comments. If you look again at the example described earlier, you might say it is clear and easy to understand. This is true for now. But what will happen after a while? Do you think you will be able to remember all the

details? Further, in the software industry field, it is common for many people to work together to develop the same program. This means developers need an effective way to communicate and exchange information. One of the methods to do that is through writing comments. Therefore, it is one of this book's objectives to help students build up the habit of writing comments whenever it is desired or needed.

Below is the revised code with suggested comments that would help in explaining the code to someone who is not familiar with it. Read the code and try to come up with your own comments that you think it is helpful to add. Note that in this example, we used two comment styles. We used **//** for single-line comments and **/* */** for multiple-line comments.

```
/*
 * Author: Your Name Goes Here.
 * Email Address: you@domain_name
 * Description: This program reads three numbers from the console and
 * calculates their average. Then prints out the result to the console .
 * Last Changed: July 10, 2023
*/

#include <iostream>
using namespace std;

int main()
{
    int number01, number02, number03; // declaring three integer variables.
    double total, average; // these variables have double type.
    cout << " Enter three numbers to calculate their average ." << endl;
    cin >> number01 >> number02 >> number03; // this statement reads three numbers from the console.
    total = number01 + number02 + number03; // students can practice writing their own comments here!
    average = total / 3; // ..... and here!
    cout << " The average = " << average << endl; // ..... and here!

    return 0;
}
```

Code_Snippet 2: Code for example 1 with comments.

OBJECT-ORIENTED PROGRAMMING (OOP)

Programming languages are a fast-evolving field of computer science. One of the remarkable advancements is a developing methodology known as Object-Oriented programming (OOP). Rather than designing programs as a set of instructions, in OOP,

a program is designed as a set of objects. Each object has its own data and algorithms to handle the data. This design methodology has proved advantageous over other methodologies by providing the following characteristics:

- **Encapsulation**: Known as an information-hiding technique that improves the security of the code, in addition to adding simplicity for developing the code.
- **Inheritance**: Objects can use code written in other objects, which can greatly simplify the code-development process.
- **Polymorphism**: This is a technique that can allow writing a code once, while it is possible to behave differently in various scenarios.

Students need not worry if those concepts are not clear, as they will be discussed again in due course.

LOOKING INTO A SIMPLE C++ PROGRAM

Code_Snippet 1 shows that any typical C++ program includes several parts. In the following subsections, a brief description is given for each of these parts to get you started writing code. More details will be given as they become needed.

KEYWORDS/RESERVED WORDS

Keywords or reserved words are a set of words used by the C++ system itself. This means you cannot use these names in ways different from their original definitions. Some examples are **int**, **double**, and **return**. Appendix A has the full list of these keywords.

As you might already have noticed, we can not do anything useful without being able to read and write data. These operations are

really complex operations to be done. Luckily, we do not need to have deep knowledge to read or write data, as C++ generously provides us with a large set of predefined classes that can do a lot of work for us. In Code_Snippet 1, ***cout*** and ***cin*** are called stream objects. While ***cout*** is used to send data to the computer console, ***cin*** is used to read data from the computer keyboard.

IDENTIFIERS AND NAMING RULES

Reading and writing data dictate the need to have a place in the computer memory to save the data; otherwise, we risk losing it. RAM is the main memory used to run programs and provides space for them to save any data they need. RAM memory locations are represented as binary or hexadecimal numbers. Binary numbers are made from digits that can hold either a zero or a one, which is known as a ***bit***. Every 8 bits consists of a ***byte***.

Once again, we are lucky not to refer to RAM location addresses using these complex and hard-to-remember numbers. Instead, we can create labels to refer to any needed memory location. These labels are formally known as variable names. In Code_Snippet 1, we have created five of those variables. Three variables to save the values of the numbers, namely, ***number01***, ***number02***, and ***number03***. Another variable to save the sum of these numbers is called the ***total***. Finally, the ***average*** variable is used to save the calculated average.

You can create and name variables as you wish. However, you must always create meaningful and descriptive variable names. Additionally, there are a couple of restrictions that you have to follow. Variable names cannot start with a number. They must start with a letter or underscore. Variable names can only contain letters, numbers, and underscores. Spaces are not allowed in variables' names.

Exercise 01

For each of the following variable names, determine whether it is an acceptable name or not. Give reasoning for your choices: (answers are provided at the end of the chapter.)

1. 12 calculate
2. First_Name
3. myLastName
4. thisIsIt
5. total-score
6. final&GPA
7. x

OPERATORS

Appendix B has a list of operators available in C++. In this section, we will only discuss the operators used in the provided Code_Snippet 1. Others will be introduced while you are reading through the book. In Code_Snippet 1, the following lines have three operators, the addition **+**, the division **/** and the assignment **=** operators:

```
total = number01 + number02 + number03 ;
average = total / 3;
```

From these two lines, you can deduce that operators are called as such because they operate on something, usually called operands, to produce some sort of a result. The addition **+** adds its operands' values, and the division **/** divides its operands. The assignment **=** operator assigns the value resolved from the right side to the variable name on the left side. That is, the assignment operator is always executed from right to left in opposition to most other operators that are executed from left to right, like the addition operator and all other similar arithmetic operators.

The following two lines have different types of operators, **<<** is called the insertion operator, while the **>>** is called the extraction operator.

```
cin >> number01 >> number02 >> number03 ;
cout << " The average = " << average << endl ;
```

As you can tell from the code, the extraction operator extracts (which means reads) data from the input stream ***cin***. In Code_Snippet 1, we are using the extraction operator to extract 'read' three numbers from the keyboard and save them into the provided three variable names on the right side of the extraction operator. On the opposite of the extraction operator, there is the insertion operator that inserts whatever is provided on the right side into the output stream ***cout*** given on the left side. In Code_Snippet 1, we are sending to the output stream the string **"The average = "**, the value stored in the variable name ***average***, and the object ***endl*** that inserts a new line. We can insert a new line by using \n in any quoted text, like this ***cout << "Hello!\n"***. Note that in C++, the semicolon ***;*** is used to end every line of code.

INCLUDING LIBRARIES

Computers are complex systems, and operating them requires sophisticated software. In the early days of programming, developers had the unfortunate task of writing all the code they needed from scratch. This practice persisted until it was recognized that different programs often require a common set of operations. For instance, operations such as reading and writing data are essential in all programs. The realization that certain operations could be developed once and reused across multiple programs led to the creation of what we know as libraries. These libraries contain collections of useful operations that many programs can utilize.

In Example 01, we needed to read from the keyboard and write to the console. Rather we developing our own streams to do that, we simply used the predefined streams included in the C++ library called **<iostream>** library. By doing so, we saved ourselves a great deal of work and technical complexity. Other libraries will be needed while you are progressing through this book. For example, to be able to create variables that are able to store entire sentences, you need to include a library called **<string>**. The following include directives do that.

```
#include <iostrem>
#include <string>
```

As libraries are usually huge, there is always a possibility of name conflicts of their contents. Therefore, in every program that you will write in this course, you must specify what namespace you want to use. The following code is called a ***use*** directive:

```
using namespace std;
```

COMMON SYNTAX ERRORS WITH SPECIAL CHARACTERS!

Writing code can contain some confusing characters that might look similar but are entirely different. You need to find those characters on your system's keyboard, as they might be located differently based on your system's manufacturer and model.

- Forward slash **/** and backward slash ****
- Underscore **_** and dash **–** . The dash character is used for the subtraction operation and to denote the negative sign as well.
- Single quote **'** and double quote **"** and the tick character **`**. The tick character, sometimes called the Grave Accent character, is usually found on the top left corner of a standard keyboard. Using two single-quote characters instead of one double-quote character will produce

errors, although they might appear to you as the same!

- Colon `:` semi colon `;` comma `,` and dot `.` characters might be mixed up by first-time programmers. Therefore, you have to pay more attention when you type your code.

OFFLINE AND ONLINE COMPILERS

When you start writing code, you have to make a choice whether to use an online compiler or an IDE. IDE stands for Integrated Development Environment. IDEs have become popular for their many features that make writing code easier and more efficient. In this book, we will use Visual Studio IDE for two reasons: it is powerful and freely available to install and use on your system. In addition, readers are free to use one of the many available online C++ compilers. They are popular for their simplicity and free users from having an IDE installed on their systems. This **`link`** takes you to a website that gives a nice comparison between top IDEs and online compilers. The projects at the end of this chapter will provide you with detailed help to install Visual Studio IDE and run your first C++ program. Additionally, it will show how to use some online compilers.

FIND AND FIX ERRORS IN YOUR PROGRAMS

Writing code dictates that you fully understand the syntax rules and follow them literally. In practice, it is extremely rare for someone, no matter how experienced they are, to write code free of syntax errors. That is, it is an essential skill whenever you write a piece of code to test it thoroughly until you are pretty sure it is free from bugs. Bugs are an informal term that means errors in code. Historically, Grace Hopper discovered the first documented computer bug.

Syntax rules are the rules that say how we should write code.

In the example discussed in this chapter, we followed all required syntax rules to declare variables, read numbers from the keyboard, and then calculate the average. If you tried to input the code yourself, you might get error messages from the compiler. IDEs are very helpful in discovering those bugs. This type of error is called a syntax error, which is relatively easier to find and fix compared to other types of errors discussed in the following sections.

When you compile your code without getting any error messages from the compiler, this means your code is free from syntax errors. However, this does not guarantee the code will run correctly. There are other error types that compilers can NOT detect, called logical and run-time errors.

LOGICAL ERRORS

Logical errors are caused by programmers when they misunderstand the purpose of the program. For example, in our earlier example in this chapter, we had a code to calculate the average of a set of numbers. If you used a wrong equation to do that, the compiler will not complain, however, the results will be wrong.

```
total = ( number01 * number02 * number03 ) ; // you mistakenly multiplied all numbers instead of adding them!
average = total / 3;
```

Another popular logical error is declaring the wrong type for a variable. In Code_Snippet 3, the result of a division operation is saved as an integer type, which causes the fraction part of the result to be lost! In this example, instead of having the result as 2.67, we get 2, as shown in Figure 8!

```cpp
#include <iostream>
using namespace std;

int main()
{
    int number01, number02, number03, total, average;
    cout << " Enter three numbers to calculate their average ." << endl;
    cin >> number01 >> number02 >> number03;
    total = number01 + number02 + number03;
    average = total / 3;
    cout << " The average = " << average << endl;

    return 0;
}
```

Code_Snippet 3: The code with a logical error.

```
 Enter three numbers to calculate their average .
1
3
4
 The average = 2
```

Figure 8: Wrong result caused by a logical error in the code.

RUN-TIME ERRORS

This type of error only occurs when you run the code. An example is when the code has a statement to divide by zero. As we all know, division by zero is not defined and cannot be calculated. Therefore, the execution of the program will be terminated and show an error message.

In the previous section example, if you changed the statement that calculates the average as below, the code will cause a run-time error!

```cpp
average = total / 0;
```

HOW SHOULD YOU TEST YOUR CODE?

Testing code is a fundamental skill that every programmer must master. In the software industry, many testing methodologies have been developed solely for this purpose. In this book, however, testing will not be presented as an independent topic; rather, it is going to be emphasized and encouraged. Students will be repeatedly encouraged to test their code while working on exercises and projects.

The major theme of the testing that they should follow is incremental testing. This means that rather than waiting until finishing writing the whole code, which probably can be dozens or even hundreds of lines of code, to start testing, incremental testing will be far more efficient. Imagine that after writing a lot of code, you received an error message from the compiler (that is a syntax error), or even worse, a logical or runtime error. Then, you would start wandering around your code trying to find that error. Certainly, you will spend a great amount of time trying to fix the code. However, if you follow our advice, you should start testing after writing a couple of lines of code. In case you get an error warning, it will be relatively easy to find and fix that error.

HOW MUCH TESTING IS ENOUGH?

When it comes to testing code, there isn't a specific number of tests that you must run. However, a good rule of thumb is to keep testing until you're confident that your code is error-free. The concept of "sufficient testing" is relative, but for every program you write, you should run multiple tests to cover all possible input scenarios.

Consider the following guidelines for effective testing:

1. **Input Domains:** If your code expects various types of input (such as numbers, strings, or characters), make sure your tests cover all of these input types.

2. **Faulty Input:** Be deliberate about testing faulty input. What happens if the user provides unexpected data? Your code should gracefully handle such scenarios, whether it's by providing informative error messages or taking appropriate actions.
3. **Comments and Console Messages:** Writing lots of meaningful comments and printing meaningful messages to the console are essential skills you must use while testing your code.

WRAP UP!

This chapter explored the basics of computers, software, and hardware. The main objective was to provide students with a better understanding of how code is written and executed within the hardware. Algorithms, C++ code structures, and preliminary information on keywords, operators, identifiers, and libraries were discussed. Additionally, this chapter emphasized the importance of thorough testing, including handling syntax, logical, and runtime errors.

ANSWERS KEY

Exercise 01:

1. Not acceptable because it starts with a number.
2. Acceptable
3. Acceptable
4. Acceptable
5. Not acceptable because it uses a not allowed special character. The only acceptable special character is the underscore.
6. Not acceptable because it uses &.

7. Acceptable, though it is not recommended because it tells nothing about its purpose.

END OF CHAPTER EXERCISES

1. In your own words, describe what hardware and software are.
2. Give examples of input and output devices other than those mentioned in this chapter.
3. What is the main function of RAM, CPU, and Hard drives?
4. What is the difference between high-level and low-level programming languages? Give an example for each.
5. Briefly explain the function of a compiler, a linker, and a loader.
6. What were punch cards used for?
7. In your own words, describe what algorithms are.
8. Write an algorithm to find out the total cost of your groceries.
9. Write an algorithm to find out whether a student is passing or not the computer science course based on their grades.
10. Why is it important to write comments within the code?
11. How can you insert comments in C++ code?
12. What is the use of ***cout*** and ***cin*** objects?
13. What are the uses of the variables? What rules do you have to follow when creating variables' names?
14. What are the uses of **<<** and **>>** operators?
15. In your own words, define what libraries are and why they

are included in C++ coding.

16. What does IDE stand for? What IDE will be used in this book? Why?
17. Find the syntax errors in this code:

```
#inlcude <iostream>;
using name_space std;

int main() {
    cout >> 'Hello, world !'.

    return 0:
}
```

18. What errors can a compiler not find?
19. In your own words, describe the three different types of errors: syntax, logical, and run-time errors.
20. Why should you test your code? How many tests, on average, should you run on your code?

PROJECTS

Project 1: The goal of this project is to get you to start using Visual Studio to write C++ code. All provided links are tested at the time of writing this book. If any link becomes inaccessible, you are advised to search the web for the updated links.

The official Microsoft Visual Studio page (**click here**) provides two links to download the Visual Studio installer for Windows and Mac users. For either platform, three options are available: Community, Professional, and Enterprise. Choose the Community option as it is free and yet powerful enough to learn to code in C++ effectively and

efficiently. Below are two suggested YouTube videos that can guide you through the installation process. However, you are advised to search for the most updated help available online.

- **Windows users.**
- **Mac users.**

After installing Visual Studio, create your first project as follows:
Step 1: Start Visual Studio and create a new project. Name the project as firstProject.Proj1
Step 2: Create a new file and rename it as yourName.cpp. Usually, you need only to submit the cpp file for grading, NOT the entire project folder.
Step 3: In the yourName.cpp file, add the following code. Pay close attention to the capitalization of the words. C++ is a case-sensitive language, which means words like myname, myName, MYNAME, and MyName are all recognized as different words.

```
1   #include <iostream>
2   using namespace std;
3
4   int main() {
5       cout << "My name is .......... " << endl;
6       cout << " This is my first project in C ++!" << endl;
7       return 0;
8
9   }
```

Step 4: Fill in your name in line 5. Run the code and examine the output.
Step 5: Writing comments is an essential skill all programmers must master. Thus, add the following comments at the top of the file. Replace the dots with your name and other appropriate details.

```
/*
* Author: Your Name Goes Here.
* Email Address: you@domain_name
```

```
* Description: ....................
* Last Changed: ....................
*/
```

Step 6: Change the keyword *cout* to *Cout* in line 5. Try to run the code and write your comments on what happened and why.

```
Cout << "My name is ........ < < endl ; // The Cout with capital C ( run/ did not) run because .......
```

Step 7: Change back the *Cout* to *cout* again to fix the error.

CHAPTER 2

C++ Fundamentals

HUSSAM GHUNAIM

Learning Objectives

At the end of reading this chapter, you will be able to:

- Write your first program
- Understanding data types in C++
- Implement branching and loops in your code
- Describe best practices for writing code

INTRODUCTION

Writing code always goes hand in hand with data. We can not think of writing any useful code without being able to handle data. Handling data in real-life scenarios usually requires data structures that you will learn about in higher-level courses. However, as this

is your first course in programming, we will start with the smallest memory units that can store data, called variables and constants.

Variables vs. Constants

As discussed in Chapter 1, to be able to run code, it must be copied into the computer's main memory, RAM. Running code needs to store data to operate on. Reserving memory locations is achieved by declaring our intentions as follows:

```
type variable_name;
```

Types will be discussed in more detail later in this chapter. For now, we can use the**int** type, which is the abbreviation for integer, or whole numbers. This specifies what type of data can be stored in this particular variable. In this example, we are saying we want to save only whole numbers or integers in the variable. Of course, we need more than that to write useful programs. After the type, we must choose a meaningful, easy-to-understand, and remember variable name, as in this example.

```
int sum;
```

To better understand how variables are created and used, Figure 1 depicts the computer's main memory, RAM. When you declare a variable, a label will be created with the variable name that points to a specific location in the RAM. Remember that RAM looks like a huge array of memory locations that can be used by all running programs, including the operating system itself. This depiction is not entirely accurate, but it is a good starting point. Later, when we discuss further the types, we can elaborate more on how data is stored in RAM.

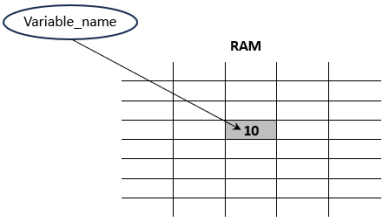


Figure 1: Assigning a memory location for a declared and initialized variable.

After declaring (that is, creating) the variable, we can assign a value to it, like in this statement. This operation is called initializing the

variable:

```
var = 10;
```

Variables, as the name suggests, have values that can be changed as many times as we wish. So in the previous assignment, we can assign 0, -100, or any desirable integer value to the declared variable. Every time we assign a new value, the older value will be overwritten and lost forever! Therefore, you must test your code many times to be sure the code produces the expected results.

On the other hand, constants are mostly declared in the same way, with one addition, which is using the keyword **const,** which prevents changing the value after the initial assignment. If you attempt to change the value of a constant, an error will be generated. Note where the **const** keyword should be written in the following statement:

```
const type constant_name;
const int MY_CONSTANT = 1100;
```

It is a common convention that constant names are capitalized to easily distinguish them from other names. Remember, two operations can be applied to both variables and constants: ***declaration*** and ***initialization***. In this example, we combined the declaration and initialization operations in one line of code rather than two separate lines.

Expressions and Operators

This topic will be discussed in more detail in Chapter 3, but for now, let us write basic expressions to start with. Operators are special symbols used to perform specific operations on one or more operands, which are often variables.

Example 01

Study the provided code that shows how to write some basic

expressions. Additionally, note how we can add comments in C++. There are two ways to do this. For single-line comments, we use `//`. For multiple-line comments, we use `/* */` . Comments are ignored by the compilers and hence they are not considered part of the code. However, they're extremely important for making the code understandable to humans. This is especially true in large projects where multiple developers need to understand each other's code.

```
/* declaration and initialization
   of variables can be accomplished
   in the same line!  */
int a = 10;
int b = 20;

int sum = a + b; // Addition, sum will be 30
int diff = a - b; // Subtraction, diff will be -10
int prod = a * b; // Multiplication, prod will be 200
int quot = b / a; // Division, quot will be 2
```

NOTE: For division, we use the operator `/` , which is called a forward slash. This is different from the other similar slash `\` , which is called a backward slash.

Exercise 01

1. Declare two integer variables, num1 and num2. Initialize num1 with 15 and num2 with 5. Then, calculate the sum of both numbers and store the result in the variable addition.
2. Using the variables in exercise 1, calculate the difference, multiplication, and division. Store the results in variables named subtraction, multiplication, and division, respectively.

3. Declare a constant integer named MAX_VALUE and initialize it with the value 1000.
4. In all previous exercises, write appropriate comments to explain the code and the expected output.

Writing Your First Program

As discussed in Chapter 1, you need either an IDE or an online service to compile and run C++ code. Whatever choice you make, you must be able to run your code for learning and testing purposes.

Example 02

Run the provided code and study how the code is written.

```
#include <iostream> // Include the iostream library
using namespace std;

int main() {
    int num = 10; // Declare an integer variable and initialize it with 10
    cout << "The number value is: " << num << endl; // Print the number
    return 0;
}
```

The first two lines will appear in almost all your programs in this book. The first line is used to include the **<iostream>** library. Libraries contain prewritten and tested code by professionals. They contain a lot of useful code that is ready for us to use. For example, to print out a message to the console, we use the stream **cout**. We do not know how exactly this stream is implemented. However, all that we need to know is how to use it, relieving a huge amount of work off our shoulders! Therefore, you will learn how to include a couple of more libraries as you proceed in this book. The second

line is used to specify what namespace we need to use. This is important to avoid any conflicts in naming. As libraries can be huge, two or more libraries may have different implementations but with the same name, which can cause conflict in your code. Later, you will learn how to create your own namespaces.

Line 4 defines the **main** function in the code. You can think of functions as containers that contain the code. This function is called main because it is the first function to execute whenever you run your code; that is, it must appear only once in every program. No more than one **main** function in the same application is acceptable. The body of the **main** function starts with open curly braces `{` and ends with the close curly braces `}` in line 8. All the **main** code must be inside those symbols.

Line 5 has the declaration and initialization of the variable **num** as explained earlier in this chapter. Line 6 prints out a message to the console. We do this by using the stream **cout** that prints everything sent to it on the console. **cout** works with the insertion operator **<<**. The message we want to print on the console is surrounded by quotation marks `" "` to differentiate it from the code. As you can note from line 6, **cout** stream can accept a sequence of pieces to print out on the console. In this example, after printing the message surrounded by the quotation, it prints the value stored in the variable **num,** followed by a blank line created by the object **endl**, which stands for 'end of the line'. Notice that when variable names appear in the **cout** line of code, the value stored in that variable will print out to the console, not the variable name itself.

Every statement in C++ must end with a semicolon. Line 7 has the **return** statement. For now, it is sufficient to understand that some functions must return something when they finish running. In this example, the main function will return `0` when it finishes running.

In many cases, we want users to enter some data. This makes programs interact with the users. We use the **cin** stream that reads users' input from the keyboard. **cin** works with the extraction

operator **>>**. Pay attention to not confusing extraction and insertion operators by noticing the direction in which the arrows point.

Example 03

Run the provided code and study how the code is written.

```
1   #include <iostream>
2   using namespace std;
3
4   int main() {
5       int num1, num2, sum;
6       cout << "Enter two numbers: ";
7       cin >> num1 >> num2; // Taking input from the user
8       sum = num1 + num2; // Adding the numbers
9       cout << "The sum is " << sum << endl; // Printing the sum
10      return 0;
11  }
```

To simplify the code, **cout** and **cin** streams allow for a sequence of operations to be done in the same line of code. For example, in line 7 of the code, **cin** is used to read values for two variables rather than writing two separate lines. This becomes possible because the **>>** operator reads input until a whitespace is encountered, effectively using spaces to separate different values. Similarly, line 9 shows how we can print out several pieces of data using the **cout** stream.

Exercise 02

Write a program that asks users to enter the distance in miles and time in hours, then it calculates the speed using this formula: **Speed = Distance / Time**. The program should print the result as follows: The speed of the vehicle is .... miles/hour!

Note: Because this program calculation might involve numbers with

decimals, using **int** as a data type will be erroneous. Therefore, you must use **double** as the declared variables' type.

Exercise 03

Write a program that asks users to enter the radius of a sphere and then calculates its volume using this formula: **sphere volume = (4.0/3.0) * PI * radius^3**. The value of the mathematical constant PI must be declared as a global constant and initialized to 3.14. The final result should be printed out to the console as "The volume of the sphere is ..... cubic units".

NOTE: In the sphere volume formula, we put the fraction as 4.0/3.0. This is important to enforce the result to be of type **double**, otherwise, if we put 4/3, the results will be 1 instead of 1.33!! This error is known as the ***integer division error***. If we used integers to perform division, the answer would be an integer as well, causing the loss of the fraction part from the result. To enforce the result to be **double**, that is, it can have a fraction part, we must write either the denominator or the numerator (or both) as a **double** number. This topic will be explained further in this chapter when we discuss type casting.

Exercise 04

Write a program to calculate simple interest. The program should ask users for the principal amount, the rate of interest entered as a whole number, and the time in years. It then calculates the simple interest using the formula: **Interest = (Principal * Rate * Time) / 100** and displays the result.

Escape Characters

Sometimes it is useful to add basic formatting to the printed text, like adding a new line , printing the quotation with the output text, or adding more horizontal space, and so on. To do that you can use escape characters where the compiler treats them differently. Any character sequence that starts with a backslash \ and is followed by another character (or digits) will make the compiler treat that combination as one special character, not as literal text. Table 01 shows how you can use some common sequence characters.

Table 01: Escape characters examples.

Escape Characters	What It Does	Example Code	Example Output
\\\\	Inserts a single backslash	cout << "Path: C:\\\\Program Files\\\\MyApp\n";	Path: C:\Program Files\MyApp
\'	Inserts a single quotation mark	cout << "It\'s C++ fun\n";	It's C++ fun
\"	Inserts double quotation marks	cout << "She said, \"Hello\"\n";	She said, "Hello"
\n	Starts a new line	cout << "Error:\nInvalid input\n";	Error: Invalid input

MORE ABOUT TYPES

So far, you have used two basic types, **int** and **double**. In this section, you will be introduced to more types used in C++. But before we start delving deeper into learning types, let us talk about why types are important. Misusing types will lead either to syntax errors or erroneous results. At the beginning of this chapter, we said that when you declare a variable, a location in the computer memory RAM will be reserved and assigned to that particular variable. The question now is how much memory is needed. The answer lies with the type chosen. Every type you use in the declaration statement will instruct the system how much memory is needed. Table 02 shows the memory size needed for some popular types used in this book.

Table 02: Some C++11 popular types.

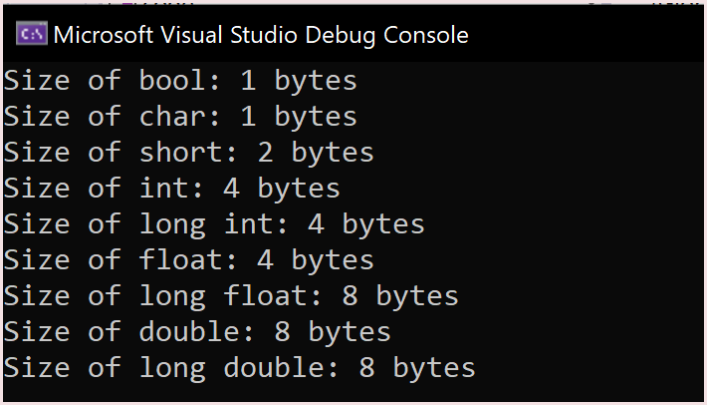
Type Name	Memory Used	Size Range
`bool`	1 byte	True or False
`char`	1 byte	-128 to 127 or 0 to 255
`short`	2 bytes	-32768 to 32767
`int`	4 bytes	-2147483648 to 2147483647
`long int`	4 bytes	-2147483648 to 2147483647
`float`	4 bytes	1.2E-38 to 3.4E+38
`long float`	8 bytes	2.3E-308 to 1.7E+308
`double`	8 bytes	2.3E-308 to 1.7E+308
`long double`	8 bytes	2.3E-308 to 1.7E+308

From the table, we can tell what data is possible to store in a variable with a particular type. Note that some types have similar sizes. This is because different compilers assign different sizes to each type. Type sizes can vary based on the system used, whether it be a 32-bit or 64-bit system. This can be problematic. Example 04 explains this problem further. In C++ 11 and later, a special type called ***auto*** was introduced. Basically, when declaring a variable with an ***auto*** type, you are leaving it to the compiler to decide what type to use.

Example 04

To find out the exact sizes of types on your system, you can use the function **sizeof**. Functions will be studied later, but for now, let us

think of this function as a utility that helps us to find out the types' sizes. Run the provided code on your system and compare the results with those I obtained on my system.




```cpp
#include <iostream>
using namespace std;

int main() {
    bool boolVar;
    char charVar;
    short shortVar;
    int integerVar;
    long int longIntVar;
    float floatVar;
    long float longFloatVar;
    double doubleVar;
    long double longDoubleVar;

    cout << "Size of bool: " << sizeof(boolVar) << " bytes" << endl;
    cout << "Size of char: " << sizeof(charVar) << " bytes" << endl;
    cout << "Size of short: " << sizeof(shortVar) << " bytes" << endl;
    cout << "Size of int: " << sizeof(integerVar) << " bytes" << endl;
    cout << "Size of long int: " << sizeof(longIntVar) << " bytes" << endl;
    cout << "Size of float: " << sizeof(floatVar) << " bytes" << endl;
    cout << "Size of long float: " << sizeof(longFloatVar) << " bytes" << endl;
    cout << "Size of double: " << sizeof(doubleVar) << " bytes" << endl;
    cout << "Size of long double: " << sizeof(longDoubleVar) << " bytes" << endl;

    return 0;
}
```

Types Compatibility

You need to pay extra attention when deciding what type to

choose. For example, in the following line of code, some compilers will issue a syntax error or at least a warning:

```
int variable = 3.93;
```

The problem here is a data type mismatch by trying to save a double number 3.93 into an integer variable. However, other compilers allow such an assignment, but with the consequence that the fraction part will be lost. That is, the stored number in the variable will be 3, not 3.93. This is very tricky because the compiler might run your code without issuing any syntax error, you will still get the wrong results, leaving you wondering what happened with your code! The opposite might not be that serious, though.

```
double varialble = 3;
```

Here, the number will be converted from 3 to 3.0, which works without issues, however, the rule of thumb is, don't mix types when working with variables!

Another example of type compatibility is **int** and **char**. **char** type is used to store characters; however, remember that all characters are actually encoded into numbers. That is, computers do not understand characters; they only understand numbers. That is, every character you can see on your keyboard is associated with a number. Figure 02 shows some of the characters encoding known as the ASCII encoding. As you can see from the table, every character has a code, including invisible characters like space, NULL, and many others. Therefore, the following code can run, although it is strongly unrecommended because this will make your code hard to read and confusing.

```
int variable = 'H';
char variable = 72;
```

Dec	Char	Dec	Char	Dec	Char	Dec	Char
0	NUL (null)	32	SPACE	64	@	96	`
1	SOH (start of heading)	33	!	65	A	97	a
2	STX (start of text)	34	"	66	B	98	b
3	ETX (end of text)	35	#	67	C	99	c
4	EOT (end of transmission)	36	$	68	D	100	d
5	ENQ (enquiry)	37	%	69	E	101	e
6	ACK (acknowledge)	38	&	70	F	102	f
7	BEL (bell)	39	'	71	G	103	g
8	BS (backspace)	40	(	72	H	104	h
9	TAB (horizontal tab)	41	)	73	I	105	i
10	LF (NL line feed, new line)	42	*	74	J	106	j
11	VT (vertical tab)	43	+	75	K	107	k
12	FF (NP form feed, new page)	44	,	76	L	108	l
13	CR (carriage return)	45	-	77	M	109	m
14	SO (shift out)	46	.	78	N	110	n
15	SI (shift in)	47	/	79	O	111	o
16	DLE (data link escape)	48	0	80	P	112	p
17	DC1 (device control 1)	49	1	81	Q	113	q
18	DC2 (device control 2)	50	2	82	R	114	r
19	DC3 (device control 3)	51	3	83	S	115	s
20	DC4 (device control 4)	52	4	84	T	116	t
21	NAK (negative acknowledge)	53	5	85	U	117	u
22	SYN (synchronous idle)	54	6	86	V	118	v
23	ETB (end of trans. block)	55	7	87	W	119	w
24	CAN (cancel)	56	8	88	X	120	x
25	EM (end of medium)	57	9	89	Y	121	y
26	SUB (substitute)	58	:	90	Z	122	z
27	ESC (escape)	59	;	91	[	123	{
28	FS (file separator)	60	<	92	\	124	|
29	GS (group separator)	61	=	93	]	125	}
30	RS (record separator)	62	>	94	^	126	~
31	US (unit separator)	63	?	95	_	127	DEL

Figure 02: ASCII encoding table.

Another example of type compatibility is **int** and **bool**. The type of **bool** accepts only two values, **true** or **false**. Those two values are encoded as 1 for **true** and 0 for **false**. Thus, the following lines of code are legal, though they should be avoided.

```
int variable = true;
bool variable = 0;
```

Exercise 05

Run the given code and find out what value is stored in each variable. Explain the results.

```
#include <iostream>
using namespace std;

int main() {
    int myInt = true;
    int myInt2 = false;
    int myInt3 = 'H';
    int myInt4 = 5.978;
    bool myBool = 0;
    bool myBool2 = -100;
    char myChar = 72;
    double myDouble = 5;
    return 0;
}
```

Types Casting

In some cases, it is necessary to explicitly change the type of a particular variable.

Example 05

Run the given code and check the correctness of the results.
We use the formula

$$\text{Fahrenheit } (^\circ\text{F}) = (\text{Celsius } (^\circ\text{C}) \times \frac{9}{5}) + 32$$

to convert temperature degrees from Celsius to Fahrenheit.

```
#include <iostream>
using namespace std;

int main() {
    double celsius = 25.0;
    double fahrenheit = celsius * (9/5) + 32.0;
    cout << "The temperature degree in fahrenheit = " << fahrenheit << endl;

    return 0;
}
```

While we expect the 25 °C to be converted to 77°F, we get the result of 57°F. The problem here is known as ***integer division,*** as explained in exercise 03. The easiest method to fix the problem is by writing either the denominator or the numerator (or both) as a **double** number. Another more formal method is using the casting syntax ***static_cast<the_desired_type>(variable)*** as follows:

```
double fahrenheit = celsius * (static_cast<double>(9)/5) + 32.0;
```

Change the code and check the correctness of the results again. You have to pay attention when using casting. The following statement still produces the wrong result. Why?

```
double fahrenheit = celsius * (static_cast<double>(9/5)) + 32.0;
```

The reason is that we put the fraction 9/5 inside the parentheses, the integer division is then performed before the casting takes place. Thus, the result will already be calculated as an integer division, and casting will not make any difference by then!

Casting can be done on various types, such as casting **int** to **char**, **bool** to **int** and vice versa.

The string Type

All the types we've studied so far are known as primitive types. This is because they are fundamentally capable of storing only a single piece of data, such as an integer, a character, and so on. However, to write more realistic programs, certainly, we certainly need data types that allow for handling more complex data. Later, you will learn about user-defined data types and classes. But for now, we can discuss one popular class type called string. This type allows storing a large amount of text. Additionally, it provides many powerful pre-defined functions that can be used to manipulate texts. To declare variables of string type, you need to include the ***<string>*** library.

Example 06

Run the provided code and comment on the outputs.

```
1   #include <iostream>
2   #include <string>
3   using namespace std;
4
5   int main() {
6       string firstName = "John";
7       string lastName = "Smith";
8       string fullName = firstName + " " + lastName;
9       int nameLength = fullName.length();
10
11      cout << "First Name: " << firstName << endl;
12      cout << "Last Name: " << lastName << endl;
13      cout << "Full Name: " << fullName << endl;
14      cout << "Name Length: " << nameLength << " characters!" << endl;
15
16      return 0;
17  }
```

When initializing strings, you must surround the string between double-quotations, as in lines 6 and 7. Line 8 shows how to combine several strings together. This operation is called concatenating strings and is performed using the + operator. Compilers are smart enough to tell what needs to be done with this statement. If you are using the + operator between two numerical values, it will be understood that the required operation is to add the two numerical values and return

their sum. However, if the + operator happens to be placed between two strings, it will be understood that the required operation is to concatenate the two strings into one string. If you try to place the + operator between a string and a numerical value, you will get an error, as this operator is not defined. Later in this book, you will learn how you can define your own operators. Additionally, line 8 shows that you can create strings without storing them in variables, ***" "***. Here, all we want to do is add a space between the first and the last names. And of course, space is a string.

In line 9, we used one of the member functions of the string class. For now, all you need to know is that a string is a complex data type that includes many components (that is, member functions) inside it. To use those member functions, you need to call them by their names using the dot notation to specify which string you want a particular function to be called upon, for example, ***fullName.length()***. In this statement, you are calling the member function ***length()*** specifically on the string variable ***fullName*** to find out how many characters are in this string.

Example 07

Run the provided code and explain the outputs.

```cpp
#include <iostream>
#include <string>
using namespace std;

int main() {
    string sentence1 = "Hello, C++ programmers!";
    string reversed(sentence1.rbegin(), sentence1.rend());
    cout << "The original sentence: " << sentence1 << endl;
    cout << "Reversed sentence: " << reversed << endl;

    string sentence2 = "Welcome to the world of C++ programming!";
    cout << "Position of 'world': " << sentence2.find("world") << endl;

    string sentence3 = "I love programming!";
    cout << "After replacement: " << sentence3.replace(7, 11, "C++") << endl;

    return 0;
}
```

To reverse a string, we can use the **reversed** function that uses two member functions of the string class, **rbegin** and **rend**. These two member functions are used to specify the beginning and the end of the string to be reversed. **find** is another useful function that finds the location of a substring in a larger string. In this example, the position is 15. Note here that in most cases in coding, counting usually starts from 0, not 1! The **replace** member function is used to replace part of a string with a substring. Again, pay attention to how numbers are chosen. The first number 7 denotes the position where the replacement should start, hence, counting starts at 0. The second number 11 denotes the number of characters that are to be replaced with the new string.

Exercise 06

Write a program to concatenate two strings without using the + operator. You need to do some research to find out how to use the string member function **append**.

BRANCHING AND LOOPS

Up to this point, the code we wrote so far was executed line by line sequentially. In some cases, we need to alter this linear execution. We can do this mainly by branching and loops. Branching can be done using `if..else` discussed in this chapter. Later in Chapter 3, we will discuss another branching technique called `switch`. There are several loops you are going to learn. In this chapter, we will cover `for` loops, `while` loops, and `do..while` loops. Later, in chapter 5, another variation of the `for` loop called the **for-each** loop will be discussed.
All of these branching and loop structures use Boolean expressions. In contrast to arithmetic expressions that can return any value, boolean expressions only return two values, **true** or **false**. The following example introduces you to the basic Boolean expressions that you need to understand before exploring this topic. In Chapter 3, you will be introduced to more operators and expressions.

Example 08

In boolean expressions, we can use these operators for any two variables a and b:

```
  a >
b       greater than

  a
>= b    greater than or equals

  a <
b       less than

  a
<= b    less than or equals

  a
!= b    is not equal

        equals (Note: this operator is
  a     different from the assignment =
== b    operator.)

 .
```

`if..else` Statements

The best way to learn about ***`if..else`*** statements is by discussing examples.

Example 09

Run the provided code and explain the output.

```
#include<iostream>
using namespace std;

int main() {
    int num = -10;
    if (num >= 0)
        cout << "The number is positive.";
    else
        cout << "The number is negative.";

    return 0;
}
```

The evaluation of the Boolean expression **num >= 0** depends on the value of the variable **num**. That is, if the expression evaluates to **true**, then the statement that comes after ***if*** will be executed, skipping the ***else*** part. However, if the expression is evaluated as **false**, then the ***else*** part will be executed skipping the ***if*** part.

To make the example more interactive, ask users to enter any number they wish, then the program will decide whether the entered number is positive or negative. You can do that by adding these statements above the ***if..else*** statement:

```
cout << "Enter any number and press 'Enter' button" << endl;
cin >> num;
```

Example 10

Run the provided code and explain the outputs.

```
#include<iostream>
using namespace std;

int main() {
    double num1, num2, sum, sub;
    char answer;
    cout << "Enter any two number." << endl;
    cin >> num1 >> num2;

    cout << "Enter + to add the numbers or - to subtract them." << endl;
    cin >> answer;

    if (answer == '+') {
        sum = num1 + num2;
        cout << "The sum of the two numbers is " << sum << endl;
    }
    else {
        sub = num1 - num2;
        cout << "The subtraction of the two numbers is " << sub << endl;
    }
    return 0;
}
```

In this code, the user is required to enter two numbers and then choose what operation they would like to operate on them, either addition or subtraction. If the user chooses **+** then this character will be saved in the variable ***answer*** and used in the boolean expression in the **if** statement where the expression ***answer == '+'*** will return **true** and execute the **if** part skipping the **else** part. Note here, we used the curly braces **{ }** to group more than one statement in both the **if** and the **else** parts. This is important; otherwise, only the first statement after **if** will be executed, ignoring the rest statements, which of course will result in wrong outputs. Compare this case with example 09, where we did not use the **{ }** because we had only one statement and did not need the grouping curly braces.

In the previous example, you might have noticed that if you enter any character other than **+** the **else** part will be executed. That is ok with us as long as we meet the program requirements. In Chapter 3, you will learn about nested **if..else** statements where you can specify more than just one condition.

Exercise 07

Write a program that asks users to enter two numbers. By using the ***if..else*** statement, the program should determine which number is greater than the other and print it out to the console. Numbers entered should be either integers or decimals.

Exercise 08

Write a program that applies a 5% discount on items' prices for members only using this formula

$$discountedPrice = itemPrice - (0.05 \times itemPrice)$$

The program should ask the user to enter the item price and then ask if the user is a member or not. If the user answers y for yes, then a new discounted price should be printed out to the console; otherwise, a message saying no discount is applied should be printed instead.

for *Loops*

The syntax of the for loop is as follows:

```
for(initialization ; boolean expression ; update){
      loop body
}
```

- ***initialization*** is the operation where we give the loop control variable an initial value. This control variable can be either declared in the loop header or somewhere else before the loop. This difference can have a significant consequence. In many compilers, including Visual Studio, if the control variable is declared inside the ***for*** loop

header, it will be considered as a local variable. This means it can be visible only inside the loop body and cannot be accessed outside it. You are strongly recommended to test the compiler that you are using, whether this fact holds or not.

- ***boolean expression*** is any expression that results in either **true** or **false** values.
- ***update*** is any operation that results in updating the loop control variable value.

Example 11

Run the given code and explain the results.

```
#include <iostream>
using namespace std;
int main() {
    int num;
    cout << "How many times do you want to print the welcome message?";
    cin >> num;
    for (int i = 1; i <= num; i++)
        cout << i << " : Welcome to your first programming course!" << endl;
    return 0;
}
```

The statement ***int i = 1*** declares and initializes the loop control variable ***i***. The loop continues as long as the expression ***i <= num*** evaluates to **true**. When this expression evaluates to **false**, the loop terminates. The statement ***i++*** increments ***i*** value by one after each iteration (conversely, the **--** operator decrements any variable value by one). At the start of the loop, ***i*** equals 1. If the user enters 5, then **num** equals 5, and the condition **1 <= 5** is **true**. After the first iteration, ***i*** is incremented to 2, and the condition **2 <= 5** is still **true**. This pattern continues until ***i*** equals 6, at which **6 <= 5** is **false**, and the loop terminates. During each iteration, the value ***i*** is printed to the console, followed by a colon and the welcome message.

Run the code with different user inputs to observe the pattern of loop iterations.

Note in example 11, we did not use **{ }** to surround the ***for*** loop body because the loop body has only one statement. However, whenever you need to add more than one statement inside the loop body, you must use the **{ }** same approach as in example 12.

Example 12

Run the given code and explain the results.

```
1   #include <iostream>
2   using namespace std;
3   int main() {
4       double grade, total = 0;
5       int counter;
6       cout << "Enter your quizzes grades to calculate the average." << endl;
7       for (counter = 1; counter <= 5; counter++) {
8           cin >> grade;
9           cout << "Quiz [" << counter << "] grade is " << grade << endl;
10          total = total + grade;
11      }
12      cout << "The average of your quizzes is " << total / (counter - 1) << endl;
13      return 0;
14  }
```

The loop control variable **counter** declared before the loop header. This is important because this variable is used to calculate the average after the loop is terminated in line 12. As an exercise, delete line 5 and declare the **counter** variable inside the loop header. When you do so, you will get an error because the scope of the **counter** variable becomes local to the loop body, thus, you cannot access its value. Another thing to note here is that the **total** variable is initialized in line 4. Again, if you do not do that, you will get an error generated by executing line 10. In line 10, we read the contents of the variable **total** and it to **grade**. And then save the new value back in the **total** variable. At the first iteration, assuming you did not initialize the **total** variable, the code in line 10 will try to read

the uninitialized variable value, which causes the error. The last detail to pay attention to is that on line 12, we divided the **`total`** over **`(counter - 1)`**. This is because the loop condition **`counter <= 5`** evaluates **`true`** until the **`counter`** value becomes 6, causing the loop to terminate. This, of course, will cause a logical error because the user entered only 5 grades, not 6. Therefore, we must subtract 1 from this variable to bring it back to the correct value of the number of entered quizzes' grades, which is 5.

Exercise 09

Write a program that asks the user to enter an integer number, then the program should print out all even numbers starting from 2 up to the number entered by the user. The program must use the **`for`** loop.

Exercise 10

Write a program that calculates the factorial of a non-negative integer number using this formula,

$$n! = n \times (n\text{-}1) \times (n\text{-}2) \times \ldots \times 3 \times 2 \times 1$$

That is, if n = 5, then

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

In math, the exclamation mark **!** is used to denote the factorial function. The program must use **`for`** loop. Test your code for several numbers to find out the largest number that your system can calculate

its factorial. On my system, using ***long long int*** data type ***n*** turns out to equal ***20***. You will get negative results, denoting an overflow occurred in your system, which means the calculations have exceeded your system's capabilities!

while *Loops*

for loops are generally easier to use whenever you know beforehand how many iterations you would need. However, in some cases, we do not know this information. In example 11, we asked the user to choose how many times they wanted to print out the welcome message to the console. Suppose now we want the user to be able to print as many as they want until to say stop without specifying the number of iterations. This can be achieved using a ***while*** loop.

Example 13

Run the given code and explain the results.

```
#include <iostream>
using namespace std;
int main() {
    char answer;
    cout << "Welcome to your first programming course!" << endl;
    cout << "One more? y/n: ";
    cin >> answer;

    while (answer == 'y') {
        cout << "Welcome to your first programming course!" << endl;
        cout << "One more? y/n: ";
        cin >> answer;
    }
    cout << "Good bye!" << endl;
    return 0;
}
```

A ***while*** loop operates by evaluating an expression, similar to ***for*** loops. The loop continues to run as long as this condition evaluates to

`true`. Once the condition evaluates to `false`, the loop terminates. It's crucial to ensure that the loop condition will eventually evaluate to `false`; otherwise, the loop will run indefinitely, resulting in an infinite loop. After the welcome message is printed for the first time, we ask the user if they want to print the message again. The user's response is checked in the ***while*** loop condition **`answer == 'y'`**. As long as this condition returns `true`, the loop will continue to repeat. When the user enters '**n**' (or any other character), the condition evaluates to `false`, and the loop terminates.

Example 14

Run the given code and explain the results.

```
#include <iostream>
using namespace std;

int main() {
    int secretNumber = 7;
    int userGuess;

    cout << "Welcome to the number guessing game!" << endl;
    cout << "Guess a number between 1 and 10: ";
    cin >> userGuess;

    while (userGuess != secretNumber) {
        cout << "Sorry, that's not correct. Try again: ";
        cin >> userGuess;
    }
    cout << "Congratulations! You guessed the number correctly." << endl;

    return 0;
}
```

This is a simple guessing game where the user keeps guessing a number until they guess it correctly. The loop condition compares the **`userGuess`** value to the preselected**`secretNumber`** value. While this condition evaluates `true`, the loop will keep running until the user guesses the correct number, causing the condition to evaluate

`false`. The code can be modified to let users guess characters or strings like places' names or people's names to make the game more fun.

Exercise 11

Modify the code in example 14 to guess words instead of numbers. Be creative with what can be guessed. At the end of this chapter, a sample solution is provided.

Exercise 12

Redo exercise 10 using **`while`** loops.

`do..while` *Loops*

In the loops studied so far, the loop condition is checked first. Then, depending on the result of this check, the loop might be executed or terminated. There is one special case that might be problematic. What if the loop condition evaluates to `false` at the very beginning of the loop? Naturally, this will terminate the loop immediately without any iterations. Generally, this can be acceptable and raises no issues, however, in the case that we want to guarantee that at least the loop will iterate once, then we must use the **`do..while`** loop. In this loop, the body of the **`do`** block will execute without any condition checking. The loop condition check comes after the **`do`** body. The following examples explain this special case loop.

Example 15

In example 13, I hope you noticed the annoying duplication of printing the welcome message, followed by reading users' input. Here, we can take advantage of ***do..while*** to improve our code readability and efficiency.

```
#include <iostream>
using namespace std;
int main() {
    char answer;

    do {
        cout << "Welcome to your first programming course!" << endl;
        cout << "One more? y/n: ";
        cin >> answer;
    } while (answer == 'y');

    cout << "Good bye!" << endl;
    return 0;
}
```

The code starts with a ***do*** block, which runs without checking any condition. In this example, that's exactly what we want. Right after the end of the ***do*** block, the ***while*** header appears, checking the loop condition. As usual, the loop will continue as long as the condition evaluates to `true`, and terminates when the condition evaluates to `false`. Note that we must add a semicolon **;** at the end of the ***while*** header. However, we do not add a semicolon in the **ordinary** ***while*** loop. If you do, it will separate the ***while*** loop header from its body, resulting in an infinite loop!

Example16

do..while loop is a popular choice when you design a menu of choices that users need to choose from. Run the provided code and notice that the code is incomplete. In order to respond to the user's choice, we usually use multiple ***if .. else*** statements or ***switch*** statements that you will learn about in Chapter 3. For now, it

is ok to just notice how this loop can be used in the future whenever the problem that you are solving includes a menu of options. In this example, we implemented user choice as **int** numbers, however, technically, you can use anything else like, characters or even words.

```
#include <iostream>
using namespace std;

int main() {
    int choice;
    do {
        cout << "Menu:" << endl;
        cout << "1) Option 1" << endl;
        cout << "2) Option 2" << endl;
        cout << "3) Option 3" << endl;
        cout << "4) Exit" << endl;
        cout << "Enter your choice: " << endl;
        cin >> choice;

        //more code to respond to user choice!
    } while (choice != 4);
    return 0;
}
```

FORMATTING OUTPUTS

At this point, you might have gotten a brief picture of how complex the code can be, and hence the output as well. Therefore, it's important as a programmer to start thinking about formatting the code and the output. This is not solely for beautifying how they look, but to make reading and understanding the code and the output a lot easier. The benefits of this way of thinking will become evident as you progress through this book.

In the next section, you will learn about the best practices for writing clear and easy-to-read code. In this section, you will learn how to write clear and easy-to-read outputs.

Example 17

Run the given code and pay attention to the outputs. The first group of output lines are called “Messy Outputs”. As the name suggests, their format is messy and hard to read and understand. However, in the second group of output lines, the same information is printed out in a formatted way. While the information is exactly the same, the way it’s formatted makes it easier to read and understand. In this code, we used a predefined function **setw,** which stands for “set the width”. This function is sometimes called a manipulator because it manipulates the output **cout** stream. **left** and **right** manipulators can be used with **setw** to align outputs. **setw** takes one **int** argument and creates a table column with the specified number of characters as the column width. You must include the ***<iomanip>*** library to be able to use the **setw** function. Try to experiment with the values used to get a better sense of how to use this formatting function.

```
#include <iostream>
#include <iomanip> // for setw

using namespace std;

int main() {
    // Messy console output
    cout << "Messy Output:\n";
    cout << "Name" << "City" << "Resident Age" << "\n";
    cout << "JohnDoe" << "NewYork" << "25" << "\n";
    cout << "JaneDoe" << "LosAngeles" << "22" << "\n";

    // Formatted console output
    cout << "\nFormatted Output:\n";
    cout << left << setw(15) << "Name" << left << setw(15) << "City" << right << setw(5) << "Resident Age" << "\n";
    cout << left << setw(15) << "John Doe" << left << setw(15) << "New York" << right << setw(5) << "25" << "\n";
    cout << left << setw(15) << "Jane Doe" << left << setw(15) << "Los Angeles" << right << setw(5) << "22" << "\n";

    return 0;
}
```

Example 18

You can be as creative as you want to present the outputs in the best way you are able to. Modify the previous example code to add boundaries to the table created. Note here to avoid writing long lines of code, some IDEs provide an option to break a single line of code into several lines to enhance readability. For example, line 10 is broken into two lines. Therefore, you either enter it as one long line on your IDE or configure your IDE to allow automatically breaking long lines. The curved arrow that appears at the end of the line is just a reminder that this line of code continues to the next line. Similarly, lines 12 and 13 are broken into two lines.

***NOTE**: If you are using Visual Studio IDE, you can configure it to automatically break long lines by following these steps:*

1. *Open Visual Studio.*
2. *From the **Tools** menu, select **Options**.*
3. *In the search bar, type "word wrap".*
4. *Click the checkbox next to the 'Word Wrap' option.*

```cpp
#include <iostream>
#include <iomanip> // for setw

using namespace std;

int main() {
    // Formatted console output
    cout << "\nFormatted Output:\n";
    cout << "+---------------+---------------+-------------+\n";
    cout << "| " << left << setw(14) << "Name" << "| " << left << setw(14) << "City" << "| " << right << setw(11) << "Resident Age" << "|\n";
    cout << "+---------------+---------------+-------------+\n";
    cout << "| " << left << setw(14) << "John Doe" << "| " << left << setw(14) << "New York" << "| " << right << setw(11) << "25" << " |\n";
    cout << "| " << left << setw(14) << "Jane Doe" << "| " << left << setw(14) << "Los Angeles" << "| " << right << setw(11) << "22" << " |\n";
    cout << "+---------------+---------------+-------------+\n";

    return 0;
}
```

Exercise 13

Write a program that produces this output:

```
Formatted Output:
+---------------+---------------+-------+
| Student       | Subject       | Grade |
+---------------+---------------+-------+
| John Doe      | Math          |     A |
| Jane Gripen   | Science       |     B |
| Tom Smith     | English       |     A |
| Emma Brown    | History       |     B |
| Oliver Davis  | Math          |     A |
+---------------+---------------+-------+
```

In addition to formatting texts, we often need to format numbers as they would become very messy.

Example 19

Run the provided code and notice the new manipulators used. In this example, we have used **setprecision** manipulator to choose how many decimal digits we desire. The **fixed** manipulator is necessary to manipulate the **cout** output stream to always print out numbers with decimal places. That is to avoid printing numbers in sceintific notation like 1.234e5, 9.998745e-13, and so on, where the letter **e** stands for the exponent.

```
#include <iostream>
#include <iomanip> // for setprecision and fixed manipulators.

using namespace std;

int main() {
    // Messy output
    cout << "Messy Output:\n";
    cout << 3.141 << " " << 2.7182008 << " " << 1.415421 << "\n";
    cout << 0.51 << " " << 1.6803 << " " << 2.3025459 << "\n";

    // Formatted output
    cout << "\nFormatted Output:\n";
    cout << fixed << setprecision(2);
    cout << setw(6) << 3.141 << setw(6) << 2.7182008 << setw(6) << 1.415421 << "\n";
    cout << setw(6) << 0.51 << setw(6) << 1.6803 << setw(6) << 2.3025459 << "\n";

    return 0;
}
```

Exercise 14

Write a program that produces this output:

```
Messy Output:
1.234 23.45 345.6
456.78 56.789 6.789
7.8901 89.012 90.123
123.456 234.567 345.678
456.789 567.89 678.901

Formatted Output:
+---------+---------+---------+
|   1.234 | 23.450 |345.600   |
| 456.780 | 56.789 |  6.789   |
|   7.890 | 89.012 | 90.123   |
| 123.456 |234.567 |345.678   |
| 456.789 |567.890 |678.901   |
+---------+---------+---------+
```

BEST PRACTICES FOR WRITING CODE

Naming Conventions

In chapter one, we mentioned some general rules to create variable names. For the importance of this topic, we re-emphasize these rules here. Until now, variable names have been created arbitrarily. Technically, you can name your variables in any way you prefer; however, you must follow the following rules:

- Names must always be one word. No spaces are allowed.
- You cannot use a keyword as a variable name, like int, double, cout, and so on.
- You cannot use any special characters in the name except the underscore.
- Names must not start with numbers. The first character

must always be either a letter or an underscore, followed by any combination of letters and numbers.

- C++ is a case-sensitive language, meaning all of these names are considered different and unique names: myVariable, MyVariable, myvariable, ... and so on.
- Create meaningful names that can tell their purpose, like sum, average, grossTotal, FtoC, or CtoF for converted degrees from Fahrenheit to Celsius and vice versa. Always think creatively and consistently.

As you have seen so far, some variable names can consist of multiple words, like grandTotal or discountedPrice. As using spaces is not allowed in naming, you can make names easier to read by either capitalizing each word or using underscores. The choice will be totally up to your preference.

Exercise 15

Determine which of the following names are correct following the common names writing rules and which are not. If a name is not correct, explain the reason why.

1. int
2. My Variable
3. totalScore
4. 123abc
5. studentName
6. numberOfItems
7. my-variable
8. isReady

9. maxValue
10. @username

Writing Styles

While compilers won't complain if you write your entire code in a single line, doing so can make it challenging for developers to read and debug. Therefore, adopting a professional coding style is essential. It serves not only as a formatting and beautification requirement but also as a means to ensure code clarity and ease of understanding, especially when multiple developers collaborate on the same project.

While there is no widespread consensus on which coding style to follow, the golden rule remains: keep your code clean and tidy. Fortunately, Integrated Development Environments (IDEs) offer significant assistance in achieving this goal. If you're using any of the widely available IDEs (some of which are free, like Visual Studio), they can greatly simplify your development process.

Writing Meaningful Comments

Example 01 explained that in real-life scenarios, comments can serve a very important role, which is communication between developers, especially in large projects. For this reason, to write useful and efficient comments, you can follow one of the many available comment structures.

Example 20

When writing code, it's essential to include certain details at the top of every program. These practices not only enhance readability but also provide valuable context for anyone who reads your code.

Consider adopting these habits from now on, and feel free to customize the content to suit your specific needs:

- **Program Description**: Start with a brief description of what the program does. This helps readers quickly understand its purpose.
- **Author Information**: Include your name or the name of the author(s) responsible for the code. It's a professional courtesy and allows others to reach out if they have questions.
- **Date Created/Modified**: Specify when the program was initially written and any subsequent modification dates. This timestamp helps track changes and ensures that everyone is working with the most up-to-date version.
- **Version Number**: If applicable, indicate the version of the program. This is especially useful for software that undergoes frequent updates.
- **Dependencies and Requirements**: List any external libraries, modules, or tools required to run the program successfully. Be specific about versions if necessary.
- **Usage Instructions**: Provide clear instructions on how to use the program. Include command-line arguments, input formats, and expected output.
- **License Information:** State the license under which the code is released. Common licenses include MIT, GPL, and Apache. Choose one that aligns with your intentions for sharing the code.
- **Acknowledgments**: If you've borrowed ideas, algorithms, or code snippets from other sources, acknowledge them. It's good practice and shows respect for the work of others.

```
/* Developer name:
* Program name:
* Program purpose:
* Version:
* Date:
* License:
*
* possible more details as needed
*/
```

Example 21

In this example, we are using the preconditions and postconditions comments that usually appear at the beginning of every function. In chapters 04 and 05, you will start writing your own functions, and you will need to write meaningful comments following this common style. Preconditions are used to explain clearly what should be true before calling a function. In code snippet 1, the function ***calculateRectangleArea*** expects to receive two ***double*** arguments. The precondition is explicitly saying that those two arguments must not be negative, as the results would be unreasonable. This comment is very helpful in informing whoever wants to use this function how to use it correctly. Similarly, the postcondition says what the function will return at the end of its execution. In this case, the function will return the calculated area as a ***double*** value. Again, if someone wants to use this function to calculate the area and save the returned value into some variable, they need to be sure that the variable has the correct type to save

the returned value. In this case, the returned value will be ***double*** and hence the variable used to save this value must be ***double***, otherwise, either an error or unexpected results will occur. Read the remaining code snippets to understand their importance in clarifying how each function must be called and used.

```
// code snippet 1: Calculate the Area of a Rectangle
// Preconditions: width and height must be positive double values.
// Postconditions: Returns the area of the rectangle as a double value.
double calculateRectangleArea(double width, double height) {

    // code implementing the function

}

// code snippet 2: Convert Fahrenheit to Celsius
// Preconditions: Fahrenheit temperature must be within a reasonable range.
// Postconditions: Returns the equivalent Celsius temperature.
double fahrenheitToCelsius(double fahrenheit) {

    // code implementing the function

}

// code snippet 3: Validate User Input for Age
// Preconditions: Age must be a non-negative integer.
// Postconditions: Returns true if age is valid; otherwise, false.
bool isValidAge(int age) {

    // code implementing the function

}
// code snippet 4: Deposit Money into a Bank Account
// Preconditions: Amount must be positive.
// Postconditions: Increases the account balance by the given amount.
void deposit(double amount) {

    // code implementing the function

}
```

Exercise 16

Write the preconditions and postconditions for the following functions:

```
1.  double squareRoot(double x)
2.  bool isValidEmail(string email)
3.  double calculateCircleArea(double
    radius)
```

Wrap up!

In this chapter, you have been introduced to the fundamental building blocks of C++ programming. You have learned how to write your first simple program, including declaring variables, using operators, and printing output to the console. You developed a strong understanding of the various data types available in C++, their memory requirements, and the importance of type compatibility. The chapter also covered key control flow structures like if-else statements and different loop constructs like for, while, and do-while. By practicing the exercises provided, you have gained hands-on experience in applying these concepts. Finally, the text emphasized the importance of following best practices, such as using meaningful variable naming, writing clear comments, and adopting consistent coding styles. With this solid foundation, you are now well-equipped to progress to more advanced C++ programming topics covered in the subsequent chapters of this book.

ANSWERS KEY

1)

```
1)
int num1 = 15; //declare the variable num1 and initialize it with 15.
int num2 = 5; //declare the variable num2 and initialize it with 5.
int addition = num1 + num2; //addition of num1 and num2 which is 20.

2)
int subraction = num1 - num2; //subtraction of num1 and num2 which is 5.
int multiplication = num1 * num2; //multiplication of num1 and num2 which is 75.
int division = num1 / num2; //dividion of num1 and num2 which is 3.

3)
const int MAX_VALUE = 1000; //declare constant MAX_VALUE and initialize it with 100.
```

2)

```
#include<iostream>
using namespace std;

int main() {
    // Declare variables for speed and time
    double distance, time;

    // Ask user for speed and time
    cout << "Enter the distance in miles: ";
    cin >> distance;
    cout << "Enter time in hours: ";
    cin >> time;

    // Calculate and display the distance
    double speed = distance / time;
    cout << "The speed of the vehicle is " << speed << " miles/hour!" << endl;

    return 0;
}
```

3)

```
#include<iostream>
using namespace std;

double const PI = 3.14;

int main() {
    // Declare variable for radius
    double radius;

    // Ask user for radius
    cout << "Enter radius of the sphere: ";
    cin >> radius;

    // Calculate and display the volume of the sphere
    double volume = (4.0 / 3.0) * PI * 3 * 3 * 3;
    cout << "The volume of the sphere is: " << volume << " cubic units" << endl;

    return 0;
}
```

4)

```cpp
#include<iostream>
using namespace std;

int main() {
    // Declare variables for principal, rate and time
    float principal, rate, time;

    // Ask user for principal, rate and time
    cout << "Enter principal amount: ";
    cin >> principal;
    cout << "Enter rate of interest as a whole number: ";
    cin >> rate;
    cout << "Enter time in years: ";
    cin >> time;

    // Calculate and display the simple interest
    float interest = (principal * rate * time) / 100;
    cout << "The simple interest is: " << interest << endl;

    return 0;
}
```

5)

myInt = 1
myInt2 = 0
myInt3 = 72 //the ASCII value of H is 72
myInt4 = 5 //the fraction part is lost!
myBool = 0 //false value is encoded as 0
myBool2 = 1; //any non-zero number is represented as true, true value is encoded as 1
myChar = H; //H is represented by ASCII value 72
myDouble = 5;

6)

```cpp
#include <iostream>
using namespace std;

int main() {
    string str1 = "Hello";
    string str2 = " World!";
    str1.append(str2);
    cout << str1 << endl;//Outputs: Hello World!

    return 0;
}
```

7)

```cpp
#include <iostream>
using namespace std;
int main() {
    double num1, num2;
    cout << "Enter two numbers: ";
    cin >> num1 >> num2;
    if (num1 > num2)
        cout << num1;
    else
        cout << num2;
    return 0;
}
```

8)

```
#include <iostream>
using namespace std;
int main() {
    char response;
    double itemPrice, discountedPrice;

    cout << "Enter your item price.";
    cin >> itemPrice;

    cout << "Are you a member? Enter y for yes or n for no.";
    cin >> response;
    if (response == 'y') {
        discountedPrice = itemPrice - (0.05 * itemPrice);
        cout << "Your discounted item price is $" << discountedPrice << endl;
    }
    else
        cout << "Sorry, there is no discount applicable to you! " << endl;
    return 0;
}
```

9)

```
#include <iostream>
using namespace std;
int main() {
    int num;
    cout << "Enter a number: " << endl;
    cin >> num;
    cout << "The even numbers list is:  ";
    for (int i = 1; i <= num; i++)
        if (i % 2 == 0)
            cout << i << "  ";
    return 0;
}
```

10)

```cpp
#include <iostream>
using namespace std;

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

    cout << "Enter a non-negative integer: ";
    cin >> n;
        for (int i = 1; i <= n; ++i) {
            factorial = factorial * i;
        }
        cout << "Factorial of " << n << " = " << factorial << endl;
    return 0;
}
```

11)

```cpp
#include <iostream>
using namespace std;

int main() {
    string userGuess, secretWord = "lion";

    cout << "Welcome to the animals guessing game!" << endl;
    cout << "Which animal known as the king of jungle?";
    cin >> userGuess;

    while (userGuess != secretWord) {
        cout << "Sorry, that's not correct. Try again: ";
        cin >> userGuess;
    }
    cout << "Congratulations! You guessed the correct animal." << endl;

    return 0;
}
```

12)

```cpp
#include <iostream>
using namespace std;

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

    cout << "Enter a non-negative integer: ";
    cin >> n;

    int i = 1;
    while (i <= n) {
        factorial = factorial * i;
        i++;
    }
    cout << "Factorial of " << n << " = " << factorial << endl;

    return 0;
}
```

13)

```cpp
#include <iostream>
#include <iomanip> // for setw

using namespace std;

int main() {
    // Formatted console output
    cout << "\nFormatted Output:\n";
    cout << "+---------------+---------------+-------+\n";
    cout << "| " << left << setw(14) << "Student" << "| " << left << setw(14) <<
       "Subject" << "| " << right << setw(5) << "Grade" << " |\n";
    cout << "+---------------+---------------+-------+\n";
    cout << "| " << left << setw(14) << "John Doe" << "| " << left << setw(14)
      << "Math" << "| " << right << setw(5) << "A" << " |\n";
    cout << "| " << left << setw(14) << "Jane Gripen" << "| " << left << setw
      (14) << "Science" << "| " << right << setw(5) << "B" << " |\n";
    cout << "| " << left << setw(14) << "Tom Smith" << "| " << left << setw(14)
      << "English" << "| " << right << setw(5) << "A" << " |\n";
    cout << "| " << left << setw(14) << "Emma Brown" << "| " << left << setw(14)
       << "History" << "| " << right << setw(5) << "B" << " |\n";
    cout << "| " << left << setw(14) << "Oliver Davis" << "| " << left << setw
      (14) << "Math" << "| " << right << setw(5) << "A" << " |\n";
    cout << "+---------------+---------------+-------+\n";

    return 0;
}
```

14)

```cpp
#include <iostream>
#include <iomanip> // for setprecision and fixed manipulators.

using namespace std;

int main() {
    // Messy output
    cout << "Messy Output:\n";
    cout << 1.234 << " " << 23.45 << " " << 345.6 << "\n";
    cout << 456.78 << " " << 56.789 << " " << 6.7890 << "\n";
    cout << 7.8901 << " " << 89.012 << " " << 90.123 << "\n";
    cout << 123.456 << " " << 234.567 << " " << 345.678 << "\n";
    cout << 456.789 << " " << 567.890 << " " << 678.901 << "\n";

    // Formatted output
    cout << "\nFormatted Output:\n";
    cout << "+---------+---------+---------+\n";
    cout << "| " << fixed << setprecision(3);
    cout << setw(7) << 1.234 << " |" << setw(7) << 23.45 << " |" << setw(7)
       << 345.6 << "   |\n";
    cout << "| " << setw(7) << 456.78 << " |" << setw(7) << 56.789 << " |"
      << setw(7) << 6.7890 << "   |\n";
    cout << "| " << setw(7) << 7.8901 << " |" << setw(7) << 89.012 << " |"
      << setw(7) << 90.123 << "   |\n";
    cout << "| " << setw(7) << 123.456 << " |" << setw(7) << 234.567 << "
      |" << setw(7) << 345.678 << "   |\n";
    cout << "| " << setw(7) << 456.789 << " |" << setw(7) << 567.890 << "
      |" << setw(7) << 678.901 << "   |\n";
    cout << "+---------+---------+---------+\n";

    return 0;
}
```

15)

1. (not correct) int (uses a reserved keyword)
2. (not correct) My Variable (contains spaces)
3. (correct) totalScore
4. (not correct) 123abc (starts with a digit)
5. (correct) studentName
6. (correct) numberOfItems
7. (not correct) my-variable (contains a hyphen)
8. (correct) isReady
9. (correct) maxValue

10. (not correct) @username (contains special characters)

16)
These are suggested answers.

1. **Precondition**: The input `x` must be non-negative.
   Postcondition: The function returns the square root of `x`.
2. **Precondition**: The input `email` is a non-empty string.
   Postcondition: The function returns `true` if the email address is valid (following common email format rules), otherwise `false`.
3. **Precondition**: The input `radius` is non-negative.
   Postcondition: The function returns the area of a circle with the given radius.

END OF CHAPTER EXERCISES

1) What is the purpose of declaring a variable in a program?
   A) To reserve a memory location to store data.
   B) To perform operations on data.
   C) To print data to the console.
   D) To connect to a database.
2) What keyword is used to declare a constant?
   A) var
   B) let
   C) const
   D) static
3) True or False: The value of a variable can be changed after it has been declared and initialized.
4) True or False: The value of a constant can be changed after it has been declared and initialized.

5) True or False: It is a common convention to capitalize constant names.
6) Write a code snippet to declare an integer variable newNumber and initialize it with the value 50, and declare a constant DEFAULT_NUM and initialize it with the value 100.
7) What is the difference between // and /**/ in C++?

A) // is used for single-line comments, and /**/ is used for multi-line comments.
B) // is used for multi-line comments, and /**/ is used for single-line comments.
C) Both // and /**/ are used for single-line comments.
D) Both // and /**/ are used for multi-line comments.

8) This statement is called...

```
int variableName = 0;
```

A) Variable declaration
B) Variable initialization
C) Variable declaration and initialization
D) Syntax error!

9) True or False:

a) The **main** function can appear more than once in a C++ program.
b) Every statement in C++ must end with a semicolon.
c) The **cout** stream is used to read user input.
d) The **cin** stream works with the extraction operator **>>**.
e) When variable names appear in a **cout** line of code, the variable name itself will print out to the console.
f) The **return 0;** statement in the **main** function signifies that the program has encountered an error.

10) What does the **#include <iostream>** line do in a C++ program?

A) It includes the input/output stream library.
B) It starts the main function.
C) It prints out a message to the console.
D) It takes user input.

11) What is the purpose of the **main** function in a C++ program?

A) It defines a variable.
B) It is the first function to execute when you run your code.
C) It includes a library.
D) It prints out a message to the console.

12) What does the **cin** stream do in a C++ program?
A) It prints out a message to the console.
B) It includes a library.
C) It reads user input.
D) It defines a variable.

13) What operator is used with the **cout** stream to print out a message?
A) >> B) << C) >= D) <=

14) What does the **return 0;** statement signify in the **main** function?
A) It signifies that the program has encountered an error.
B) It signifies that the program has run successfully.
C) It signifies that the program needs user input.
D) It signifies that the program is about to start.

15) Write a program that takes a user's name as input and prints "Hello, [Name], nice to meet you!".

16) Write a program that declares two integer variables **`a`** and **`b`**, initializes them with values 5 and 10, respectively, and then swaps their values. The program should print the values of the variables before and after the swap.

17) Extend the simple interest calculator program you created in exercise 04 to calculate and display the total amount in the account after the interest has been added. The total amount should be the sum of the principal and the interest.

18) What will be the output if we replace ***sentence3.replace(7, 11, "C++")*** with ***sentence3.replace(7, 4, "C++")*** in example 07?
A) I love programming C++!
B) C++ I love programming!
C) I love C++ramming!
D) Syntax error

19) True or False:

a) The ***find()*** function returns the position of the first character of the first match of the specified substring.

b) The + operator can be used to concatenate two strings in C++

c) The ***length()*** function is used to print a string backward

20) Write a program that does the following:

- Declares a string variable **str.**
- Initializes **str** variable to the value "I just started learning C++.".
- Use the member function ***replace()*** to replace the word C++ with Java. Both sentences, before and after the replacement, should be printed out to the console.
- Use the ***append()*** function to add "It is so much fun!" to the same variable and print out the new sentence.
- Use the ***find()*** function to find the position of the word ***it*** in the new sentence. The position value should be printed to the console.

21) Write a program that asks the user to enter their grades for quizzes, the midterm, and the final. If the total is equal to or greater than 75, print a message to the console saying "You passed, congratulations!". Otherwise, print a message saying "Sorry, you have to redo your work!".

22) What will the following code print?

```
int x = 10;
if (x > 5)
    cout << "Hello ";
else
    cout << "Hi ";
cout << "World!";
```

A) Hello World!
B) Hi World!
C) World!
D) Hello, Hi World!

23) True or False:

a) When using an ***if***-statement, we must use an ***else***-statement with it as well.

b) The expression inside the ***if***-statement must always be a boolean.

24) True or False:

a) A ***for*** loop is used when the number of iterations is known beforehand.

b) A ***do-while*** loop checks the condition before executing the loop body.

c) A ***do-while*** loop is guaranteed to execute its body at least once.

25) What will be the output of the following C++ code?

```
int i = 0;
while (i < 5) {
cout << i << " ";
i++;
}
```

a) 0 1 2 3 4 5
b) 0 1 2 3 4
c) 1 2 3 4 5

d) 1 2 3 4

26) What will be the output of the following code?

```
int i = 0;
do {
cout << i << " ";
i++;
} while (i < 5);
```

a) 0 1 2 3 4 5
b) 0 1 2 3 4
c) 1 2 3 4 5
d) 1 2 3 4

27) Write a program that prints the numbers from 1 to 10 using a ***for*** loop.

28) Write a program that calculates the factorial of a number using a ***while*** loop.

29) Write a program that prints the numbers from 10 to 1 in reverse order using a ***for*** loop.

30) Write a program that calculates the sum of the numbers from 1 to 100 using a ***for*** loop.

31) Write the preconditions and postconditions for the following functions:

- `int add(int a, int b)`
- `double divide(double a, double b)`
- `bool isPrime(int n)`
- `void swap(int a, int b)`

32) Explain how to use ***setprecision***, ***setw,*** and ***fixed*** manipulators. Provide brief examples.

PROJECTS

Project 01: Grocery Store Billing System

Write a program that simulates a grocery store billing system. The program should prompt the user to enter the quantities of three types of fruits: apples, bananas, and oranges. Then, it should calculate the total cost of the items purchased based on their quantities and prices. Additionally, if the total cost exceeds $10, a 10% discount should be applied. Finally, the program should display the total cost to the user, with or without the discount applied.

Your program should do the following:

1. Prompt the user to enter the number of apples, bananas, and oranges.
2. Calculate the total cost of items purchased using the following prices declared as constants:
    - Apples: $1.00 each
    - Bananas: $0.50 each
    - Oranges: $1.50 each
3. Display the total cost of items to the user.
4. If the total cost exceeds $10, apply a 10% discount and display the new total cost.
5. Ask the user if they wish to repeat the calculations using the `do..while` loop.
6. Write appropriate comments to explain the code where necessary.
7. Test your program with various input values to ensure its correctness and reliability.

Project 02: Weekly Budget Planner

Create a program for managing weekly expenses using the Weekly Budget Planner. The program should allow users to add expenses for food, transportation, and entertainment categories. Additionally, users should be able to print the budget to see the total expenses for each category and the grand total.

Your program should do the following:

1. Display a menu with these options:
    - Add food expenses
    - Add transportation expenses
    - Add entertainment expenses
    - Print budget
2. Prompt the user to enter their choice (1, 2, 3, or 4) for the category they want to add expenses to.
3. Depending on the user's choice, prompt them to enter the value for the corresponding category's expenses.
4. Repeat steps 2 and 3 until the user chooses to print the budget by entering '4'.
5. After the user chooses to print the budget, display the total expenses for each category and the grand total.
6. Thank the user for using the Weekly Budget Planner.
7. Write appropriate comments to explain the code where necessary.
8. Test your program with various input values to ensure its correctness and reliability.

Ensure that your program follows these specifications:

- Use appropriate variable names and data types to store

expenses and user inputs.

- Utilize a `do..while` loop to repeatedly prompt the user for input until they choose to print the budget.
- Provide clear instructions and prompts to guide the user through the program.

Project 03: Prime Number Checker

Create a console application that can determine whether a given number is prime or not.

1. Prompt the user to input an integer.
2. Process the input and determine whether it is a prime number.
3. Display the result to the user.
4. Write appropriate comments to explain the code where necessary.
5. Test your program with various input values to ensure its correctness and reliability.

Hints:

- A prime number is a number greater than 1 that has no positive divisors other than 1 and itself.
- Consider how you can optimize the program to reduce the number of checks needed to determine if a number is prime.
- Think about how you can handle different types of inputs and potential errors.

CHAPTER 3

More on Branching and Loops

HUSSAM GHUNAIM

Learning Objectives

At the end of reading this chapter, you will be able to:

- Write complex expressions.
- Solving problems using
  - nested *if .. else* statements
  - *switch* statements
  - nested loops

WRITING COMPLEX EXPRESSIONS

In this section, we will explore operators in more detail. You have already learned in chapter 2, to write if... else statements and loops like for loops and while loops. We always need to write appropriate expressions to control the behavior of these statements. To master

writing expressions, let's first look at the types of operators you are going to use in writing C++ code.

Operators Types

There are many operators used in writing code. To better understand them, we can group them into several groups. The easiest group to start with is the ***Arithmetic Operators***, Table 01. These are self-explanatory operators, as you can tell. However, the modulo operator `%` may be new to some readers. This operator is used to find the remainder after dividing two numbers. Figure 01 shows two simple examples to differentiate the `%` operator from the division `/` operator.

Table 01: Arithmetic Operators

+	Addition
–	Subtraction
*	Multiplication
/	Division
%	Modulo (Remainder)

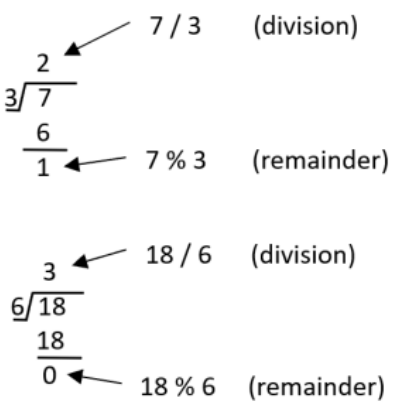


Figure 01: Modulo examples

Example 01

Examples of modulo.

- 13 % 3 = 1 (13 divided by 3 is 4 with a remainder of 1.)
- 27 % 7 = 6 (27 divided by 7 is 3 with a remainder of 6.)
- 33 % 11 = 0 (33 divided by 11 is 3 with a remainder of 0.)
- 18 % 7 = 4 (18 divided by 7 is 2 with a remainder of 4.)
- 15 % 5 = 0 (15 divided by 5 is 3 with a remainder of 0.)

Exercise 01

Evaluate the following modulo expressions.

- 75 % 25 =
- 101 % 3 =
- 62 % 6 =
- 224 % 10 =
- 555 % 5 =

Table 02: Logical Operators

AND && Operator
true && true returns true
true && false returns false
false && true returns false
false && false returns false
OR || Operator
true || true returns true
true || false returns true
false || true returns true
false || false returns false
NOT ! Operator
!true returns false
!false returns true

The second group of operators is the ***Logical Operators***. We have three of them: AND, OR, and NOT. In C++ code, these operators must be written as **`&&`** , **`||`**, and **`!`** respectively. Logical operators always operate on Boolean expressions and return a Boolean as well. The syntax of using logical operators is

left_expression ***logical_operator*** *right_expression*

The result depends on the expressions and the logical operator used. For example, if left and right expressions return `true`, then *`left_expression`* **`&&`** *`right_expression`* returns **`true`**, that is, **`true && true`** returns **`true`**.

However, there is an exception, the NOT operator. This operator operates on a single expression and negates it. Table 02 shows the truth table for the three logical operators.

Table 03: Comparison Operators

==	Equal to
!=	Not equal to
>	Greater than
>=	Greater than or equal
<	Less than
<=	Less than or equal

The third group of operators is the ***Comparison Operators***, Table 03. Operators in this group take two expressions and return a Boolean depending on the result of the comparison. To write useful comparisons, we usually combine both arithmetic and comparison operators as illustrated in the following examples.

Example 02

Study carefully the following examples.

- 5 < 5 returns false
- 5 <= 5 returns true
- (4*2) == (48/6) returns true
- (5+6) != (4+7) returns false
- 8 >= 8 returns true
- (9/3) == (18/3) returns false
- (7+2) != (5+5) returns true

Exercise 02

Find the results of the following expressions:

- (12 % 3) >= (6 – 7)
- 22 != (11 * 2)
- -11 < 11
- (15 % 4) >= (8 – 9)
- 30 != (15 * 2)
- 20 <= -20

The fourth group of operators is the ***Assignment Operators*** group, Table 04. In the previous chapters, you have seen some assignment expressions using the operator **=** that assigns a value from the right side to a variable located on the left side. In addition to this operator, there are a couple of other assignment operators that can be used as a shortcut for writing longer expressions. For example, instead of writing *sum = sum + 1*, we can write *sum += 1*. Both lines of code do exactly the same thing, which is adding **1** to the variable **sum** and then saving the result into the same variable **sum**. Similarly, we can write other assignment expressions like *sum -= 2*, *sum *= 0.5*, and so on to have whatever desired outcomes we are looking for.

Table 04: Assignment Operators

=	Assign
+=	Add and assign
-=	Subtract and assign
*=	Multiply and assign
/=	Divide and assign
%=	Madulo and assign

Exercise 03

Without writing additional code, find the new values of the variables in the following expressions:

```
int counter = 0;
counter += 5;

int num = 10;
num -= 10;

int grade = 90;
grade /= 3;

int TheRemaining = 45;
TheRemaining %= 5;
```

Exercise 04

Run the following code to find out the outputs. Explain the results.

```cpp
#include <iostream>
using namespace std;

int main() {
    double z = 10;
    int d = 10, x = 10, y = 10;

    for (int i = 0; i < 3; i++) {
        x -= 1;
        cout << "x = " << x << endl;
    }

    for (int i = 0; i < 3; i++) {
        y *= 2;
        cout << "y = " << y << endl;
    }

    for (int i = 0; i < 3; i++) {
        z /= 2;
        cout << "z = " << z << endl;
    }

    for (int i = 0; i < 3; i++) {
        d %= 3;
        cout << "d = " << d << endl;
    }

    return 0;
}
```

Table 05: Unary Operators

++	Incrementing by 1
−−	Decrementing by 1
+	Positive sign
−	Negative sign

All of the operators we have used so far are called binary operators because they operate on two operands, that is, the left operand and the right operand. In addition to that, there is a group of operators that is called unary operators, which means they operate only on one operand, Table 05. The **++** operator is called the incrementing operator, while the **–– (no space in between)** is called the decrementing operator. Intuitively, the **++** increments a variable by one while the **––** decrements a variable by one. These operators are popular for their ease of use. Finally, the sign operators, **–** and **+** are used to give a sign to numerical variables.

Obeying Operators' Rules of Precedence

Table 06: Operators' Rules of Precedence. Operators at the top have the highest precedence, while those at the bottom have the lowest.

`++ -- ! + -`	Unary operators
`% / *`	Arithmetic operators
`+ -`	Arithmetic operators
`>= <= > <`	Logical operators
`!= ==`	Logical operators
`&&`	Logical operators
`||`	Logical operators
`= += -= *= /= %=`	Assignments operators

As you have seen so far that operators are going to be used frequently and combined in complex expressions. Thus, most of the time we need to evaluate complex expressions with multiple operators combined together. For that reason, we need to come up with a way to control the sequence of their evaluations. This sequence is called the Operators' Rules of Precedence. For example, what is the result of evaluating this expression, 2 * 3 – 1? If we multiply 2 and 3 and then subtract 1, we will get 5 as the result. However, if we subtract 1 from 3 first and then multiply by 2, we will get 4 as the result. Which answer should we accept? To solve the problem, all programming languages have established precedence rules that mandate which operators must be evaluated first. A list of precedence rules is provided in Appendix B. However, Table 06 shows a short list. The parenthesis always has the highest precedence.

Exercise 05

Based on the operators' precedence rules provided in Table 06, evaluate the following expressions:

```
1) int result = 3 + 5 * 4 / 2;
2) int result = (3 + 5) * 4 / 2;
3) bool result = (2 + 2 == 4) && (3 + 3 == 7);
4) bool result = 2 + 2 == 4 && 3 + 3 == 6;
5) bool a = true, b = false, c = true;
   bool result = a && b || c;
```

Confusing Operators

Some operators can cause different effects depending on their position in expressions. Example 03 below shows how the position of the incrementing and decrementing unary operators produces different results. Note that ++ and –– can appear as a prefix (on the left of a variable) or postfix (on the right of a variable).

Example 03

Run the provided code and then evaluate the following expressions. Explain the results

```cpp
#include <iostream>
using namespace std;

int main() {
    // part 1:
    int x = 5;
    x++; // same as x = x + 1;
    cout << "x = " << x << endl;

    // part 2:
    int y = 10;
    y--; // same as y = y - 1;
    cout << "y = " << y << endl;

    // part 3:
    int a = 3;
    int b = ++a;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;

    // part 4:
    int c = 12;
    int d = --c;
    cout << "c = " << c << endl;
    cout << "d = " << d << endl;

    // part 5:
    int e = 6;
    int f = e++;
    cout << "e = " << e << endl;
    cout << "f = " << f << endl;

    // part 6:
    int g = 8;
    int h = g--;
    cout << "g = " << g << endl;
    cout << "h = " << h << endl;

    return 0;
}
```

In example 3, parts 1 and 2 work as expected to increment or decrement the variables. Part 3 shows using ++ as a prefix to increment the value of the variable **`a`** and then assign the new value to **`b`**. The same procedure applies to part 4. However, in part 5, the ++ operator is used as a postfix. Note that in this case, the value of the variable **`e`** will be assigned to the variable **`f`** first, then the value of **`e`** will be incremented by 1. The same procedure applies to part 6.

Exercise 06

Run the provided code and then evaluate the following expressions. Explain the results

```cpp
#include <iostream>
    using namespace std;

    int main() {
        // part 1:
        int counter = 0;
        counter++; // same as counter = counter + 1;
        cout << " counter = " << counter << endl;

        // part 2:
        int grade = 85;
        grade--; // same as grade = grade - 1;
        cout << " grade = " << grade << endl;

        // part 3:
        int first = 3;
        int second = ++first;
        cout << " first = " << first << endl;
        cout << " second = " << second << endl;

        // part 4:
        int var1 = 10;
        int var2 = --var1;
        cout << " var1 = " << var1 << endl;
        cout << " var2 = " << var2 << endl;

        // part 5:
        int begin = 6;
        int end = begin--;
        cout << " begin = " << begin << endl;
        cout << " end = " << end << endl;

        // part 6:
        int g = 8;
        int h = g++;
        cout << " g = " << g << endl;
        cout << " h = " << h << endl;

        return 0;
    }
```

Short Circuits

Writing efficient code is always a concern for programmers. Therefore, whenever we write a piece of code, we need to double-check if our code is efficient or not. Efficiency generally means how fast a code can run and how much memory it needs. Through this book, we will visit this topic frequently. For now, we can explain how we can achieve a faster way to run code if we are using logical operators.

In writing **if** statements, **while**, or **for** loops, sometimes we need to use logical expressions. The following are some examples,

```
if (counter > 10 && grade != 50)
while (score < 100 || repetitions == 5)
for (int i = 0; i < 10 && i % 2 == 0; i++)
```

Note that, when using the && operator, if the left operand is false, the result will always be false regardless of the right operand, whether it is false or true. Table 02 shows all the cases where the && operator returns false. The only case that the operator && returns true is when both left and right operands are true. This means, when running the code, if the left operand of the && is found to be false, the system can return false immediately without evaluating the right operand. This operation is called short-circuiting.

Similarly, when using the || operator. Table 02 shows that the || operator always returns true if the left operand is true. However, if the left operand is false, we must check the right operand to determine the final result.

NESTED *IF .. ELSE* STATEMENTS

Table 07: Scholarships based on students' category.

	On-Campus	On-Line
Freshman	%20	%15
Otherwise	%10	%5

In Chapter 2, you learned how to write ***if..else*** statements. There is another technique to write conditional statements, which is called nested ***if..else*** statements.

Example 04

Write a program that asks users whether they are on-campus, online, freshman, or non-freshman students. Then the program should assign a specific scholarship to every student category, Table 07.

The best way to solve any problem is to start by writing an algorithm, similar to the example discussed in Chapter 1. If you have not done so already, read that section before attempting this example. Before looking at the provided solution, grab a piece of paper and write in simple sentences what the program should do in order to assign the correct scholarship for every student category.

Now, compare your algorithm with the one provided below. Note that almost always, it is possible to solve the same problem in many different ways. Therefore, when you compare your solutions with the ones provided to you, do not hesitate to unleash your creativity by finding different alternatives to solve the problem at hand.

The Algorithm:

1. Ask the user if she or he is an on-campus student.
2. Ask the user if she or he is a freshman.
3. If the student is on campus and a freshman, print a message saying they will get a 20% scholarship off their tuition.
4. If the student is on campus and not a freshman, print a message saying they will get a 10% scholarship off their

tuition.

5. If the student is online and a freshman, print a message saying they will get a 15% scholarship off their tuition.
6. If the student is online and not a freshman, print a message saying they will get a 5% scholarship off their tuition.

The second step that comes after writing the algorithm is to start thinking about how we can convert the simple sentences into C++ code. However, at this step, it is not necessary to write complete or runnable code. It is OK if you are not sure how to do that at this point. Let's start with the first item. To ask a user to enter data, we can always print out a message explaining what is needed from the user to enter. So far, we know that we can use ***cout*** to print such messages to the console and ***cin*** to read users' input. So we can write something like "Are you an on-campus student?". But now we realize that we need to declare a variable to save the user's answer. So let's say to do the first step in the algorithm, we need the following code:

```
string onCampus;
cout << " Are you an on - campus student ? Answer Yes or No" << endl;
cin >> onCampus;
```

The second item is similar to the first one, but we realize that we need a second variable to save the user's response to the second question.

Next, we need to think about how we would implement the third item. From the wording of this item, we need to make a decision on one option or the other. Of course, we know that we can use **`if..else`** to do that, but the problem here is that we have another decision to make. It is not a simple decision; rather, it is a compound decision. Up to this point, we can come up with the following pseudocode. Pseudocode is a technical term that means code-like statements but does not necessarily follow any particular

syntax. Writing pseudocode is very useful because it saves us from worrying about syntax and focusing only on the solution.

```
if (user is an online student)
    and if (user is a freshman)
        cout << " You got 20% percent scholarship !"
```

In addition to writing pseudocode, programmers use visualization a lot to help them understand the problem that they want to solve. UML (Unified Modeling Language) is a popular example. UML is not in the scope of this book; however, simple visualization techniques will be used frequently to make understanding and writing code easier for first-time programmers. Figure 02 depicts the decisions the code should make. By examining the figure, we can have an idea of how we should construct the needed code to solve the problem.

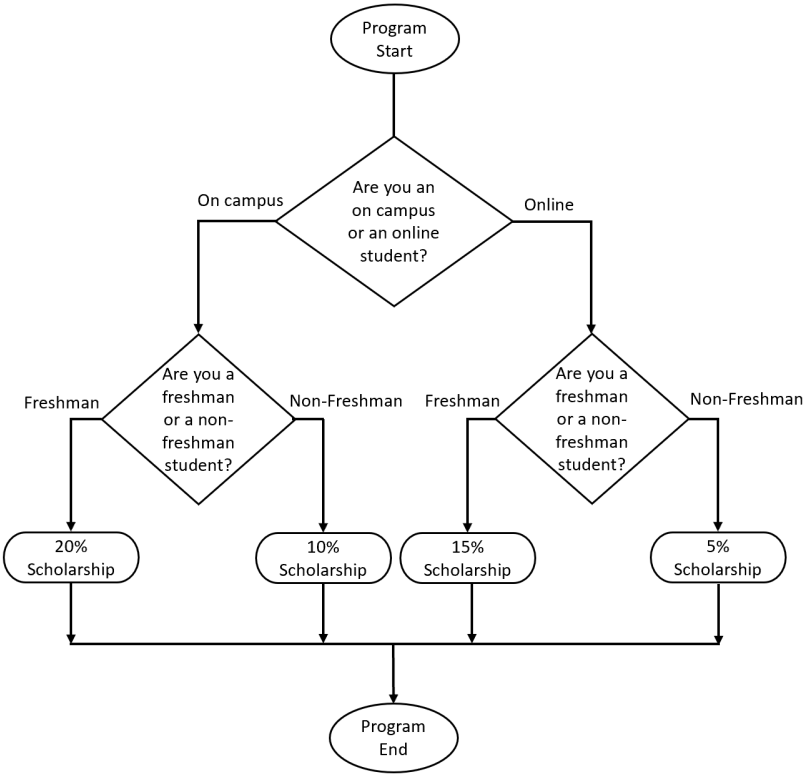


*Figure 02: UML for nested **if..else** Statements.*

Now, we are definitely in a better position to start writing code as compared to if we had started writing code right away. Before looking at the provided solution, you have to try to write your own code and test it to check if it produces the expected outcomes. Remember, it is absolutely fine to come up with a different solution provided that your code produces the same expected outcomes.

```
#include <iostream>
using namespace std;

int main() {
    string onCampus, freshman;
    cout << " Are you an on-campus student ? Answer Yes or No" << endl;
    cin >> onCampus;
    cout << " Are you a freshman student ? Answer Yes or No" << endl;
    cin >> freshman;

    if (onCampus == " Yes") {
        if (freshman == " Yes") {
            cout << " You will get scholarship worth of %20 of your tuition !" << endl;
        }
        else {
            cout << " You will get scholarship worth of %10 of your tuition !" << endl;
        }
    }
    else {
        if (freshman == " Yes") {
            cout << " You will get scholarship worth of %15 of your tuition !" << endl;
        }
        else {
            cout << " You will get scholarship worth of %5 of your tuition !" << endl;
        }
    }
    return 0;
}
```

Exercise 07

Use the nested *if..else* statements discussed in this section to solve the following problem.

Ask a user to enter three numbers. The program must find and print out the largest number. Start first with writing an algorithm and drawing a UML diagram. Then, convert your algorithm into code.

IF .. ELSE STATEMENTS WITH COMPOUND CONDITIONS

Another technique to solve problems using the **`if..else`** statements is by writing compound conditions.

Example 05

First, study and run the following code, and then explain the output. After that, modify the ***firstCondition*** and ***secondCondition*** values to get the other two possible outputs.

```
#include <iostream>
using namespace std;

int main() {
    int firstCondition = 10;
    int secondCondition = 20;

    if (firstCondition > 0 && secondCondition < 30) {
        cout << "Both conditions are true." << endl;
    }
    else if (firstCondition > 0 || secondCondition < 50) {
        cout << "At least one condition is true." << endl;
    }
    else {
        cout << "Both conditions are false!" << endl;
    }
    return 0;
}
```

Exercise 08

By using **`if..else`** statement with compound conditions: Write a program that prints the following output messages.

"You are a young student."
"You are not a student but a young adult."
"You are neither a student nor a young adult."

Hint: You may use an integer ***age*** variable to hold the age of a person and a ***isStudent*** Boolean variable to hold whether a person is a student

or not. Assume that people aged between 18 to 30 years old are considered young adults.

SWITCH STATEMENT

You have seen how using an ***if..else*** statement is an efficient method to solve problems with a couple of options to choose from. However, in cases where we need many options to choose from, using nested ***if..else*** statements can be confusing and error-prone. For such scenarios, the ***switch*** statement provides a nicer way to handle many options to choose from. After having enough experience, you will be able to decide when using ***if..else*** statements are more suitable than using ***switch*** statements, and vice versa.

Example 06

Write a program using a ***switch*** statement to print the day of the week based on user input. That is, if a user enters 1, the program must print out Monday, if a user enters 2, the program must print out Tuesday, and so on.

Study and run the given solution.

```cpp
#include <iostream>
using namespace std;

int main() {
    int day;
    cout << "Enter a number between 1 and 7: ";
    cin >> day;

    switch (day) {
    case 1:
        cout << "Monday \n";
        break;
    case 2:
        cout << "Tuesday \n";
        break;
    case 3:
        cout << "Wednesday \n";
        break;
    case 4:
        cout << "Thursday \n";
        break;
    case 5:
        cout << "Friday \n";
        break;
    case 6:
        cout << "Saturday \n";
        break;
    case 7:
        cout << "Sunday \n";
        break;
    default:
        cout << "Invalid input !\n";
    }
    return 0;
}
```

From the example, you can see that to use the ***switch***, you must provide an expression inside the parentheses. This expression can be as simple as a single variable, or it can be a complex expression, as you learned at the beginning of this chapter. In this example, the ***switch*** expression is a simple day variable. The value of this variable will be matched with only one ***case***. The statement ***case*** can ONLY take numbers, characters, enums, or Boolean. You can add as many lines of code for each ***case*** as you want. Note the importance of providing the ***break*** statement at the end of every ***case***. If you did not do that, the execution will continue to run lines of code in the next ***case***. In this example, this is wrong; however, in the next example, this is exactly what we want. It always depends on what problem you are trying to solve. Finally, the ***default*** statement is optional, which means, if you did not add it, the compiler will not complain and your code will run. However, adding a ***default*** statement is useful to catch any

values that do not match any of the provided cases. In this example, we used the *default* statement to catch any wrong input from users. When testing the code, try inputting unacceptable values for the day variable, like 12, 100, or even a negative number. The *default* statement will print out an error message telling users that what they input was wrong.

Example 07

Write a program that asks users to enter ONLY one letter to decide whether this letter is a vowel or a consonant. For simplicity, we assume that users will always enter letters and there is no chance of mistakes like entering numbers or other unacceptable inputs. In this example, start by writing an algorithm that describes your solution, then write the code. When you are done, compare your solution with the provided one.

The Algorithm:

1) Ask users to enter only one character

2) We need to write 26 cases to accommodate all English letters.

3) Print out an appropriate message for vowels and constant letters.

Refining the Algorithm!

After a second thought. Instead of writing 26 cases, which is technically possible but a tedious task to do, we can use a trick as shown in the given code below. Try to explain how this trick works. Note that we did not provide the *break* statement for the first four cases, which will result in continuing the execution until *case* 'u'. You can use this trick whenever you have several cases that share the same output or behavior. All other letters will be caught by the *default* case.

```cpp
#include <iostream>
using namespace std;

int main() {
    char c;
    cout << "Enter a character: ";
    cin >> c;
    switch (c) {
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
        cout << "The character is a vowel.\n";
        break;
    default:
        cout << "The character is a consonant.\n";
        break;
    }
    return 0;
}
```

Exercise 09

Write a program using a *switch* statement to determine the season based on what month entered by a user. If a user enters 12, 1, or 2, WINTER must be printed out to the console. Similarly, 3, 4, or 5 are the SPRING months, 6, 7, and 8 are the SUMMER months, and 9, 10, and 11 are the FALL months. Start by writing an algorithm and then translate the algorithm to code.

NESTED LOOPS

In Chapter 2, you have learned how to write *for* and *while* loops. In almost all structures that you are going to learn in this book, you can nest inside each other. However, the rule of thumb in programming is that whenever you find yourself writing such complex code, you MUST revise your algorithm and try to find a simpler way to solve the problem at hand.

Example 08

Study and run the following code. Comment on the outputs and explain how the code works before reading the explanation.

```
#include <iostream>
using namespace std;

int main() {
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j <= i; j++) {
            cout << "* ";
        }
        cout << endl;
    }
    return 0;
}
```

The best way to understand how code works is by **tracing** an example. **Tracing** is a technique used by programmers to **trace** the execution of the code line by line. In this way, they can find errors or notice any unusual behavior. In the outer loop, the variable `i` will start with the value `i = 0`. Then, for every iteration of the outer loop, the inner loop will iterate for the entire range of `j` values. That is, in the first iteration and when `i = 0`, `j` will range from `0` to `0`, because of the inner loop condition `j <= i`. In this case, only one `*` will be printed out to the console. When the inner loop finishes its entire range of allowable iterations, it is now time for the outer loop to move on and increment `i` value to become `i = 1`. In this case, the inner loop will loop for `j` values as `j = 0` and `j = 1`. In this case, two `**` will be printed out to the console. This pattern will continue until `i` value becomes `i = 9`, and `j` values range from `j = 0` up to `j = 9`. In the last iteration when `i = 10`, the outer loop condition `i < 10` will return false and terminate the execution of the nested loops.

Tracing can involve many complex techniques to debug and analyze code. To make it easier to follow the execution of this example, we can add a couple of ***cout*** statements to trace how the variables' values change. This can be particularly helpful if the code compiles and runs but produces unexpected results. The case is known as having **Logical Errors**. Reenter and run the code again

after adding the suggested ***cout*** statements in the code below. Comment on the results.

Example 08 (*continued*)

```
#include <iostream>
using namespace std;

int main() {
    for (int i = 0; i < 5; i++) {
        cout << "  i = " << i << endl;
        for (int j = 0; j <= i; j++) {
            cout << "* ";
            cout << "j = " << j << endl;
        }
        cout << "Inner loop is terminated!" << endl;
        cout << endl;
    }
    cout << "Outer loop is terminated!" << endl;
    return 0;
}
```

Example 09

Write nested loops to print the 10X5 multiplication table. Use *for* as the outer loop and *while* as the inner loop. As always, try to solve this problem by yourself before checking the provided solution.

```cpp
#include <iostream>
using namespace std;

int main() {
    int rows = 10;
    int columns = 5;

    for (int i = 1; i <= rows; i++) {
        int j = 1;
        while (j <= columns) {
            cout << i << " X " << j << " = " << i * j << "\t";
            j++;
        }
        cout << endl;
    }
    return 0;
}
```

This example shows that loops can be nested in any way we deem suitable. That is, we can nest a ***while*** loop inside a ***for*** loop and vice versa. Of course, we can write nested ***while*** loops as well. Use the tracing technique to run and examine the code. Be sure to understand how variables' values change!

Exercise 10

Choose true or false and explain your choices:

1) The number of iterations in nested loops is determined by the product of the iteration counts of each individual loop.
2) It is possible to have multiple levels of nesting in C++ loops.
3) The inner loop in a nested loop structure completes all its iterations before the outer loop moves to the next iteration.
4) It is possible to break out of both the inner and outer loops simultaneously in a nested loop structure using a single break statement.

Exercise 11

Modify example 8 code to change the asterisk pattern to:

a) square shape
b) diamond shape

Exercise 12

Using nested loops, write a program to calculate the total number of rooms in a building. The program should prompt the user to enter the number of floors in the building, and then for each floor, prompt the user to enter the number of rooms on that floor. The program should display the total number of rooms in the building after gathering all the necessary information.

WRAP UP!

This chapter delved into the intricacies of composing intricate expressions and their application within structures such as the *if..else* construct. Mastery of this skill can wield significant influence, particularly in crafting nested *if..else* statements, where multiple conditions dictate program flow. Additionally, we explored the switch statement as a versatile alternative, enabling users to select from an array of available options within a program's framework. Lastly, we introduced nested loops, indispensable for tackling complex problems requiring iterative solutions. These constructs empower programmers to navigate through various scenarios with clarity and efficiency, enhancing the robustness of their code.

ANSWERS KEY

Exercise 01

- 0
- 2
- 2
- 4
- 0

Exercise 02

- true
- false
- true
- true
- false
- false

Exercise 03

- 5
- 0
- 30
- 0

Exercise 04

- x = 9
  x = 8
  x = 7
- y = 20
  y = 40
  y = 80
- z = 5

z = 2.5
z = 1.25

- d = 1
  d = 1
  d = 1

Exercise 05

- 13
- 16
- false
- true
- true

Exercise 06

1) counter = 1
2) grade = 84
3) first = 4, second = 4
4) var1 = 9, var2 = 9
5) begin = 5, end = 6
6) g = 9, h = 8

Exercise 07

The Algorithm:

1) Create three variables x, y, and z
2) Ask the user to enter 3 numbers
3) Save the entered numbers into x, y, and z variables
4) If x > y and x > z then x is the largest
5) If x > y and x < z then z is the largest
6) If x < y and y > z then y is the largest
7) If x < y and y < z then z is the largest

The UML of the algorithm:

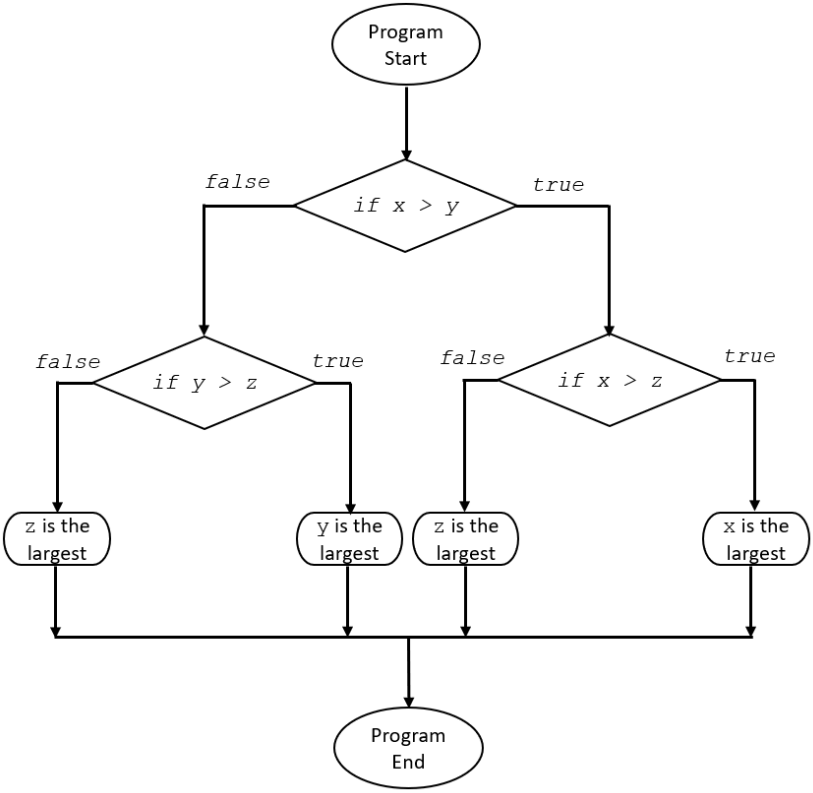


Figure 03: UML for exercise 07.

The code:

```
#include <iostream>
using namespace std;

int main() {
    double x, y, z;
    cout << "Please enter three numbers, then hit the ";
    cout << "Enter key. Numbers can be either integers ";
    cout << "or decimals, positive or negative." << endl;
    cin >> x >> y >> z;

    if (x > y) {
        if (x > z) {
            cout << x << " is the largest number." << endl;
        }
        else {
            cout << z << " is the largest number." << endl;
        }
    }
    else {
        if (y > z) {
            cout << y << " is the largest number." << endl;
        }
        else {
            cout << z << " is the largest number." << endl;
        }
    }
    return 0;
}
```

Exercise 08

The Algorithm:

1) Create the variables, int age and bool isStudent
2) Assign values to the age and isStudent variables
3) If age is in the range of 18 to 30 years old and isStudent is true, then print this message ”You are a young student.”
4) If age is in the range of 18 to 30 years old and isStudent is false, then print this message:” You are not a student but a young adult.”
5) If age is not in the range of 18 to 30 years old and

isStudent is true, then print this message: " You are a student but not a young adult."

6) If none of the previous conditions match, print this message:" You are neither a student nor a young adult."

The code:

```
#include <iostream>
using namespace std;

int main() {
    int age = 25;
    bool isStudent = true;

    // you have to change the values of the age and isStudent.
    // variables to get all possible output messages.
    // Note how isStudent boolean variable is used in this exercise.

    if (age >= 18 && age <= 30 && isStudent) {
        cout << " You are a young student." << endl;
    }
    else if (age >= 18 && age <= 30 && !isStudent) {
        cout << " You are not a student but a young adult." << endl;
    }
    else if (!(age >= 18 && age <= 30) && isStudent) {
        cout << " You are a student but not a young adult." << endl;
    }
    else {
        cout << " You are neither a young adult nor a student." << endl;
    }
    return 0;
}
```

Exercise 09

The Algorithm:

1) Ask the user to enter a number from 1 to 12.

2) Create cases to match the 12-month numbers. Remember, you can use cases without break statements for cases that have the same output.

The code:

```
#include <iostream>
using namespace std;

int main() {
    int month;
    cout << "Enter a month (1 - 12): ";
    cin >> month;

    switch (month) {
    case 12: case 1: case 2:
        cout << "It is WINTER !\n";  break;

    case 3: case 4: case 5:
        cout << "It is SPRING !\n";  break;

    case 6: case 7: case 8:
        cout << "It is SUMMER !\n";  break;

    case 9: case 10: case 11:
        cout << "It is FALL !\n";    break;

    default:
        cout << "Invalid month number!\n";
        break;
    }
    return 0;
}
```

Exercise 10

1) true: Because for every iteration of the outer loop, the inner loop will loop through its entire range of iterations. That is, if an outer loop loops **n** times and the inner loop loops **m** times, then the total number of nested loops iterations is **n X m**.

2) true: It is evident from the given examples.

3) false: Because it is possible to add *if* and *break* statements to terminate the loop earlier than its entire range of iterations.

4) false: You can terminate either the outer or the inner loop by using code similar to *if(condition) break;*

Exercise 11

a) The same code as example 8 with only one change. In the inner **for**-loop, change the condition from **j <= i** to **j <= 10**.

b)

```
#include <iostream>
using namespace std;

int main() {
    int n = 5; // n is the number of rows in the top half of the diamond

    // Print the top half of the diamond
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n - i; j++) {
            cout << " ";
        }
        for (int j = 0; j < 2 * i + 1; j++) {
            cout << "*";
        }
        cout << endl;
    }

    // Print the bottom half of the diamond
    for (int i = n - 2; i >= 0; i--) {
        for (int j = 0; j < n - i; j++) {
            cout << " ";
        }
        for (int j = 0; j < 2 * i + 1; j++) {
            cout << "*";
        }
        cout << endl;
    }

    return 0;
}
```

END OF CHAPTER EXERCISES

1. What is the result of the following expression?

```
int result = (5 + 2) * 3 / 2 - 1;
```

(a) 10
(b) 9
(c) 8
(d) 7

2. What is the value of z after executing the following code?

```
int x = 5;
int y = 3;
int z = (x += y) * (x - y);
```

(a) -45
(b) 0
(c) 11
(d) 40

3. Which expression is equivalent to `!(x || y)`?

(a) `!x && !y`
(b) `x && y`
(c) `!x || !y`
(d) `x || y`

4. What is the value of x after evaluating the following expression?

```
int x = 10 % 3 + 2 * 4 - 8 / 2;
```

(a) 5
(b) 9
(c) 0
(d) -4

5. Order the following operators according to their precedence level.

(a) Assignment (=)
(b) Multiplication (*)

(c) Logical AND (&&)
(d) Addition (+)

6. Write a nested `if-else` statement that checks if a number is positive, negative, or zero.
7. Write a nested `if-else` statements that check if a student's grade is "A", "B", "C", or "Fail" based on their score.
8. Can you nest `if-else` statements within the `else` block of another `if-else` statement? Explain your answer.
9. Write a nested `if-else` statement that determines if a number is divisible by both 2 and 3.
10. Write a nested `if-else` statement to determine if a given character is a vowel or a consonant.
11. What will be the output of the following code snippet?

```
int x = 5;
int y = 10;
if (x > 0 && y < 20) {
    cout << " Condition is true ." << endl;
}
else {
    cout << " Condition is false ." << endl;
}
```

(a) Runtime Error.
(b) Condition is false.
(c) Compiler Error.
(d) Condition is true.

12. What will be the output of the following code snippet?

```
int m = 20;
int n = 30;
if (m > 10 && n < 25) {
    cout << "Condition 1 is true." << endl;
}
else if (m > 10 && n > 25) {
    cout << "Condition 2 is true." << endl;
}
else {
    cout << "No condition is true." << endl;
}
```

(a) Condition 1 is true.
(b) Condition 2 is true.
(c) No condition is true.
(d) Logical error!

13. What will be the output of the following code snippet?

```
int age = 25;
bool isStudent = true;
bool hasDiscount = false;
if ((age >= 18 && age < 25) && (isStudent || hasDiscount)) {
    cout << "You are eligible for a discounted price." << endl;
}
else {
    cout << "You are not eligible for a discounted price." << endl;
}
```

(a) You are eligible for a discounted price.
(b) You are not eligible for a discounted price.
(c) Syntactical error!
(d) Logical error!

14. Write an `if .. else` statement using compound conditions that checks if a student's grade is between 70 and 100 (inclusive) and if their attendance is at least 80%. If both conditions are met, output "Passing grade". Otherwise, output "Failing grade".

15. Write an `if .. else` statement using compound conditions

that checks if a person's age is either less than 18 or greater than 65, and their membership status is active. If either condition is true, output "Eligible for special discount". Otherwise, output "Not eligible for special discount".

16. Write an `if .. else` statement using compound conditions that checks if a customer's total purchase amount is greater than 100$ and they have a valid coupon code, or their membership level is "Premium". If either condition is true, output "Discount applied". Otherwise, output "No discount is available".

17. Write a `switch` statement that takes a variable direction of type char representing a cardinal direction ('N', 'S', 'E', or 'W') and prints a corresponding message indicating the full name of the direction. If the
direction is not recognized, print "Unknown direction".

18. Write a `switch` statement that takes a variable day of type `int` representing the day number in a month (1 for the first day, 2 for the second day, etc.) and prints a corresponding message indicating the suffix of the day. For example, if the day is 1, print "1st", if the day is 17, print "17th", if the day is 23, print
"23rd", and so on. If the day is not valid (less than 1 or greater than 31), print "Invalid day".

19. Write a `switch` statement that takes a variable month of type `int` representing the month number (1 for January, 2 for February, etc.) and prints the number of days in that month. January, March, May, July, August, October, and December have 31 days. April, June, September, and November have 30 days.
February has either 28 or 29 days. If the month number is less than 1 or greater than 12, print "Invalid
month".

20. Write a nested `for` loops that print the following pattern:

```
    1
  2 3
```

4 5 6
7 8 9 10

21. Write a nested `for` loops that print the following output:

Sum of numbers from 1 to 1: 1
Sum of numbers from 1 to 2: 3
Sum of numbers from 1 to 3: 6
Sum of numbers from 1 to 4: 10
Sum of numbers from 1 to 5: 15

22. Rewrite exercise 21 to sum only odd numbers.

Hint: You can use the `if (condition) continue;` statement to skip the current iteration of a loop.

23. Rewrite exercises 21 and 22 using nested `while` loops.

PROJECTS

Project 01 Prime Numbers: Write a program that prompts the user to enter an integer greater than 2 (denoted as N) and finds and prints all the prime numbers between 3 and N. A prime number is a number that can only be evenly divided by 1 and itself, such as 3, 5, 7, 11, 13, 17, and so on.

To solve this problem, you can use a nested loop. The outer loop will iterate from 3 to N, while the inner loop will check if the current value of the outer loop counter is prime. One way to determine if a number n is prime is to loop from 2 to n-1 and check if any of these numbers evenly divide n. If any number from 2 to n-1 divides n evenly, then n cannot be prime. Conversely, if none of the values from 2 to n-1 evenly divide n, then n must be prime. The program must ask the user if they wish to rerun the code again. (Note that there are more efficient algorithms to solve this problem, but for simplicity, we will use this approach.)

Project 2: Rock, Paper, Scissors: This is a simple game where two players choose either rock, paper, or scissors. The program can determine the winner based on the rules of the game :

- Rock beats scissors
- Scissors beats paper
- Paper beats rock

CHAPTER 4

Writing Basic Functions

HUSSAM GHUNAIM

Learning Objectives

At the end of this chapter, you will be able to

- Use many of the pre-defined functions from various C++ libraries
- Write your own user-defined functions
- Differentiate between different types of functions, like *void* functions and functions that return values

PRE-DEFINED FUNCTIONS

Functions can be thought of as containers that contain code inside them. You have been using only one function in all programs written so far, called the *main* function. This was ok, as all the programs you wrote were relatively small. However, imagine if you need to write larger programs that contain hundreds or even

thousands of lines of code. These programs will be very hard to review, debug, and modify. That is the reason why you must think of your code as built of various small pieces.

This way of thinking will perfectly match the algorithmic way of thinking. When you first write an algorithm to solve a certain problem, usually every task or step in the algorithm is translated into one function. In this chapter, you will learn to write and call many more functions called ***user-defined*** functions. Nevertheless, many other functions are already written and available for us to use right away, called ***pre-defined*** functions. These functions are saved in larger containers called libraries.

All functions written in C++ follow the same format:

```
returnType functionName(parameterType parameterName) {
//body of the function
}
```

To use functions correctly (either pre-defined or user-defined), you must understand very well the types of ***parameters***[1] (if any), the function expects to have, and the returned type as well. If a function is defined to accept a ***string*** parameter, but you call it with a numeric type, like ***int*** or ***double***, an error message will be thrown. Similarly, knowing the returned type of a function is important because, generally, you need to declare a variable to hold the returned value. For example, if you call a function that returns a ***char*** type, you must declare a variable with the same type to save the returned value.

Functions can have zero or more parameters included inside the parentheses that appear after the function name. If a function takes more than one parameter, a comma must be used to

1. Parameters are the variables declared in a function definition. They define the types and order of values that the function expects to receive when it is called. Parameters act as placeholders for the actual values that will be passed as arguments.

separate the parameters from each other. Example 3 shows this case. Every parameter added must be preceded with it is type. Additionally, functions must have a return type except if it does not return any value. In this case, the return type must be replaced by the keyword ***void***. This topic will be discussed further in the upcoming section.

Table 1 shows some common pre-defined functions and their corresponding libraries. From the Table, you can learn that to call a function, you must use the parentheses associated with its name. These parentheses are used to pass any needed ***arguments***[2]. Later in this chapter, when you start defining your own functions, you will notice that most functions need more information passed to them as arguments to perform their desired operations. Some functions do not need any arguments to work; however, it is still required to write empty parentheses associated with the name of the function. This is particularly important for the compilers to distinguish between the function name from any other variable name in the code.

2. . Arguments are the actual values that are passed to a function when it is called. They correspond to the parameters defined in a function declaration. Arguments provide the specific values that will be used in the execution of a function.

Table 1: Examples of some common pre-defined functions.

Standard Library	Function Name	Function Description	Examples			
			Arguments Types (if needed!)	Returned Type	How to call the function	Returned value
cmath	sqrt	Calculates the square root of the given parameter.	double	double	sqrt(25.0)	5.0
	pow	Raises the first parameter to the power of the second parameter.	double	double	pow(5.0 , 2.0)	25.0
	ceil	Finds the smallest integer greater than or equal to the given parameter value; however, the returned type is double!	double	double	ceil(1.1)	2.0
	floor	Finds the smallest integer less than or equal to the given parameter value; however, the returned type is double!	double	double	floor(1.7)	1.0.
	round	Rounds decimal numbers to the nearest integer value. When the fractional part is less than 0.5, it is rounded down. Conversely, when the fractional part is 0.5 or greater, it is rounded up.	double	double	round(4.45) round(4.5)	4.0 5.0
cstdlib	abs	Finds the absolute value of the given parameter.	int	int	abs(-12)	12

	rand	Generates pseudorandom integer values. It typically generates integer values in the range 0 to RAND_MAX, where RAND_MAX is a constant defined in the <cstdlib> library. The exact value of RAND_MAX may vary depending on the implementation.	void	int	rand()	values range from 0 to RAND_MAX
	srand	Passes a seed value to the rand function.	unsigned int	void	srand(10)	none

Unsigned Types:

Notice in Table 1, `srand` function expects an **unsigned int** as the argument type. In C++, unsigned types are integer types that can only represent non-negative numbers. This is in contrast to signed types, which can represent both positive and negative numbers. The keyword **unsigned** is used to declare unsigned types.

Table 2 compares the common signed and unsigned types in C++ along with their corresponding ranges. One advantage of using unsigned types is that to provide larger ranges whenever you need such a large range to solve problems.

Table 2: Comparison between signed and unsigned types.

Type	Size (bits)	Size (bytes)	Minimum Value	Maximum Value
char	8	1	-128	127
unsigned char	8	1	0	255
int	32	4	-2,147,483,648	2,147,483,647
unsigned int	32	4	0	4,294,967,295
long int [3]	64	8	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
unsigned long int [4]	64	8	0	18,446,744,073,709,551,615

To understand how unsigned types can provide larger ranges, we can recall that numbers are represented in computer memory as binary numbers. That is, if we have only 1-bit to represent numbers, then we can only represent two numbers, namely 0 and 1. If we choose to use 2-bits, then we can have four numbers, 00, 01, 10, and 11. By noticing the pattern, we can see that the number of possibilities equals 2^n, where n is the number of bits used. For example, in Table 2, when raising 2 to the power of 8 bits, you will get the range of numbers available for the unsigned char type (notice that we start counting from 0). Similarly, you can calculate the range for unsigned int and long int by raising 2 to the power of the corresponding number of bits. On the other hand, signed types need to take one bit to represent the sign (usually 0 means positive and 1 means negative). That is, in the case of a signed char, 1 bit will be used to represent the sign, leaving 7 bits to represent the range; hence, the range now will split into two halves. One half represents the positive range starting from 0, and the other half represents the negative range. Similarly, for the signed int and long int, you can

3. The exact type size depends on your system and compiler.
4. The exact type size depends on your system and compiler.

recalculate the range after taking away 1 bit to represent the sign and raising 2 to the power of the remaining bits.

Example 01

Run the following code and notice how **sqrt** and **pow** functions are used.

```
#include <cmath>
#include <iostream>
using namespace std;

int main() {
    //what is the square root of 64?
    double x = 64;
    double y = sqrt(x);
    cout << "The square root of 64 is " << y << endl;

    //what is the result if you raised 3 to the power of 5?
    double z = pow(3, 5);
    cout << "3 to the power of 5 is " << z << endl;

    return 0;
}
```

rand function is used to generate pseudorandom numbers, which means not real random numbers. Random number generators use mathematical formulae to calculate these numbers; hence, they are predictable. However, their use is generally acceptable in many applications since pseudorandom numbers have statistical properties similar to true random numbers. Random number generators use a ***seed*** value to generate random numbers, which results in a specific set of random numbers for every seed value selected. If you need to generate a different set of random numbers every time a program runs, you need to find a way to randomly choose different seeds. Example 2 explains how you can do that.

Example 02

Run the provided code and notice that you are getting the same set of random numbers every time you run the code. To get a different set of random numbers, remove the double forward slash characters **'//'** to activate line 6 and run **srand(5)** function.

Although you get a new set of random numbers, they are also the same every time you run the code. If you want to get a different set of random numbers every time you run the code, you need to provide a different seed value to the **srand** function for every run.

The easiest way to do this is by calling the pre-defined **time** function. This function returns the system's current time in seconds. This value can be used as the seed value. To see how this works, replace line 6 with this code: **srand (time (0));** and run the code several times.

Notice how the **time** function has been passed as an argument to the **srand** function. Remember to include the **<ctime>** library to be able to use the **time** function.

```
1   #include <cstdlib>
2   #include <iostream>
3   using namespace std;
4
5   int main() {
6       //srand(5);
7       //5 can be replaced by any unsigned integer
8       for (int i = 0; i < 5; i++)
9           cout << rand() << "\t";
10
11      return 0;
12  }
```

Exercise 01

Write a program that asks a user to enter a floating-point number (a number with decimals) and prints the ceiling and floor values of that number using the pre-defined functions **ceil** and **floor**.

Exercise 02

Write a program that asks a user to enter a positive number and prints the square root, ceiling, and floor values of that number using **sqrt**, **ceil**, and **floor** functions.

USER-DEFINED FUNCTIONS

Writing functions in C++ involves two steps: function declaration and definition. You can not use a function before it has been declared and defined. Otherwise, the compiler will complain that it is not able to execute the called function. Another restriction is that you cannot define a function inside another function. That is, every function must be declared and defined independently from any other function. There are two methods to declare and define functions.

Method 1:

First, you have to declare the function ***prototype*** above the *main* function. The function is called a ***prototype*** at this point because it is still not functional. The function still needs to be defined.

Second, define the function by writing all necessary code inside the body of the function below the *main* function.

Example 03

Declare and define a function called ***average*** that accepts two parameters of type ***int*** and returns the average of type ***double***.

Answer

Let us use the general syntax for writing functions,

```
returnType functionName(parameterType parameterName) {
//body of the function
}
```

First, we declare the ***average*** function above the ***main*** function.

```
double average(int num1, int num2);
//the function prototype or header
```

Second, define the ***average*** function below the ***main*** function.

```
double average(int num1, int num2){
      double ave = (num1 + num2)/2.0;
return ave;
}
```

The following code is the complete example code. Note the positions where the ***average*** function is declared and then defined. Once the new function is declared and defined, it is now ready to be called. You can call functions in a similar manner to how you would call any other variable name. Lines 9, 11, and 13 show several ways to call the ***average*** function. It is important here to visualize how the code is executed. When you run this code, the execution will start from the beginning until it reaches line 9. When the ***average*** function is called, the execution will jump to the definition of the function in line 19 and execute its body. When the function completes its execution, the execution will go back to line 9, where it was exactly stopped, and continue from there. The same process happens when the function is called again in lines 11 and 13.

```cpp
#include <iostream>
using namespace std;

// function declaration
double average(int num1, int num2);

int main() {

    cout << "The average of 4 and 8 is " << average(4, 8) << endl;

    double newVariable = average(4, 8);
    double calculationResult = (5 + newVariable * 2) / 3;
    double anotherResult = (5 + average(5, 15) * 2) / 3;

    return 0;
}

//function definition
double average(int num1, int num2) {
    double ave = (num1 + num2) / 2.0;
    return ave;
}
```

Method 2:

If you really do not like to declare functions and then define them, it is possible to immediately define functions above the *main* function without the need to declare them first.

Exercise 03

Write a function called **add** that takes two `double` parameters and returns their sum. Declare the function prototype at the top of your code and define the function below the *main*. In the *main*, call the **add** function with the arguments `10.05` and -7, and store the result in a variable. Print the result to the screen.

Exercise 04

Write a function called ***myName*** that takes no arguments. The function should read the user name from the console and return it back to the caller. In the *main* function, ***myName*** must be called twice, one time to read the user's first name, and then the second call is to read the user's last name. The *main* function must print to the screen the following message: "Nice meeting you firstName secondName!".

Exercise 05

What is the difference between declaring and defining functions in C++?

PRECONDITIONS AND POSTCONDITIONS

Writing meaningful comments is one of the essential skills in software engineering. For first-time programmers, it is an even more crucial skill to focus on and practice. Therefore, in this book, you will frequently be asked to provide appropriate comments in the code. In Chapter 2, ***preconditions*** and ***postconditions*** comments were introduced and are reemphasized here. In these comments, programmers describe what functions expect to receive and what they are expected to return. If they are well-written, you can use the function confidently without issues because you know how to work with the function. Writing well-formatted comments is a very common practice in designing and developing software.

Example 04

Write precondition and postcondition comments for a function that calculates the square root of a number.

Answer

```
// precondition: the given number x must be a non-negative number.
// postcondition: the square root of x will be calculated and returned.
```

Example 05

Write precondition and postcondition comments for a function that calculates the factorial of a number.

Answer

```
// precondition: the given number n must be a non-negative.
// postcondition: the factorial of n will be calculated as follows: n! = 1 X 2 X .. X (n-2) X (n-1 ) X n.
```

Exercise 06

Write precondition and postcondition comments for the following function headers:

1. double calculateHypotenuse(double leg1, double leg2);
2. bool isLeapYear(int year);
3. bool isValidEmailAddress(string email);
4. double calculateMonthlyPayment(double principal,

```
    double interestRate, int loanTerm);
5.  double calculateCirclePerimeter(double radius);
6.  double kilometersToMiles(double kilometers);
7.  bool isValidEmail(string email);
```

NUMERIC FUNCTIONS

Functions can be written to satisfy almost any need or requirement. In this section, the focus will be on functions that perform a set of calculations and return the result back to the caller.

In example 4, you have written the precondition and postcondition for the square root function. Let us now develop the code for it. Calculating the square root of a positive number can be done through various algorithms. Here we will use a simplified algorithm called the Babylonian Algorithm. This algorithm iteratively approximates the square root. Note that this iterative method is an approximation and may not provide the exact square root.

Example 06

Write a function that takes any positive number as a parameter and returns its square root using the Babylonian algorithm.

The Babylonian Algorithm:

- Step 1: Make an initial guess for the square root of the given number. This can be any positive number. For simplicity, we will let the ***initial guess = given number***.

- Step 2: Divide the given number by the ***guess*** to get the ***quotient***.
- Step 3: Average the ***guess*** and the ***quotient*** to get a new guess. Let us call it the ***root***.
- Step 4: If the absolute value of the difference between the ***root*** and ***guess*** is less than the precision value, return the current value of the ***root*** as the acceptable square root, otherwise, update the guess value and repeat steps 2, 3, and 4.

The code:

```
#include <iostream>
#include <cstdlib>
using namespace std;

// declaration of sqrRoot prototype
// precondition: the given number must be a non-negative number.
// postcondition:the square root of the given number will be
//               approximated by the Babylonian algorithm.
double sqrRoot(double number);

int main() {
    double number;
    cout << "Enter a number: ";
    cin >> number;
    cout << "Square root of " << number << " = " << sqrRoot(number) << endl;
    return 0;
}

//definition of sqrRoot function
double sqrRoot(double number) {
    double precision = 0.00001;
    double guess = number;   //step 1
    double root, quotient;

    while (true) {
        quotient = (number / guess);    //step 2
        root = (guess + quotient) / 2;  //step 3
        if (abs(root - guess) < precision) return root;  //step 4
        guess = root;  //step 4
    }
}
```

In example 5, you have written the precondition and postcondition for the factorial function. Let us now develop the code for it.

Example 07

Write a function that takes an *int* as a parameter and returns its factorial.

The Algorithm:

Calculating the factorial of any positive number is quite simple and straightforward. All that we have to do is create a loop that multiplies integers starting from 1 and incrementing by 1 up to the required number. This formula can help us visualize this procedure: `n! = 1 X 2 X .. X (n-2) X (n-1 ) X n.`

In designing solutions, giving examples is very helpful as well:

0! = 1 // by definition

1! = 1

2! = 1 X 2 = 2

3! = 1 X 2 X 3 = 6

4! = 1 X 2 X 3 X 4 = 24

.. and so on.

The code:

```
#include <iostream>
using namespace std;

//declaring the factorial function prototype.
// precondition:  the given number n must be a non-negative.
// postcondition: the factorial of n will be calculated as follows:
// n! = 1 X 2 X .. X (n-2) X (n-1 ) X n.
int factorial(int n);

int main() {
    int n;
    cout << "Enter a non-negative integer: ";
    cin >> n;
    if (n >= 0)
        cout << "Factorial of " << n << " is " << factorial(n) << endl;
    else
        cout << "You must enter a non-negative number!";
    return 0;
}

//definition of fatorial function
int factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++) {
        result *= i;
    }
    return result;
}
```

Test the provided code for several n values like 5, 10, and 20. What do you notice?! This code will be able to provide accurate results up to factorial 12. If you entered values for n larger than 12, surprisingly, you will get unreasonable results, like either negative numbers or zero! The problem here comes from the int type that, in most C++ implementations, has a size of 4 bytes (32 bits).

This means that any variable declared with the int type will be capable of holding integers ranging from -2,147,483,648 to 2,147,483,647. This range represents the positive and negative values that can be stored in a signed 32-bit integer using two's complement representation. The most significant bit is used to represent the sign of the number, leaving 31 bits for the

magnitude. Therefore, for n > 12, the code will cause a memory overflow, resulting in unexpected outcomes.

There are some specialized libraries that can help in solving this problem. However, here we can partially solve this problem with a larger type than int, which is long long int. This type uses 8 bytes to be saved in the memory and can hold numbers ranging from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

Exercises 07

Modify the code in example 07 by replacing the type int with long long int. What is the largest n you can correctly calculate n! for it on your system?

Exercise 08

Write a program that asks the user to enter a radius value. Then, the ***main*** function will call two functions, ***circleArea***, and ***sphereVolume,*** to calculate the area of the circle and volume of the sphere using the given radius. Both functions take one parameter of type ***double*** and return a ***double*** result. The code must have adequate preconditions and postconditions comments.

$$\text{Area of the Circle: } A = \pi r^2$$

$$\text{Volume of the Sphere: } V = \frac{4}{3}\pi r^3$$

BOOLEAN FUNCTIONS

Some functions are designed to help us in making decisions. Rather than including the needed logic in the ***main*** function, for clarity, we can delegate this job to another function to do so. Study the following examples and notice how ***Boolean*** functions can be used mainly as testing functions.

Example 08

Write a function that takes an int number and returns true if the number is even, otherwise, it returns false.

```
bool isEven(int num) {
   return num % 2 == 0;
}
```

Example 09

Write a function that takes a char variable and returns true if the character is a vowel, otherwise, it returns false.

```
bool isVowel(char ch) {
   ch = tolower(ch);
   return ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o
}
```

Note: This function uses a pre-defined function ***tolower*** from the **<cctype>** library. Intuitively, this function converts any upper-case letter to a lower-case letter.

Exercise 09

Write a function that returns true if a number has five digits, otherwise, it returns false.

Exercise 10

Sometimes, we can use *Boolean* functions to enhance the code readability. For example, consider the following code:

The *if-statement* condition can be hard to read and contribute to the overall complexity of the code. Alternatively, we can move this condition into a B*oolean* function with an appropriate name: *if (checkEligiblity())*. The *checkEligiblity()* function will return either true or false to the *if-statement*.

```cpp
#include <iostream>
#include <string>
using namespace std;

int main() {
    double GPA = 3.75; //GPA range from 0 - 4 points
    double SAT = 1350; //SAT scores range from 400 - 1600
    string status = "sophomore"; //freshman, sophomore, junior, senior

    if (GPA >= 3.5 && SAT >= 1100 && (status == "junior" || status == "senior"))
        cout << "Congratualtions! You are eligible for a scholarship." << endl;
    else
        cout << "Sorry! You are not elegible for a scholarship." << endl;

    return 0;
}
```

Rewrite the given code by implementing the *checkEligiblity()* function.

NOTE: Practically, we might want users to enter values for GPA, SAT, and status variables. For the sake of simplicity, we hard-coded those values in this exercise.

Exercise 11

True or False:

1. A function that returns a Boolean value can have multiple return statements throughout its body.
2. A Boolean function in C++ cannot take parameters.
3. The return type of a Boolean function can be any data type.
4. A Boolean function can call other functions within its body.
5. A Boolean function can be used as a while loop condition.

VOID FUNCTIONS

void functions are a special type of functions that do not return any value back to the caller. However, to comply with the compiler requirements, we must add the keyword *void* as a placeholder in place of the regular type. *void* functions are useful when your needs are basically performing actions rather than calculations.

Example 10

Define a *void* function that takes a user name as a *string* type and prints a welcome message to the console.

```cpp
#include <iostream>
#include <string>
using namespace std;

//declare the welcomeMessage function
void welcomeMessage(string userName);

int main() {
    cout << "What is your name? ";
    string name;
    cin >> name;

    //calling the void function
    welcomeMessage(name);

    return 0;
}

//define welcomeMessage function
void welcomeMessage(string userName) {
    cout << "Welcome " << userName << " to our program!" << endl;
    cout << "We hope you will find it useful." << endl;
}
```

Note in example 10 that *void* functions do not have a return statement. This makes sense as *void* functions are not supposed to return any values. However, adding the return statement does not cause errors. Additionally, if the user enters their full name or full name with a middle initial, the code will only print out the first name. This is because the stream extraction operator **>>** reads input until a whitespace is encountered, effectively using spaces to separate different values. If you want to read everything the user enters, you can replace the code ***cin >> name;*** with ***getline(cin, name);***. The predefined ***getline*** function reads the entire line and saves it in the ***name*** variable. Be sure to include the `<string>` library to be able to use this function.

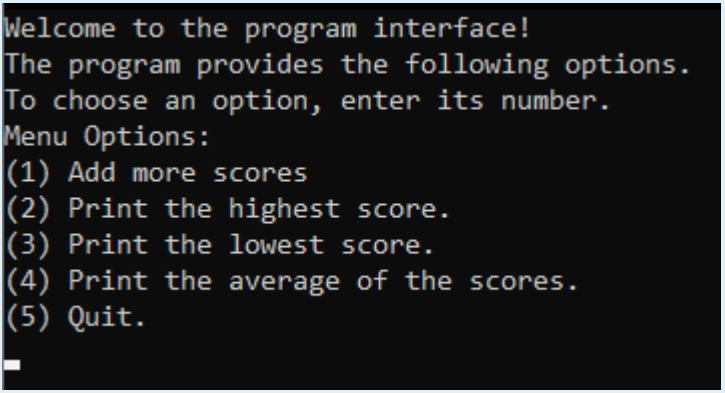
Example 11

Write a *void* function called *printTable* that takes two *double* parameters, num1 and num2, and prints them to the console formatted in a table shape. In the *main* function, the square and cubic roots will be calculated and passed to the *printTable* function.

```
#include <iostream>
#include <cmath>
#include <iomanip>
using namespace std;

//declare the printTable function
void printTable(double num1, double num2);

int main() {
    double squareRoot = sqrt(2);
    double cubicRoot = pow(2, 1.0/3.0);

    //calling the void function
    cout << "The square and the cubic roots of 2 are: " << endl;
    printTable(squareRoot, cubicRoot);

    return 0;
}

//define printTable function
void printTable(double num1, double num2) {
    cout << fixed << setprecision(10);
    cout << "-----------------------------------------" << endl;
    cout << "| Square Root\t | \t" << num1 << "\t |" << endl;
    cout << "| Cubic Root\t | \t" << num2 << "\t |" << endl;
    cout << "-----------------------------------------" << endl;
}
```

Exercise 11

Write a *void* function called *openingScreen* that takes no parameters. The function's purpose is to provide information to users on how to use the program. Below is what the function has to print to the console.



In the *main* function, call *openingScreen* function and write code to read user's choice. You do not need to implement any of the menu options.

Exercise 12

True or False:

1. *void* functions always return a value.
2. *void* functions can have parameters and arguments.
3. *void* functions can be called from other functions.
4. *void* functions must always have a return statement.
5. *void* functions can be used to perform complex calculations and print them on the screen.

WRAP UP!

In this chapter, you've delved into the world of functions in C++, exploring both pre-defined and user-defined functions. You've

learned that functions serve as containers for code, allowing for better organization and reusability in your programs. With user-defined functions, you can break down complex tasks into smaller, manageable pieces, enhancing readability and maintainability. Pre-defined functions, housed in libraries, provide a wealth of functionality ready for use in your programs. You've gained an understanding of function syntax, including parameters, return types, and function prototypes. Additionally, you've explored Boolean functions for decision-making and void functions for actions without return values. Through exercises and examples, you've honed your skills in writing, declaring, and calling functions, setting the stage for more advanced programming concepts ahead. As you continue your journey in C++, remember that functions are powerful tools in your programming arsenal, empowering you to write clear, efficient, and scalable code.

ANSWERS KEY

Exercise 01:

```
#include <iostream>
#include <cmath>
using namespace std;

int main() {
    double num, theCeilingValue, theFloorValue;
    cout << "Enter a floating-point number: ";
    cin >> num;

    theCeilingValue = ceil(num);
    theFloorValue = floor(num);

    cout << "The ceiling value: " << theCeilingValue << endl;
    cout << "The floor value: " << theFloorValue << endl;

    return 0;
}
```

Exercise 02:

```cpp
#include <iostream>
#include <cmath>
using namespace std;

int main() {
    double num, sqrRoot, ceilingValue, floorValue;
    cout << "Enter a positive number: ";
    cin >> num;

    sqrRoot = sqrt(num);
    ceilingValue = ceil(sqrRoot);
    floorValue = floor(sqrRoot);

    cout << "Square Root: " << sqrRoot << endl;
    cout << "Ceiling of Square Root: " << ceilingValue << endl;
    cout << "Floor of Square Root: " << floorValue << endl;

    return 0;
}
```

Exercise 03:

```cpp
#include <iostream>
using namespace std;

// Declare function prototype
double add(double first, double second);

int main() {
    // Call add function and store the result
    double result = add(10.05, -7);
    // Print the result
    cout << " The sum of the two numbers is: " << result << endl;
    return 0;
}

// Define add function
double add(double first, double second) {
    return first + second;
}
```

Exercise 04:

```cpp
#include <iostream>
#include <string>
using namespace std;

//declare function prototype wihtout parameters.
string myName();

int main() {
    // readding the user first name
    cout << "What is your first name?" << endl;
    string firstName = myName();

    // readding the user last name
    cout << "What is your last name?" << endl;
    string lastName = myName();

    cout << "Nice meeting you " << firstName << " " << lastName << "!" << endl;

    return 0;
}

//define myName function
string myName() {
    string name;
    cin >> name;
    return name;
}
```

Exercise 05:

In C++, a function declaration, also known as a function prototype, provides the compiler with information about the function's name, return type, and parameters. It does not include the function's body. A function declaration typically appears at the top of a source code file. However, a function definition includes both the function's prototype and its body. The body of the function contains the statements that define what the function does. A function definition typically appears at the bottom of a source code file.

Exercise 06:

1.

// Preconditions: Both legs of the triangle must be positive double values.

// Postconditions: Returns the length of the hypotenuse as a double value.

2.

// Preconditions: Year must be a positive integer.
// Postconditions: Returns true if the year is a leap year; otherwise, false.
3.
// Preconditions: Email address must follow the standard format rules.
// Postconditions: Returns true if the email address is valid; otherwise, false.
4.
// Preconditions: Principal, interest rate, and loan term must be positive values.
// Postconditions: Returns the monthly loan payment amount.
5.
// Preconditions: Radius must be a positive double value.
// Postconditions: Returns the perimeter of the circle as a double value.
6.
// Preconditions: Distance in kilometers must be a positive double value.
// Postconditions: Returns the equivalent distance in miles.
7.
// Preconditions: Email must be a string.
// Postconditions: Returns true if the email is valid; otherwise, false.

Exercise 07

The modified code will calculate n! correctly up to n = 20 on my system.

```cpp
#include <iostream>
using namespace std;

//declaring the factorial function prototype.
// precondition:  the given number n must be a non-negative.
// postcondition: the factorial of n will be calculated as follows:
// n! = 1 X 2 X .. X (n-2) X (n-1 ) X n.
long long int factorial(int n);

int main() {
    int n;
    cout << "Enter a non-negative integer: ";
    cin >> n;
    if (n >= 0)
        cout << "Factorial of " << n << " is " << factorial(n) << endl;
    else
        cout << "You must enter a non-negative number!";
    return 0;
}

//definition of fatorial function
long long int factorial(int n) {
    long long int result = 1;
    for (int i = 1; i <= n; i++) {
        result *= i;
    }
    return result;
}
```

Exercise 08:

```cpp
#include <iostream>
#include <cmath>
using namespace std;

const double PI = 3.14159;//an acceptable approximation for the pi value

// Decleration of circleArea function
// Precondition: radius must be a positive value
// Postcondition: returns the area of the circle
double circleArea(double radius);

// Decleration of sphereVolume function
// Precondition: radius must be a positive value
// Postcondition: returns the volume of the sphere
double sphereVolume(double radius);

int main() {
    double radius;
    cout << "Enter the radius: ";
    cin >> radius;
    cout << "Area of the circle: " << circleArea(radius) << endl;
    cout << "Volume of the sphere: " << sphereVolume(radius) << endl;

    return 0;
}

// Definition of circleArea function
double circleArea(double radius) {
    return PI * pow(radius, 2);
}

// Definition of sphereVolume function
double sphereVolume(double radius) {
    return (4.0 / 3.0) * PI * pow(radius, 3);
}
```

Exercise 09:

```cpp
bool isFiveDigits(string num) {
        bool result = true;
        if (num.length() != 5) result = false;
        return result;
}
```

Exercise 10:

```cpp
#include <iostream>
#include <string>
using namespace std;

//declaring the fucntion
bool checkEligiblity();

int main() {

    if (checkEligiblity())
        cout << "Congratualtions! You are eligible for a scholarship." << endl;
    else
        cout << "Sorry! You are not elegible for a scholarship." << endl;

    return 0;
}

//defining the function
bool checkEligiblity() {
    double GPA = 3.75; //GPA range from 0 - 4 points
    double SAT = 1350; //SAT scores range from 400 - 1600
    string status = "sophomore"; //freshman, sophomore, junior, senior

    return GPA >= 3.5 && SAT >= 1100 && (status == "junior" || status == "senior");
}
```

Exercise 11:

1. true
2. false
3. false
4. true
5. true

Exercise 11:

```cpp
#include <iostream>
using namespace std;

//declare the openingScreen function
void openingScreen();

int main() {
    //calling the void function
    openingScreen();

    int menuOption;
    cin >> menuOption;
    cout << "You have selected option: " << menuOption << endl;
    return 0;
}

//define openingScreen function
void openingScreen() {
    cout << "Welcome to the program interface!" << endl;
    cout << "The program provides the following options." << endl;
    cout << "To choose an option, enter its number." << endl;
    cout << "Menu Options:" << endl;
    cout << "(1) Add more scores" << endl;
    cout << "(2) Print the highest score." << endl;
    cout << "(3) Print the lowest score." << endl;
    cout << "(4) Print the average of the scores." << endl;
    cout << "(5) Quit." << endl;
}
```

Exercise 12:

1. False
2. True
3. True
4. False
5. True

END OF CHAPTER EXERCISES

1. Write a function that converts Fahrenheit temperature

degrees to Celsius using this formula: Celsius = (Fahrenheit – 32) * 5/9. The function should take a ***double*** parameter and returns a ***double***. Add appropriate preconditions and postconditions comments. Hint, in C++ coding, the fraction 5/9 will result in 0 (why?!). To avoid this logical error, you should use 5.0/9.0 instead.

2. In exercise 1, add a while loop to ask users if they wish to do another conversion.
3. In exercise 2, the while loop condition must be a ***bool*** function that returns true if the user wishes to repeat the calculation; otherwise, the function will return false to terminate the loop.
4. In Geometry, the distance between two points in the Cartesian Coordinate System is calculated by:

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

Write a function that asks users to enter the values x_1, x_2, y_1, y_2, and returns the distance between the two points. Round the result to the nearest integer. Add appropriate preconditions and postconditions comments.

5. Write a function that takes a ***string*** as input and returns an ***int*** that represents the number of vowels in the ***string***.
6. Write a program that calculates the sum of all even numbers between 1 and 100. The program must use a Boolean function ***isEven*** that returns true if a number is even, otherwise, it returns false.
7. Write a ***bool*** function that checks whether a given year is a leap year or not. The function takes one ***int*** parameter representing the year value and returns true if the year is a leap year; otherwise, it returns false. You must include

appropriate preconditions and postconditions comments describing the implemented algorithm. This is a good practice if you intend to share your code and want to be sure that other developers will understand how you solved the problem at hand. It would be helpful to conduct some research to gain a better understanding of the reasons behind the occurrence of leap years and their importance in calendar tracking.

The Algorithm:

Step 1: If the year is evenly divisible by 4, continue to step 2; otherwise, return false

Step 2: If the year is evenly divided by 100, continue to step 3; otherwise, return true

Step 3: If the year is evenly divisible by 400, return true; otherwise, return false

Thus, years such as 1996, 1992, and 1988 are leap years because they are divisible by 4 but not by 100. Years that are divisible by 4 and by 100 but not by 400 are not leap years. However, years that are divisible by 4, 100, and 400 are leap years, such as years 1600 and 2000.

8. Write a function that checks if a given sentence is a palindrome. A palindrome sentence is a sentence that reads the same forwards and backwards, considering only alphanumeric characters and ignoring spaces, punctuation, and capitalization. Do not forget the preconditions and postconditions comments to explain your algorithm.

Examples of Palindrome Sentences:

Madam, in Eden, I'm Adam.

Step on no pets.

Never odd or even.

Was it a car or a cat I saw?
No lemon, no melon.

9. Write a *void* function that takes a *string* parameter and prints its reverse.

10. Write a *void* function *drawGraph*(*int* rows) that prints a random number of * for every row it draws. A user must be allowed to choose how many rows they want to be drawn by passing an integer to the function argument. The total number of * printed on every row must not exceed 100. Below is a sample output:

Hint: You need to use nested loops to solve this problem.

```
How many rows you want to add to your graph? 11

***
******
**********
************
****
******************
**********
*****
***********
**************
**************
```

PROJECTS

Project 1: Write a program that simulates a simple calculator.

- The calculator should have the following functions: add, sub, div, mul, squareRoot, power, repeat, and menu.
- For the first four functions, users must be able to enter as many numbers as they want by calling a Boolean function

called repeat that takes no parameters. The repeat function must recognize the responses, y, Y, yes, and Yes. If a user enters anything else, it will be considered as no.

- Use predefined functions to implement the squareRoot and power functions.
- For the div function, add code to prevent the error of dividing by zero.
- Menu is a void function that prints a list of options users can choose from to perform the desired calculation.
- All functions must have adequate preconditions and postconditions comments.

Project 2: The metric system and the American system (also known as the Imperial system) are two different systems of measurement used in different parts of the world.

- Write a program that converts values from one system to another.
- Each of the provided conversions must be implemented in a separate function.
- The program must have a Boolean repeat function that allows users to perform conversions as often as they want.
- The program must have a void menu function that prints menu options for users to choose from.
- All functions must have adequate preconditions and postconditions comments.

Length:

- 1 meter (m) = 3.281 feet (ft)
- 1 kilometer (km) = 0.6214 miles (mi)
- 1 centimeter (cm) = 0.3937 inches (in)
- 1 millimeter (mm) = 0.03937 inches (in)

Mass/Weight:

- 1 kilogram (kg) = 2.205 pounds (lb)
- 1 gram (g) = 0.03527 ounces (oz)

Volume:

- 1 liter (L) = 0.2642 gallons (gal)
- 1 liter (L) = 33.814 fluid ounces (fl oz)
- 1 cubic meter (m^3) = 35.3147 cubic feet (ft^3)

Temperature:

- To convert Celsius (°C) to Fahrenheit (°F): Multiply by 9/5 and add 32.
- To convert Fahrenheit (°F) to Celsius (°C): Subtract 32 and multiply by 5/9.

Project 3 Rock, Paper, Scissors: Read Project 2 from Chapter 3 again and redo it by allowing a player to play against the computer. That is, you have to add code that randomly selects one choice from the three available choices (rock, paper, scissors). Redesign the project by writing a couple of suitable functions rather than adding all the code in the main function as you did in Chapter 3.

CHAPTER 5

More on Functions

HUSSAM GHUNAIM

Learning Objectives

At the end of reading this chapter, you should be able to:

- Write functions that pass arguments by value and by reference.
- Correctly identify and utilize local and global variables.
- Write functions that call other functions.
- Overload functions.
- Write functions that use default arguments.
- Correctly use the scope resolution operator `::` to access global variables.

CALL BY VALUE VS. CALL BY REFERENCE

In the functions that you have written so far, you have used a

technique called ***passing arguments by value***. Which simply means, the arguments are merely copies of the original values. The following example explains this principle further.

Example 01

Enter and run the given code. Then try to explain the outputs.

In the ***main*** function, the variable **x** is declared and initialized to the value of 5 and printed out on the screen. When the function ***increment*** is called, the argument **x** is passed and incremented by 1 inside the function, and printed out the new value of the **x** variable which now becomes 6. When the function ***increment*** terminates and the execution continues back to line 13, the variable **x** is printed again, showing the old value of **x, NOT** the incremented one. This result proves that the variable **x** incremented inside the function ***increment*** is different from the variable **x** declared in the ***main*** function. To explain this, we can say that when ***increment (x)*** is called, a copy of the original variable is made and passed as an argument to the function ***increment***.

```
1   #include <iostream>
2   using namespace std;
3
4   void increment(int x) {
5       x++;
6       cout << "Inside the increment function, x = " << x << endl;
7   }
8
9   int main() {
10      int x = 5;
11      cout << "Before the increment function is called, x = " << x << endl;
12      increment(x);
13      cout << "After the increment function is called, x = " << x << endl;
14      return 0;
15  }
```

If this behavior does not satisfy our needs, we can use another technique called passing ***arguments by reference***. As the name suggests, we are not passing the actual variable value; instead, we are passing the reference to it, in other words, “its address”. In this case, whatever changes happen inside the ***increment*** function

will be applied to the actual variable and can be seen in both the *increment* and *main* functions. The symbol ***ampersand &*** is used to pass arguments by reference. The code in the previous example will be entirely the same with only one little difference in the *increment* function header as follows, `void increment(int &x)` . Note that the symbol ***ampersand &*** can be placed anywhere between the argument name and its type.

Now, change the code in example 01 to pass the *increment* function argument by reference and examine the outcomes. When the code runs, you will notice that the incremented value of x is printed from both functions, *increment* and *main*. Additionally, you can pass some arguments to the same function by value and the other arguments by reference.

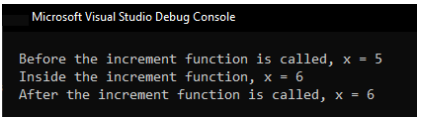


Figure 01: Passing arguments by reference.

Exercise01

The given code swaps two numbers. Run the code and then answer the questions. Did the code swap the numbers as expected? Explain why? Now, fix the problem. The solution is provided at the end of the chapter.

```cpp
#include <iostream>
using namespace std;

void swap(int x, int y) {
    int temp = x;
    x = y;
    y = temp;
}

int main() {
    int a = 5, b = 10;
    cout << "Before swapping: a = " << a << ", b = " << b << endl;
    swap(a, b);
    cout << "After swapping: a = " << a << ", b = " << b << endl;
    return 0;
}
```

Exercise02

Write a void **sortAscending** function that accepts two integer arguments passed by reference. The function must check if the two passed numbers are sorted correctly; otherwise, the numbers must be swapped. The program should print these messages:

Before sorting: x = , y =

After sorting: x = , y =

SCOPE, LOCAL, AND GLOBAL VARIABLES

The **scope** can simply be explained as the variables' visibility. In other words, when a variable is declared, we need to know where in the code we can use that variable. In example 01, the variable **x** was declared in the *main* function on line 10. However, when the variable **x** was passed to the *increment* function as an argument, the argument **x** becomes a new variable that is visible only within the *increment* function boundaries. Similarly, in the *main*, the

variable **x** is visible only within the *main* function. Here we can realize the power of scoping as a way to create variables with the same names and yet without causing any conflicts.

Scope boundaries can generally be determined by the block in which they were created. In other words, block scope can be defined by using { } symbols. Any variable declared in a block is called a **local** variable in that particular block. If you try to call that variable from another block, an error message will pop up! In some cases, we might need a variable to be accessed from different locations in the code. To do that, the variable needs to be declared as a global variable.

Example02

In the given code, **x** is declared outside of all code blocks. Hence, it is referred to as a **global** variable that can be accessed from anywhere. Run the code and examine the outputs to check your understanding.

```
#include <iostream>
using namespace std;

int x = 10; // global variable

void func() {
    cout << "The value of x from inside func: " << x << endl;
}

int main() {
    func();
    cout << "The value of x from inside main: " << x << endl;
    return 0;
}
```

Generally, writing global variables is not recommended as they can make the code harder to understand and debug. To elaborate, the following example uses both global variables and global constants. We can declare a constant by using the keyword *const* in front of

the constant type. Once constants are declared and initialized, any attempt to change their values will cause an error!

Example03

In the given code, one global variable and one global constant are declared and initialized. The first two lines in the *main* function print their values. When *func* is called, the first line would return an error if it is executed because it attempts to change the constant value. However, changing the global variable value is possible. Now it is clear what problem this change can cause. Imagine that you created a function and called the global variable. At this point, there is no guarantee of what value the global variable would have because any other function in the code can change its value. In developing large systems, this can be confusing to programmers and generally causes errors.

```
#include <iostream>
using namespace std;

const int globalConstant = 10;
int globalVariable = 20;

void func() {
    //globalConstant = 30; //this statement generates an error message, why?!
    globalVariable = 40;
    cout << "From inside func: globalConstant = " << globalConstant;
    cout << ", and globalVariable = " << globalVariable << endl;
}

int main() {
    cout << "Before calling func: globalConstant = " << globalConstant;
    cout << ", and globalVariable = " << globalVariable << endl;
    func();
    cout << "After calling func: globalConstant = " << globalConstant;
    cout << ", and globalVariable = " << globalVariable << endl;
    return 0;
}
```

Exercise03

Write a program that declares one global variable and one local variable for both the *main* function and the *printVar* function. All variables must have the same name but different values. The program should print the values of all variables.

NOTE: When a local variable has the same name as a global variable, calling that particular variable will by default access the local variable value. To enforce accessing the global variable value instead, we can use the scope operator **::** to do that.

::variableName; // This code will access the global variable named *variableName*.

Exercise04

As scopes can be defined using the block braces **{ }**, nesting these blocks will result in nesting scopes. Write a program that demonstrates the concept of nested scopes. That is, define two variables with the same name but different values in two nested blocks. Print and compare the variables' values.

FUNCTION CALLS ANOTHER FUNCTION

You have already practiced calling functions from within other functions. When you first introduced the predefined functions, these functions were called from within the *main* function. This technique becomes powerful as the problem we are trying to solve becomes larger and larger, and the solution becomes more complex. Let's here practice using this concept a little more.

Example 04

Run the given code that calculates the average of a set of numbers entered by a user. The *average* function calculates the average by dividing two of its local variables, **total** and **counter**. The **counter** variable is needed to count how many numbers are entered by a user. To keep the *average* function as simple as possible, the code responsible for getting numbers from the user is moved to another function called *input*. In this case, the *average* function calls the *input* function to do its job. The *input* function has a *while* loop that keeps asking the user if they still wish to enter more numbers until they answer 'n'. Note that the *input* function is *void* and does not return any variables. Instead, the *input* function passes its arguments by reference. Although **total** and **counter** are local variables to the *average* function and hence, they can not be called by any other function besides *average*, the *input* function has access to them through their references, that is, their addresses. Lastly, the *average* function returns the result to its caller, the *main* function.

```cpp
#include <iostream>
using namespace std;

void input(double &total, int &counter) {
    double num;
    char more = 'y';

    cout << "Enter the numbers that you want to find their average." << endl;
    while (more == 'y') {
        cin >> num;
        total += num;
        counter++;

        cout << "Do you want to add more, (y/n)" << endl;
        cin >> more;
    } //end of while loop.
}

double average() {
    double total = 0.0;
    int counter = 0; //counter is used to find how many numbers are entered.

    input(total, counter); //input() function job is to gather users input.

    double average = total / counter;
    return average;
}

int main()
{
    cout << "This program lets you enter as many numbers ";
    cout << "as you wish and finds their average for you!" << endl;
    cout << "average = " << average() << endl;

    return 0;
}
```

In the above example, instead of keep asking the user whether they want to enter more numbers, we can use another trick that is useful only in some specific cases. In our case, we want to calculate the average of a set of numbers and do not expect users will ever need to input negative numbers, although we understand this is not always the case. Therefore, if we agree that it is safe to make this assumption, we can rewrite the ***input*** function to keep looping until a negative number is entered. Below is a new version of the ***input*** function that you can use in example 04.

```
// A new version!
void input(double& total, int& counter) {
    cout << "Enter positive numbers to find their average. Enter -1 to stop." << endl;

    double num;
    cin >> num;
    while (num >= 0) {
        total += num;
        counter++;
        cin >> num;
    } //end of while loop.
}
```

Figure 02: A new version of the input function.

Exercises05

Write a program that calculates the area of a rectangle. The program should have three functions: *getLength*, *getWidth*, and *calculateArea*. The *getLength* and *getWidth* functions should ask the user for the length and width of the rectangle, respectively, and return their values. The *calculateArea* function should get the length and width values by calling their respective functions, calculating the area, and printing the result. The *main* function should call only the *calculateArea* function.

Exercise06

Write a program that calculates the power of a number. The program should have two functions: *getNumber*(**string prompt**), and *calculatePower*(**double base**, **int exponent**). The *getNumer* function takes a string argument that labels the number entered by the user, whether it is the base or the exponent, and returns the entered number as a **double**. The *calculatePower* function should take the base and exponent as parameters, calculate the power based on the given algorithm below, and print the result. The main function should call these functions to perform the task.

Algorithm: To find the power of a number, we multiply a number by itself a certain number of times. This is often written as $base^{exponent}$.

- The **base** is the number that is being multiplied.
- The **exponent** tells us how many times to multiply the base by itself.

For example, if we want to calculate 2^3, we multiply 2 by itself 3 times: 2 X 2 X 2, which equals 8.

FUNCTIONS OVERLOADING

Sometimes it is useful to use the same function name several times in the same program. We know this is not possible and returns errors if we try to do it. However, as far as the compiler is concerned, all it wants is a way to distinguish between functions' calls. To do that, we can use a different number of parameters or types to make the distinction. This technique is called ***overloading***. All the following functions can legally be used in the same program.

```
func()
func(type1 parameter1)
func(type1 parameter1, type2 parameter2)
```

Note that we can not overload functions based on their return type. The reason is that when a function is called, its arguments will be specified with the call but not the return type. Therefore, there is no way for the compiler to tell what the return type is until run time.

Example05

Run the given code and notice that the function *display* is overloaded three times. The first and second functions are overloaded by providing different argument types. The third function is overloaded by providing a different number of arguments.

```
#include<iostream>
using namespace std;

void display(int num) {
    cout << "Displaying an integer: " << num << endl;
}

void display(double num) {
    cout << "Displaying a double: " << num << endl;
}

void display(int num1, double num2) {
    //this line should be entered in a single line!
    cout << "Displaying an integer and a double: " << num1 << ", " <<
      num2 << endl;
}

int main() {
    display(5);
    display(6.7);
    display(5, 6.7);
    return 0;
}
```

Example06

This example shows functions that can be overloaded by changing the order of the arguments.

```cpp
#include <iostream>
#include <string>
using namespace std;

// Function to print details when name is first and age is second
void welcomeMessage(string name, int age) {
    cout << "Hello, " << name << "! You are " << age << " years old.
      \n";
}

// Function to print details when age is first and name is second
void welcomeMessage(int age, string name) {
    cout << "Hello, " << name << "! You turned " << age << " this
      year.\n";
}

int main() {
    // Calling the functions
    welcomeMessage("Alice", 25);
    welcomeMessage(30, "Bob");

    return 0;
}
```

Exercise07

Write two overloaded ***calculateArea*** functions. One should calculate the area of a rectangle, and the other should calculate the area of a circle.

Exercise08

Write two overloaded ***calculate*** functions. The first version of the ***calculate*** function should take two **integers** as parameters and return their sum. The second version of the ***calculate*** function should take two **integers** and a **char** as parameters. If the char is '*', the function should return the product

of the two integers. Otherwise, it should print "**Invalid operation. Defaulting to addition**" and return the sum of the two integers.

DEFAULT ARGUMENTS

In writing functions, sometimes it is useful to provide users with different ways to call a function without the need to overload it. This can reduce the size of the overall written code and increase flexibility.

Example07

Run the given code and explain the outputs before continuing reading. The *calculateVolume* function is written with three parameters, each of which has a default value assigned. Of course, the type of the default value must match the type of the parameter. In the *main* function, *calculateVolume* is called in four different ways without overloading. In the first call, no arguments were provided by the user; hence, the compiler is forced to use all available default values. In the second call, only one argument was provided; hence, the compiler will use only the last two available default values. By examining the other calls, it's important to note that the arguments provided by users will override the default values. This process happens in order. This means that there is no way for users to skip the first or second argument and provide values only for the last

argument.

```cpp
#include <iostream>
using namespace std;

double calculateVolume(double length = 1.0, double width = 1.0, double height = 1.0) {
    return length * width * height;
}

int main() {
    // using all default arguments
    cout << "Volume: " << calculateVolume() << "\n";
    // using only the last two default arguments
    cout << "Volume: " << calculateVolume(5.0) << "\n";
    // using only the last default argument
    cout << "Volume: " << calculateVolume(5.0, 3.0) << "\n";
    // no default argument is used
    cout << "Volume: " << calculateVolume(5.0, 3.0, 2.0) << "\n";
    return 0;
}
```

Example08

When you run the provided code, you will notice that two welcome messages were printed. One greets 'Guest' and the other one greets 'Alice'.

```cpp
#include <iostream>
using namespace std;

void printGreeting(string name = "Guest") {
    cout << "Hello, " << name << "! Welcome to our website.\n";
}

int main() {
    printGreeting();
    printGreeting("Alice");
    return 0;
}
```

Exercise09

Write *printDate* function that takes three **int** parameters: day, month, and year. Provide default values for these parameters. In the *main* function, call *printDate* function in four different ways, similar to example 7.

Exercise10

Write *calculatePower* function that takes two parameters: a **double** for the **base** and an **int** for the **exponent**. Provide a default value only for the exponent parameter. In the *main* function, call the *calculatePower* function twice. In the first call, provide only one argument, and in the second call, provide two arguments. **Explain the results.**

Question: What will happen if you try to call the *calculatePower* function without providing any arguments? Why?

Note: If a function definition has both parameters with default values and other parameters without default values, the parameters with default values must always be placed last.

USING THE SCOPE :: OPERATOR

Because the **std** namespace is really huge, using the entire namespace might result in conflicting names, especially in large programs. Alternatively, it is considered good coding practice to use the scope operator **::** to identify only the needed names.

Example 09

We will rewrite example 08 code without the statement "**using**

namespace std;". Notice how the scope operator is used to identify every name, like the **string** type and the output stream *cout*. However, user-defined names do not need to be identified like variables and functions' names.

```
#include <iostream>

void printGreeting(std::string name = "Guest") {
    std::cout << "Hello, " << name << "! Welcome to our website.\n";
}

int main() {
    printGreeting();
    printGreeting("Alice");
    return 0;
}
```

Example 10

Run the given code and notice the places where the scope operator is used.

```
#include <iostream>
#include <cmath>

int main() {
    double base, exponent;
    std::cout << "Enter the base: ";
    std::cin >> base;
    std::cout << "Enter the exponent: ";
    std::cin >> exponent;

    //this line should be written in a single line!
    std::cout << "The value of the base " << base << " raised to the exponent "
       << exponent << " is " << std::pow(base, exponent) << std::endl;
    return 0;
}
```

Exercise11

Write a function named *check*. This function should take an integer

as an argument and return a string. If the integer is even, the function should return 'even', otherwise, it should return 'odd'. In the *main* function, prompt the user to enter an integer and call the *check* function with this integer passed as an argument. Finally, print out a message in the following format: The entered number [number] is an [even/odd] number. You **MUST** use the scope resolution operator (`::`) to access standard library functions and objects. Do not use the *using namespace std;* statement.

Exercise12

Write a program that generates and prints 10 random numbers between 1 and 100. The program should use the *srand*, *time*, and *rand* predefined functions. The program should not use the *using namespace std;* statement, but instead use the scope resolution operator (`::`). Each random number should be printed on a new line with a message indicating its position in the sequence.

BEST PROGRAMMING PRACTICES

In real-life C++ development, large systems often involve writing many functions, each handling a specific task. The *main* function typically serves only as the program's entry point, starting the system, so it's often called a ***driver function***. During testing, especially when the full system isn't complete, it's helpful to use ***stub*** functions. A stub is a minimal placeholder function (often empty or returning a fixed value) that stands in for a function not yet implemented. By using stubs, you can compile and test parts of the system even before all the functions are fully written. Once development progresses, the stubs are replaced with real implementations.

In the development life cycle, developers always wonder how much testing is enough. The rule of thumb here is to test for all valid and invalid inputs for a function, including what is known as *boundary values*.

WRAP UP!

In C++, functions are a fundamental building block of the language, and understanding their complexities is crucial for effective programming. One such complexity is the difference between Call by Value and Call by Reference. Call by Value involves passing a copy of the variable to the function, meaning any modifications made within the function do not affect the original variable. Conversely, Call by Reference passes the actual variable to the function, so any changes made within the function directly affect the original variable.

Another important concept is the scope of variables. Variables in C++ can be either local or global. Local variables are declared within a function and can only be accessed within that function. In contrast, global variables are declared outside all functions and can be accessed by any function in the program. Understanding the scope is essential for managing data and avoiding conflicts or unexpected behavior in your code.

Functions in C++ can also call other functions. This capability is particularly useful in larger programs where tasks can be broken down into smaller, more manageable functions. A function that is called by another function is often referred to as a helper function.

Function overloading is another powerful feature in C++. It allows you to define multiple functions with the same name but different parameters. The compiler determines which function to use based on the number, types, and order of the arguments. This feature enables varied functionality under a single function name, making your code more readable and organized. Default arguments in

functions provide flexibility by allowing optional parameters. If an argument is not provided when the function is called, the default value is used. This feature can make your functions more versatile and easier to use.

Lastly, the scope resolution operator (`::`) is used to access global variables or functions. This operator is especially useful when a local variable has the same name as a global variable. By using the scope resolution operator, you can ensure the correct variable is referenced in your code. This operator is also used to avoid using an entire namespace and include only the needed names.

In summary, these advanced concepts deepen the understanding of functions in C++, which is essential for writing efficient and effective code.

ANSWERS KEY

Exercise 01:

No, the code did not swap the numbers as expected.

The reason is that the function *swap* passed the arguments **a** and **b** by value. This means **a** and **b** variables swapped inside the *swap* function are different copies from the original **a** and **b** arguments declared inside the *main* function.

To fix this error, we only need to change the `swap` function header to pass arguments by reference. The rest of the code will remain the same.

```
void swap(int &x, int &y)
```

Exercise 02:

```cpp
#include <iostream>
using namespace std;

void sort(int& x, int& y) {
    if (x > y) {
        int temp;
        temp = x;
        x = y;
        y = temp;
    }
}

int main() {
    int x = 10, y = 5;
    cout << "Before sorting: x = " << x << ", y = " << y << endl;
    sort(x, y);
    cout << "After sorting: x = " << x << ", y = " << y << endl;
    return 0;
}
```

Exercise 03:

```cpp
#include<iostream>
using namespace std;

int var = 10; // Global variable

void printVar() {
    int var = 20; // Local variable to printVar function
    cout << "Local variable to printVar function: " << var << endl;
}

int main() {
    int var = 30; // Local variable to main function
    cout << "Local variable to main function: " << var << endl;
    printVar();
    // Using scope resolution operator :: to access global variable
    cout << "Global variable: " << ::var << endl;

    return 0;
}
```

Exercise 04:

```cpp
#include<iostream>
using namespace std;

int main() {
    int x = 10; // Outer 'x'
    cout << "Outer x: " << x << endl;
    {
        int x = 20; // Inner 'x'
        cout << "Inner x: " << x << endl;
    }
    cout << "Outer x after inner scope: " << x << endl;
    return 0;
}
```

Exercise 05:

```cpp
#include<iostream>
using namespace std;

double getLength() {
    double length;
    cout << "Enter the length of the rectangle: ";
    cin >> length;
    return length;
}

double getWidth() {
    double width;
    cout << "Enter the width of the rectangle: ";
    cin >> width;
    return width;
}

void calculateArea() {
    double length = getLength();
    double width = getWidth();
    double area = length * width;
    cout << "The area of the rectangle is: " << area << endl;
}

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

Exercise 06:

```cpp
#include<iostream>
using namespace std;

double getNumber(string prompt) {
    double num;
    cout << prompt;
    cin >> num;
    return num;
}

void calculatePower(double base, int exponent) {
    double result = 1;
    for (int i = 0; i < exponent; i++) {
        result *= base;
    }
    // this line should be entered as a single line
    cout << base << " raised to the power of " << exponent << " is: "
      << result << endl;
}

int main() {
    double base = getNumber("Enter the base: ");
    double exponent = getNumber("Enter the exponent: ");
    calculatePower(base, exponent);
    return 0;
}
```

Exercise 07:

```cpp
#include <iostream>
using namespace std;

const double PI = 3.1416; // the mathmatical constant pi
// Function to calculate the area of a rectangle
double calculateArea(double length, double width) {
    return length * width;
}

// Function to calculate the area of a circle
double calculateArea(double radius) {
    return PI * radius * radius;
}

int main() {
    cout << "Area of rectangle: " << calculateArea(5.0, 3.0) << "\n";
    cout << "Area of circle: " << calculateArea(4.0) << "\n";

    return 0;
}
```

Exercise 08:

```cpp
#include <iostream>
using namespace std;

// Function to calculate the sum of two integers
int calculate(int a, int b) {
    return a + b;
}

// Function to calculate the product of two integers
int calculate(int a, int b, char operation) {
    if (operation == '*') {
        return a * b;
    }
    else {
        cout << "Invalid operation. Defaulting to addition.\n";
        return a + b;
    }
}

int main() {
    cout << "Sum: " << calculate(5, 3) << "\n";
    cout << "Product: " << calculate(5, 3, '*') << "\n";

    return 0;
}
```

Exercise 09:

```cpp
#include <iostream>
using namespace std;

void printDate(int day = 1, int month = 1, int year = 2000) {
    cout << "Date: " << day << "/" << month << "/" << year << "\n";
}

int main() {
    printDate();  // Output: Date: 1/1/2000
    printDate(5);  // Output: Date: 5/1/2000
    printDate(5, 3);  // Output: Date: 5/3/2000
    printDate(5, 3, 2024);  // Output: Date: 5/3/2024
    return 0;
}
```

Exercise 10:

The first call will raise the base of 5 to the default power of 2,

resulting in 25.
The second call will raise the base of 5 to the power of 3, resulting in 125.
If you call the *calculatePower* function without providing any arguments, an error will be thrown specifying that at least one argument must be provided.

```
#include <cmath>
#include <iostream>
using namespace std;

double calculatePower(double base, int exponent = 2) {
    return pow(base, exponent);
}

int main() {
    cout << "Power: " << calculatePower(5.0) << "\n";
    cout << "Power: " << calculatePower(5.0, 3) << "\n";
    return 0;
}
```

Exercise 11:

```
#include <iostream>
#include <string>

std::string check(int num) {
    if (num % 2 == 0)
        return "even";
    else
        return "odd";
}

int main() {
    int num;
    std::cout << "Enter an integer: ";
    std::cin >> num;
    std::string result = check(num);
    std::cout << "The entered number " << num << " is an " << result << " number." << std::endl;
    return 0;
}
```

Exercise 12:

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>

int main() {
    // Use the time function to get a "seed" value for srand
    std::srand(std::time(0));

    // Generate and print 10 random numbers between 1 and 100
    for (int i = 0; i < 10; i++) {
        int randomNumber = std::rand() % 100 + 1;
        std::cout << "Random number " << i + 1 << ": " << randomNumber << std::endl;
    }

    return 0;
}
```

END OF CHAPTER EXERCISES

1. Write a **void** *min_max* function that takes two integer parameters **x** and **y** passed by value and two integer parameters **min** and **max** passed by reference. The function should compare **x** and **y** and assign the smaller value to **min** and the larger value to **max**. The function should not return a value, as the **min** and **max** variables should be updated directly via the passed references. The *main* function will ask the user to enter two integers and test whether **x** and **y** are equal. If **x** and **y** are equal, the *main* should print this message and quit the program: "Both entered integers are equal!"; otherwise, call the *min_max* function, pass the two entered integers, and print out the results.
2. Write a **void** *sum_ave* function that takes two **double** parameters, **sum** and **average,** passed by reference. The *sum_ave* function should ask users to enter as many positive numbers (whole or decimal) as they want and calculate their sum and average. The *sum_ave* function should not return any values. The *main* function will call *sum_ave* and print out the results.

3. True or False, both call by value and call by reference allow changes to the arguments in the calling function.
4. True or False, call by reference is more efficient than call by value for large data structures.
5. Which of the following correctly describes call-by-value?
   A. The actual parameter is copied into the formal parameter.
   B. The formal parameter is a reference to the actual parameter.
   C. Both A and B
   D. Neither A nor B
6. When should you use call-by-value?
   A. When you want to modify the passed argument.
   B. When you don't want to modify the passed argument.
   C. When you want to pass a large data structure.
   D. All of the above
7. True or False, a local variable can be accessed from anywhere in the program.
8. True or False, a global variable and a local variable can have the same name.
9. True or False, changing the value of a global variable will affect all functions that use this variable.
10. Which of the following correctly describes the global variable?
    A. A variable that is declared within a function.
    B. A variable that is declared outside all functions.
    C. A variable that is declared within a block of code.
    D. None of the above.
11. What is the default value of a global variable if it is not initialized?
    A. 0

B. *NULL*
C. Depends on the type of the variable
D. None of the above

12. What happens if a local variable has the same name as a global variable?
A. Compile error
B. The global variable is used.
C. The local variable is used.
D. None of the above

13. Declare a global variable and assign it a value. The *main* function must do the following:
    1. Print out the global variable value.
    2. Call the increment() function that increments and prints the incremented value of the global value.
    3. Call the decrement() function that decrements and prints the decremented value of the global value.

14. When you run the attached code, a sequence of values will be printed out to the console for the variable `i`. Uncomment line 8 and rerun the code. What will happen? Explain why? Fix the generated error.

```
1   #include<iostream>
2   using namespace std;
3
4   int main() {
5       for (int i = 0; i < 5; i++) {
6           cout << "i = " << i << endl;
7       }
8       //cout << "i after loop = " << i << endl;
9       return 0;
10  }
```

15. True or False, a function in C++ can call itself.

16. True or False, a function can not call multiple other

functions.

17. What happens when a function is called?
    A. The program jumps to the first line of the main function.
    B. The program jumps to the first line of the called function.
    C. The program jumps to the last line of the called function.
    D. None of the above

18. Write a program where a function ***printMessage()*** calls another function ***getMessage()*** to get a message and then prints it.

19. Which of the following is not a correct way to overload a function?
    A. By changing the return type of the function.
    B. By changing the number of parameters of the function.
    C. By changing the types of parameters of the function.
    D. All of the above are correct ways to overload a function.

20. Which of the following can be used to distinguish overloaded functions?
    A. The return type of the functions.
    B. The name of the functions.
    C. The number and types of parameters of the functions.
    D. None of the above.

21. Which of the following correctly describes function overloading?
    A. Functions that have the same name and the same number and types of parameters.
    B. Functions that have the same name but different numbers or types of parameters.
    C. Functions that have different names but the same

number and types of parameters.
D. None of the above.

22. Write two functions with the same name ***findMax*** that find the maximum of two integers and two doubles, respectively. Write a ***main*** function to test the overloaded ***findMax*** functions.
23. Overload the ***fullName*** function to print full names in two different formats, **firstName , lastName** and **firstName , middleInitial . lastName**. The ***main*** function should ask the user to enter either their first name and last name, or their first name, middle initial, and last name, and then call the corresponding overloaded function.
24. True or False, functions can have both default and non-default arguments.
25. True or False, parameters with default values must always be placed first in functions' definitions.
26. True or False, the default value of a parameter can not be modified in the function call.
27. What is the output of the following code?

```
#include<iostream>
using namespace std;
void display(int x = 10, int y = 20) {
    cout << x << " " << y << endl;
}
int main() {
    display(30);
    return 0;
}
```

A) 10 20

B) 30 20
C) 30 10
D) 10 30

28. Write the *calculateArea* function that takes two integer parameters: ***length*** and ***width***, and returns an integer. The width parameter has a default value of -1. When *calculateArea* function is called, it must check for the width value; if it equals -1, this means that the user did not provide a width value, and it should assume the area required to calculate is the area of a square. Otherwise, the area returned should be ***width*** **X*length***. In the *main* function, call *calculateArea* function twice, one with only one argument and another call with two arguments.

29. The scope resolution operator `::` can be used to access global variables when a local variable has the same name.

30. What is the output of the following code?

```
#include<iostream>
using namespace std;
int x = 10;
int main() {
    int x = 20;
    cout << ::x << endl;
    return 0;
}
```

A) 10
B) 20
C) Error
D) None of the above

PROJECTS

Project 01: Number Guessing Game:

The objective of this project is to create a simple number-guessing game. The game will generate a random number, and the player will have a limited number of guesses to guess the correct number.

Your program should include the following features:

- The scope resolution operator :: must be used throughout the entire project code.
- Global Constant: Define a global constant for the maximum number of guesses allowed.
- Global Variables: Define global variables for the number to be guessed and the number of guesses made.
- In the main function, start the game by calling the startGame function and enter a game loop where the player can make guesses until they run out of attempts or guess correctly. If the player guesses the number correctly, it prints "You win!" and the game statistics. If the player runs out of guesses, it prints "You lose!" and the game statistics.
- startGame(): This function should initialize the game's global variables. generateRandomNumber function should be called.
- generateRandomNumber(int min, int max): This function should generate a random number within a given range. The range is defined by the parameters min and max. The generated number must be different in every run of the game.
- makeGuess(int guess, int& numGuesses): This function should take a guess and a reference to the number of guesses, increment the number of guesses, and return

true if the guess is correct.

- printStatistics(int numGuesses): This function should print the number of guesses made and the number to be guessed. Format the output appropriately.
- Ensure that the program is well-documented, providing comments for each function and explaining the purpose of the code.

Projec 02: Membership Discount System:

The goal of this project is to calculate and apply a discount to a price based on a customer's membership category. The program should support three categories: "gold", "VIP", and "non-member". The discount rates for "gold" and "VIP" members should be 20% and 30%, respectively, while "non-members" should receive no discount. The project should include at the minimum the following functions:

- calculateDiscountedPrice(double& price, double discountRate): This function calculates the discounted price of a product. It takes two parameters: price (the original price of the product) and discountRate (the discount rate to be applied). The discounted price is stored in the price variable, which is passed by reference, so the change will affect the original variable in the calling function.
- getDiscountRate(string category): This function determines the discount rate based on the membership category. It takes one parameter: category (the membership category of the customer). If the category is "gold", the function returns a discount rate of 0.2 (or 20%). If the category is "VIP", it returns a discount rate of 0.3 (or 30%). For any other category, it returns the default discount rate of 0.0 (or 0%), which must be declared as a

global constant.

- applyDiscount(double& price, string category): This is an overloaded function that applies a discount to a price based on a specified membership category. It takes two parameters: price (the original price of the product) and category (the membership category of the customer). The function first gets the appropriate discount rate by calling the getDiscountRate function. It then calculates the discounted price using the calculateDiscountedPrice function. Finally, it prints out the discounted price.
- applyDiscount(double& price): This is another overloaded version of the applyDiscount function. It applies a discount to a price without a specified membership category. It takes one parameter: price (the original price of the product). The function sets the category to "without" and then calls the other version of the applyDiscount function to calculate and print the discounted price.
- main(): It first asks the user to enter the original price of a product. Then it asks the user if they have a membership. If the user answers "yes", it asks for the membership category and applies a discount based on that category. If the user answers anything other than "yes", it applies a discount without a specified category. Finally, it prints out the original price and ends the program.
- Ensure that the program is well-documented, providing comments for each function and explaining the purpose of the code.

Project 03: Grade Statistics Calculator:

The objective of this project is to create a program that calculates and displays statistics for a set of students' grades.

The project should include at the minimum the following:

    - Use the scope operator `::` instead of including the entire standard namespace ***std***.

- Declare all grades as global constants: grade1, grade2, grade3, grade4, and grade5. Initialize these constants to a set of grades of your choice.
- Create the calculateAverage function to calculate and return the average of the grades.
- Create the findMax function to find and return the maximum grade.
- Create the findMin function to find and return the minimum grade.
- Create the calculateRange function to calculate and return the range of the grades. The range is the difference between the max and min grades.
- In the main function, display each grade, along with a separator, and then print the calculated statistics as shown in the screenshot.
- Ensure that the program is well-documented, providing comments for each function and explaining the purpose of the code.

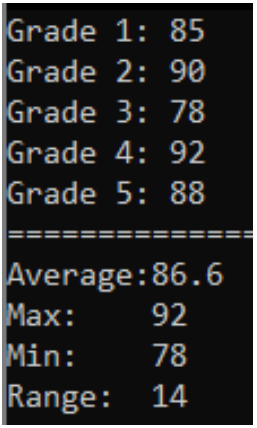


Figure 03: Project 3 output screenshot.

CHAPTER 6

Arrays

HUSSAM GHUNAIM

Learning Objectives

At the end of this chapter, you will be able to :

- Use Arrays to solve real-world problems
- Pass Arrays as Parameters for Functions
- Search and Sort Arrays
- Use Multi-dimensional Arrays to solve real-world problems

INTRODUCING ARRAYS

Figure 01 in Chapter 2 shows what happens in the computer's main memory RAM when you declare a variable of any primitive type. Primitive types are perfect whenever you need to store a single datum. However, what if you need to store multiple data of the

same type, for example, a couple of numbers, characters, or even strings and objects? In this case, declaring many variables to store every needed datum will be time-consuming and very difficult to work with.

The code below declares both a single-value variable and an array. The only difference is the use of brackets [ ] to indicate the number of elements the array will store, Figure 01.

```
  int var;
int var[5];
```

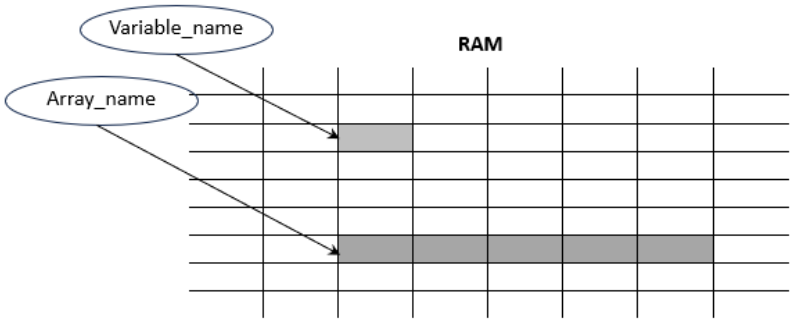


Figure 01: RAM representation of an array.

After declaring an array, the next step is initialization. For a single-valued variable, the process is as straightforward as you've experienced. However, initializing arrays offers several methods.

The simplest method is to assign values directly to each slot in the array. But this raises a question: how do we specify the exact slot for a particular value? The solution lies in using indices to number each slot, which are automatically generated by the system upon array declaration. As Figure 02 illustrates, indices start at 0, meaning the highest index in an array is the array's size minus one. We can then initialize the array slots as follows:

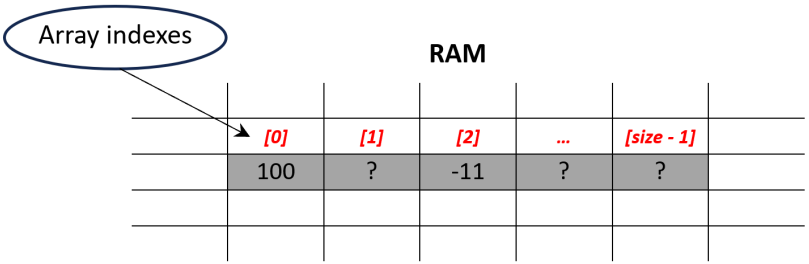


Figure 02: RAM representation of an array initialized and uninitialized memory slots.

```
var[0] = 100;
var[2] = -11;
// and so on...
```

As demonstrated, it's not necessary to fill all the slots in sequence; you can assign values as desired. However, it's crucial to remember that upon declaration, all array slots are initially uninitialized. Accessing an uninitialized slot can lead to undefined behavior. If you choose to populate your array non-sequentially, you must keep track of which slots are filled and which remain unassigned. To prevent issues, you may want to initialize the entire array to default values at declaration. While this approach can be cumbersome for large arrays, using loops, as shown in Example 01, is often more practical.

Example 01

Run the provided code and comment on the results. The first thing you might notice is the use of a constant for the array size at the declaration. This is generally considered good programming practice because it enhances the code's readability and modifiability. That is, if you decide to change the array size, you can do so in one location

rather than modifying the size value throughout the code. Secondly, you can assign any desired value to any desired index in the array using the syntax ***myArray[i] = anyValue***. However, ensure that the value you store is of the same type as the array, and the index does not exceed ***SIZE – 1***. Lastly, you can access any value stored at the index ***i*** with the code ***myArray[i]***, assuming there is a value stored at that location; otherwise, you will get surprising outputs instead. One more thing to pay attention to is the for loop condition ***i < SIZE***, which effectively ensures that the largest index value will be ***SIZE – 1***.

```
#include <iostream>
using namespace std;

int main() {
    const int SIZE = 10; // Size of the array
    int myArray[SIZE]; // Declaration of the array

    // Populating the array using a for loop
    for (int i = 0; i < SIZE; ++i)
        myArray[i] = i * 2; // For example, populating with even numbers

    // Printing the array to verify
    for (int i = 0; i < SIZE; ++i)
        cout << "Element at index " << i << ": " << myArray[i] << endl;

    return 0;
}
```

Example 02

Run the provided code and comment on the results. In this example, we declare an array with a base type of **char**. The first loop fills the array with the 26 letters of the alphabet, starting from 'a' to 'z'. In Chapter 2, we discussed type compatibility, where **int** and **char** types can be used interchangeably. Remember that characters are encoded with integer values following the ASCII standard encoding. Once again, pay attention to the loop condition ***i < 26***, which will terminate the loop when the value of ***i*** becomes 26, to avoid an out-of-boundary error.

The second loop does the opposite by printing out the contents of the array. The last loop prints the contents of the array in reverse order. Study this loop header carefully to understand how it works, noticing that the index values go from 25 down to 0.

```
#include <iostream>
using namespace std;

int main() {
    // Declare a char array with 26 elements for the alphabet
    char alphabet[26];

    // Fill the array with letters 'a' to 'z'
    for (int i = 0; i < 26; ++i)
        alphabet[i] = 97 + i; // 97 is the letter 'a' ASCII code

    // Print the characters in normal order
    for (int i = 0; i < 26; ++i)
        cout << alphabet[i] << " ";

    // Add a new line for clarity
    cout << endl;

    // Print the characters in reverse order
    for (int i = 25; i >= 0; --i)
        cout << alphabet[i] << " ";

    return 0;
}
```

Another method of declaring and initializing an array is as follows:

```
int anyName[5] = {4, 0, -11, 34, 7};
```

This code assigns each value to its corresponding place in the array, starting from index 0 and continuing up to index 4 in this example. Because the compiler can determine the size of the required array from the number of values provided for initialization, it is possible to omit the array size altogether:

```
int anyName[] = {4, 0, -11, 34, 7};
```

Example 03

Run the given code and comment on the results. In this code, we use a variation of the for-loop called the for-each loop. This loop iterates through the array from start to end, conveniently eliminating concerns about index boundaries. The syntax for this loop is:

for (variable to hold each item in the array : array)

In this example, the loop variable is declared as a *string* to match the array's base type. Note that at the beginning of the loop, the loop variable holds the first item in the array. With each iteration, the loop variable holds the next item, overriding the previously stored item, and so on until the last item in the array.

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    // Declare and initialize an array of strings
    string strArray[] = { "Hello", "World,", "C++", "Programming", "Array!" };

    // Output the contents of the array using for-each loop
    for (string str : strArray)
        cout << str << " ";

    // add a new line for clarity.
    cout << endl;
    return 0;
}
```

Exercise 01

Write a program that reads the user's input of five numbers. The entered numbers can be positive or negative and can be either integers or decimals. The program must calculate and print out the average, sum, maximum (max), and minimum (min) of the entered numbers with appropriate formatting. To find the max, initialize the

max variable to the smallest possible value on your system, typically -1e308, which is scientific notation for negative 10 raised to the power of 308. Then, compare each value stored in the array with this max variable, updating the max value whenever a new maximum is found. Use the same approach to find the min by initializing the min variable to the largest possible value, 1e308.

Exercise 02

Write a program that prompts the user to enter five words. The program should then print out the longest and shortest words entered.

Hint: In Chapter 2, you learned how to use some of the member functions of the string class, such as ***length()***. You may want to reread that chapter to refresh your memory. For instance, to find the length of a string stored in a variable named ***word***, you can use the expression ***word.length()***. Similarly, when working with arrays, you can determine the length of a string stored at a specific index in the array with ***arrayName[index].length()***.

PASSING ARRAYS AS FUNCTION PARAMETERS

Returning to the basic principle of robust programming, which involves dividing a program into several subproblems, each solvable by a single function, it's essential that functions can accept arrays as parameters. The upcoming example will demonstrate how an array's stored values can be accessed individually. Subsequently, another example will illustrate how an entire array can be passed to a function.

Example 04

Run the provided code and comment on the results. Lines 14 and 16 demonstrate how individual values stored in an array can be passed as parameters to a function. From the compiler's perspective, it is looking for an integer parameter to match the *squared* function definition. So, when the expressions ***a[0]*** and ***a[3]*** are evaluated, they will be replaced by the values stored at those respective locations in the array. Similarly, in line 22, the compiler will attempt to resolve the expression ***a[i]*** to an integer value. If it fails, an error will be returned. Otherwise, the code can continue executing without any issues, provided that the expression ***a[i]*** always evaluates to a valid index within the defined array boundaries.

```
1   #include <iostream>
2   #include <cmath>
3   using namespace std;
4
5   // Function declaration
6   void squared(int num);
7
8   int main() {
9       // Initialize array
10      int a[5] = { 1, 2, 3, 4, 5 };
11
12      // Pass individual elements of array to function
13      cout << "The value stored at a[0] squared is: ";
14      squared(a[0]);
15      cout << "The value stored at a[3] squared is: ";
16      squared(a[3]);
17
18      // Or pass elements in a loop
19      cout << "\n\nSquaring the entire array stored values:- " << endl;
20      for (int i = 0; i < 5; i++) {
21          cout << "The square of the value stored at a[" << i << "] squared is: ";
22          squared(a[i]);
23      }
24
25      return 0;
26  }
27
28  // Function definition
29  void squared(int num) {
30      cout << pow(num,2) << endl;
31  }
```

Example 5 demonstrates how to pass an entire array as a parameter. In Chapter 5, we discussed two methods of passing

parameters: by value and by reference. However, passing arrays as parameters does not fall into either of these categories. For now, it's acceptable to consider it similar to passing parameters by reference, but it's important to note that it's not exactly the same.

Example 05

Run the provided code and comment on the results. It is recommended to test the code several times to enhance your understanding and trace the results back to the code. When passing arrays to functions as parameters, it's important to distinguish between the two different syntaxes.

Firstly, when declaring or defining a function, you must specify that the parameter is an array variable, as shown in line 4. In this example, we have only defined the ***userInput*** function at the top of the source file. However, if you choose to declare the function at the top of the source file and then define it at the bottom, you must ensure that the same syntax is followed in both the declaration and definition of the function.

Secondly, when calling the function in line 23, you should pass the array name without the square brackets **[ ]**. This is because the compiler already recognizes this variable as an array variable, as declared in line 21. Therefore, calling the function and passing the array name along with square brackets **[ ]** will cause an error, as it would appear as if you are declaring the array again.

Lastly, it's necessary to pass the array size along with the array. This ensures that when accessing the array items, you do not go beyond the legal boundaries of the array.

```cpp
#include <iostream>
using namespace std;

void userInput(int classScores[], int size) {
    cout << "Enter the classScores for " << size << " students:\n";
    for (int i = 0; i < size; i++) {
        cin >> classScores[i];
    }
}

char grade(int studentScore) {
    if (studentScore >= 90) return 'A';
    else if (studentScore >= 80) return 'B';
    else if (studentScore >= 70) return 'C';
    else if (studentScore >= 60) return 'D';
    else return 'F';
}

int main() {
    const int num_students = 5;
    int classScores[num_students];

    userInput(classScores, num_students);

    cout << "\nStudents Grades:\n";
    for (int i = 0; i < num_students; i++) {
        cout << "Student " << i + 1 << ": " << grade(classScores[i]) << "\n";
    }

    return 0;
}
```

Example 06

As always, it is important to run the code several times and trace the outputs back to the code for a better understanding of how the code works. This is a simple guessing game that utilizes two functions. The first function, ***initializeOcean***, is used to initialize the array by placing a '1' at a random location in the array to denote the ship (or anything else you'd like to symbolize). The rest of the array will be initialized to '0's. The second function, ***playGame***, asks users to enter their guesses from 0 to 9 and checks if they guessed correctly. Both functions take an array as a parameter. It's important to distinguish between how the functions are defined and called in the code. ***break;*** statement is used in the second ***if*** statement in the ***while*** loop to terminate the loop

once the user makes a correct guess. This makes sense as you do not want to keep asking users to enter more guesses if they have already guessed right. As a practice, delete the *break;* statement to see the effect it causes. Do not forget to put it back when you are done testing.

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

// Function to initialize the ocean with '0's and place a '1' at a random index
void initializeOcean(int ocean[], int size) {
    srand(time(0)); // seed the random number generator
    int shipLocation = rand() % size; // generate a random index for the ship

    for (int i = 0; i < size; i++) {
        if (i == shipLocation) {
            ocean[i] = 1; // place the ship
        }
        else {
            ocean[i] = 0; // empty water
        }
    }
}
// Function to let the player guess the ship location
void playGame(int ocean[], int size) {
    int guess;
    cout << "Guess the location of the ship (0-" << size - 1 << "):\n";

    while (true) {
        cin >> guess;

        // Check if the guess is within the ocean boundaries
        if (guess < 0 || guess >= size) {
            cout << "Enter a valid location!\n";
        }
        else if (ocean[guess] == 1) {
            cout << "Hit! You found the ship.\n";
            break;
        }
        else {
            cout << "Miss! Try again.\n";
        }
    }
}
int main() {
    const int oceanSize = 10;
    int ocean[oceanSize];

    initializeOcean(ocean, oceanSize);
    playGame(ocean, oceanSize);

    return 0;
}
```

On a side note, using *continue;* statement in a loop skips the current iteration and proceeds to the next iteration. If there are any statements in the loop after the *continue;* statement, they are not executed for the current iteration. The *break;* statement, on the other hand, terminates the entire loop. Once a *break;* statement is encountered, the control exits the loop, and no further iterations are performed. As a practice, replace *break*; statement with *continue;* in the previous example to examine the effects. Do not forget to put *break;* back when you are done testing.

Exercise 03

Write a simple adventure game as follows:

1. The game has an array that stores these strings “Forest”, “River”, “Mountain”, “Cave”, “Desert”.
2. A ***printLocation*** function that takes two parameters, the array and an integer variable representing an index to print the current location of the player. Initially, the current location will be at index 0, which means at the “Forest”.
3. A ***playGame*** function that takes two parameters, the array and its size. The job of this function is to ask the user to enter one of these commands, which are actually strings: “forward”, “backward”, “quit”. Depending on the user input, the function should either increment or decrement the array index and print the new location by calling the ***printLocation*** function.
4. Users should be allowed to move linearly from one location to another. Overjumping is not permitted.
5. The ***playGame*** function must have an infinite loop that will keep asking the user to enter their chosen command indefinitely until they choose “quit”.

6. The ***palyGame*** function must check for invalid user inputs commands.
7. The ***main*** function should have minimal code for merely declaring and initializing the array and calling the ***playGame*** function.

Exercise 04

Write a simple guessing game as follows:

1. The game has two helper functions.
2. A ***playGame*** function that takes no parameters. The tasks of this function are:
    - Generate a random number between 1 and 100.
    - Declare an integer array to store all guesses made by a user.
    - An indefinite loop that will keep asking a user to make guesses until they guess right. To help users guess right, print one of these messages with every guess: “Too low! Try again” and “Too high! Try again”.
    - If a user enters a guess outside of the allowed range, an error message should be printed out and prompt the user to enter another guess.
3. A ***processGuesses*** function that takes two parameters, the array and its size. This function should be called from the

playGame function when the user hits a right guess to print out all guesses made.

4. The ***main*** function should have minimal code for merely calling the ***playGame*** function.

SEARCH AND SORT ARRAYS

By now, you have gained good experience working with arrays. It's time to explore how arrays can be instrumental in solving real-life problems. This section will discuss two fundamental problems in computer science: searching and sorting.

It's hard to imagine any software that doesn't incorporate at least one, if not both, of these functionalities. We'll begin with the simplest search algorithm, Linear Search, and then progress to a more efficient search algorithm, Binary Search.

Linear Search

Figure 03 depicts an array named ***anyArray***. To perform a linear search, all we need to do is compare the value we are searching for, which we'll call ***target***, with every value stored in the array starting at index 0: ***target == anyArray[0];***

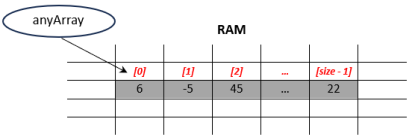


Figure 03: Linear search.

If the condition returns **true**, we've successfully found our ***target***. If not, we increment the index and perform the next comparison, ***target == anyArray[1]***. If the condition now returns **true**, that means we've found our ***target***. Otherwise, we continue incrementing the index until we either find the target or reach the end of the array. At that point, we can print a message stating that the target does not exist in ***anyArray***.

Example 07

The code is straightforward and self-explanatory. For a better understanding, modify the target value to include both existing and non-existing values in the array. This will allow you to test whether the code behaves as expected in both scenarios.

```
#include <iostream>
using namespace std;
int main() {
    int arr[] = { 1, 2, 3, 4, 5 }, size = 5, target = 3;
    for (int i = 0; i < size; i++) {
        if (arr[i] == target) {
            cout << "Target is found at index " << i << endl;
            return 0;
        }
    }
    cout << "Target is not Found!" << endl;
    return 0;
}
```

Example 08

Execute the code with various values for the ***bookName*** variable, including both existing and non-existing values in the array, to test the code's behavior in both scenarios. In this example, the ***linearSearch*** function is responsible for executing the linear search, adhering to the best programming practices of having functions perform only one specific task. Additionally, note that the array is passed to this function as a constant array using the **const** modifier, line 5. Although this is not required, it also aligns with best programming practices. The ***linearSearch*** function is designed to perform a linear search without modifying the array contents. To safeguard the array from any accidental changes, the **const** modifier is used. Consequently, any attempt to modify the array within

the ***linearSearch*** function will result in an error or, at the very least, a warning in most compilers. However, the modification on line 27 will not cause any errors as it is executed in the ***main*** function, which does not have any restrictions on modifying the array contents.

```
1   #include <iostream>
2   #include <string>
3   using namespace std;
4
5   bool linearSearch(const string arr[], int size, string bookName) {
6       for (int i = 0; i < size; i++) {
7           if (arr[i] == bookName) {
8               return true;
9           }
10      }
11      return false;
12  }
13
14  int main() {
15      string books[] = { "Java", "Python", "Ruby", "Prolog", "C++"};
16      int size = 5;
17      string bookName = "C++";
18
19      if (linearSearch(books, size, bookName)) {
20          cout << bookName << " is in the collection." << endl;
21      }
22      else {
23          cout << bookName << " is not in the collection." << endl;
24      }
25
26      // Modifying the array
27      books[0] = "JavaScript";
28      cout << "The first book is now: " << books[0] << endl;
29
30      return 0;
31  }
```

Using the modifier **const** with arrays can become tricky. In C++, if an outer function is declared to receive a **const** array as a parameter and it calls an inner function, the inner function must also be declared to receive a **const** array. This is because the **const** modifier promises that the function will not modify the array. If the inner function were allowed to modify the array, it would contradict this promise. Therefore, to ensure the integrity of the **const** promise, any function that is passed a **const** array must also

promise that any inner function it calls will not modify the array contents.

Exercise 05

A store manager needs to check whether an item exists or not in their store. To assist with this, write a program that implements the following functions:

1. A **void** *fillArray* function that takes 2 parameters, the array and its size. This function task is to fill the array with the item codes that the user enters.
2. A **bool** *linearSearch* function that takes three parameters: the array, the array size, and an item code. The function returns true if the code is found in the array; otherwise, it returns false.
3. In the *main* function, an array to store the item codes is created. For simplicity, create an **int** array that can store five item codes. The main function should print out appropriate messages depending on what the *linearSearch* function returns to announce the outcomes of the search.

```
Enter item[1] code: 123
Enter item[2] code: 456
Enter item[3] code: 789
Enter item[4] code: 147
Enter item[5] code: 258
Enter the item code you want to search: 369
Item is not in stock.
```

Figure 04: Sample output of the required program.

Use the attached output sample as guidance, Figure 04.

Exercise 06

A simple Employee Management System. The program should allow users to enter employee information (ID, name, and salary) and search

for employees by their ID. Users should be able to input as many employees as they want up to the maximum number allowed. To keep the code simple, the code only needs to read employees' last names (one word). Your implementation should include the following components.

1. A constant MAX_EMPLOYEES is set to 100.
2. A function findEmployeeById that performs a linear search on an array of employee IDs.
3. A function enterEmployeeInfo, that allows users to input employee data (ID, name, salary) and store it in three separate arrays.
4. A main function that:
    - Declares three arrays for employee IDs, names, and salaries with appropriate types.
    - Calls enterEmployeeInfo to populate the arrays.
    - Prompts the user for an ID to search.
    - Uses findEmployeeById to search for the employee.
    - Displays the found employee's information or a 'not found' message.

Binary Search

The binary search algorithm is significantly more efficient than the linear search. For instance, consider an array with 1,000,000 elements. In the worst-case scenario, a linear search would require 1,000,000 comparisons to either find the target or determine that the target does not exist in the array. In contrast, the binary search algorithm can accomplish the same task with only 20 comparisons,

which is $\mathbf{log_2(1,000,000) = 20}$!! The base-2 logarithm is used because of the way the algorithm operates. In each step of a binary search, the algorithm halves the size of the search space. This is achieved by selecting the middle element and discarding either the left half or the right half of the search space. Consequently, the number of steps it takes to reduce the search space from ***n*** elements to 1 is equivalent to the number of times you can divide ***n*** by 2 until you reach 1.

Binary Search Algorithm

1. **Start with a Sorted Array:** Binary search requires the array to be sorted in ascending or descending order. If the array is not sorted, we must sort it first.
2. **Find the Middle Element:** Calculate the middle index of the array. This can be done by adding the start index and the end index of the array and then dividing the sum by 2. If this division results in a fractional number, decide whether to round up or down.
3. **Compare the Middle Element with the Target Value:**
    1. If the middle element is equal to the target value, then we have found the target and the search is complete.
    2. If the middle element is greater than the target value (in an ascending sorted array), then the target must be in the left half of the array.
    3. If the middle element is less than the target value (in an ascending sorted array), then the target must be in the right half of the array.
4. **Repeat the Process:** If the target is not found, repeat the process with the appropriate half of the array (either left or right, depending on the comparison in Step 3).
5. **End Condition:** The search ends when the target is found

or when the start index is greater than the end index, which means the target is not in the array.

Example 09

Run and test the provided code for several target values. To solidify your understanding, trace the results back to the code to understand how it actually finds the correct results with the help of Figure 05. As usual, the *binarySearch* function takes three parameters: the array, its size, and the target.

Because the Binary Search algorithm keeps halving the array, we need to track the **start** and **end** indices for each chosen half. Figure 05 shows that at the beginning, the start index is 0, and the end index is set to 9, following the code in line 7. The *while* loop condition will keep checking whether **start** is smaller or equal to **end**. When this condition fails, it means the array size is reduced to 0 without finding the target. At this point, line 23 executes and returns -1.

Otherwise, the *while* loop will start its first iteration by calculating the **middle** index value on line 10, which is *middle = (0 + 9) /2 = 4.5*. However, the **middle** variable type was declared as **int**. In other scenarios, this should be considered a logical error. However, in this particular example, it is used to get rid of the fraction part of the division as the indices must be integers. Therefore, the **middle** variable is set to equal 4.

In the *if-else* statements, we check three possible scenarios: line 13 checks if the value at the calculated **middle** index equals the **target**. If this is **true**, then line 14 returns the index to announce that the **target** is found. Otherwise, the second condition checks whether the value at the calculated **middle** index is smaller than the **target**. If this is **true**, we can ignore the entire lower half of the array because we know that the array is sorted in ascending order and it is impossible to find the **target** in that half. Line 17 modifies the **start** index accordingly. On the other hand, if this condition returns **false**, then we can conclude

the opposite, which is that the **target** must be in the lower half of the array. Line 20 modifies the **end** index accordingly.

Figure 5 shows the *while* loop iterations that will lead to the position where the **target** exists. The *binarySearch* function will return either the index where the target is found or -1 to indicate that the target was not found. Based on what the *binarySearch* function returns, the **main** function will print the appropriate message.

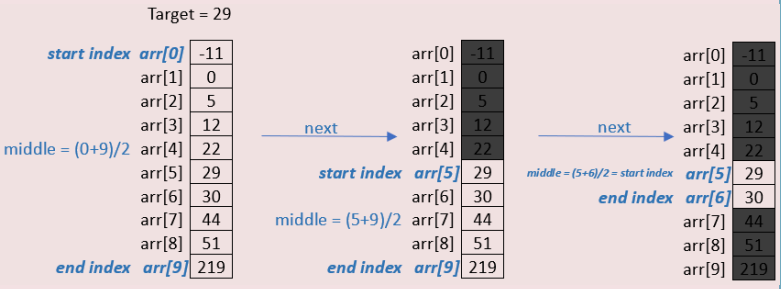


Figure 05: Binary search example.

```cpp
#include <iostream>
using namespace std;
const int MAX_Array_Size = 10;

int binarySearch(int arr[], int size, int target) {
    int start = 0;          // Starting index
    int end = size - 1;  // Ending index

    while (start <= end) {
        int middle = (start + end) / 2;// Calculate the middle index

        // Check if the target is present at the middle
        if (arr[middle] == target) {
            return middle;
        }
        else if (arr[middle] < target) {
            start = middle + 1;
        }
        else {
            end = middle - 1;
        }
    }
    return -1; // Return -1 if the target is not found
}

int main() {
    int arr[MAX_Array_Size] = { -11, 0, 5, 12, 22, 29, 30, 44, 51, 219 };//Sorted array
    int target = 29;

    int result = binarySearch(arr, MAX_Array_Size, target);

    if (result != -1) {
        cout << "Element " << target << " found at index: " << result << endl;
    }
    else {
        cout << "Element " << target << " not found in the array" << endl;
    }
    return 0;
}
```

Example 10

Run and test the provided code for several target values ***(note that line 48 must be entered as one line)***. In this example, we can examine how the binary search algorithm can be useful in solving real-world problems. For instance, national public libraries' catalogs can contain up to several million books. Searching such extensive catalogs necessitates very efficient search algorithms.

Study the code and notice how we removed all braces from the if..else statements. Since all branches have only one statement in them, there is no need for any braces. Additionally, we used three

separate arrays to hold the books' acquisition numbers, titles, and authors' names.

In the following section, we will discuss multidimensional arrays. For now, it's acceptable if we have to declare a couple of separate arrays.

```cpp
#include <iostream>
#include <string>
using namespace std;

int binarySearch(int acqNumbers[], int size, int targetAcqNumber) {
    int left = 0;
    int right = size - 1;

    while (left <= right) {
        int mid = (left + right) / 2;

        if (acqNumbers[mid] == targetAcqNumber) return mid;
        else if (acqNumbers[mid] < targetAcqNumber)  left = mid + 1;
        else right = mid - 1;
    }
    return -1;
}

int main() {
    const int SIZE = 5;
    //array to store the books acquisition numbers.
    int acqNumbers[SIZE] = { 102, 145, 231, 378, 492 };
    string titles[SIZE] = {
        "To Kill a Mockingbird",
        "Pride and Prejudice",
        "1984",
        "The Hobbit",
        "The Great Gatsby"
    };
    string authors[SIZE] = {
        "Harper Lee",
        "Jane Austen",
        "George Orwell",
        "J.R.R. Tolkien",
        "F. Scott Fitzgerald"
    };

    int targetAcqNumber = 102;
    int result = binarySearch(acqNumbers, SIZE, targetAcqNumber);

    if (result != -1) {
        cout << "Book found:" << endl;
        cout << "Title: " << titles[result] << endl;
        cout << "Author: " << authors[result] << endl;
        cout << "Acquisition Number: " << acqNumbers[result] << endl;
    }
    else {
        cout << "Book with Acquisition Number " << targetAcqNumber << ➥
          " not found in the catalog" << endl;
    }
    return 0;
}
```

Exercise 07

Write a program implementing a Binary Search algorithm to search a sorted array of words. Your implementation must include a *binarySearch* function that takes three parameters: the array of words, the size, and the target. This function must return either the index of the target if found, or -1 if the target word does not exist in the array. The **main** function should call the *binarySearch* function and print appropriate messages to the user based on what the *binarySearch* function returns.

Hint: In this exercise, you will compare words (i.e., strings) with each other. This is possible because characters are encoded according to the ASCII scheme. When comparing strings, you are actually comparing their ASCII codes character by character until a difference is found or the end of the string is reached.

Exercise 08

Answer these multiple-choice questions:

Question 1: What is the primary condition for applying a binary search algorithm on an array?

a) The array must be unsorted.

b) The array must be sorted.

c) The array must contain only positive numbers.

d) The array must contain unique elements.

Question 2: If an array has 1,000,000 elements, how many comparisons does the binary search algorithm need in the worst case to either find the target index or determine that the target does not exist in the array?

a) $\mathbf{\log_{10}(1,000,000) = 20}$

b) $\mathbf{\log_{2}(1,000,000) = 20}$

c) 0
d) It is impossible to know the answer

Question 3: Which of the following correctly initializes the ***left*** and ***right*** indices for a binary search?
a) int left = 1, right = size;
b) int left = 0, right = size;
c) int left = 1, right = size – 1;
d) int left = 0, right = size – 1;

Question 4: What does the binary search algorithm do if the middle element is equal to the target?
a) It returns the middle element.
b) It continues to search the right half of the array.
c) It returns the index of the middle element.
d) It continues to search the left half of the array.

Sorting Algorithms

There are several sorting algorithms in use. Their efficiency varies; however, we will discuss the simplest algorithm in this chapter, the Bubble-Sort algorithm.

Bubble-Sort Algorithm

1. Compare the first element with the second element in the unsorted array.
2. If the first element is larger than the second element, swap them; otherwise, no change is necessary.
3. Repeat step 2 for each pair of adjacent elements until reaching the end of the array.
4. At the end of step 3, the largest element will be at the last position in the array.
5. Repeat steps 1 to 3 for the remaining unsorted portion of the array (i.e., from the first element to the last unsorted element).

6. After each pass, the next largest element will be placed in its correct position, just before the previously sorted elements.
7. Repeat step 5 until the entire array is sorted and no more comparisons are needed.

Example 11

Run and test the provided code with several different array values. Be sure to carefully read and understand the Bubble Sort algorithm for smoother comprehension of this example. Figure 06 visualizes the algorithm's dynamics. At the heart of this algorithm, the largest values bubble up to the right of the array, effectively sorting the array in ascending order.

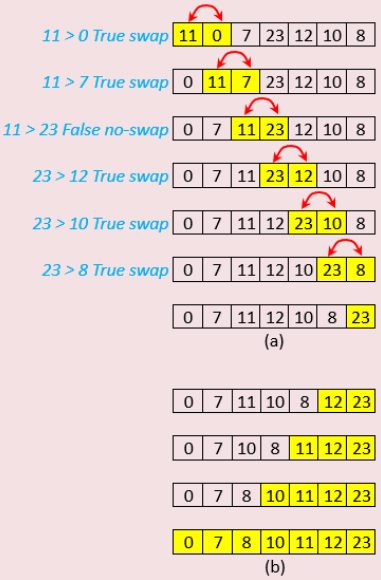


Figure 06: Bubble-Sort algorithm operation.

Figure 06(a) shows how the largest number in the array, 23, is placed in the last position at the far right. This process starts by comparing the first two positions of the array. The result of each comparison bubbles the larger number to the right. Initially, the number 11 was swapped with 0 and 7. Later, number 23 surpassed number 11 and swapped with the remaining values all the way to the rightmost position in the array. These comparisons and swaps are handled by an inner loop that compares every two adjacent elements using the condition ***arr[j***

] > arr[j + 1]. It is crucial not to exceed the array's boundaries, so the inner loop condition *j < size – i – 1* is used.

The condition *j < size* would normally stop the loop at the last position in the array. However, in this algorithm, we are comparing the last element with the element before it, requiring the condition to be *j < size – 1*. Additionally, we subtract **i** from **size** because, after placing the largest number at the last position in the array, we must exclude that particular element from comparisons in subsequent passes. Hence, the correct condition is *j < size – i – 1*. The inner loop passes through the entire array, but we also need an outer loop to keep the algorithm running for each element in the array. Therefore, the outer loop condition must be *i < size – 1*.

Figure 06(b) shows the effect of the outer loop on every element in the array, causing the array to be incrementally sorted at every iteration. Note that although the first three elements happen to be in sorted order, "0, 7, and 8", the loop will continue systematically running until all loop conditions are exhausted. While this is not the most efficient approach, the code can be modified to terminate early if needed.

```cpp
#include <iostream>
using namespace std;

// Function to print the array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

// Function to perform Bubble Sort
void bubbleSort(int arr[], int size) {
    // Loop through each element in the array
    for (int i = 0; i < size - 1; i++) {
        // Loop to compare adjacent elements
        for (int j = 0; j < size - i - 1; j++) {
            // If the current element is greater than the next element,
            // swap them, otherwise, skip to next comparisons.
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;

                // Print the array after each swap for testing only. You
                // can remove this line if you want.
                printArray(arr, size);
            }
        }
    }
}

int main() {
    const int size = 7;
    int arr[] = { 11, 0, 7, 23, 12, 10, 8 };

    cout << "Original array: ";
    printArray(arr, size);

    // Perform Bubble Sort
    bubbleSort(arr, size);

    cout << "Sorted array: ";
    printArray(arr, size);

    return 0;
}
```

Exercise 09

Modify example 11 code to sort an array of characters.

Exercise 10

Modify example 11 code to sort an array of strings.

Exercise 11

Answer the following multiple-choice questions:

Question 1: In the bubbleSort function, how many times does the outer loop execute for an array of size n?

A) n

B) n – 1

C) n + 1

D) 2n

Question 2: What does the variable **temp** represent in the bubbleSort function?

A) The total number of comparisons made.

B) The current index of the array being processed.

C) The temporary storage for swapping elements.

D) The size of the array.

Question 3: What is the primary purpose of the outer loop in the bubbleSort function?

A) To compare adjacent elements of the array.

B) To initialize the array and its size.

C) To control the number of passes through the array.

D) To print the elements of the array after each swap.

Question 4: What is the primary role of the inner loop in the bubbleSort function?

A) To print the elements of the array.

B) To calculate the sum of all elements in the array.
C) To find the maximum element in the array.
D) To compare adjacent elements and swap them if necessary.

MULTI-DIMENSIONAL ARRAYS

Arrays are flexible enough to contain other arrays within them, creating what is known as Multi-Dimensional arrays. For simplicity, we will discuss only the 2-dimensional arrays. Visually, a 2D array resembles a table with rows and columns. Higher dimensions are technically possible but are more challenging to visualize or conceptualize.

The syntax to declare a 2-dimensional array is:

baseType variableName[rows];

```
int anArray[5][7];
```

This declaration creates an array with 5 rows and 7 columns. It's important to note that array indices start from 0. Therefore, the row indices range from 0 to 4, and the column indices range from 0 to 6.

Example 12

Figure 07 illustrates how 2-dimensional arrays (or simply 2D) should be visualized as sets of rows and columns. The challenge lies in addressing each location within this array. The solution is to use two indices: one for rows and another for columns. It's essential to

	column 0	column 1	...	column j
row 0	[0][0]	[0][1]	...	[0][j]
row 1	[1][0]	[1][1]	...	[1][j]
⋮	⋮	⋮	⋱	⋮
row i	[i][0]	[i][1]	...	[i][j]

Figure 07: 2D Array indices.

remember that the first index refers to rows, while the second index refers to columns.

The provided code demonstrates how to declare and initialize the array in a single line. First, specify the number of rows and columns required for the array. Then, populate the array with values. Note how the code assigns values to the array row by row within curly braces, separated by commas except for the last row. To access any location in the array, you must specify the exact indices, as shown in Figure 07. Since multidimensional arrays are often large, nested loops are commonly used to access or modify values. Notice how the ranges of the outer and inner loops are structured to match the dimensions of the array and prevent out-of-bounds errors. For better comprehension, you can uncomment the second ***cout*** line inside the inner loop (and comment out the other ***cout*** line) to print the contents of the array along with their corresponding indices.

```
#include <iostream>
using namespace std;

int main() {
    // Declare and initialize a 2D array
    int matrix[3][4] = {  {1, 2, 3, 4},
                          {5, 6, 7, 8},
                          {9, 10, 11, 12}
                       };

    // Access and print elements of the 2D array
    cout << "2D Array Contents:" << endl;
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 4; ++j) {
            cout << matrix[i][j] << " ";
          //cout << "matrix[" << i << "][" << j << "]=" << matrix[i][j] << " ";
        }
        cout << endl;
    }
    return 0;
}
```

Example 13

Run the code and trace the outputs for a better understanding. This step is

crucial for comprehending how individual locations in a 2D array can be accessed. There are two functions in this example: ***populateArray***, which allows users to enter daily temperatures, and ***displayArray***, which prints the array's contents and calculates the daily temperature average.

The ***populateArray*** function takes a 2D array as a parameter, specifically of type **double** to store temperatures with decimals. As is typical with 2D arrays, it requires nested loops to access each location. In this implementation, locations are accessed row by row. Initially, the outer loop selects the first row ***(i = 0)***, and the inner loop iterates through each column ***(j = 0*** to ***j < TIMES)***. This process repeats for each subsequent row until the outer loop terminates when ***i < DAYS*** becomes **false**. This allows the user to populate the entire array with daily temperatures for 7 days.

The ***displayArray*** function also takes the same array as a parameter to print its contents and calculate the daily temperature average. Using a similar nested loop structure, it iterates through rows with an outer loop and columns with an inner loop. Within the inner loop, line 32 sums the temperatures for the day, while in the outer loop, line 34 calculates the average temperature for each day. Line 35 prints the average. Line 25 uses ***fixed*** and ***setprecision*(1)** manipulators to ensure that all printed temperatures have only one decimal place. These manipulators, which will be discussed in detail in Chapter 7, are included from the **<iomanip>** library. The tab escape character **\t** is used in lines 22, 28, and 31 to format the output in a tabular shape.

The **main** function primarily declares the 2D array and calls the ***populateArray*** and ***displayArray*** functions.

```cpp
#include <iostream>
#include <iomanip> // Include for setting precision
using namespace std;

// Constants for array dimensions
const int DAYS = 7; // Number of days in a week
const int TIMES = 3; // Number of times per day (morning, afternoon, evening)

// Function to populate the temperature array
void populateArray(double temperatures[DAYS][TIMES]) {
    for (int i = 0; i < DAYS; ++i) {
        cout << "Enter temperatures for day " << i + 1 << " (morning, afternoon, evening): ";
        for (int j = 0; j < TIMES; ++j) {
            cin >> temperatures[i][j];
        }
    }
}

// Function to display the temperature array and calculate daily averages
void displayArray(double temperatures[DAYS][TIMES]) {
    cout << "\nTemperature readings for the week:\n";
    cout << "Day\tMorning\t\tAfternoon\tEvening\t\tAverage\n";

    // Set precision for cout to 1 decimal place
    cout << fixed << setprecision(1);

    for (int i = 0; i < DAYS; ++i) {
        cout << i + 1 << "\t";
        double sum = 0;
        for (int j = 0; j < TIMES; ++j) {
            cout << temperatures[i][j] << "\t\t";
            sum += temperatures[i][j];
        }
        double average = sum / TIMES;
        cout << average << "\n";
    }
}

int main() {
    double temperatures[DAYS][TIMES]; // 2D array to store temperatures

    // Populate the array with user input
    populateArray(temperatures);

    // Display the populated array and calculate daily averages
    displayArray(temperatures);

    return 0;
}
```

Exercise 12

This exercise is similar to Example 13, so you can reuse some of its code. Write a program to enter students' grades and calculate the maximum, minimum, and average of these grades using 2D arrays. In this exercise, you must follow standard C++ coding best practices by first declaring all functions at the top of the file (above the main

function) and then defining the functions below the main function. The program must include at least the following functions:

1) main: This function must have minimal code, primarily to declare the 2D array and call other functions.

2) initializeGrades: This function takes the 2D array as a parameter and initializes its entries with default values of its base type. Remember the importance of avoiding accessing any uninitialized memory locations, especially with large arrays, when it becomes difficult to track all memory locations.

3) inputGrades: This function takes the 2D array as a parameter and allows users to enter students' grades, which can be decimals. The number of students and the number of grades must be declared as constants.

4) printGrades: This function takes the 2D array as a constant parameter to prevent any accidental changes to its values. The tasks of this function are to find the maximum, minimum, and average grades, and then print all grades along with these statistics in a formatted tabular presentation. You can use the same manipulators used in Example 13.

Exercise 13

You are tasked with developing a program to manage seating in a movie theater using a 2-dimensional array. Each seat in the theater can either be available **' . '** or booked **'X'**. Your program should print the current seating arrangement, allow the booking of seats, and handle invalid seat selections gracefully.

Program functions:

1) initializeTheater(): Initializes a 2-dimensional array (theater) to represent the seating arrangement. Initially, all seats are set to **' . '** to denote availability.

2) The theater array must be declared globally to ensure accessibility across all functions without passing it as a parameter.

3) printSeating(): Displays the current seating arrangement in a grid format, showing row and column numbers along with seat availability **' . '** or **'X'**.

4) bookSeat(int row, int col): Books a seat specified by row and column numbers by checking if the seat is available **' . '**, books it by changing the value to **'X'.** This function must provide appropriate feedback for successful bookings, attempts to book already booked seats, and invalid seat selections.

5) main function: Contains minimal code, primarily calling other functions to simulate booking seats at various rows and columns. It should print the seating arrangement **before and after each booking attempt** to demonstrate changes in seat availability.

Note: Figure 08 shows a sample run of the program.

```
Current seating arrangement:
  1 2 3 4 5 6 7 8 9 10
1 . . . . . . . . . .
2 . . . . . . . . . .
3 . . . . . . . . . .
4 . . . . . . . . . .
5 . . . . . . . . . .

Seat at Row 1, Seat 2 has been booked.
Seat at Row 3, Seat 5 has been booked.
Seat at Row 5, Seat 10 has been booked.
Seat at Row 2, Seat 3 has been booked.
Seat at Row 3, Seat 5 is already booked. Please choose another seat.
Seat at Row 10, Seat 10 is invalid seat selection. Please choose a seat within the valid range.

Current seating arrangement:
  1 2 3 4 5 6 7 8 9 10
1 . X . . . . . . . .
2 . . X . . . . . . .
3 . . . . X . . . . .
4 . . . . . . . . . .
5 . . . . . . . . . X
```

Figure 08: A sample run.

WRAP UP

This comprehensive introduction to arrays covers a wide range of essential concepts and techniques. Starting with the basics, readers learn how to declare, initialize, and access elements in one-dimensional arrays. The material then progresses to more advanced array manipulation, demonstrating various methods for working with arrays, including using loops to populate and display their contents.

An important aspect covered is the passing of arrays to functions, highlighting the differences between passing individual elements and entire arrays. This knowledge is crucial for developing modular and efficient code. The text also delves into fundamental search algorithms, introducing both linear search for unsorted arrays and the more efficient binary search for sorted arrays. In addition, the Bubble Sort algorithm is explained as an example of how to sort array elements, providing a foundation for understanding more complex sorting techniques.

Multi-dimensional arrays are introduced, with a focus on 2D arrays. Readers learn techniques for declaring, initializing, and manipulating these more complex data structures. Throughout the material, various examples and exercises demonstrate real-world applications of arrays, such as storing temperatures, managing student grades, and implementing a simple theater seating system. These practical scenarios help reinforce the theoretical concepts and illustrate the versatility of arrays in solving diverse programming problems.

The content emphasizes important programming practices, including avoiding uninitialized memory access, using constants for array sizes, and properly structuring code with function declarations and definitions. By adhering to these best practices, readers can develop more robust and maintainable code. Overall, this material provides a solid foundation in array usage, equipping

students with the necessary skills to tackle more advanced data structures and algorithms in their future studies.

ANSWER KEY

Exercise 01:

```cpp
#include <iostream>
using namespace std;

int main() {
    double numbers[5]; // Declare an array of doubles with size 5
    // max initialized to smallest number possible
    // min initialized to largest number possible
    double sum = 0.0, max = -1e308, min = 1e308;

    cout << "Please enter 5 numbers:" << endl;

    // Loop to get user input and store it in the array
    for (int i = 0; i < 5; ++i) {
        cout << "Enter number " << (i + 1) << ": ";
        cin >> numbers[i];
        sum += numbers[i]; // Add the number to the sum
        if (numbers[i] > max) max = numbers[i]; // Check for new maximum
        if (numbers[i] < min) min = numbers[i]; // Check for new minimum
    }

    double average = sum / 5; // Calculate the average

    // Print the results
    cout << "Sum: " << sum << endl;
    cout << "Average: " << average << endl;
    cout << "Max: " << max << endl;
    cout << "Min: " << min << endl;

    return 0;
}
```

Exercise 02:

```cpp
#include <iostream>
#include <string>
using namespace std;

int main() {
    string words[5]; // Declare an array of strings with size 5
    // Initialize to empty and space for comparison
    string longestWord = "", shortestWord = " ";

    cout << "Please enter 5 words:" << endl;

    // Loop to get user input and store it in the array
    for (int i = 0; i < 5; ++i) {
        cout << "Enter word " << (i + 1) << ": ";
        cin >> words[i];

        // Check for new longest and shortest words
        if (words[i].length() > longestWord.length())
            longestWord = words[i];

        if (shortestWord == " " || words[i].length() < shortestWord.length())
            shortestWord = words[i];
    }

    // Print the results
    cout << "==============================" << endl;
    cout << "Longest word: " << longestWord << endl;
    cout << "Shortest word: " << shortestWord << endl;

    return 0;
}
```

Exercise 03:

A sample solution:

```
#include <iostream>
#include <string>
using namespace std;

void printLocation(string locations[], int currentLocation) {
    cout << "You are currently at the " << locations[currentLocation] << ".\n";
}

void playGame(string locations[], int size) {
    int currentLocation = 0;
    string command;

    while (true) {
        printLocation(locations, currentLocation);
        cout << "Enter a command (backward, forward, quit): ";
        cin >> command;

        if (command == "backward" && currentLocation - 1 >= 0) {
            currentLocation--;
        }
        else if (command == "forward" && currentLocation + 1 < size) {
            currentLocation++;
        }
        else if (command == "quit") {
            break;
        }
        else {
            cout << "Invalid command.\n";
        }
    }
}

int main() {
    const int num_locations = 5;
    string locations[num_locations] = { "Forest", "River", "Mountain", "Cave", "Desert" };

    playGame(locations, num_locations);

    return 0;
}
```

Exercise 04:

A sample solution:

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

const int MAX_ATTEMPTS = 100;

void processGuesses(int guesses[], int size) {
    cout << "Here are your guesses:\n";
    for (int i = 0; i < size; i++) {
        cout << "Attempt " << (i + 1) << ": " << guesses[i] << "\n";
    }
}

void playGame() {
    srand(time(0)); // seed random number generator
    int number = rand() % 100 + 1; // generate a random number between 1 and 100
    int guesses[MAX_ATTEMPTS];
    int guess;
    int attempts = 0;

    while (true) {
        cout << "Enter your guess (1-100): ";
        cin >> guess;

        // Check if the guess is out of range
        if (guess < 1 || guess > 100) {
            cout << "Out of range. Please enter a number between 1 and 100.\n";
            continue;
        }

        guesses[attempts] = guess;
        attempts++;

        if (guess < number) {
            cout << "Too low! Try again.\n";
        }
        else if (guess > number) {
            cout << "Too high! Try again.\n";
        }
        else {
            cout << "Congratulations! You found the number in " << attempts << " attempts.\n";
            processGuesses(guesses, attempts);
            break;
        }
    }
}

int main() {
    cout << "Welcome to the number guessing game!\n";
    playGame();
    return 0;
}
```

Exercise 05:

A sample solution:

```cpp
#include<iostream>
using namespace std;

void fillArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        cout << "Enter item[" << i + 1 << "] code: ";
        cin >> arr[i];
    }
}

bool linearSearch(int arr[], int size, int itemCode) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == itemCode) {
            return true;
        }
    }
    return false;
}

int main() {
    const int size = 5;
    int itemCodes[size];

    fillArray(itemCodes,size);

    int searchItem;
    cout << "Enter the item code you want to search: ";
    cin >> searchItem;

    if (linearSearch(itemCodes,size, searchItem)) {
        cout << "Item is in stock." << endl;
    }
    else {
        cout << "Item is not in stock." << endl;
    }

    return 0;
}
```

Exercise 06:

A sample solution:

```cpp
#include <iostream>
#include <string>
using namespace std;

const int MAX_EMPLOYEES = 100;

int findEmployeeById(const int ids[], int size, int targetId) {
    for (int i = 0; i < size; i++) {
        if (ids[i] == targetId) return i;
    }
    return -1;
}

int enterEmployeeInfo(int ids[], string names[], double salaries[]) {
    int count = 0, id;
    while (count < MAX_EMPLOYEES) {
        cout << "Enter employee ID (or -1 to finish): ";
        cin >> id;
        if (id == -1) break;

        ids[count] = id;
        cout << "Enter name: ";
        cin >> names[count];
        cout << "Enter salary: ";
        cin >> salaries[count];
        count++;
    }
    return count;
}

int main() {
    int ids[MAX_EMPLOYEES];
    string names[MAX_EMPLOYEES];
    double salaries[MAX_EMPLOYEES];

    int count = enterEmployeeInfo(ids, names, salaries);

    cout << "Enter ID to search: ";
    int id;
    cin >> id;

    int index = findEmployeeById(ids, count, id);

    if (index != -1) {
        cout << "Found: ID: " << ids[index] << ", Name: " << names[index]
            << ", Salary: $" << salaries[index] << endl;
    }
    else {
        cout << "Employee not found." << endl;
    }

    return 0;
}
```

Exercise 07:

A sample solution:

```cpp
#include <iostream>
#include <string>
using namespace std;

int binarySearch(string arr[], int size, string target) {
    int left = 0;
    int right = size - 1;

    while (left <= right) {
        int mid = (left + right) / 2;

        if (arr[mid] == target) return mid;
        else if (arr[mid] < target)  left = mid + 1;
        else right = mid - 1;
    }
    return -1;
}

int main() {
    const int SIZE = 10;
    string words[SIZE] = { //alphabetically sorted array.
        "apple", "banana", "cherry", "date", "elderberry",
        "fig", "grape", "honeydew", "kiwi", "lemon"
    };

    string target;
    cout << "Enter a word to search for: ";
    cin >> target;

    int result = binarySearch(words, SIZE, target);

    if (result != -1)
        cout << "The word '" << target << "' was found at index " << result << endl;
    else
        cout << "The word '" << target << "' was not found in the array" << endl;

    return 0;
}
```

Exercise 08:

Question 1: B

Question 2: B

Question 3: D

Question 4: C

Exercise 09:

The required changes are:

1) Declare an array of characters instead of an array of integers:

char arr[size] = { 'd', 'g', 'e', 'b', 'c', 'f', 'a' };

2) Change the *printArray* and *bubbleSort* functions' headers to take

an array of characters as a parameter:

```
void printArray(char arr[], int size)
void bubbleSort(char arr[], int size)
```

3) Change the **temp** variable inside the *if* statement to a *char* type:

```
char temp = arr[j];
```

Exercise 10:

The required changes are:

1) Include the **<string>** library.

2) Declare an array of strings instead of an array of integers:

```
string arr[size] = { "Zara", "John", "Mike", "Alice", "Bob" };
```

3) Change the *printArray* and *bubbleSort* functions' headers to take an array of strings as a parameter:

```
void printArray(string arr[], int size)
void bubbleSort(string arr[], int size)
```

4) Change the **temp** variable inside the *if* statement to a string type:

```
string temp = arr[j];
```

Exercise 11:

Question 1: B

Question 2: C

Question 3: C

Question 4: D

Exercise 12:

A sample solution:

```cpp
#include <iostream>
#include <iomanip> // Include for formatting output
using namespace std;

const int STUDENTS = 3;
const int QUIZZES = 5;

//////// Declaring functions (prototypes) ////////
void initializeGrades(double grades[STUDENTS][QUIZZES]);
void inputGrades(double grades[STUDENTS][QUIZZES]);
void printGrades(const double grades[STUDENTS][QUIZZES]);

int main() {
    double grades[STUDENTS][QUIZZES];
    initializeGrades(grades);
    inputGrades(grades);
    printGrades(grades);
    return 0;
}

//////// Defining functions ////////
void initializeGrades(double grades[STUDENTS][QUIZZES]) {
    for (int i = 0; i < STUDENTS; ++i) {
        for (int j = 0; j < QUIZZES; ++j) {
            grades[i][j] = 0.0;
        }
    }
}

void inputGrades(double grades[STUDENTS][QUIZZES]) {
    for (int i = 0; i < STUDENTS; ++i) {
        cout << "Enter grades for student " << i + 1 << ":\n";
        for (int j = 0; j < QUIZZES; ++j) {
            cout << "  Quiz " << j + 1 << ": ";
            cin >> grades[i][j];
        }
    }
}

void printGrades(const double grades[STUDENTS][QUIZZES]) {
    cout << "\nGrades:\n";
    cout << "Student\tQ1\tQ2\tQ3\tQ4\tQ5\tMax\tMin\tAverage\n";
    for (int i = 0; i < STUDENTS; ++i) {
        double sum = 0, maxGrade = grades[i][0], minGrade = grades[i][0];
        cout << i + 1 << "\t";
        for (int j = 0; j < QUIZZES; ++j) {
            cout << grades[i][j] << "\t";
            sum += grades[i][j];
            if (grades[i][j] > maxGrade) maxGrade = grades[i][j];
            if (grades[i][j] < minGrade) minGrade = grades[i][j];
        }
        double average = sum / QUIZZES;
        cout << maxGrade << "\t" << minGrade << "\t" << fixed << setprecision(1) << average << endl;
    }
}
```

Exercise 13:

A sample solution:

```
#include <iostream>
using namespace std;

const int ROWS = 5;        // Number of rows in the theater
const int COLS = 10;       // Number of seats per row
char theater[ROWS][COLS];  // 2D array to store seat availability ('O' for available, 'X' for booked)

// Function to initialize the theater with available seats
void initializeTheater() {
    for (int i = 0; i < ROWS; ++i) {
        for (int j = 0; j < COLS; ++j) {
            theater[i][j] = '.';  // '.' represents an available seat
        }
    }
}

// Function to display the current seating arrangement
void printSeating() {
    cout << "Current seating arrangement:" << endl;
    cout << "   ";
    for (int j = 0; j < COLS; ++j) {
        cout << j + 1 << " ";  // Display column numbers
    }
    cout << endl;
    for (int i = 0; i < ROWS; ++i) {
        cout << i + 1 << " ";  // Display row numbers
        for (int j = 0; j < COLS; ++j) {
            cout << theater[i][j] << " ";  // Display seat status ('O' or 'X')
        }
        cout << endl;
    }
    cout << endl;
}

// Function to book a seat based on user input
void bookSeat(int row, int col) {
    if (row >= 1 && row <= ROWS && col >= 1 && col <= COLS) {  // Check if input is within valid range
        if (theater[row - 1][col - 1] == '.') {  // Check if seat is available
            theater[row - 1][col - 1] = 'X';    // Book the seat ('X' represents booked)
            cout << "Seat at Row " << row << ", Seat " << col << " has been booked." << endl;
        }
        else {
            cout << "Seat at Row " << row << ", Seat " << col << " is already booked. Please choose another seat.\n";
        }
    }
    else {
        cout << "Seat at Row " << row << ", Seat " << col << " is invalid seat selection. Please choose a seat within the valid range.\n\n";
    }
}

int main() {
    initializeTheater(); // Initialize the theater with available seat
    printSeating(); // Display initial seating arrangement

    bookSeat(1, 2); // Simulate booking seats
    bookSeat(3, 5);
    bookSeat(5, 10);
    bookSeat(2, 3);
    bookSeat(3, 5); // Attempt to book the same seat again
    bookSeat(10, 10); // Attempt to book an invalid seat

    printSeating(); // Display updated seating arrangement after bookings

    return 0;
}
```

END OF CHAPTER EXERCISES

Multiple Choice Questions:

1) What is the correct way to declare an array of 5 integers?

a) int var(5);

b) int var{5};

c) int var[5];

d) int var = 5;

2) What is the index of the last element in an array of size 5?
a) 5
b) 4
c) 3
d) 0

3) Which loop is used in Example 03 to iterate through the array?
a) while loop
b) do-while loop
c) for loop
d) for-each loop

4) What is the benefit of using a constant for array size declaration?
a) It makes the code run faster
b) It allows for dynamic resizing of the array
c) It prevents resizing the array at runtime, which is not allowed in C++
d) It reduces memory usage

5) In C++, array indices start at:
a) 0
b) 1
c) -1
d) Depends on the array type

True/False Questions:

6) Arrays can store multiple types of data.
7) It's necessary to fill all array slots in sequence when initializing.
8) Accessing an uninitialized array slot can lead to undefined behavior.
9) The for-each loop eliminates concerns about index boundaries when iterating through an array.
10) In C++, you can omit the array size when initializing it with values.

Coding Questions:

11) Write a C++ code snippet to declare and initialize an array of base type of double with values 1.2, 2.98, 3.01, 4.33, and 5.5.

12) Write a for loop to print all elements of an integer array named "numbers" of size 10.
13) Declare a character array to store the alphabet (a-z) and initialize it using a for loop.
14) Write a for-each loop to print all elements of a string array named "words". Assume the array "words" is already declared and initialized.
15) Write a code snippet to find the maximum value in an integer array named "values" of size 5. Assume the array "Values" is already declared and initialized.

Multiple Choice Questions:

16) When passing an array to a function, how should you pass the array name?
a) With square brackets []
b) Without square brackets
c) With parentheses ()
d) With curly braces {}

17) What does the ***break;*** statement do in a loop?
a) Skips the current iteration
b) Terminates the entire loop
c) Restarts the loop
d) Has no effect on the loop

18) What does the ***continue;*** statement do in a loop?
a) Terminates the entire loop
b) Restarts the loop
c) Skips the current iteration
d) Has no effect on the loop

19) How is passing an array as a parameter different from passing by value or by reference?
a) It's exactly the same as passing by value
b) It's exactly the same as passing by reference
c) It's a unique category, different from both
d) It depends on the compiler

20) When passing an array to a function, what additional

information is typically needed?
a) The size of the array
b) The memory address of the array
c) The data type of the array
d) The name of the array

True/False Questions:

21) Passing individual array elements to a function is the same as passing any other variable of the same type.
22) When calling a function with an array parameter, you should include the square brackets with the array name.
23) In C++, arrays are always passed by reference to functions.

Coding Questions:

24) Write a function declaration for a void function named processArray that takes an integer array and its size as parameters.
25) Write a code snippet that demonstrates how to pass an individual array element to a function named squareNumber.
26) Write a void function that takes an integer array of size 10 as a parameter and initializes it with random numbers between 1 and 100.
27) Write a function that finds and returns the index of a given value in an array, or -1 if the value is not found. Assume that the array stores integer values.
28) Write a function that reverses the order of elements in an array. Assume that the array stores integer values.
29) Write a code snippet to call ***calculateAverage*** function that takes a double array myArray and its size as parameters. The function should return the average. Do NOT implement the function.

Multiple Choice Questions:

30) What is the primary characteristic of a linear search algorithm?
a) It requires the array to be sorted
b) It compares the target with every element in the array sequentially

c) It divides the array in half repeatedly
d) It uses a hash function to locate elements

31) What does the ***const*** modifier do when used with an array parameter in a function?
a) It makes the array elements constant
b) It prevents the function from modifying the array contents
c) It allows the function to modify only certain elements of the array
d) It has no effect on arrays

32) In the context of the Employee Management System exercise, what does the findEmployeeById function do?
a) It adds a new employee to the system
b) It deletes an employee from the system
c) It performs a linear search on an array of employee IDs
d) It modifies an employee's information

True/False Questions:

33) In a linear search, each item in the array is sequentially compared to the target value until the target is found or the end of the array is reached.

34) When using const with array parameters, inner functions called by the outer function don't need to use const.

35) The linear search algorithm requires the array to be sorted before searching.

Coding Questions:

36) Write a function that implements a linear search on an integer array. The function should take three parameters (the array, its size, and the target) and return the index of the target if found, or -1 if not found. The function should not be able to modify any of the array values.

37) Re-implement the fillArray function described in Exercise 05 by filling the array with random 3-digit codes ranging from 100 to 999.

38) Write a function called countDuplicates that takes a character array and its size as parameters. The function's task is to determine if the array contains any duplicate characters and count the occurrences of these duplicates. The function must not modify the

array contents.
39) Which of the following statements is true about binary search?
a) Binary search can be applied to either sorted or unsorted arrays.
b) Binary search can only be applied to arrays sorted in descending order.
c) Binary search divides the search space into two halves in each step.
d) Binary search can only be applied to arrays sorted in ascending order.

40) Why is the base-2 logarithm used in measuring the efficiency of the binary search algorithm?
a) Because the algorithm compares each element sequentially.
b) Because the algorithm halves the search space at each step.
c) Because the algorithm sorts the array first.
d) Because the algorithm multiplies the search space at each step.

41) Write a program that uses two arrays: one to store employee IDs and another to store employee names. Implement a binary search function to search the ID array. This function should return the index of the ID if found, or -1 if the ID doesn't exist. In the main function, create an indefinite loop that:

- Prompts users to enter an ID they want to look up
- If the ID is found, it prints both the employee ID and name
- If the ID is not found, it prints a message stating that the employee ID does not exist
- Terminates the loop if the user enters -1

Multiple Choice Questions:

42) What is the main characteristic of the Bubble Sort algorithm?
a) It sorts in descending order
b) It uses recursion
c) The largest (or smallest) values bubble up to the right of the array
d) It requires a separate array for sorting

43) In the Bubble Sort algorithm, how many elements are guaranteed to be in their correct position after the first pass?
a) All elements
b) Half of the elements
c) One element
d) Two elements

44) What is the condition used in the inner loop of the Bubble Sort algorithm to avoid exceeding array boundaries?
a) j < size
b) j < size – 1
c) j < size – i
d) j < size – i – 1

45) Why is the variable 'i' subtracted in the inner loop condition (j < size – i – 1)?
a) To speed up the sorting process
b) To exclude already sorted elements from comparisons
c) To reverse the sorting order
d) To count the number of swaps

46) What is the outer loop condition in the Bubble Sort algorithm?
a) i < size
b) i < size – 1
c) i < size + 1
d) i <= size

True/False Questions:

47) In Bubble Sort, the largest unsorted element is placed in its correct position after each pass.
48) The Bubble Sort algorithm always requires the same number of passes, regardless of the initial order of elements.
49) Bubble Sort can be modified to terminate early if the array becomes sorted before all passes are completed.
50) The inner loop in Bubble Sort compares every two adjacent elements in the unsorted portion of the array.
51) Modify the bubbleSort function in example 11 to stop early if the array becomes sorted before all passes are completed.

52) Write a function that uses the Bubble Sort algorithm to find the ***k-th*** largest element in an unsorted array.

Multiple Choice Questions:

53) When accessing elements in a 2D array, what does the first index typically refer to?

a) Columns
b) Rows
c) Depth of the array
d) Width of the array

54) What is the correct way to initialize a 2D array in C++?

a) int arr[][] = {{1,2}, {3,4}};
b) int arr[2][2] = {1,2,3,4};
c) int arr[2][2] = {{1,2}, {3,4}};
d) int arr[2,2] = {1,2,3,4};

55) What is the purpose of using nested loops with 2D arrays?

a) To sort the array
b) To access or modify values in the array
c) To create the array
d) To delete the array

True/False Questions:

56) The indices of a 2D array in C++ start from 1.

57) It's possible to have arrays with more than two dimensions in C++.

58) Using the 'const' keyword when passing a 2D array as a parameter prevents accidental modifications to the array.

59) Write a function to initialize a 2D array of size 3×3 with all elements set to 0.

60) Write a function to print the contents of a 2D array of size 3×3.

61) Write a function to find the sum of all elements in a 2D array of size 3×3.

62) Write a function to find the maximum element in a 2D array of size 3×3.

63) Write a function to transpose a 2D array of size 3×3 (swap rows with columns).

PROJECTS

Project 01: Voting System

Overview

You will create a program that simulates a simple voting system. The program will generate random votes for a set of candidates, determine the winner based on the number of votes each candidate received, and display the results.

Steps to Follow

Include Necessary Libraries

- Use the <iostream> library for input and output functions.
- Use the <string> library to handle string data types.
- Use the <cstdlib> library for random number generation.
- Use the <ctime> library to seed the random number generator with the current time.

Generate Random Votes Function

Create a function that generates random votes for the candidates. The function should take an array of integers (to store votes), the number of candidates, and the number of votes to generate as parameters.

Use a loop to generate random votes and increment the vote count for a randomly selected candidate.

Find the Winner Function

Create a function that finds the candidate with the highest number of votes. The function should take an array of strings (candidates), an array of integers (votes), the number of candidates, and two reference parameters (winner and maxVotes) to store the result. Use a loop to compare the votes and update the winner and maxVotes variables accordingly.

Print Results Function

Create a function that prints the number of votes each candidate

received and announces the winner. The function should take an array of strings (candidates), an array of integers (votes), the number of candidates, the winner, and the maxVotes as parameters. Use a loop to print the results for each candidate and display the winner.

Voting System Function

Create a function that sets up the candidates and votes arrays, generates random votes, finds the winner, and prints the results.

- Initialize an array of strings with the names of the candidates.
- Initialize an array of integers with zeros to store the votes.
- Call the function to generate random votes.
- Call the function to find the winner.
- Call the function to print the results.

Main Function

In the main function,

- Seed the random number generator with the current time.
- Prompt the user to enter the number of votes to generate.
- Call the voting system function with the number of votes.

Example Output

Welcome to the voting system!
Enter the number of votes to generate: 1000000
Generating 1000000 random votes for candidates: Alice, Bob, Charlie, Dave, Eve.
Alice received 200203 votes.
Bob received 200329 votes.
Charlie received 199176 votes.

Dave received 200157 votes.
Eve received 200135 votes.
The winner is 'Bob' with 200329 votes.

Project 02: A Simple X / O (Tic-Tac-Toe) Game

Overview

In this project, you will create a simple text-based X / O(Tic – Tac – Toe) game using the C++ programming language. This game is a two-player game where players take turns marking a spot on a 3×3 grid with either 'X' or 'O'. The goal is to get three of your marks in a row—horizontally, vertically, or diagonally—before your opponent does. If the grid is completely filled and neither player has three marks in a row, the game ends in a draw.

Learning Objectives

By the end of this project, you will learn :

- How to work with 2D arrays: You'll create and manage a 3×3 game board using a 2D array.
- Basic control structures: You'll use loops and conditional statements to control the flow of the game.
- Functions: You'll break down the game logic into smaller, manageable functions.
- User input: You'll learn how to accept and validate user input for their moves.

Step-by-Step Instructions

1) Set Up the Game Board: You'll start by creating a 3×3 grid that represents the game board. This will be done using a 2D array of characters, where each spot on the board can hold either an 'X', 'O', or a space (' ') if it's empty. Write a function ***initializeBoard()*** that will set all spots on the board to empty(' ').

2) Display the Game Board: Write a function ***printBoard()*** that will display the current state of the game board on the screen. This function should show the grid with the marks and make it easy for

players to see which spots are taken and which are still available.
3) Check for a Win: Write a function ***checkWin(char player)*** that checks whether the current player has won the game. This function will look for three marks in a row in the rows, columns, or diagonals.
4) Check for a Draw: Write a function ***checkDraw()*** that checks whether the game has ended in a draw. This happens when all spots on the board are filled, and no player has won.
5) Main Game Loop: In the ***main()*** function, you'll set up a loop that allows players to take turns making moves. The loop continues until someone wins the game or the game ends in a draw.

- After each move, the program will :
- Update the game board.
- Check if the move resulted in a win using ***checkWin()***.
- If no one has won, check if the game is a draw using ***checkDraw()***.
- Switch turns between the players 'X' and 'O'.

6) Player Input: Prompt the current player to enter the row and column where they want to place their mark. Make sure to validate the input: check if the chosen spot is within the grid and if it's empty. If the input is invalid, prompt the player to enter a valid move.
7) Ending the Game: When the game ends, either declare a winner or announce that the game is a draw, depending on the outcome.
Example Output

```
Current Board:
  |   |
--|---|--
  |   |
--|---|--
  |   |
Player X, enter your move (row and column) numbers start from zero: 1 1


Current Board:
  |   |
--|---|--
  | X |
--|---|--
  |   |
Player O, enter your move (row and column) numbers start from zero: 0 0


Current Board:
O |   |
--|---|--
  | X |
--|---|--
  |   |
Player X, enter your move (row and column) numbers start from zero: 0 0
Invalid move. Try again.


Current Board:
O |   |
--|---|--
  | X |
--|---|--
  |   |
Player X, enter your move (row and column) numbers start from zero: 0 1


Current Board:
O | X |
--|---|--
  | X |
--|---|--
  |   |
Player O, enter your move (row and column) numbers start from zero: 2 0


Current Board:
O | X |
--|---|--
  | X |
--|---|--
O |   |
Player X, enter your move (row and column) numbers start from zero: 2 1


Current Board:
O | X |
```

CHAPTER 7

C++ I/O Streams

HUSSAM GHUNAIM

projectSkeletonLearning Objectives

At the end of this chapter, you will be able to:-

- Explain what Classes and Objects are and how to differentiate between them
- Use I/O Console Objects to solve real-life problems
- Use I/O File Objects to solve real-life problems
- Formatting I/O Streams
- Practice using Low-Level I/O Functions

INTRODUCING CLASSES AND OBJECTS

Up to this point, you have mainly declared and initialized variables with primitive data types, like *int*, *double*, and others. We call these types primitive because they are capable of holding only a single

datum, such as a number or a character. This works fine for simple applications, but in real-world applications, we often need larger data structures that can hold more data and provide operations on them, that is, classes. One way to understand classes is to think of them as custom data types or 'tiny programs'. You have already used one of these classes, the ***string*** class. In Chapter 2, you used several of this class's member functions, such as ***length()***, ***find()***, and ***replace()***, among others.

Objects are instances created from classes, Figure 01. The vertical dots in this figure denote the possible presence of additional code. Classes are blueprints that define the structure and behavior of objects, including data members (we call them either variables or fields; both words are synonyms) and member functions (methods). However, a class itself is not executable code. To utilize the structure and behavior defined by the class, we instantiate the class by creating an instance of an object from it. The created object is not exactly a 'copy' of the class, but an 'instance' that follows the structure and behavior defined by the class. Once an object is instantiated, we can assign data to its fields and call its member functions.

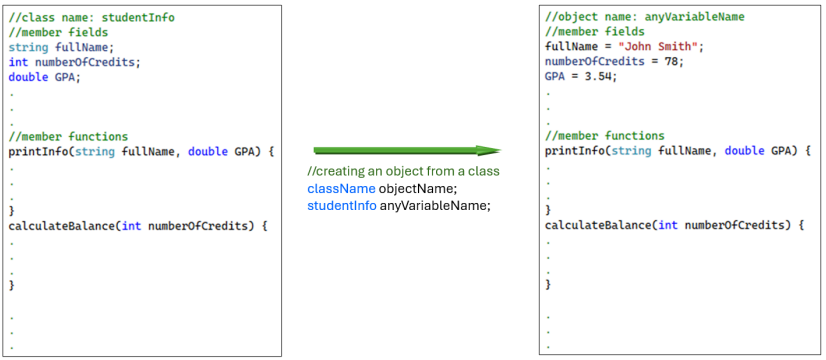


Figure 01: Creating an instance of an object from a class.

I/O CONSOLE OBJECTS

In this chapter, we will focus on stream objects. Later, you will learn how you can write your own classes. These objects are called streams because they are specifically designed to control data reading and writing. Technically speaking, computers read and write data as streams of data. As you already know, computers can read from and write to many input and output devices. Each input and output device has a specific input or output stream associated with it.

In all previous chapters, you have used the *cout* output stream that prints data to the console and *cin* input stream that reads data from the console. These are a special kind of objects because they do not need to be declared or instantiated. These are standard I/O streams that are ready to be used right away.

I/O FILE OBJECTS

Writing programs always involve working with data input and output. So far, we have managed to enter the needed data manually. But imagine if we need to enter a large amount of data every time we run our program. Further, imagine if we want to keep the output that results from running the program. For now, all outputs will disappear once the execution of a program is terminated. Files come to the rescue. Working with files is fantastic to serve both needs: to read data that programs need to function properly, and to save outputs for further use. Therefore, it is an essential skill to become comfortable working with files.

As you expect, there is a stream to read from a file and another stream to write to a file. Working with files requires two operations: declaring the stream and then instantiating the stream.

```
ifstream myInputFile; // declaring the file input object
myInputFile.open("inputFile.txt"); // instantiating the file
```

```
input object
ofstream myOutputFile; // declaring the file output object
myOutputFile.open("outputFile.txt"); // instantiating the file output object
```

In the declaration line, we use the class name, which is effectively the type name, to declare a file input stream object called myInputFile. This operation is similar to all primitive declarations that you have already done. For example, when writing *int myVar;* you are declaring a variable named *myVar* with the specified type **int**. The same thing goes for declaring the file output stream object. However, the initialization looks different. In primitive types like **int**, after declaring the variable, you can simply initialize it by assigning a value to it, like *myVar = 0;* When working with objects, we must use their member functions to initialize them.

The **open** function is a member function of both input and output streams, but of course, they are different functions. To specify which **open** function we intend to call, we use the dot . notation. In the provided code, we use the member function **open** from the input stream to attach the inputFile.txt to the input stream. Similarly, we use the member function **open** from the output stream to attach the outputFile.txt to the output stream.

Note that we use two different files for input and output to make it easy to differentiate between them; however, technically, it is possible to use the same file for input and output. C++ is a powerful language that allows you to read and write to virtually any file type. However, we will focus in this book on text files because they are the easiest type of files to work with.

Example 01

This example is about counting how many words are in the input file and saving the result in the output file. The location of files is very important. You need to specify exactly where the files are located. Of

course, both files can be located at the same location or at different locations. The problem here is that files' addresses are written differently on different systems. On Windows, the backslash **** is used; however, on Unix or Mac systems, the forward slash **/** is used. The one that can be used in C++ code is the forward slash. Therefore, if you are using a Windows system, you cannot use the backslash alone because it will confuse the C++ compiler and be interpreted as the escape character. If you still want to use the Windows file address style, you must replace every backslash with double backslashes to avoid this conflict. In the example, the two methods are used for illustration, but you can choose your preferred way of writing files' addresses.

Another solution to this problem is to put all files in the same location as the source code file and avoid adding file paths altogether. This method will be used throughout the chapter.

In the provided code, change the files' addresses corresponding to your system. Then, run the example code and explain the results.

```cpp
#include <fstream>
#include <string>

using namespace std;

int main() {
    // Create ifstream and ofstream objects
    ifstream inputFile;
    ofstream outputFile;

    // Open the files
    inputFile.open("C:\\Users\\hmghunaim\\Downloads\\ch06\\input.txt");
    outputFile.open("C:/Users/hmghunaim/Downloads/ch06/output.txt");

    // Count the words in the input file
    string word;
    int wordCount = 0;
    while (inputFile >> word) {
        ++wordCount;
    }

    // Write the word count to the output file
    outputFile << "The input file contains " << wordCount << " words.\n";

    // Close the files
    inputFile.close();
    outputFile.close();

    return 0;
}
```

First of all, when you run the code, you will notice that nothing has been printed on the console. This is expected as there is no code to do so. Even we did not include the **<iostream>** library because, in this particular example, we do not intend to use either **cout** or **cin** streams.

Second, assuming you entered your filenames along with their locations correctly, you should have the total number of words printed in the output file. When creating the input file, add a couple of words or sentences to double-check if the code works correctly.

Third, notice how the insertion operator **<<** and the extraction operator **>>** work with files' streams exactly in the same way as they work with console streams. In the **while** loop condition, every time the condition is checked, a word will be read from the input file and

saved in the ***word*** variable. If a word is successfully read, the condition returns **true**, causing the loop to continue. When reaching the end of the file, reading a word fails, causing the condition to return **false** and terminating the loop. After the loop terminates, a message will be sent to the output file along with the total number of words saved in the ***wordCount*** variable.

Finally, as a good programming practice, you must always close all open streams for many reasons, such as releasing allocated resources and avoiding data corruption.

Example 02

Run the provided code, examine both input and output files, and explain the results. Before doing that, you must create the input file 'rawData.txt' and add a few integers and decimal numbers to it, such as ***5 0 7.2 8 0 3 3.14 0 2 9.8***. If you forgot to create the output file, it will be created automatically for you.

In this example, we added code to handle errors when working with files. Errors can occur for many reasons, such as misspelling files' names, corrupted files, and so on. The *is_open*() function is a member function in both streams, the input and the output. If a file cannot be opened, the *is_open*() function will return **false**. If this occurs, the *if*-statement condition, **!inputFile.***is_open*() will be **true,** causing the message "Unable to open input file." to print out to the console. Similar logic applies to the output file as well. Note that we are using both file streams and console streams.

The *while* loop condition will perform two tasks. First, it will read numbers from the input stream and save them one by one in the ***number*** variable. The second task is to return **true** if the read operation was successful and initiate the following iteration until the condition returns **false**, thereby terminating the *while* loop. For every

iteration when the variable ***number*** gets a new value, this value is used to perform the four arithmetic operations. At the end of the loop, the variables: ***addition***, ***subtraction***, ***division***, and ***multiplication*** will have their final results.

After the ***while*** loop terminates, all results will be printed out to the console along with appropriate messages. Finally, both streams will be closed.

```cpp
#include <fstream>
#include <iostream>

using namespace std;

int main() {
    // Create ifstream and ofstream objects
    ifstream inputFile;
    ofstream outputFile;

    // Open the files
    inputFile.open("rawData.txt");
    outputFile.open("results.txt");

    // Check if the files were opened successfully
    if (!inputFile.is_open()) {
        cout << "Unable to open input file.\n";
        return 1; //it is common to return a non-zero value when errors occur.
                  //The exact value can depend on the specific error that occurred
    }

    if (!outputFile.is_open()) {
        cout << "Unable to open output file.\n";
        return 1; //it is common to return a non-zero value when errors occur.
                  //The exact value can depend on the specific error that occurred
    }

    // Read all numbers from the input file and perform calculations.
    double number, addition = 0, subtraction = 0, multiplication = 1, division = 1;
    while (inputFile >> number) {
        addition += number;
        subtraction -= number;
        if (number != 0) {
            division /= number;
            multiplication *= number;
        }
    }

    // Write the results to the output file
    outputFile << "The sum of the numbers is: " << addition << "\n";
    outputFile << "The subtraction of the numbers is: " << subtraction << "\n";
    outputFile << "The product of the numbers is: " << multiplication << "\n";
    outputFile << "The division of the numbers is: " << division << "\n";

    // Close the files
    inputFile.close();
    outputFile.close();

    cout << "Calculation operation completed successfully.\n";

    return 0;
}
```

Exercise 01

Modify example 1 to allow the code to count the number of characters instead of words.

Hint: You only need to make a tiny change!

Exercise 02

Write a program that reads a text file that contains both words and numbers, and then counts how many words and numbers are in the file. The program should meet the following specifications:

1) Both input and output files should be checked to ensure they were opened correctly. If not, an error message must be printed out to the console.

2) Use the isdigit() function found in the <cctype> library to check whether a character is a number or not.

3) The results should be printed out to an output file like this: "The input file contains ..... numbers and ..... words."

4) At the end of the program execution, print this message to the console: "File operation completed successfully!".

FORMATTING I/O STREAMS

In Chapter 2, you have learned how to use some manipulators to format the output console stream ***cout***. You can think of manipulators as special functions that can be called to manipulate the stream behavior, for example, ***setw***(), ***left***, ***right***, ***fixed***, and ***setprecision***().

Example 03

Run the code and explain the results. To do that, you need to create two text files, inFile.txt and outFile.txt. The inFile.txt must contain some unformatted data to work on, like,

```
Quiz01 Quiz02 Quiz03 Average
12.5 16 15.8 14.76666666
9 19.9 16 14.9666666
19.5 17.7 18 18.36666666
```

The code will print the unformatted data to the console and write the formatted data to the output file.

When reading from an input stream, an internal marker is used to keep track of the current reading position. This marker advances every time a successful reading operation is executed. The stream begins in a '*good*' state if opened successfully. It remains in this state until it encounters an issue, such as reaching the end of the file, at which point the *eof* flag is set. The stream's status can be checked at any time to determine if operations have been successful or if any errors have occurred.

As usual, we first must declare and instantiate the file streams, lines 10 to 16. Lines 19 to 24 print the content of the input file to the console unformatted. This is done using a *while* loop. This loop continues as long as the input stream is in a valid state by checking the condition *while (inputFile >> word)*. The loop reads the file word by word, printing each word to the console. After each word is read, we use the *peek()* function to check if the next character is a newline. If it is, we print a new line to the console, ensuring that the output maintains the line structure of the input file. This process continues until the end of the file is reached, at which point the stream enters a *fail* state, terminating the loop.

After reading the file once, the stream still has all the input, but the reading marker is positioned at the end of the stream. Therefore, we can not read it again. To read the stream again, we call *inputFile.clear()*

to reset any error flags and*inputFile.seekg(0)* to move the read position back to the beginning of the stream. The second *while* loop reads the input stream for a second time, writing the formatted output to the output file. The output file stream is set to use fixed-point notation, always show the decimal point, and use 2 decimal places for floating-point numbers, lines 31 to 33. As it reads each word, the code checks if the first character is a digit. If so, it's converted to a double using *stod()* (string to double) function and written as a number; otherwise, it's written as a string. This step is important because words are read from the input stream as strings and will not be affected by the formatting settings in lines 31 to 33. These settings only apply to numbers. All output is left-aligned in fields 15 characters wide. Newlines are preserved from the input file structure.

```cpp
#include <iostream>
#include <fstream>
#include <string>
#include <iomanip> // for using manipulators
#include <cctype> // for isdigit() and isalpha()

using namespace std;

int main() {
    // Declare input and output streams objects
    ifstream inputFile;
    ofstream outputFile;

    // Instantiate input and output streams objects
    inputFile.open("inFile.txt");
    outputFile.open("outFile.txt");

    // Messy input file contents
    string word;
        while (inputFile >> word) {
            cout << word;
            if (inputFile.peek() == '\n')
                cout << '\n';
        }

    // prepare the stream to be read again!
    inputFile.clear();
    inputFile.seekg(0);

    // Formatted file output
    outputFile.setf(ios::fixed);
    outputFile.setf(ios::showpoint);
    outputFile.precision(2);

    double number;
    while (inputFile >> word) {
        if (isdigit(word[0])) {
            number = stod(word);
            outputFile << left << setw(15) << number;
        }
        else {
            outputFile << left << setw(15) << word;
        }
        if (inputFile.peek() == '\n')
            outputFile << '\n';
    }
    inputFile.close();
    outputFile.close();
```

Example 04

In this example, we will do a similar formatting as we did in example 03 with some additions. First of all, run the code and try to explain the outcomes to familiarize yourself with the code. To run the code, you need to create an input file that has the following data without formatting.

```
First Name Last Name Score Letter Grade
John Doe 85
Jane Smith 92
Alice Johnson 78
Bob Brown 88
Carol Davis 95
David Wilson 67
Emma Taylor 74
Frank Anderson 81
Grace Thomas 90
Henry Jackson 59
```

Instead of entering files' names hard-coded, we will let users input their chosen file name, lines 56 and 57. Of course, the file name should include the file extension, which is ***.txt***. Then, we declare and initialize both the input and output streams' objects. You have already seen that whenever you write data to the output file, all preexisting contents will be overwritten. As we are using, in this example, the same file as an input and output file, we must be able to keep the preexisting content and append the new data below the older data. To do this, we use the open mode flag ***ios::app*** when initializing the output stream, line 61.

Following the best programming practices, we should not put all the code in the ***main*** function; rather, other functions must be created to handle different tasks. Therefore, we created three functions: ***Grades***, ***displayUnformattedData***, and ***writeFormattedData***. The ***Grades*** function takes a ***double*** score value and returns the

corresponding grade letter. *displayUnformattedData* function takes the input stream as a parameter passed by reference. To read from this input stream, it uses the *getline* function from the **<string>** library. *getline* function takes two parameters: the input stream and the variable used to store each line. *while* loop will keep iterating while *getline* reads lines successfully until hitting the end of the file, when the loop condition fails, terminating the loop. In each iteration, the line is printed out to the console using the *cout* stream.

The *writeFormattedData* function takes both the input and output streams as parameters passed by reference. Lines 30 to 33 print the header of the formatted table. To avoid duplication, line 36 skips reading the table header from the input file by simply reading the first line and doing nothing with it. The *while* loop condition reads the first three words in every line from the input stream and saves them into ***firstName***, ***lastName***, and ***score*** variables sequentially. ***letterGrade*** variable gets its value by calling the *Grades* function. Several manipulators are called to format the output in the way we want. *ignore* function in line 45 is called to clear any remaining characters in the line to ensure that in the next iteration, the reading will start from the second line. *ignore* takes two parameters: the maximum number of characters to be cleared and the end-of-line character. *ignore* will stop clearing the stream when hitting either of those two parameters' values. The number 1000 is an arbitrary value chosen here to be large enough to clear all characters remaining in any specific line.

```cpp
#include <iostream>
#include <fstream>
#include <string>
#include <iomanip> // for using manipulators
using namespace std;

// Function to calculate letter grade based on score
char Grades(double score) {
    if (score >= 90) return 'A';
    else if (score >= 80) return 'B';
    else if (score >= 70) return 'C';
    else if (score >= 60) return 'D';
    else return 'F';
}

// Function to display unformatted data to the console
void displayUnformattedData(ifstream& inputFile) {
    string line;
    cout << "\nThe UnFormatted Table:\n";
    while (getline(inputFile, line)) {
        cout << line << endl;
    }
}

// Function to write formatted data to the file
void writeFormattedData(ofstream& outputFile, ifstream& inputFile) {
    string line, firstName, lastName;
    double score;

    outputFile << "\n\nFormatted Table:\n";
    outputFile << "|------------|------------|-------|--------------|\n";
    outputFile << "| First Name | Last Name  | Score | Letter Grade |\n";
    outputFile << "|------------|------------|-------|--------------|\n";

    // Skip the header line
    getline(inputFile, line);

    while (inputFile >> firstName >> lastName >> score) {
        char letterGrade = Grades(score);
        outputFile << "| " << left << setw(10) << firstName << " | "
            << left << setw(10) << lastName << " | "
            << right << setw(5) << fixed << setprecision(2) << score << " | "
            << right << setw(12) << letterGrade << " |\n";
        // Clear any remaining characters on this line
        inputFile.ignore(1000, '\n');
    }
    outputFile << "|------------|------------|-------|--------------|\n";
}

int main() {
    // Declare input and output streams objects
    ifstream inputFile;
    ofstream outputFile;
    string fileName;

    cout << "Enter the file name: ";
    cin >> fileName;

    // Instantiate input and output streams objects
    inputFile.open(fileName);
    outputFile.open(fileName, ios::app); // Open in append mode

    // Display unformatted data
    displayUnformattedData(inputFile);

    // Prepare the stream to be read again
    inputFile.clear();
```

Exercise 03

Write a program that does the following:

1) Read a text file containing the following unformatted data:

```
   WeekDay     Sat  Sun Mon      Tue  Wed    Thu Fri
      Average
week01  85    92   81  79    95   86         90
week02  81          93      77   64        89  96
82
week03          71      73 87     85   94      88   79
week04 83    97               91   80 82          91   84
```

2) Write a function that prints out the unformatted data to the console. This function takes the input stream as a parameter passed by reference.

3) Write a function that formats the data in a table with boundaries using the '**|**' and '**–**' characters. Calculate the average for each week and add the values to the same table. The formatted data must be appended under the unformatted data in the same file. This function takes both the input and output streams as parameters passed by reference.

4) The user must be allowed to enter the full input file name.

5) In the formatted table, all columns must be left-aligned except the ***Average*** column, which must be right-aligned.

6) The average values must be formatted to have only two decimal places.

7) Before exiting the program, write to the console this message: "The formatted data is written to the output file, please check!".

Hints:

- There are many similarities between this exercise and example 02 that allow you to reuse some of its code.
- You can declare and instantiate either input or output streams in one line:

- *ifstream inputFile(fileName);*
- *ofstream outputFile(fileName, ios::app);*

Exercise 04

1. What are manipulators in the context of I/O streams?
2. How does the internal marker of an input stream work?
3. What happens when an input stream reaches the end of a file?
4. How can you reset an input stream to read from the beginning again?
5. What is the purpose of the peek() function when reading from a file?
6. How can you append new data to an existing file instead of overwriting it?
7. What does the ignore() function do in the context of input streams?
8. How can you format floating-point numbers to always show a specific number of decimal places?
9. What is the purpose of the stod() function?
10. How can you ensure that words read as strings are not affected by numeric formatting settings?

LOW-LEVEL I/O FUNCTIONS

The console and file streams we have discussed so far offer a powerful way to read and write data. These streams are convenient because they allow us to use extraction and insertion operators, which automate tasks such as skipping spaces and end-of-line characters. However, in some cases, we might prefer more control over the data being read or written. C++ provides lower-level

functions, such as `get()`, `put()`, and `putback()`, which allows for this level of control.

Example 05

Run the given code and explain the outputs. We use the **cin** to read from the input stream. However, **cin** is designed to stop reading when encountering the first white space. This can be convenient for some scenarios but not desirable for others. In this code, we use the*get()* method to read everything from the input stream. This includes white spaces, tabs, special characters, and even the end-of-line character '\n', which is invisible.

In this example, the code reads an entire line of text from the console and prints out the reversed text. The line can contain letters, numbers, tabs, and special characters. To test whether the code works as expected or not, use this line: "*This is a test line_ (234) !*". *Note*: The long space is created by a *tab*.

We use an array to store the line characters and later print them out in reverse. The *while* loop condition has 3 parts. The first part, *cin.get(ch)*, returns **true** as long as the stream has characters in it. The second part, *ch !='\n'*, returns **false** when reading the end-of-line character. The last part tests whether the array becomes full or not. In the *for* loop, we use the *put()* function to put characters into the output stream in a reversed order. Note that *get()* is a member function of every input stream, like **cin** and **ifstream**, and *put()* is a member function of every output stream, like **cout** and **ofstream**.

```cpp
#include <iostream>
using namespace std;

int main() {
    const int MAX_LENGTH = 1000; // Maximum length of the input line
    char input[MAX_LENGTH];
    char ch;
    int length = 0;

    cout << "Enter a line of text: ";

    // Read characters one by one using get() and store them in the array
    while (cin.get(ch) && ch != '\n' && length < MAX_LENGTH) {
        input[length++] = ch;
    }

    cout << "Reversed text: ";

    // Output characters in reverse order using put()
    for (int i = length - 1; i >= 0; --i) {
        cout.put(input[i]);
    }

    cout << endl;
    return 0;
}
```

Example 06

Run the given code and explain the outputs. To do so, create an input file containing dollar amounts and other text. For example:

> *The concert tickets cost \$50 each, and I plan to buy 4 for my friends. After spending \$75 on dinner, we'll head to the show, which features 10 opening acts. If I buy a souvenir for \$85, that will leave me with \$40 for snacks. Overall, I have saved \$400 for the event, hoping to enjoy a great night out. I'm excited to spend around \$400 on this unforgettable experience!*

This example shows how you can manipulate inputs in any way you wish. The code processes the input file "exa06.txt" and writes the processed output to the same file, appending the new content to the end.

The outer *while* loop will keep reading the input stream character by character until the end of the file is reached, when the condition *inputFile.get(ch)* returns **false**. The first *if*-statement on line 21 checks if the read character is a '\$'. If it is, the code will surround the number, including the '\$', with parentheses; otherwise, the *else* statement on line 36 puts the character as-is to the output stream using the *put()* function.

The second if statement on line 24 reads the character right after the dollar sign and checks if it is a digit. When both conditions in the *if*-statement are **true**, the *do-while* loop will read the entire number regardless of how many digits are in it. Note that the *do-while* loop does not use curly braces because it has only one statement.

After the inner *do-while* loop terminates, line 28 puts the closing parenthesis into the output stream. Line 30 checks if the end of the file has been reached, and if it has, the outer *while* loop terminates.

```cpp
#include <iostream>
#include <fstream>
#include <cctype>  // For isdigit()

using namespace std;

int main() {
    char ch;
    ifstream inputFile("exa06.txt");
    ofstream outputFile("exa06.txt", ios::app);

    // Check if files opened successfully
    if (!inputFile.is_open() || !outputFile.is_open()) {
        cout << "Error opening input file!" << endl;
        return 1;
    }

    outputFile << "\n\nThe processed text!\n";
    // Read character by character using get()
    while (inputFile.get(ch)) {
        if (ch == '$') {
            outputFile.put('('); // Put opening parentheses before the $
            outputFile.put(ch); // Write the $ sign
            if (inputFile.get(ch) && isdigit(ch)) {
                do outputFile.put(ch); // Write each digit
                while (inputFile.get(ch) && isdigit(ch));

                outputFile.put(')'); // Put closing parentheses after the number

                if (inputFile.eof()) break; // Break if it's EOF after digits

                outputFile.put(ch); // Write the next character (non-digit)

            } else outputFile.put(ch); // Write the non-digit character after $

        } else outputFile.put(ch); // Write non-$ characters normally
    }

    inputFile.close();
    outputFile.close();

    cout << "The input file has been processed correctly!" << endl;
    return 0;
}
```

Example 07

Run the given code and explain the results. When you run this code, line 7 will execute normally, asking for your first name initial. Line 8 will read your entry. However, when you are asked to enter your last name initial, you will miss the opportunity to respond, and the program terminates immediately.

What happens here is that when reading input using the extraction operator >>, it stops reading at the first space, tab, or newline it encounters. However, the input stream is not empty; it still contains the newline character '\n' as a leftover. Although invisible, this character is still there.

Thus, when we try to read the last name initial in line 11, it actually reads this leftover character from the previous operation. This causes the system to think it has already executed the line of code completely and continues running the remaining lines.

This problem can be solved in two ways:

1. Avoid reading the input stream using the ***get()*** function after reading the same stream using the extraction operator. If you rearrange the code in this example to read the first initial using the ***get()*** function and then read the second initial using the extraction operator, it will work correctly.

2. Use the ***cin.ignore(1000, '\n')*** function before reading the input stream with the ***get()*** function. This ensures that the stream is cleared of any leftover characters from previous operations, as we used in example 4, line 45.

```
1   #include <iostream>
2   using namespace std;
3
4   int main() {
5       char firstNameIntial, lastNameInitial;
6
7       cout << "Enter your first name initial: ";
8       cin >> firstNameIntial;
9
10      cout << "Enter your last name initial: ";
11      cin.get(lastNameInitial);
12
13
14      cout << "Your full name initials are: ";
15      cout << firstNameIntial << ". " << lastNameInitial << endl;
16      return 0;
17  }
```

Exercise 05

Modify the example 05 code to read and write to the same file.
Hint: All you need to do is replace console streams with file streams.

Exercise 06

In example 06, we wrote code to detect monetary numbers and add parentheses around them. The logic was checking for the \$ to distinguish the monetary numbers from everything else. In this exercise, you will use similar logic to sum all detected monetary numbers by assuming that monetary numbers must start with \$ sign. The result should be printed out to the console. Create an input text file that contains this text as an example to test your code:

> *Students Planner for 2024/2025 Academic Year*
> ————————————————————————
> *Attending college requires significant financial investment. The average tuition for a 4-year public university is around \$25,000 per year. In addition to tuition, students can expect to spend \$1,200 on textbooks and supplies each semester. Housing, food, transportation, and entertainment costs can roughly add another \$15,000 annually, depending on the student's lifestyle. Students should plan to spend 15-20 hours per week on studying and 10-15 hours per week on entertainment and fun activities. Most universities follow a 4-month semester system, with each semester divided into 16 weeks of instruction and exams.*

The challenge in this exercise is that when reading from a text file, we need a way to convert detected monetary numbers into integers. For simplicity, we will assume all monetary numbers are integers. However, if you are interested, you can find a way to work with decimal numbers as well. When a monetary number is detected, and the first digit is stored in a variable called ***ch***, performing the subtraction ***ch – '0'*** converts the character into its integer value. To clarify, remember that all characters are encoded using the ASCII

encoding system; thus, if *ch = '5'*, then *ch – '0'* equals *53 – 48 = 5*. Now, we need to give this digit its correct value by using something similar to this line:

*currentAmount = currentAmount * 10 + (ch – '0');*

Example: Converting the string "123" to the integer number 123. Suppose the code is reading the characters '1', '2', and '3' one by one from the input file from left to right. Initially, we have *currentAmount = 0*.

1) Reading '1' and saving it in ch, then performing ch – '0' = '1' – '0' = 49 – 48 = 1.
2) Run currentAmount = currentAmount * 10 + (ch – '0') which results to currentAmount = 0 * 10 + 1 = 1.
3) Reading '2' and saving it in ch, then performing ch – '0' = '2' – '0' = 50 – 48 = 2.
4) Run currentAmount = currentAmount * 10 + (ch – '0') which results to currentAmount = 1 * 10 + 2 = 12
5) Reading '3' and saving it in ch, then performing ch – '0' = '3' – '0' = 51 – 48 = 3.
6) Run currentAmount = currentAmount * 10 + (ch – '0') which results to currentAmount = 12 * 10 + 3 = 123 which is the final value.

CHARACTER FUNCTIONS

Character functions are designed to manipulate individual characters, such as *isupper*, *islower*, *isalpha*, *isdigit*, *isspace*, *tolower*, and *toupper*. Table 01 provides further details. To use these functions, you need to include the **<cctype>** library.

Table 01: Character functions imported from <cctype> library.

Function	Input Type	Output Type	Brief Description	Example Usage
isupper	char	bool	Returns true if the character is an uppercase, otherwise, returns false.	isupper('A') returns true.
islower	char	bool	Returns true if the character is a lowercase; otherwise, returns false.	islower('D') returns false.
isalpha	char	bool	Returns true if the character is a letter of the alphabet; otherwise, returns false.	isalpha('*') returns false.
isdigit	int	bool	Returns true if the character is a digit (0-9); otherwise, returns false.	isdigit('5') returns true.
isspace	char	bool	Returns true if the character is a space; otherwise, returns false.	isspace(' ') returns true.
tolower	char	char	Converts an uppercase letter to its lowercase equivalent. If the character is not an uppercase letter, it returns the character unchanged.	tolower('A') returns 'a'.
toupper	char	char	Converts a lowercase letter to its uppercase equivalent. If the character is not a lowercase letter, it returns the character unchanged.	toupper('a') returns 'A'.

Example 08

Run the given code and explain the results. To do so, create an input file containing this text or any other similar text:

> *Hello World! 123 SeCreT MeSsAgE 456. ThIs Is A lOnGeR eXaMpLe 789. NoW wE hAvE mOrE tExT tO tEsT 012. UpPeR aNd LoWeR cAsE 345. DiGiTs AnD sPaCeS 678. PuNcTuAtIoN!@# %^ &*(TeStInG 901.*

Note that in this example, we use the *fail()* stream member function to check whether the file opened correctly. In previous examples, we used the *is_open* member function, which serves a similar purpose. However, the *fail()* function can also check for errors in other operations beyond opening files, such as reading and writing errors. Additionally, we use the *exit(1)* function when an error is detected in the stream. Calling the *exit(1)* function immediately terminates the entire program. In previous examples, we simply used the *return* statement, which terminates only the current function, allowing the calling function to continue running. Conventionally, *exit(0)* indicates that the program completed its execution successfully, while any non-zero value signifies that an error has occurred.

Most of the code is straightforward and self-explanatory. The only

exception is line 21, where we use the only ternary operator in C++ that has this syntax:

condition ? expression_if_true : expression_if_false;

This usage makes the code more compact and clearer. First, the *if*-statement in line 21 checks if the character **c** is alphabetical. If this is true, we continue to run the ternary operator; otherwise, the code continues to the following *if-else* statement. In the ternary operator, we run the condition *isupper(c)*. If it returns true, we run *tolower (c)*. Otherwise, we run *toupper (c)*. Whatever lettercase is returned for the character **c** will be appended to the string variable **decoded**.

Lines 26 and 27 are used to count how many uppercase and lowercase letters the text has. They do that by calling the *back()* function that returns the last character in the variable **decoded**.

```cpp
#include <iostream>
#include <fstream>
#include <cctype>
#include <string>
using namespace std;

int main() {
    ifstream file("exa08.txt");
    if (file.fail()) {
        cout << "File error!";
        exit(1);
    }

    string message, decoded;
    getline(file, message);
    file.close();
    cout << "The Original Message: \n" << message << endl;

    int upper = 0, lower = 0, digit = 0, space = 0;
    for (char c : message) {
        if (isalpha(c)) decoded += isupper(c) ? tolower(c) : toupper(c);
        else if (isdigit(c)) { decoded += '*'; digit++; }
        else if (isspace(c)) { decoded += '_'; space++; }
        else decoded += c;

        if (isupper(decoded.back())) upper++;
        if (islower(decoded.back())) lower++;
    }
    cout << "\nThe Decoded Message: \n" << decoded << "\n\nCounts:\n";
    cout << "Uppercase: " << upper << "\nLowercase: " << lower;
    cout << "\nDigits: " << digit << "\nSpaces: " << space << endl;

    return 0;
}
```

WRAP UP

This chapter has provided a comprehensive overview of essential C++ concepts, focusing on classes, objects, and input/output operations. We began by exploring the fundamentals of object-oriented programming in C++, introducing classes as blueprints and objects as their instantiations. This foundational knowledge sets the stage for more advanced programming paradigms.

Our discussion then shifted to input and output operations, covering both console and file I/O. We examined the standard streams `cout` and `cin` for console operations, as well as

`ifstream` and `ofstream` for file handling. Emphasis was placed on the importance of proper stream management, including initialization, usage, and closure.

The chapter also delved into more advanced I/O techniques. We explored various manipulators for output formatting and low-level I/O functions for finer control over data streams. Additionally, we covered character manipulation functions, basic error handling techniques, and methods for stream manipulation. These topics are crucial for developing robust and efficient C++ programs.

By mastering these concepts, you've significantly expanded your C++ programming toolkit. These skills form the backbone of many sophisticated applications and will prove invaluable as you tackle more complex programming challenges. As with any programming skill, regular practice will be key to fully internalizing these concepts and applying them effectively in your future projects.

ANSWER KEY

Exercise 01:

The entire code will be similar to the example 01 code. The only change that you need to make is to change the declaration ***string word*** to ***char character***. This will force the ***while*** loop condition ***inputFile >> character*** to read characters instead of words.

Exercise 02:

```cpp
#include <iostream>
#include <fstream>
#include <cctype>
#include <string>

using namespace std;

int main() {
    // Create ifstream and ofstream objects
    ifstream inputFile;
    ofstream outputFile;

    // Open the files
    inputFile.open("input.txt");
    outputFile.open("output.txt");

    // Check if the files were opened successfully
    if (!inputFile.is_open()) {
        cerr << "Unable to open input file\n";
        return 1;
    }

    if (!outputFile.is_open()) {
        cerr << "Unable to open output file\n";
        return 1;
    }

    // Read numbers and strings from the input file
    string read;
    int numberCount = 0;
    int wordCount = 0;
    char ch;
    while (inputFile >> ch) {
        if (isdigit(ch)) {
            inputFile >> read;
            ++numberCount;
        }
        else {
            inputFile >> read;
            ++wordCount;
        }
    }

    // Write the counts to the output file
    outputFile << "The input file contains " << numberCount << " numbers and " << wordCount << " words.\n";

    // Close the files
    inputFile.close();
    outputFile.close();

    cout << "File operation completed successfully.\n";

    return 0;
}
```

Exercise 03:

```cpp
#include <iostream>
#include <fstream>
#include <string>
#include <iomanip> // for using manipulators
using namespace std;

// Function to display unformatted data to the console
void displayUnformattedData(ifstream& inputFile) {
    string line;
    cout << "\nThe UnFormatted Table:\n";
    while (getline(inputFile, line)) {
        cout << line << endl;
    }
}

// Function to write formatted data to the file
void writeFormattedData(ofstream& outputFile, ifstream& inputFile) {
    string line, weekNumber;
    int sat, sun, mon, tue, wed, thu, fri;
    double average;

    outputFile << "\n\nFormatted Table:\n";
    outputFile << "|---------|-----|-----|-----|-----|-----|-----|---------|\n";
    outputFile << "| WeekDay | Sat | Sun | Mon | Tue | Wed | Thu | Fri | Average |\n";
    outputFile << "|---------|-----|-----|-----|-----|-----|-----|-----|---------|\n";

    // Skip the header line
    getline(inputFile, line);

    while (inputFile >> weekNumber >> sat >> sun >> mon >> tue >> wed >> thu >> fri) {
        average = static_cast<double>(sat + sun + mon + tue + wed + thu + fri) / 7;
        outputFile << "| " << left << setw(7) << weekNumber
                   << " | " << left << setw(3) << sat
                   << " | " << left << setw(3) << sun
                   << " | " << left << setw(3) << mon
                   << " | " << left << setw(3) << tue
                   << " | " << left << setw(3) << wed
                   << " | " << left << setw(3) << thu
                   << " | " << left << setw(3) << fri
                   << " | " << right << setw(7) << fixed << setprecision(2) << average << " |\n";

        // Clear any remaining characters on this line
        inputFile.ignore(1000, '\n');
    }
    outputFile << "|---------|-----|-----|-----|-----|-----|-----|-----|---------|\n";
}

int main() {
    string fileName;
    cout << "Enter the file name: ";
    cin >> fileName;

    // Declare and instantiate input and output streams objects
    ifstream inputFile(fileName);
    ofstream outputFile(fileName, ios::app);

    if (!inputFile.is_open() || !outputFile.is_open()) {
        cout << "Error opening file for reading!" << endl;
        return -1;
    }

    // Display unformatted data
    displayUnformattedData(inputFile);

    // Prepare the stream to be read again
    inputFile.clear();
    inputFile.seekg(0);

    // Write formatted data
    writeFormattedData(outputFile, inputFile);

    inputFile.close();
    outputFile.close();

    cout << "\nThe formatted data is written to the output file, please check!" << endl;
    return 0;
}
```

Exercise 04:

1. Q: What are manipulators in the context of I/O streams?

A: Manipulators are special functions that can be called to manipulate stream behavior, such as setw(), left, right, fixed, and setprecision().

2. Q: How does the internal marker of an input stream work?

A: The internal marker keeps track of the current reading position in the stream. It advances every time a successful reading operation is executed.

3. Q: What happens when an input stream reaches the end of a file?

A: When an input stream reaches the end of a file, the ***eof*** flag is set, and the stream enters a fail state.

4. Q: How can you reset an input stream to read from the beginning again?

A: You can reset an input stream by calling inputFile.clear() to reset any error flags and inputFile.seekg(0) to move the read position back to the beginning of the stream.

5. Q: What is the purpose of the peek() function when reading from a file?

A: The peek() function is used to check the next character in the stream without advancing the read position. In the example, it's used to check for newline characters to maintain the line structure of the input file.

6. Q: How can you append new data to an existing file instead of overwriting it?

A: You can use the open mode flag ios::app when initializing the output stream to append new data to the end of an existing file.

7. Q: What does the ignore() function do in the context of input streams?

A: The ignore() function clears any remaining characters in the current line of the input stream. It takes two parameters: the maximum number of characters to clear and the end-of-line character.

8. Q: How can you format floating-point numbers to always show a specific number of decimal places?
A: You can use the fixed manipulator to use fixed-point notation and the setprecision() manipulator to set the number of decimal places. For example: outFile << fixed << setprecision(2);

9. Q: What is the purpose of the stod() function?
A: The stod() function converts a string to a double. It's used when reading numeric data from a file as strings and converting it to actual numeric types.

10. Q: How can you ensure that words read as strings are not affected by numeric formatting settings?
A: Numeric formatting settings only apply to numbers. Words read as strings need to be explicitly converted to numbers (using functions like stod()) before they will be affected by numeric formatting.

Exercise 05:

```cpp
#include <iostream>
#include <fstream>

using namespace std;

int main() {
    const int MAX_LENGTH = 1000; // Maximum length of the input line
    char input[MAX_LENGTH];
    char ch;
    int length = 0;

    ifstream inputFile("exe05.txt");
    ofstream outputFile("exe05.txt", ios::app);

    if (!inputFile || !outputFile) {
        cerr << "Error opening file!" << endl;
        return 1;
    }

    // Read characters one by one using get() and store them in the array
    while (inputFile.get(ch) && ch != '\n' && length < MAX_LENGTH) {
        input[length++] = ch;
    }

    // Output characters in reverse order using put()
    for (int i = length - 1; i >= 0; --i) {
        outputFile.put(input[i]);
    }

    inputFile.close();
    outputFile.close();

    cout << "Reversed text has been written to output.txt" << endl;

    return 0;
}
```

Exercise 06:

```cpp
#include <iostream>
#include <fstream>
#include <cctype>  // For isdigit()
using namespace std;

int main() {
    char ch;
    int currentAmount = 0, totalAmount = 0;
    bool inDollarAmount = false;
    ifstream inputFile("exe06.txt");

    // Check if the file opened successfully
    if (!inputFile.is_open()) {
        cout << "Error opening input file!" << endl;
        return 1;
    }

    // Read character by character using get()
    while (inputFile.get(ch)) {
        if (ch == '$') {
            // Start reading a dollar amount
            inDollarAmount = true;
            currentAmount = 0; // Reset current amount
        }
        // Accumulate the digit to form the full number
        else if (inDollarAmount && isdigit(ch))
            currentAmount = currentAmount * 10 + (ch - '0');

        // Ignore commas in large numbers like 1,000
        else if (inDollarAmount && ch == ',') continue;

        else if (inDollarAmount && !isdigit(ch)) {
            // When a non-digit is found, end of dollar amount
            totalAmount += currentAmount;
            inDollarAmount = false;
        }
    }
    // Ensure any last dollar amount is added
    if (inDollarAmount) totalAmount += currentAmount;

    inputFile.close();
    cout << "The total sum of dollar amounts is: $" << totalAmount << endl;
    return 0;
}
```

END OF CHAPTER EXERCISES

1) What is the main limitation of primitive data types in C++?

a) They are too complex for simple applications
b) They can hold only a single datum
c) They cannot be used in real-world applications
d) They don't have member functions

2) How can classes be understood in simple terms?
a) As custom data types or 'tiny programs'
b) As primitive data types
c) As complex data structures
d) As executable code

3) What is the relationship between a class and an object?
a) A class is an instance of an object
b) An object is a copy of a class
c) A class is a blueprint, and an object is an instance created from it
d) Objects and classes are the same thing

4) What components do classes typically include?
a) Only data members
b) Only member functions
c) Both data members and member functions
d) Neither data members nor member functions

5) What happens when an object is instantiated from a class?
a) The class becomes executable code
b) The object becomes a blueprint
c) The object becomes an exact copy of the class
d) We can assign data to the object's fields and call its member functions

6) What are stream objects specifically designed for?
a) Controlling file permissions
b) Managing memory allocation
c) Controlling data reading and writing
d) Handling network connections

7) Which of the following is NOT a standard I/O stream in C++?
a) fstream
b) cout

c) cin
d) None of the above

8) What is the correct way to declare a file input stream object?
a) ifstream myInputFile;
b) ofstream myInputFile;
c) fstream myInputFile;
d) inputstream myInputFile;

9) Which operator is used to read data from an input file stream?
a) <<
b) ->
c) ::
d) >>

10) What function should be called to check if a file was opened successfully?
a) open()
b) is_open()
c) check_open()
d) file_status()

True/False Questions:

11) The cout and cin streams need to be declared and instantiated before use.

12) C++ allows reading from and writing to the same file.

13) The backslash character () can be used directly in file paths on all operating systems in C++ code.

14) The insertion operator (<<) is used for writing data to output streams.

15) The is_open() function returns true if a file cannot be opened.

16) Write a program that reads a text file and counts how many words contain vowels and how many words do not contain vowels. The program must check if the input file was opened correctly and print out the results to the console.

17) Write a C++ program to read a text file and write all words that have vowels to an output file called “wordsWithVowels.txt”, and write all words without vowels to a file called

"wordsWithoutVowels.txt". Only the input file must be checked to see if it was opened correctly. The user does not need to create the output files, as they will be created automatically by the system.

18) Write a program that reads an input file and counts how many times a user-specified word appears in the file. If the word does not exist in the file, display an appropriate message to the console. The result should be printed to the console.

19) What is the purpose of manipulators in C++ I/O streams?

a) To control the formatting of input data
b) To manipulate the internal state of the stream
c) To handle file operations
d) To read and write data to/from files

20) How does the peek() function work when reading from a file stream?

a) It reads the next character from the stream and advances the read position
b) It checks the next character without advancing the read position
c) It returns the current read position
d) It sets the read position to the beginning of the file

21) What happens when an input stream reaches the end of a file?

a) The *eof* flag is set
b) An exception is thrown
c) The stream is automatically closed
d) The stream remains in a good state

22) How can you append data to an existing file instead of overwriting it?

(a) Use the ios::binary flag when opening the output file
(b) Use the ios::out flag when opening the output file
(c) Use the ios::in flag when opening the output file
(d) Use the ios::app flag when opening the output file

23) What is the purpose of the stod() function?

a) To read a line from a file

b) To convert a double to a string
c) To convert a string to a double
d) To write a line to a file

True/False:

24)After reading the file once, the stream can be read again without resetting the reading marker.

25) The ignore function is used to clear any remaining characters in the line to ensure the next iteration starts from the second line.

26) ***left***, ***right***, and precision manipulators can only be used to format cout stream.

27) We must import the <iostream> library to be able to use the getline() function.

28) To prepare the input stream to be read for a second time, we can use the clear() and seekg(0) member functions.

29) Write a C++ program that reads a list of integers from a file named "numbers.txt" and outputs the integers to the console, ensuring that each integer is right-aligned in a field of width 10. Use the `setw()` manipulator for formatting. Include code to handle any potential errors. Use this suggested input data to test your program:

5
12
123
4567
890
23456
789012
3
999999
42

30) Create a C++ program that reads from a file named "data.txt" and prints the contents to both the console and a new file named "output.txt." Include code to handle any potential errors.

31) Develop a C++ program that reads a file "grades.txt"

containing floating-point numbers. The program should compute the average of these numbers and write the result to a file “average.txt”, formatting the output to show exactly two decimal places. Include code to handle any potential errors. Use this suggested input data to test your program:
85
90.0
76.3
88.44
92.7
78
95.555
81.2
65.1
99.99
70

32) Write a C++ program that reads a file “students.txt” containing student names and scores. The program should output a formatted table to the console, showing names left-aligned in a field of width 20 and scores right-aligned in a field of width 10. Scores should have only one decimal place. Each column should have a header. Include code to handle any potential errors. Use this suggested input data to test your program:
Alice 89
Bob 76.25
Charlie 92.5
Diana 85
Ethan 78.75
Fiona 95.0
George 67.5
Hannah 88.1
Ian 79.0
Jasmine 91.333

33) Implement a C++ program that processes a file named

"weekly_data.txt" containing weekly scores. The program should output the unformatted data to the console, then reformat the data into a structured table with boundaries (using '|' and '-') and append it to the same file. The table should include a third column titled "Status." If the score is greater than 100, the status will be "verified"; otherwise, it will be "unverified." Scores must be formatted to two decimal places and aligned to the right, while the weekly data should also be right-aligned. The Status column should be aligned to the left. Include code to handle any potential errors. Use this suggested input data to test your program:

week01 85.05
week02 92
week03 78.25
week04 900
week05 88.75
week06 761
week07 84.3333
week08 901.1
week09 89.999
week10 182

34) Which of the following functions is used to read a single character from the input stream, including white spaces and special characters?

a) cin >>
b) cin.get()
c) cin.put()
d) cin.ignore()

35) What is the purpose of the ***put()*** function in C++?

a) To read a character from the input stream
b) To write a character to the output stream
c) To skip spaces and end-of-line characters
d) To clear the input stream

36) In Example 05, what does the ***while*** loop condition ***cin.get(ch)*** check for?

a) If the input stream is empty
b) If the character read is a newline character
c) If the stream has characters in it
d) If the array is full

37) What problem does Example 07 illustrate when using the extraction operator >>?

a) It stops reading at the first space, tab, or newline
b) It reads all characters, including white spaces
c) It clears the input stream
d) It reads characters in reverse order

38) How can the issue in Example 07 be resolved when reading input using the extraction operator >>?

a) By using the put() function
b) By using the cin.clear() function
c) By using the cin.get() function
d) By using the cin.ignore() function

True/False questions:

39) The ***cin*** function stops reading input at the first whitespace character when using the extraction operator >>.

40) The ***ignore()*** function is used to clear leftover characters from the input stream in the context of this chapter's examples.

41) Character functions like ***isupper*** and ***isdigit*** require the inclusion of the <***string***> library to be used.

42) The ***putback()*** function is used to push a character back into the input stream.

43) The condition ***inputFile*** >> ***word*** returns ***true*** until reaching the end of the input file.

44) The ***exit(1)*** function call in C++ indicates successful program termination.

45) The ***isspace()*** function returns ***true*** if the character is a space, tab, or newline.

46) The ***put()*** function is a member function of every input stream in C++.

47) Write a program using ***cin.get()*** to read a line of text, store it in

a character array, and then print it back to the console. All characters, including spaces and special characters, should be read until the end of the line.

48) Modify the code in exercise 47 to read the line from a file and print it out to the console.

49) Write a program that counts the number of vowels in a given input file. The program should print out the line number and the position in the line where each vowel is found. This is a sample of the program output. The exact output depends on your input file contents.

```
Vowel 'a' found at line 1, position 35
Vowel 'o' found at line 1, position 40
Vowel 'i' found at line 2, position 5
Vowel 'a' found at line 2, position 8
Vowel 'o' found at line 2, position 11
Vowel 'e' found at line 2, position 13
Vowel 'u' found at line 3, position 39
Vowel 'e' found at line 3, position 41
Vowel 'a' found at line 4, position 16
Vowel 'i' found at line 5, position 4
Vowel 'e' found at line 5, position 9
Vowel 'a' found at line 5, position 11
Total number of vowels: 12
```

50) Write a program to generate random words and save them to a file. The program should allow users to enter how many words they wish to generate. The generated words must have random lengths ranging from 1 to a maximum of 8 characters long.

PROJECTS

Project 01: Caesar Cipher

Write a program that encrypts and decrypts a text file's contents using the Caesar Cipher. This ancient and simple algorithm works by replacing every letter with another letter selected after shifting forward a certain number of positions. The shift value can be any integer value. For example, if we choose the shift to be 5, then the word 'hello' will become 'mjqqt' following the normal English

alphabet order. Similarly, to decipher this encrypted text, we must use the same shift value to move backward the same number of positions to retrieve the original text. This algorithm is based on the 26 English alphabet to cipher texts. Today, we can cipher all characters presented in the ASCII table. As every character in the ASCII table has an integer value, it is possible to shift this value to get a different character.

Develop a program that must meet the following requirements:

1. Use the attached project skeleton to complete the project code.
2. Check whether the input and output streams are initialized correctly.
3. Write adequate preconditions and postconditions comments for all functions.
4. Close streams appropriately.
5. Use the attached imaginary input text file to test your project code.

Note that you MUST change the extension of the projectSkeleton.pdf file to a .cpp file, and the input_Text_Project01.pdf to a .txt file.

projectSkeleton.pdf
input_Text_Project01.pdf

Project 02 (not ready yet): Text Analyzer

– Count how many times every character occurs in the input text file.
– Draw a diagram of stars representing the numbers.
– Compare the most and the least used characters in the text.

CHAPTER 8

C-Strings, Strings, and Vectors

HUSSAM GHUNAIM

Learning Objectives

At the end of reading this chapter, you will be able to :

- Implement C-strings in your programs
- Solve real-world problems using ***string*** class in your programs.
- Understand the differences between vectors and arrays.

C-STRINGS

C-strings have been used in C++ for a long time to handle strings. Recently, it has been replaced with a more advanced and flexible string class that will be discussed in the next section. Although C-strings are outdated, they are still

in common use. The reason for that is many legacy systems are built by implementing the C-strings. This means that programmers are still required to be familiar with and comfortable working with them. In some applications, C-strings might be favored for their reduced overhead and enhanced performance.

C-strings are actually arrays of characters delimited with the ***null character*** '**\0**' to determine the end of the stored string. For example, executing the code ***char variable[15] = "C++ Course";*** creates the following character array in the memory:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
C	+	+		C	o	u	r	s	e	**\0**	?	?	?	?

The null character is crucial when working with C-strings. Since it is just a character, it can be mistakenly deleted. Therefore, we must always check for the null character when reading the contents of C-string variables to know when to stop. If the null character is lost, we might continue reading through uninitialized areas of the array or, even worse, reading beyond the boundaries of the array.

It is acceptable to declare and initialize a C-string variable without specifying the desired size. The previous example can be rewritten as ***char variable[ ] = "C++ Course";*** In this syntax, the system will create an array with 11 memory locations to accommodate the string and the null character. However, this syntax ***char var[ ] = {'h', 'e', 'l', 'l', 'o'};*** will create a normal array of characters, **NOT** a C-string type, which means the null character will not be created and added to the array. Therefore, it is recommended that you avoid using this syntax.

Example 01

Run the provided code and explain the outputs.

```
1   #include <iostream>
2   using namespace std;
3
4   int main() {
5       // Declaring C-string variable.
6       char cStringVar[10] = "cString";
7
8       // Printing the C-string variable contents.
9       int index = 0;
10      while (cStringVar[index] != '\0') {
11          cout << cStringVar[index];
12          index++;
13      }
14      return 0;
15  }
```

Line 6 will create a C-string variable to store the string "cString" followed by automatically adding the null character '\0' located at index 7. The string characters are stored at indexes from 0 up to 6. When running the code, the while loop will print the cStringVar contents up to the null character when the loop terminates, avoiding reading the rest of the array that includes uninitialized array locations.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
C	S	t	r	i	n	g	**\0**	?	?

Let us now experiment with what would happen if we accidentally lose '\0'.

Experiment 1: Add the below line of code just below line 6 to replace '\0' with any other character like 'a' and run the code again. Explain what happens.

cStringVar[7] = 'a';

The exact behavior will depend on the specific compiler you are

using. Running this code on Visual Studio on a Windows system, the code still printed the string correctly, followed by the added 'a' character without any complications. This happened because the compiler's automated operation happened behind the scenes. The array will be automatically initialized to its base type at the time of declaration. This means that although the code did not explicitly initialize the memory locations at indexes [8] and [9], the system automatically initialized them for us based on the array base type, which is the null character '\0'.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
C	S	t	r	i	n	g	**a**	**\0**	**\0**

Experiment 2: Now let us examine another scenario. Change the size of the string to 8 instead of 10. In this case, the '\0' character will be the last character in the array. Therefore, when you replace the null character with any other character, the array will no longer behave as a C-string array. Run the code and explain the results.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
C	S	t	r	i	n	g	**a**

The code now will print all characters stored in the array in addition to undefined contents from outside the array boundary. To avoid reading and writing to memory locations outside of the array boundary, we can add additional protection to the while loop condition like this:

const int SIZE = 8;

while (cStringVar[index] != '\0' && (index < SIZE)) { .....

This change guarantees that the loop terminates at either reaching the null character or reaching the end of the array.

Operators' Constraints with C-Strings

One of the challenges we face when working with C-strings is that

they cannot be used directly with many common operators, such as the assignment operator (=) and comparison operators (e.g., == , < , >). This is because C-strings are essentially arrays of characters, and arrays in C++ cannot be assigned or compared using these operators by default. For example, these expressions are illegal,

```
char CString[15], anotherCString[15];
CString = "C++ Strings"; //illegal
anotherCString = "another C++ Strings"; //illegal
CString == anotherCString; //illegal
```

The second and the third lines are illegal because C-style strings are fixed-size character arrays and cannot be reassigned after declaration. You cannot use the = operator to assign a new string literal to a C-string. Similarly, C-style strings cannot be compared using the == operator. This operator only checks if the two strings are the same array, not if they contain the same characters. Therefore, the following comparison always produces **false**, although the two strings have the same contents because they are indeed two different arrays.

```
char str1[] = "string";
char str2[] = "string";
bool stringsComp = (str1 == str2);
```

To address these limitations, the **<cstring>** library provides several functions to manipulate C-strings. Table 01 lists some of these functions along with their descriptions.

Table 01: Some common functions from the <cstring> library.

Function	Description	Example
strlen(string)	Returns the number of characters in a string (not counting the \0 null terminator).	strlen(“Hello”) → 5
strcpy_s(dest, destsz, source)	Copies the string source into dest (overwrite the contents of dest).	char dest[10]; strcpy_s(dest, 10, “Hello”); → dest contains “Hello”
strncpy_s(dest, destsz, source, n)	Copies up to n characters from source to dest.	char dest[10]; strncpy_s(dest, 10, “Hello”, 2); → dest contains “He”
strcmp(string1, string2)	Compares string1 and string2. Returns 0 if they are the same, a positive number if string1 is lexicographically greater, and a negative if string1 is lexicographically less.	strcmp(“apple”, “banana”) → -1 (because “apple” is lexicographically less than “banana”)
strncmp(string1, string2, n)	Compares up to n characters of string1 and string2.	strncmp(“hello”, “helium”, 3) → 0 (because the first 3 characters “hel” are the same in both strings)
strcat_s(dest, destsz, source)	Adds the characters from the source to the end of the dest (concatenates them).	char dest[20] = “Hello”; strcat_s(dest, 20, ” World”); → dest contains “Hello World”
strncat_s(dest, destsz, source, n)	Adds up to n characters from source to the end of dest.	char dest[20] = “Hello”; strncat_s(dest, 20, ” World”, 3) → dest contains “Hello Wor”

Example02

Run the code and explain the results. The main purpose of this example is to illustrate the usage of some of the C-string functions. In lines 7 to 10, we declare and populate C-string variables. As always, we have a concern about overflowing C-string variables; it is a good idea to check their sizes using the ***strlen*** function from the **<cstring>** library, lines 13 and 14. ***strlen*** function returns the number of characters saved in a C-string variable – excluding the null character – not the declared size. That is, if a C-string variable is declared to have the size of 10 and then used to save 5 characters value, ***strlen*** will return 5.

Because the full name variable was declared to be of a specific size, we want to check if the combined first and last names will fit. If not, we must warn the user utilizing the ***if ... else*** statement in line 17. If the size of both first and last names + one space between them + the null character '\0' fit the fullName variable size, then we can run the ***else*** block. Firstly, we copy the firstName variable into the fullName using the ***strcpy_s*** function. The '_s' part denotes the safe version of this function introduced in C++ 11 in comparison with the older version. Many of the **<cstring>** functions have been modified and have the '_s" part to signify this fact. ***strcpy_s*** function takes three arguments: the destination variable, the size of the destination variable, and the source variable, respectively. Again, it is critical to check that the destination variable size is large enough to hold the source variable contents. Line 23 adds a space to separate the two names, and line 24 adds the lastName contents into the fullName variable using the ***strcat_s***. The ***cat*** part of this function name denotes the concatenate action, which means adding the second string to the end of the first string. This is in contrast to the ***strcpy_s*** function behavior, in which the contents of the source string overwrite the contents of the destination

variable. Finally, line 26 prints out the fullName contents to the console.

It is very important to remember that all of the mentioned functions must handle the null character appropriately for every operation. That is, ***strlen*** returns the size of the string, excluding the null character. Both the ***strcpy_s*** and ***strcat_s*** will ensure that the destination variable is always delimited with the null character appropriately.

```
1   #include <iostream>
2   #include <cstring>  // For C-string functions (safe versions)
3   using namespace std;
4
5   int main() {
6       // Declare C-string variables to hold the names
7       const int FIRST_NAME_SIZE = 10, LAST_NAME_SIZE = 10, FULL_NAME_SIZE = 10;
8       char firstName[FIRST_NAME_SIZE], lastName[LAST_NAME_SIZE], fullName[FULL_NAME_SIZE];
9       cout << "Enter your first name: "; cin >> firstName;
10      cout << "Enter your last name: ";  cin >> lastName;
11
12      // Check the length of the first and last names using strlen()
13      cout << "\nLength of first name: " << strlen(firstName) << endl;
14      cout << "Length of last name: " << strlen(lastName) << endl;
15
16      //Check if the total name length does not exceed the fullName size
17      if (strlen(firstName) + strlen(lastName) + 2 > FULL_NAME_SIZE) {
18      // +2 to accomodate the space between names and the null character
19          cout << "Warning: Your name is too long!" << endl;
20      }
21      else {
22          strcpy_s(fullName, FULL_NAME_SIZE, firstName);// Copy the first name safely
23          strcat_s(fullName, FULL_NAME_SIZE, " ");// Add space between first and last name
24          strcat_s(fullName, FULL_NAME_SIZE, lastName);// Add the last name
25
26          cout << "Full name: " << fullName << endl;
27      }
28      return 0;
29  }
```

Example03

To better understand how to work with C-strings, let us do a couple of tests on example 02 code. The exact behavior depends on your compiler and system. To run all the code in this book, I used Visual Studio IDE and Windows 11.

Test 1: Change the firstNameSize to 4 and enter a large first name like John, which will not leave a space for the null character. When running the code, an exception is thrown saying, “Stack around the

variable 'firstName' was corrupted," which prevents running the remaining of the code. Similar results will occur if the lastNamesize is changed in the same manner.

Test 2: Change firstNameSize, lastNameSize, and fullNameSize back to 10. This time, be sure the first and last names you entered fit within the allowed limit; however, the combined names will not fit the full name size allowance, such as John Smith. Of course, in this case, the *if*-statement will detect the problem and terminate the execution safely.

Test 3: Keep all changes in test 2 and change the *if*-condition to become ***if(strlen(firstName) + strlen(lastName) + 2 > 20)*** . We do this only to trick the *if*-condition and allow the *else* block to execute. This time, by using the same example, 'John Smith,' the *if*-statement will fail to detect the problem, and the code will continue executing the *else* block. Lines 22 and 23 will execute normally as the size of the lastName will fit both the first name and the additional space. However, line 24 will generate the exception again because it causes a memory overflow.

Exercise 01

Assume *strlen* function is not implemented in the **<cstring>** library. Implement a function called *stringLength* that counts how many characters are in a C-string, excluding the null character. The function should be able to count up to 100 characters.

Note: In your *stringLength* implementation, it is ok to return an integer number for the string length. However, the *strlen* member function returns the string length in *size_t* type, which is an unsigned integer type. The unsigned types are discussed in Chapter 4.

Exercise 02

Write a program that asks a user to enter their first and last names. Then, it extracts the first 3 characters from each name to create a username. Finally, attach the created username to the email domain '@example.com'. The program must utilize the *strcpy_s* and *strcat_s* functions.

To further increase the safety of the strcpy_s and strcat_s functions, other versions exist, *strncpy_s* and *strncat_s*, table 1. In these versions, a fourth parameter is designated to pass the integer *n*, which specifies the number of characters from the source string that must either be copied or concatenated with the destination string.

Example 04

Run the code and explain the results. In this example, we are testing how the *strncpy_s* and *strncat_s* functions can provide additional safety when working with C-Strings. The constants in lines 7 to 10 provide reasonable string sizes to run the code without issues. However, we are going to modify these sizes to test the behavior of both *strncpy_s* and *strncat_s* functions. Note that we used the *getline* member function of the input stream *cin* to read an entire line of text.

Test 1: change the constant MAX_LABEL_LENGTH size to 5 and provide a large first name like 'Smith'. When you run the code, the *strncpy_s* function at line 24 will cause the run-time error "Debug Assertion Failed!" to terminate the execution gracefully after failing to assert that the source size fits the destination size.

Test 2: keep the same MAX_LABEL_LENGTH size and enter a four-character first name like 'John'. This time the *strncpy_s* function at line 24 will run correctly, however, the function *strncat_s* at line 27 will now

complain about the source size – which is only 1 character – being larger than the destination size.

Test 3: as an exercise, make similar changes to test functions at lines 30, 33, and 36. Comment on the results.

```
1   #include <iostream>
2   #include <cstring>
3
4   using namespace std;
5
6   int main() {
7       const int MAX_FIRST_NAME_LENGTH = 10;
8       const int MAX_LAST_NAME_LENGTH = 10;
9       const int MAX_ADDRESS_LENGTH = 40;
10      const int MAX_LABEL_LENGTH = 60;
11
12      char firstName[MAX_FIRST_NAME_LENGTH];
13      char lastName[MAX_LAST_NAME_LENGTH];
14      char address[MAX_ADDRESS_LENGTH];
15      char shippingLabel[MAX_LABEL_LENGTH] = "";
16
17      cout << "Enter your first name: "; cin >> firstName;
18      cout << "Enter your last name: "; cin >> lastName;
19      cin.ignore(1000, '\n');
20
21      cout << "Enter your address: "; cin.getline(address, MAX_ADDRESS_LENGTH);
22
23      // Safely copy the first name to the shipping label
24      strncpy_s(shippingLabel, MAX_LABEL_LENGTH, firstName, strlen(firstName));
25
26      // Add a space after the first name
27      strncat_s(shippingLabel, MAX_LABEL_LENGTH, " ", 1);
28
29      // Safely append the last name to the shipping label
30      strncat_s(shippingLabel, MAX_LABEL_LENGTH, lastName, strlen(lastName));
31
32      // Add a newline after the name
33      strncat_s(shippingLabel, MAX_LABEL_LENGTH, "\n", 1);
34
35      // Safely append the address to the shipping label
36      strncat_s(shippingLabel, MAX_LABEL_LENGTH, address, MAX_ADDRESS_LENGTH - 1);
37
38      cout << "\nShipping Label:\n";
39      cout << shippingLabel << endl;
40
41      return 0;
42  }
```

Comparing C-Strings

We use the *strcmp* function to compare two C-strings. This function returns **0** if both strings are equal, a negative number if the first string is lexicographically less than the second, and a positive number if the first string is lexicographically greater.

Remember that all characters are encoded using the ASCII encoding scheme discussed in chapter 02 – figure 02.

Example 05

In this example, we will test what values are returned by the ***strcmp*** for various scenarios. For clarity, first, test the code using single-character strings. Then, you can use larger strings. Compare and explain the results.

```
  Test 1: str1 = a          str2 = b
Test 2: str1 = b          str2 = a
Test 3: str1 = a          str2 = A
Test 4: str1 = B          str2 = b
Test 5: str1 = a          str2 = a
Test 6: str1 = A          str2 = A
Test 7: str1 = abc        str2 = abc
Test 8: str1 = abca       str2 = abcb
Test 9: str1 = abcb       str2 = abca
Test 10: str1 = abca      str2 = abcA
```

```cpp
#include <iostream>
#include <cstring>

using namespace std;

int main() {
    const int MAX_STRING_LENGTH = 50;
    char str1[MAX_STRING_LENGTH];
    char str2[MAX_STRING_LENGTH];

    cout << "Enter the first string: ";
    cin.getline(str1, MAX_STRING_LENGTH);

    cout << "Enter the second string: ";
    cin.getline(str2, MAX_STRING_LENGTH);

    int result = strcmp(str1, str2);

    if (result < 0) {
        cout << "The comparison result is: " << result << endl;
        cout << "str1 is lexicographically less than str2." << endl;
    }
    else if (result > 0) {
        cout << "The comparison result is: " << result << endl;
        cout << "str1 is lexicographically greater than str2." << endl;
    }
    else {
        cout << "The comparison result is: " << result << endl;
        cout << "str1 is equal to str2." << endl;
    }

    return 0;
}
```

Another version of the *strcmp* function is *strncmp*. This variant allows you to compare strings up to a specified number of characters, rather than comparing the entire strings. Example 06 demonstrates two of the scenarios where this feature is desirable.

Example 06

In the provided code, line 16, we are only interested to know whether the entered command starts with the word ***help*** or not. This

can be part of a larger system where users can enter a large number of commands and this information can be useful to optimize the system performance. To test this functionality, you can enter something like ***help ls***, ***help /s/d***, ***clear dir***, ***hellp/ cd/***, or anything else.

Similarly, in line 28, we want to know the type of username entered, assuming there are many different types of usernames. For testing the code, you can enter something like ***admin_master, admin level 1***, ***user_1***, ***admn-user***, or anything else.

```
1   #include <iostream>
2   #include <cstring>
3
4   using namespace std;
5
6   int main() {
7       const int MAX_STRING_LENGTH = 50;
8       char str1[MAX_STRING_LENGTH];
9       char str2[MAX_STRING_LENGTH];
10
11      // Scenario 1: Prefix Matching (Command-line interface)
12      cout << "Enter a command: ";
13      cin.getline(str1, MAX_STRING_LENGTH);
14
15      // Check if the command starts with "help"
16      if (strncmp(str1, "help command_name", 4) == 0) {
17          cout << "Help command recognized." << endl;
18      }
19      else {
20          cout << "Command not recognized." << endl;
21      }
22
23      // Scenario 2: Security (Username Validation)
24      cout << "Enter username: ";
25      cin.getline(str1, MAX_STRING_LENGTH);
26
27      // Check if the username starts with "admin" (simplified example)
28      if (strncmp(str1, "admin_support_team", 5) == 0) {
29          cout << "Username starts with 'admin'." << endl;
30      }
31      else {
32          cout << "Username does not start with 'admin'." << endl;
33      }
34
35      return 0;
36  }
```

Exercise 03

Modify example 04 to create a name badge instead of the shipping label.

Requirements:

- Add a title (Mr./Ms./Mrs./Dr., etc.) before the name with a maximum length of 4 characters
- Add a job title after the name with a maximum length of 20 characters
- The name badge format should be: Title FirstName LastName – JobTitle
- Total badge length should not exceed 40 characters
- Validate that no buffer overflow occurs

Exercise 04

Write a program that captures and displays book details. The program should:

1) Prompt the user to enter the book name, author's name, publication date, and ISBN.

2) Safely copy and concatenate these details into a single string using the *strncpy_s* and *strncat_s* functions.

3) Display the combined book details in a formatted manner.

Example Output:

Enter the book name (max 49 characters):
Enter the author's name (max 49 characters):
Enter the publication date (max 49 characters):
Enter the ISBN (max 49 characters):

Book Details:

[The Very Hungry Caterpillar] | Author: Eric Carle | Publication Date: June 3, 1969 | ISBN: 978-0399226908

Exercise 05

Write a C++ program that compares two strings. The program should:

1) Prompt the user to enter two strings.
2) Ensure the maximum length for each string is 49 characters.
3) Check if the entire strings are identical using ***strcmp***. If not, find the maximum number of identical characters at the beginning of both strings using ***strncmp***.
4) Display the results to the user.

Example Output:

Enter the first string (max 49 characters): HelloWorld
Enter the second string (max 49 characters): HelloUniverse
The strings are not completely identical.
The maximum number of identical characters at the beginning of both strings is: 5

STRINGS

Strings are a fundamental part of programming in C++. They allow us to work with sequences of characters, making it easier to handle text. In this section, we will experiment with how working with strings is a lot easier and safer than working with CStrings discussed in the earlier section. In C++, the string class is defined in the **<string>** library. Thus, to use strings, you must include this library. Strings are first introduced in chapter two. We discuss this topic again here to easily compare strings' functionality with CStrings.

Example 07

Run the attached code and comment on the results. This example shows how easy to declare, initialize, and concatenate strings. You might want to compare how you did these operations with CStrings. In line 7, we declared four string variables in one line, initializing the first variable and leaving others without initialization. After reading two strings from the console in line 9, line 10 concatenates them together using the + operator. Compilers are smart enough to tell what needs to be done in this statement. If you are using the + operator between two numerical values, it will be understood that the required operation is to add the two numerical values and return their addition. However, if the + operator happens to be placed between two strings, it will be understood that the required operation is to concatenate the two strings into one string. If you try to place the + operator between a string and a numerical value, you will get an error as this operator is not defined. Line 11 shows how easy it is to print out string variables contents in any desired format.

```
1   #include <iostream>
2   #include <string>
3
4   using namespace std;
5
6   int main() {
7       string str = "Hello", firstName, lastName, fullName;
8       cout << "What is your name?\n";
9       cin >> firstName >> lastName;
10      fullName = firstName + " " + lastName;
11      cout << str << ", " << fullName << "!" << endl;
12      return 0;
13  }
```

Member Functions

The string class is a powerful class that contains many member functions. In this section, we will discuss a couple of them.

Example 08

This example code tests whether a given text can be read exactly the same in the reversed direction. For example, the text racecar can be read the same from right to left. Similarly, the sentence "No lemon, no melon." This wordplay is called Palindrome. To simplify the code, the given text must be stripped of all punctuation, including spaces. Additionally, upper-case letters need to be converted to lowercase.

The provided code has two functions: *cleanString* and *isPalindrome*. The program starts execution from line 27 in the *main* function when a user is prompted to enter a sentence and save it into the ***sentence*** variable. Then, the *isPalindrome* function is called in line 30, where it returns either **true** or **false**. The *isPalindrome* function takes the entered text as a parameter passed by reference to make the parameter accessible to all functions. The **const** modifier is used to protect this parameter from being changed unintentionally. The first line in the *isPalindrome* function calls the *cleanString* function, line18. As the name suggests, *cleanString* function job is to remove all punctuations and convert upper-case characters to lower-case. The first task can be achieved by calling the predefined *isalnum* function, line 10. This function returns **true** for only letters and numbers. The *tolower* function converts upper-case letters to lower-case. *cleanString* function uses the ***for-each*** loop in line 9 that reads the text saved in the ***str*** variable character by character and saves it in the ***c*** variable.

When the *cleanString* function returns the cleaned string to the *isPalindrome* function in line 18, a **for** loop is used to reverse it. Note how the **for** loop is written to loop backward from the end of the string to its beginning. Although the *length*() function returns values in **size_t** type, which is an unsigned integer type, we used **int** instead. The reason is in this implementation ***i,*** must become negative to terminate the loop, which is not possible with **size_t** type. Additionally, although string variables can be used as whole objects, it is allowable to access particular characters using the square brackets syntax resembling arrays' indexes, line 21. Line 23 tests whether the

actual entered text by the user *cleanStr* equals the reversed text. Thus, *isPalindrome* returns either **true** or **false**.

```cpp
#include <iostream>
#include <string>
#include <cctype> // For isalnum and tolower

using namespace std;

string cleanString(const string& str) {
    string clean = "";
    for (char c : str) {
        if (isalnum(c)) {
            clean += tolower(c);
        }
    }
    return clean;
}

bool isPalindrome(const string& str) {
    string cleanStr = cleanString(str);
    string reversed;
    for (int i = cleanStr.length() - 1; i >= 0; --i) {
        reversed += cleanStr[i];
    }
    return cleanStr == reversed;
}

int main() {
    string sentence;
    cout << "Enter a sentence: "; getline(cin, sentence);

    if (isPalindrome(sentence)) {
        cout << sentence << " is a palindrome." << endl;
    }
    else {
        cout << sentence << " is not a palindrome." << endl;
    }
    return 0;
}
```

Example 09

Run the attached code and comment on the results. In this

example, we use the *find* member function that searches a string for a specific character. If the character is found, *find* returns its integer position starting from 0. If not, *find* returns a special constant defined in the string class called ***npos***, which means 'no position'.

The code counts how many vowels are there in any given text. This is done by the *countVowels* function that takes the entered text by a user as a parameter. Then, it compares every character with a predefined list of vowels, including both upper and lower case characters, as in line 8.

The ***for-each*** loop is used to iterate over all characters stored in the ***str*** variable one by one. The ***if-statement*** condition returns **true** only if the character is found in the vowels list. Otherwise, it returns ***npos*** constant where the condition returns **false**, skipping counting that particular character.

We use the referencing operator **::** when we need to tell the compiler where to find a specific definition, thus we wrote ***string::npos*** to say that the constant ***npos*** is defined in the string class.

To clarify how the *find* function works, add the below line inside the***if-statement*** block. This line prints what *find* function returns for every found character c .

cout << "The character (" << c << ") position is " << vowels.find(c) << endl;

```cpp
#include <iostream>
#include <string>

using namespace std;

int countVowels(const string str) {
    int count = 0;
    string vowels = "aeiouAEIOU";

    for (char c : str) {
        if (vowels.find(c) != string::npos) {
            count++;
        }
    }
    return count;
}

int main() {
    string sentence;
    cout << "Enter a sentence: ";
    getline(cin, sentence);

    int vowelCount = countVowels(sentence);
    cout << "Number of vowels: " << vowelCount << endl;

    return 0;
}
```

Example 10

Run the provided code several times with different input texts, then comment on the results.

In this example, we want to create two copies of the user input text, vowelGroup and consonantGroup. The first loop modifies the ***vowelGroup*** by replacing every consonant letter with _ ignoring punctuation and numbers. The member function ***replace*** – line 17 – takes three arguments: the index of the character in the string, the number of characters to replace starting at the first argument index,

and the characters to replace the existing characters in the string. The first part of the **if** condition checks whether a given character exists in the ***vowels*** variable defined in line 10. As explained in example 09, the *find* member function returns ***npos*** constant if a character is not found. This means the condition ***vowels.find(vowelGroup.at(i))*** == ***string::npos*** returns **true** for all non-vowel characters. The second part checks whether the character is alpha to ignore punctuation and numbers. *isalpha* member function defined in the **<cctype>** library returns **true** only for upper and lower case letters.

Note in this example, we used the string class member function *at( )* to read a specific character in the string rather than using square bracket syntax. Both notations return the same result; however, using *at( )* is safer because it checks whether the provided index is a legal index or not. If not, *at( )* returns an exception. Using square brackets does not perform this check. Risking reading areas out of the string boundaries. Both notations are popular when working with strings.

The second loop works with similar logic. However, the **if** condition does not call *isalpha* because once we know that a character is a vowel, it is certainly an alpha character.

```cpp
#include <iostream>
#include <string>
#include <cctype> // for isalpha function
using namespace std;

int main() {
    string input;
    cout << "Enter a string: "; getline(cin, input);

    string vowels = "aeiouAEIOU";
    string vowelGroup = input;
    string consonantGroup = input;

    // Replace all consonants with underscores in vowelGroup
    for (size_t i = 0; i < vowelGroup.length(); ++i) {
        if (vowels.find(vowelGroup.at(i)) == string::npos && isalpha(vowelGroup.at(i))) {
            vowelGroup.replace(i, 1, "_");
        }
    }

    // Replace all vowels with underscores in consonantGroup
    for (size_t i = 0; i < consonantGroup.length(); ++i) {
        if (vowels.find(consonantGroup.at(i)) != string::npos) {
            consonantGroup.replace(i, 1, "_");
        }
    }
    cout << "Vowels only group:\t" << vowelGroup << endl << endl;
    cout << "Consonants only group:\t" << consonantGroup << endl;

    return 0;
}
```

Exercise 06

In example 08,

1) Explain how the ***for***-each loop works without specifying the boundaries of the string.

2) Explain how the *isPalindrome* function reverses the cleaned string.

3) In the *isPalindrome* function, the ***for***-loop copies the contents of the ***cleanStr*** from end to start, that is, backward:

```
for (int i = cleanStr.length() – 1; i >= 0; –i) {
        reversed += cleanStr[i];     }
```

Modify only the ***for***-loop to copy the contents of the ***cleanStr*** from start to end, that is, forward.

Exercise 07

Modify example 09 to count vowels, consonants, numbers, spaces, punctuation, and any other characters in a given text. Use this sample text to test your code:

*Hello, World! This is a test text to include all types of characters: 1) Vowels: a, e, i, o, u (both uppercase and lowercase) 2) Consonants: b, c, d, f, g, h, j, k, l, m, n, p, q, r, s, t, v, w, x, y, z (both uppercase and lowercase) 3) Numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 4) Spaces: (spaces between words) 5) Punctuation: . , ; : ! ? ‘ ” ( ) [ ] { } – _ + = / \ | @ # \$ % ^ & * ~ ` < > 6) Special Symbols: © ® ™ § ¶ 7) Control Characters: \n (newline), \t (tab), \r (carriage return) 8) Extended ASCII/Unicode:*

Exercise 08

Write a program that replaces a desired text with another. You must use *find* and *replace* member functions in your solution.

Hint: *find* has an overloaded version that lets you search starting at any desired position. *find* ***(wordToFind, position)***

Sample run 1:

```
Enter a text: I am learning C++ programming language. C++ is a very popular language.
Enter the word to be replaced: C++
Enter the word to replace with: Java
New text: I am learning Java programming language. Java is a very popular language.
```

Sample run 2:

```
Enter a text: I am learning C++ programming language. C++ is a very popular language.
Enter the word to be replaced: c++
Enter the word to replace with: Java
The word 'c++' was not found in the text.
```

String Comparison

Strings can be compared using comparison operators like **==**, **!=**, **<**, **>**, **<=**, and **>=**. This is possible because characters are encoded according to the ASCII scheme. When comparing strings, you are actually comparing their ASCII codes character by character until a difference is found or the end of the string is reached. In the

following example, we will demonstrate using **==** and **<** operators. It should be clear how to sue all other operators in the same manner.

Example 11

Run the provided code and comment on the results. This is a simple word battle game where two players are prompted to enter a single word of their choice. Then, inside the ***for***-loop, the two words will be compared, and the user who chooses a word that comes before the opponent word will be the winner. Of course, to make the game interesting, you can request that only real words be used.

```
#include <iostream>
#include <string>

using namespace std;

int main() {
    string player1Word, player2Word;
    int rounds, player1Score = 0, player2Score = 0;

    cout << "Welcome to Word Battle!" << endl;
    cout << "Enter the number of rounds: "; cin >> rounds;

    for (int i = 1; i <= rounds; ++i) {
        cout << "\nRound " << i << endl;
        cout << "Player 1, enter your word: "; cin >> player1Word;
        cout << "Player 2, enter your word: "; cin >> player2Word;

        // Compare words and award points
        if (player1Word == player2Word) cout << "\nIt's a tie! Both words are equal." << endl;
        else if (player1Word < player2Word) {
            cout << "\nPlayer 1's word place is before Player 2's word." << endl;
            player1Score++;
        } else {
            cout << "\nPlayer 2's word place is before Player 1's word." << endl;
            player2Score++;
               }
        cout << "\nCurrent Score| Player 1: " << player1Score << ", Player 2: " << player2Score << endl;
    }

    // Determine the winner
    cout << "\nFinal Score| Player 1: " << player1Score << ", Player 2: " << player2Score << endl;
    if (player1Score > player2Score)        cout << "Player 1 wins!" << endl;
    else if (player1Score < player2Score)   cout << "Player 2 wins!" << endl;
    else                                    cout << "It's a tie!" << endl;

    return 0;
}
```

Exercise 09

Implement the linear search algorithm for strings. The linear search algorithm is discussed in Chapter 6.

Sample run1:

```
Enter 5 strings:
I enjoy eating fruits.
I like bananas.
Cherry blossoms are beautiful.
I had a delicious dessert.
Elderberry syrup boosts the immune system.
Enter the string to search for: I like banana.

String is not found.
```

Sample run2:

```
Enter 5 strings:
I enjoy eating fruits.
I like bananas.
Cherry blossoms are beautiful.
I had a delicious dessert.
Elderberry syrup boosts the immune system.

Enter the string to search for: I had a delicious dessert.

String is found at index 3.
```

Converting Strings to Numbers and Vice Versa

In some scenarios, it is safer or more convenient to read numerical input as a string and then convert it to a numerical type like **int** or **double**. For example, if we ask users to enter a monetary value, some users might include symbols, such as \$45,000. Clearly, you will get an error if you try to read this input directly as an **int** or **double**. Similarly, if we ask users to enter dates, users might choose different formats, such as 4/14/2025 or 4-14-2025. The following example clarifies how we can handle such scenarios.

Example 12

Run the attached code and comment on the results. The code will first ask users to enter a monetary value. Because we do not know how the value will be entered, the entered value is initially saved as a string. Then, the **for** loop will remove all non-digit characters from the entered value. Note how the *erase* member function of the string class is used to achieve this goal. Lastly, the *stod* is used to convert the string to **double** and makes it ready to be included in any desired calculations. *stod* function throws an exception if its argument contains non-digit characters. There are other similar functions that convert strings to numerical types like *stoi* (string to integer) , *stof* (string to float), and others.

The second part of the code is about handling dates. To avoid confusion caused by which delimiter users might use, like **/** or **** or **–** , the code will focus on reading the numerical values for month, day, and year, ignoring any other non-digit characters. **for**-each loop is used to iterate over all characters stored in the string **input**. Each character is tested, whether being a digit or not, utilizing the *isdigit* function. Note how the **part** variable is utilized to separate month, day, and year values. The **+=** operator is overloaded to work with both strings and characters by concatenating the character at the left side to the end of the string at the right side. If month, day, and year strings need to be further processed to check if they fit within a plausible range, they must be converted to integers first. See exercise 10.

```cpp
#include <iostream>
#include <string>
using namespace std;

int main() {
    // Example for monetary value
    string moneyInput;
    cout << "Enter a monetary value (e.g., $39,999.99): ";
    getline(cin, moneyInput);

    // Remove any non-digit characters
    for (int i = 0; i < moneyInput.length(); ++i) {
        if (!isdigit(moneyInput[i])) {
            moneyInput.erase(i, 1);
            --i; // Adjust index after erasing
        }
    }

    // Convert string to double
    double moneyValue = stod(moneyInput);
    cout << "Monetary value: " << moneyValue << endl;

    // Example for date
    string input;
    cout << "Enter a date (e.g., 4/14/2025 or 4-14-2025): ";
    getline(cin, input);

    string month, day, year;
    int part = 0; // 0 for month, 1 for day, 2 for year

    for (char c : input) {
            if (isdigit(c)) {
                if (part == 0) month += c;
                else if (part == 1) day += c;
                else if (part == 2) year += c;
            }
            else part++;
        }
    cout << "Month: " << month << endl;
    cout << "Day: " << day << endl;
    cout << "Year: " << year << endl;
    return 0;
}
```

Exercise 10

Modify example 12 to check the entered date fits within a plausible range like months from 1 to 12, days from 1 to 31 and year 1900 to 2100 (or any similar range).

On the other hand, there are some scenarios where we want to convert strings to numbers. For example, to format numbers in a certain way or to prevent overflow exceptions. All numerical types have specific range limits to what numbers they can save, such as **int** or **double**; review Table 1 in Chapter 2. If a user enters numbers outside of the expected range, an overflow exception might be thrown, breaking the execution of the code. To avoid this problem, it is useful to first read numbers as strings and then test them before converting them into a numerical type. In example 13, we will examine the formatting scenario where converting numbers to string type can be desirable.

Example 13

Run the given code and test it with these numbers: 100999.87888, 123888456, -129900.654, etc. Comment on the results.

The provided code formates the entered numbers by adding a comma after every three digits, rounding decimal numbers to only two decimal places, and adding a \$ sign. The code works by first asking a user to enter any number to format; then, it calls the *formatNumber* function. The first thing this function does is round numbers to two decimal places, line 34. If a user enters an integer, this line adds two zeros decimal places to format all numbers consistently. To simplify the code, line 35 converts negative numbers to positive numbers using the absolute value *fabs* function from **<cmath>** library. The*to_string* from *<string>* library converts numbers to strings.*to_string*

function returns numbers with six decimal places. For example, if you entered 123 or 123.45, *to_string* returns 123.000000 or 123.450000. Therefore, the rounded number stored in the ***number*** variable will have four extra 0s added to it. Line 36 trims those extra four 0s utilizing *subStr* member function of the string class. This function returns part of a string by specifying the start and the end of the returned sub-string.

Now, we are ready to add commas to make reading large numbers easier. The *insertCommas* function called in line 38 takes the ***numStr*** formatted so far as an argument. To add commas at the correct places, a *for*-loop is used to iterate through the entire ***numStr*** string in reverse order. The idea here is to read characters in the ***numStr*** from right to left and to save them into a new string variable ***formattedNumber*** using the *insert* member function of the string class. This function takes two arguments: the position where we want to insert a string and the contents of this string. In line 15, we read characters one by one, starting at the end of the ***numStr*** and adding them at the beginning of the ***formattedNumber*** variable. Thus, although we are reading the ***numStr*** reversely, the ***formattedNumber*** will be in the correct order at the end of the *for*-loop.

To accomplish our goal, we have to add commas after detecting the decimal point and reading each digit from right to left in sets of three. To understand how the code works, let's trace through an example, like 1234567.89. At the very first iteration of the *for*-loop, the *if*-statement returns **false**. Then, the *else if* part returns **false** as well, allowing moving forward and reading the second character, which will have the same result. The third read character will be detected as the decimal point character by the *if*-statement setting ***pastDecimal*** to **true** and ***count*** to 0. At the following iteration, the *else if* block will be selected, incrementing the ***count*** variable and testing whether its value equals 3, which is not at this point. After two more iterations and when ***count*** equals 3, a comma is inserted at the beginning of

the *formattedNumber* variable. This process repeats until reading the entire *numStr* string.

```cpp
#include <iostream>
#include <string>
#include <cmath>

using namespace std;

// Function to insert commas into the number string
string insertCommas(const string numStr) {
    string formattedNumber;
    int count = 0;
    bool pastDecimal = false;

    // Iterate through the numStr in reverse
    for (int i = numStr.length() - 1; i >= 0; --i) {
        formattedNumber.insert(formattedNumber.begin(), numStr[i]);
        if (numStr[i] == '.') {
            pastDecimal = true;
            count = 0; // Reset count after the decimal point
        }
        else if (pastDecimal) {
            count++;
            // Insert a comma after every three digits before the decimal point
            if (count == 3 && i > 0) {
                formattedNumber.insert(formattedNumber.begin(), ',');
                count = 0;
            }
        }
    } // end of for loop
    return formattedNumber;
}

// Function to format a number with commas and two decimal places
string formatNumber(double number) {
    number = round(number * 100) / 100; // Round the number to two decimal places
    string numStr = to_string(fabs(number)); // Convert number to string
    numStr = numStr.substr(0, numStr.length() - 4); // trim numStr to have two decimal places

    string formattedNumber = insertCommas(numStr); // Insert commas into the formatted number
    formattedNumber.insert(formattedNumber.begin(), '$'); // Add the dollar sign

    if (number < 0) formattedNumber.insert(formattedNumber.begin(), '-');//Handle negative numbers
    return formattedNumber;
}
int main() {
    double number;
    cout << "Enter a number to format: "; cin >> number;
    cout << "The formatted number is: " << formatNumber(number) << endl;

    return 0;
}
```

Exercise 11

In example 12, change the first regular for loop into a for-each loop. Similarly, change the second for-each loop into a regular for loop.

Exercise 12

After studying example 13, write a program that formats phone numbers as (xxx) xxx – xxxx. The code must return an error message if a 10-digit number is not entered.

Converting CStrings to Strings and Vice Versa

While strings offer a more powerful and user-friendly interface for string manipulation, CStrings are still widely used. CStrings are rooted in the C/C++ programming languages, which means they are widely used in legacy codebases and systems that require compatibility with C libraries. Understanding CStrings is essential for maintaining and interfacing with such code, especially in embedded systems or performance-critical applications where C is prevalent. In certain scenarios, CStrings can be more efficient in terms of memory usage and performance. For example, CStrings do not have the overhead associated with the dynamic memory management that string employs. This can lead to faster execution in situations where string manipulation is minimal and predictable. CStrings provide a lower-level interface to string data, which can be beneficial when you need fine-grained control over memory allocation and string operations. This is particularly useful in systems programming or when working with hardware where resource constraints are a concern. Therefore, there are still valid reasons to learn how to convert strings to CStrings and vice versa.

Because strings are developed after the date of developing the CStrings, strings are designed to recognize CStrings and convert them to string type automatically. For, example,

```
char CString[] = "C-string";
string str;
str = CString; // legal statement
```

However, the opposite is not true. That is, CStrings cannot recognize strings, and hence converting them to CString type as in

these examples is illegal:
CString = str; //illegal statement
strcpy(CString, str); //illegal statement
The first line is illegal because CStrings does not recognize the = operator. The second line is illegal because the *strcpy* function only accepts two arguments of CString type.
To overcome this shortage, the string class has a member function *c_str* that converts the string object to CString type. Thus, this line becomes legal now: *strcpy (CString, str.c_str());* However, this line is still illegal: *CString = str.c_str() ;* Why?

VECTORS

We have seen previously how convenient it is to use arrays due to their ability to provide fast, direct access to elements through indexing. However, arrays have a notable limitation: fixed size. Once an array is declared with a specific size in C++, that size cannot be changed. This limitation can be restrictive, especially when the number of elements needed is not known in advance or may vary during runtime.

In response to this limitation, vectors were designed. Vectors are dynamic array-like containers from the Standard Template Library (STL). It is part of the C++ Standard Library and provides an efficient way to handle collections of data that can change in size during runtime. A vector can grow or shrink in size as needed. This makes vectors ideal for situations where the number of elements may change during execution, providing both flexibility and ease of use.

The syntax of declaring an empty vector with base type **int** is *vector<int> vectorName;* Vectors are template classes, which means they can accept and handle various types. Therefore, we must declare what type we are interested in when declaring the vector using the angel brackets syntax **<typeName>**.

After declaring the vector, we need to initialize it using one of its

member functions called ***push_back()***. This function adds elements to the end of the vector, increasing its size, for example:

```
vector<double> sample;
sample.push_back(0.0);
sample.push_back(-12.15);
sample.push_back(22.34);
```

Vector elements, once initialized, can be either read or written using a similar syntax to accessing array elements, using square brackets and indexes starting from 0. However, the square brackets syntax can **NOT** be used to initialize a vector element. This can be tricky because if you initialize a vector to ten values using the ***push_back()*** function, you can only access these initialized elements using the square brackets syntax. As we know, vectors can grow in size; however, if you try to assign values outside the initialized range, unexpected behavior can result.

Example 14

Run the given code and comment on the results. After declaring the vector called **temperatures**, the vector member function ***push_back()*** is used to initialize it. In the *for* loop, the vector elements are accessed using a syntax similar to the array syntax, **temperatures[i]**. The vector member function ***size()*** returns the number of elements currently stored in the vector, **NOT** the amount of space that has been allocated for the vector, which can be larger than the number of elements it currently holds. Note that the *for* loop variant is declared as ***size_t*** type, which is ***unsigned int*** to match the returned type by ***size()***. The type casting ***(char)248*** is used to print the degree symbol ° in the results. On my system, the ASCII code for this symbol is 248.

```cpp
#include <iostream>
#include <vector>
using namespace std;

int main() {
    vector<double> temperatures; // declaring the Vector called temperatures

    temperatures.push_back(20.5);  // Monday
    temperatures.push_back(22.0);  // Tuesday
    temperatures.push_back(19.5);  // Wednesday
    temperatures.push_back(23.5);  // Thursday
    temperatures.push_back(24.0);  // Friday
    temperatures.push_back(21.0);  // Saturday
    temperatures.push_back(18.5);  // Sunday

    // Calculate the total temperature sum
    double total = 0.0;
    for (size_t i = 0; i < temperatures.size(); i++) {
        total += temperatures[i];
        cout << "Temperature on day (" << i + 1 << ") is " << temperatures[i] << (char)248 << "C" << endl;
    }

    // Calculate average temperature
    double average = total / temperatures.size();
    cout << "Average temperature for the week: " << average << (char)248 << "C" << endl;

    return 0;
}
```

Alternate Vector Initialization

Using *push_back()* can be daunting. Therefore, other methods are available to do so. A list of elements can be assigned to a vector similar to the array syntax: *vector<int> sample = {1, 2, 3, 4, 5};* Additionally, we can call a vector constructor that takes an integer argument and initializes that number of elements to 0s: *vector<int> sample(10);* If we call the *size()* on this vector, *sample.size( )* would return 10. Now, [ ]'s can be used to assign elements 0 through 9. However, if you want to add more elements beyond index 9, you must use *push_back()* to do so.

Example 15

The attached code has 6 tests to help you understand the behavior of vectors. At the beginning, all code lines are commented out. Then, you will need to comment and uncomment portions of the code to run each test separately. Visual Studio and other similar IDEs provide buttons and keyboard shortcuts to do this easily.

Test 1: Uncomment lines 7 and 8 and run the code. *aVector.size()* function returns 0 as the size of a declared but not initialized vector.

Test 2: Comment out the previous lines and uncomment lines 11 to 14. After declaring the aVector and initializing the first 10 locations, the *aVector.size()* function returns 10 as the current size of the vector. Now, it is permissible to use the [ ] notation to read only the initialized locations. Notice that all values stored in the vector are 0s, which are the default values for the type **int**.

Test 3: Comment out the previous lines and uncomment lines 17 to 20. In this test, we set the *for* loop condition as *aVector.size()* **+ 1** to try accessing a location outside the initialized range. As you expected, an error message will pop up.

Test 4: Comment out the previous lines and uncomment lines 23 to 29. This test shows that after declaring a vector, we can use the square brackets **[ ]** notation to write to any of the initialized locations.

Test 5: Comment out the previous lines and uncomment lines 32 to 38. Similar to test 3, we try to write outside the initialized range of the vector. Of course, an error message will pop up.

Test 6: Comment out the previous lines and uncomment lines 41 to 49. In this test, we use the *if .. else* statement to check whether we are writing within the initialized range of the vector locations. If it is **true**, we can use the square brackets **[ ]** notation to add values, otherwise, we must use the *push_back()* function to do so.

```cpp
#include <iostream>
#include <vector>
using namespace std;

int main() {
    // test 1: what is the size of an initialized vector?
    //vector<int> aVector;
    //cout << "The uninitialized vector size is " << aVector.size() << endl;

    // test 2: Reading in the boundary range
    //vector<int> aVector(10); // declaring the Vector and initializing the first 10 locations
    //cout << "aVector size is " << aVector.size() << endl;
    //for (size_t i = 0; i < aVector.size(); i++)
    //    cout << "element (" << i << ") value is " << aVector[i] << endl;

    // test 3: Reading out of the boundary range
    //vector<int> aVector(10); // declaring the Vector and initializing the first 10 locations
    //cout << "aVector size is " << aVector.size() << endl;
    //for (size_t i = 0; i < aVector.size() + 1; i++)
    //    cout << "element (" << i << ") value is " << aVector[i] << endl;

    // test 4 Writing in the boundary range
    //vector<int> aVector(10); // declaring the Vector and initializing the first 10 locations
    //cout << "test 4 Writing in the boundary range\n";
    //for (size_t i = 0; i < aVector.size(); i++) {
    //    aVector[i] += i + i;
    //    cout << "element (" << i << ") value is " << aVector[i] << endl;
    //}
    //cout << "aVector size is " << aVector.size() << endl;

    // test 5 Writing out of the boundary range
    //vector<int> aVector(10); // declaring the Vector and initializing the first 10 locations
    //cout << "test 5 Writing out of the boundary range\n";
    //for (size_t i = 0; i < aVector.size() + 5; i++) {
    //    aVector[i] += i + i;
    //    cout << "element (" << i << ") value is " << aVector[i] << endl;
    //}
    //cout << "aVector size is " << aVector.size() << endl;

    // test 6: using push_back() to add more elements
    //cout << "test 6: using push_back() to add more elements: " << endl;
    //vector<int> aVector(10); // declaring the Vector and initializing the first 10 locations

    //for (size_t i = 0; i < 15; i++) {// infinite loop i < aVector.size() + 5
    //    if (i < aVector.size()) aVector[i] += i + i;
    //    else aVector.push_back(i + i);
    //    cout << "element (" << i << ") value is " << aVector[i] << endl;
    //}
    //cout << "aVector size is " << aVector.size() << endl;

    return 0;
}
```

What makes vectors powerful is the capability of growing and shrinking their size. You can do this by utilizing the vector class member functions like ***capacity()***, ***reserve()***, and ***resize()***.

Example 16

In this example, we will examine vectors' behavior as they are

designed to grow dynamically to meat programs needs. The code has 3 tests. You will need to comment out and comment on portions of the code as instructed.

Test 1: Uncomment lines 11 to 17 and run the code. From the results, we can notice that the capacity increases in a nonlinear manner. In the beginning, it grows slowly by adding a few locations. But later, it grows much faster to meet the program demands.

Test 2: Comment out the previous lines and uncomment lines 20 to 23. In the case that you know the needed vector size, you can reserve a specific amount using the reserve() member function. Using reserve() function can enhance the program performance by avoiding the excessive vector dynamic growing overhead. What happens behind the scenes is that every time the vector becomes full, a new vector is created with a larger capacity. Then, all stored data are copied from the old vector to the new vector. Lastly, the old vector is destroyed. These operations can affect the overall performance of programs.

Although executing line 20 will reserve 100 locations in the vector, line 21 shows that the current size is 0 because reserve() function does not actually initialize the reserved locations. Therefore, if line 23 is executed, an error message will pop up on the screen.

Test 3: Comment out the previous lines and uncomment lines 26 to 47. resize() is similar to size() in that it initializes the number of locations equal to the passed argument. However, resize() can set a new size to the vector either by increasing or decreasing its original size.

Running lines 26 to 31, resize the vector to 10. Then, it is ok to use the [ ] notation to add values to these locations because they are already initialized. Lines 34 to 37 resize the vector to a new size of 20 and print the contents of the entire vector. You will notice that after printing the first 10 values added in the previous step, 10 zeros are printed out as well. Those zeros are the default values added to the initialized locations in the vector after increasing its size from 10 to

20. Lines 41 to 45 replace zeros with values and then print the entire vector contents.

Test 4: Keep the commented-out lines from test 3 unchanged and uncomment lines 49 to 53. When you run these lines, an error message will pop up on the screen. The reason is that after resizing the vector down to 15, the last 5 locations were lost and cannot be accessed anymore.

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main() {
    // Create an empty vector of integers
    vector<int> numbers;

    // test 1: capacity() behavior
    //for (int i = 0; i < 200; ++i) {
    //    numbers.push_back(i); // Add i value to the vector
    //    // Print the current size and capacity of the vector
    //    cout << "Iteration " << i + 1 << ": ";
    //    cout << "\t Size = " << numbers.size() << ", ";
    //    cout << "\t Capacity = " << numbers.capacity() << endl;
    //}

    // test 2: reserve() behavior
    //numbers.reserve(100);
    //cout << numbers.size() << endl;
    //for (int i = 0; i < 100; ++i)
    //    cout << numbers[i]<< endl;

    // test 3: resize() behavior
    //numbers.resize(10);
    //cout << numbers.size() << endl;
    //for (int i = 0; i < 10; ++i) {
    //    numbers[i] = i*i;
    //    cout << numbers[i] << endl;
    //}
    //cout << "==============================\n";

    //numbers.resize(20);
    //cout << numbers.size() << endl;
    //for (int i = 0; i < 20; ++i)
    //   cout << numbers[i] << endl;

    //cout << "==============================\n";

    //for (int i = 10; i < 20; ++i)
    //    numbers[i] = i * i;
    //
    //for (int i = 0; i < 20; ++i)
    //    cout << numbers[i] << endl;

    //cout << "==============================\n";

    //numbers.resize(15);
    //for (int i = 0; i < 20; ++i) {
    //    cout << numbers[i] << endl;
    //}
    //cout << "==============================\n";

    return 0;
}
```

Exercise 13

Write a program that has a function called ***enterExpenses*** that allows users to enter their daily expenses. From the ***main*** function, print out the entire list of entered values and calculate and print out the total and the average. As we do not know how many values the user will enter, using Vectros is the best choice. The program should be able to handle the case of an empty vector. To this end, the vector member function ***empty()*** can be helpful.

Exercise 14

Complete this table for the discussed member functions of the vector class: resize(), capacity(), reserve(), and empty().

Function	Purpose	Return Type	Key Features	Example
size()	Returns the number of elements currently in the vector.	size_t	Shows the number of elements stored. Does not change the vector contents.	vector v = {1, 2, 3}; cout << “Size: ” << v.size();

Exercise 15

Correct the syntax errors in the following statements:

1)

```
vector<int> numbers;
numbers.push_back;
```

2)

```
vector<int> numbers;
numbers.size(5);
```

3)

```
vector<int> numbers;
numbers.reserve;
```

4)

```
vector<int> numbers = {1, 2, 3};
if (numbers.isEmpty() == false) {
   cout << "Vector is not empty.";
}
```

5)

```
vector<int> numbers = {1, 2, 3};
numbers.capacity() = 10;
```

WRAP UP

This chapter provides an overview of handling C-strings, strings, and vectors. It begins by explaining C-strings, which are arrays of characters ending with a null character (`\0`). Despite being outdated, C-strings are still used in legacy systems for their performance benefits. The chapter highlights common issues with C-strings, such as the potential loss of the null character, and discusses the limitations of using operators directly with C-strings.

To address these limitations, functions from the <cstring> library, such as ***strlen***, ***strcpy_s***, and ***strcmp***, are introduced.

The chapter then transitions to discuss the advantages of using the string class. Strings are easier and safer to use compared to C-strings, with simple initialization and concatenation using the **+** operator. The string class also offers powerful member functions like ***find***, ***substr***, ***length***, ***insert***, and ***replace***, which facilitate various string manipulations.

Vectors, a dynamic array-like container from the Standard Template Library (STL), are also covered. Vectors can grow or shrink in size, making them ideal for situations where the number of elements may change during execution. The chapter explains how to initialize vectors using member functions like ***push_back***, ***size***, ***capacity***, ***reserve***, and ***resize***, and handle dynamic growth efficiently.

Finally, the chapter addresses converting between C-strings and strings, noting that strings can be converted to C-strings using the ***c_str*** member function. Practical examples and exercises are included throughout the chapter to reinforce learning and ensure safe and efficient string manipulation and dynamic array handling.

ANSWER KEY

Exercise 01:

```cpp
#include <iostream>
using namespace std;

int stringLength(char anyString[]) {
    int length = 0;
    while (anyString[length] != '\0') length++;
    return length;
}

int main() {
    char anyString[101];
    cout << "Enter a string of up to 100 characters!" << endl;
    cin.getline(anyString, 101);

    int length = stringLength(anyString);
    cout << "The length of the string is: " << length << endl;

    return 0;
}
```

Exercise 02:

```cpp
#include <iostream>
#include <cstring>

using namespace std;

int main() {
    char firstName[50];
    char lastName[50];
    char userName[10] = "";//To store first 3 characters from first, last names
    char email[50] = "";

    cout << "Enter your first name: "; cin >> firstName;
    cout << "Enter your last name: ";  cin >> lastName;

    // Index to track position in userName
    int index = 0;
    // Copy the first 3 characters of firstName into userName
    for (int i = 0; i < 3 && firstName[i] != '\0'; ++i) {
        userName[index++] = firstName[i];
    }

    // Append the first 3 characters of lastName to userName
    for (int i = 0; i < 3 && lastName[i] != '\0'; ++i) {
        userName[index++] = lastName[i];
    }

    // Null-terminate userName
    userName[index] = '\0';

    // Create the email by appending "@example.com"
    strcpy_s(email, userName);
    strcat_s(email, "@example.com");

    cout << "Generated email: " << email << endl;

    return 0;
}
```

Exercise 03

```cpp
#include <iostream>
#include <cstring>
using namespace std;

int main() {
    const int MAX_TITLE_LENGTH = 4; // For Mr./Ms./Dr.
    const int MAX_FIRST_NAME_LENGTH = 10;
    const int MAX_LAST_NAME_LENGTH = 10;
    const int MAX_JOB_TITLE_LENGTH = 20;
    const int MAX_BADGE_LENGTH = 40;

    char title[MAX_TITLE_LENGTH + 1]; // +1 for null terminator
    char firstName[MAX_FIRST_NAME_LENGTH + 1];
    char lastName[MAX_LAST_NAME_LENGTH + 1];
    char jobTitle[MAX_JOB_TITLE_LENGTH + 1];
    char badge[MAX_BADGE_LENGTH + 1] = "";

    // Get user input
    cout << "Enter title (Mr./Ms./Dr.): "; cin >> title;
    cout << "Enter first name: "; cin >> firstName;
    cout << "Enter last name: ";  cin >> lastName;

    cin.ignore(1000, '\n');  // Clear the input buffer
    cout << "Enter job title: ";
    cin.getline(jobTitle, MAX_JOB_TITLE_LENGTH);

    // Create the badge string with format: Title FirstName LastName - JobTitle
    // Copy title
    strncpy_s(badge, MAX_BADGE_LENGTH, title, strlen(title));

    // Add space after title
    strncat_s(badge, MAX_BADGE_LENGTH, " ", 1);

    // Add first name
    strncat_s(badge, MAX_BADGE_LENGTH, firstName, strlen(firstName));

    // Add space
    strncat_s(badge, MAX_BADGE_LENGTH, " ", 1);

    // Add last name
    strncat_s(badge, MAX_BADGE_LENGTH, lastName, strlen(lastName));

    // Add separator
    strncat_s(badge, MAX_BADGE_LENGTH, " - ", 3);

    // Add job title
    strncat_s(badge, MAX_BADGE_LENGTH, jobTitle, strlen(jobTitle));

    // Display the badge
    cout << "\nName Badge:\n";
    cout << "==============================\n";
    cout << badge << endl;
    cout << "==============================\n";

    // Validate inputs length
    cout << "\nBadge length: " << strlen(badge) << " characters\n";
    cout << "Maximum allowed: " << MAX_BADGE_LENGTH << " characters\n";

    return 0;
}
```

Exercise 04:

```cpp
#include <iostream>
#include <cstring> // For strncpy_s and strncat_s

using namespace std;
const int MAX_LENGTH = 50; // Maximum length for each string

int main() {
    char bookName[MAX_LENGTH];
    char author[MAX_LENGTH];
    char publicationDate[MAX_LENGTH];
    char isbn[MAX_LENGTH];

    cout << "Enter the book name (max " << MAX_LENGTH - 1 << " characters): ";
    cin.getline(bookName, MAX_LENGTH);

    cout << "Enter the author's name (max " << MAX_LENGTH - 1 << " characters): ";
    cin.getline(author, MAX_LENGTH);

    cout << "Enter the publication date (max " << MAX_LENGTH - 1 << " characters): ";
    cin.getline(publicationDate, MAX_LENGTH);

    cout << "Enter the ISBN (max " << MAX_LENGTH - 1 << " characters): ";
    cin.getline(isbn, MAX_LENGTH);

    // Safely copy and concatenate strings using strncpy_s and strncat_s
    char bookDetails[4 * MAX_LENGTH]; // Buffer to store all details together

    // Copy book name
    strncpy_s(bookDetails, 4 * MAX_LENGTH, bookName, MAX_LENGTH - 1);

    // Concatenate author
    strncat_s(bookDetails, 4 * MAX_LENGTH, " | Author: ", MAX_LENGTH - 1);
    strncat_s(bookDetails, 4 * MAX_LENGTH, author, MAX_LENGTH - 1);

    // Concatenate publication date
    strncat_s(bookDetails, 4 * MAX_LENGTH, " | Publication Date: ", MAX_LENGTH - 1);
    strncat_s(bookDetails, 4 * MAX_LENGTH, publicationDate, MAX_LENGTH - 1);

    // Concatenate ISBN
    strncat_s(bookDetails, 4 * MAX_LENGTH, " | ISBN: ", MAX_LENGTH - 1);
    strncat_s(bookDetails, 4 * MAX_LENGTH, isbn, MAX_LENGTH - 1);

    // Display the combined book details
    cout << "\nBook Details:\n" << bookDetails << endl;

    return 0;
}
```

Exercise 05:

```
#include <iostream>
#include <cstring> // For strcmp and strncmp

using namespace std;
const int MAX_LENGTH = 50;

int main() {
    char string1[MAX_LENGTH], string2[MAX_LENGTH];

    cout << "Enter the first string (max " << MAX_LENGTH - 1 << " characters): ";
    cin.getline(string1, MAX_LENGTH);

    cout << "Enter the second string (max " << MAX_LENGTH - 1 << " characters): ";
    cin.getline(string2, MAX_LENGTH);

    // Using strcmp to check if the entire strings are identical
    if (strcmp(string1, string2) == 0) {
        cout << "\nThe strings are identical.\n" << endl;
    }
    else {
        cout << "\nThe strings are not identical." << endl;
    }

    // Find the length of the shorter string
    int minLength = min(strlen(string1), strlen(string2));

    // Using strncmp to find the maximum number of identical characters
    int maxIdenticalChars = 0;
    for (int i = 1; i <= minLength; ++i) {
        if (strncmp(string1, string2, i) == 0) maxIdenticalChars = i;
        else break;
    }
    cout << "The number of identical characters at the beginning of both strings
      is: " << maxIdenticalChars << endl;

    return 0;
}
```

Exercise 06:

1) In the cleanString function, the *for*-each loop is written as for (char c : str). This loop iterates over each character in the string *str* without explicitly specifying the boundaries. The *for*-each loop automatically handles the iteration from the beginning to the end of the string. Here's how it works:

- Initialization: The loop initializes c to the first character of *str* .
- Condition: The loop continues as long as there are characters left in *str* .

- Iteration: After each iteration, the loop moves to the next character in ***str*** .

This makes the code concise and easy to read, as you don't need to manually manage the loop boundaries.

2) In the ***isPalindrome*** function, the cleaned string ***cleanStr*** is reversed using the ***for*** loop.

```
for (int i = cleanStr.length() – 1; i >= 0; –i) {
reversed += cleanStr[i];    }
```

Here's how this loop works:

- Initialization: The loop starts with ***i*** set to the last index of ***cleanStr***, which is***(cleanStr.length() – 1)***.
- Condition: The loop continues as long as ***i*** is greater than or equal to ***0***.
- Iteration: In each iteration, the character at index ***i*** of ***cleanStr*** is appended to the ***reversed*** string, and ***i*** is decremented by ***1***.

This effectively constructs the ***reversed*** string by appending characters from the end of ***cleanStr*** to the beginning.

3) This ***for*** loop copies the contents of the cleanStr from start to end, that is, forward.

```
for (int i = 0; i < cleanStr.length(); ++i) {
    reversed = cleanStr[i] + reversed;
}
```

Exercise 07:

```cpp
#include <iostream>
#include <string>
#include <cctype> // For isalpha, isdigit, isspace, ispunct
using namespace std;

int main() {
    string sentence, vowelChars = "aeiouAEIOU";
    int vowels = 0, consonants = 0, numbers = 0, spaces = 0, punctuation = 0, others = 0;
    cout << "Enter a sentence: "; getline(cin, sentence);

    for (char c : sentence) {
        if (vowelChars.find(c) != string::npos) vowels++;
        else if (isalpha(c)) consonants++;
        else if (isdigit(c)) numbers++;
        else if (isspace(c)) spaces++;
        else if (ispunct(c)) punctuation++;
        else others++;
        }

    cout << "Number of vowels: " << vowels << endl;
    cout << "Number of consonants: " << consonants << endl;
    cout << "Number of numbers: " << numbers << endl;
    cout << "Number of spaces: " << spaces << endl;
    cout << "Number of punctuation marks: " << punctuation << endl;
    cout << "Number of other characters: " << others << endl;

    return 0;
}
```

Exercise 08:

```cpp
#include <iostream>
#include <string>
using namespace std;

int main() {
    string text, wordToFind, wordToReplace;

    cout << "Enter a text: "; getline(cin, text);
    cout << "Enter the word to be replaced: "; getline(cin, wordToFind);
    cout << "Enter the word to replace with: "; getline(cin, wordToReplace);

    size_t pos = text.find(wordToFind);
    if (pos == string::npos)
        cout << "The word '" << wordToFind << "' was not found in the text." << endl;
    else {
        while (pos != string::npos) {
            text.replace(pos, wordToFind.length(), wordToReplace);
            pos = text.find(wordToFind, pos + wordToReplace.length());
        }
        cout << "New text: " << text << endl;
    }
    return 0;
}
```

Exercise 09:

```
#include <iostream>
#include <string>
using namespace std;

int linearSearch(string arr[], int n, string target) {
    for (int i = 0; i < n; ++i)
        if (arr[i] == target) return i;//Return index of found element
    return -1; // Return -1 if the element is not found
}

int main() {
    int const size = 5;
    string arr[size];

    cout << "Enter 5 strings:" << endl;
    for (int i = 0; i < size; ++i) {
        getline(cin, arr[i]);
    }

    string target;
    cout << "\nEnter the string to search for: "; getline(cin, target);
    int result = linearSearch(arr, size, target);

    if (result != -1) cout << "\nString is found at index " << result << ".\n";
    else cout << "\nString is not found." << endl;

    return 0;
}
```

Exercise 10:

Add the following lines at the end of the example 12 code:

```
// Convert strings to integers
int monthInt = stoi(month);
int dayInt = stoi(day);
int yearInt = stoi(year);

// Check if the date is within a plausible range
bool validDate = true;
if (monthInt < 1 || monthInt > 12) validDate = false;
if (dayInt < 1 || dayInt > 31) validDate = false;
if (yearInt < 1900 || yearInt > 2100) validDate = false;

if (validDate)
    cout << "Valid date: " << monthInt << "/" << dayInt << "/" << yearInt << endl;
else cout << "Invalid date entered." << endl;
```

Exercise 11:

The first for loop can be modified as:

```
string cleanedInput;
for (char c : moneyInput)
    if (isdigit(c))
        cleanedInput += c;//Add the digit to cleanedInput string

// Convert string to double
double moneyValue = stod(cleanedInput);
cout << "Monetary value: " << moneyValue << endl;
```

The second for loop can be modified as:

```
for (int i = 0; i < input.length(); ++i) {
    char c = input[i];
    if (isdigit(c)) {
        if (part == 0) month += c;
        else if (part == 1) day += c;
        else if (part == 2) year += c;
    }
    else part++;
}
```

Exercise 12:

```cpp
#include <iostream>
#include <string>
using namespace std;

// Function to format a phone number string
string formatPhoneNumber(const string numStr) {
    string formattedNumber;

    // Ensure that the string is exactly 10 digits long (for a standard phone number)
    if (numStr.length() != 10) return "Invalid phone number length.";

    // Format the string in the form of (XXX) XXX-XXXX
    formattedNumber += '(';
    formattedNumber += numStr.substr(0, 3);  // Area code (first 3 digits)
    formattedNumber += ") ";
    formattedNumber += numStr.substr(3, 3);  // First part of the number (next 3 digits)
    formattedNumber += "-";
    formattedNumber += numStr.substr(6, 4);  // Second part of the number (last 4 digits)

    return formattedNumber;
}

int main() {
    string phoneNumber;
    cout << "Enter a 10-digit phone number: "; cin >> phoneNumber;
    cout << "The formatted phone number is: " << formatPhoneNumber(phoneNumber) << endl;

    return 0;
}
```

Exercise 13:

```cpp
#include <iostream>
#include <vector>
using namespace std;

// Function to allow users to enter expenses
void enterExpenses(vector<double>& expenses) {
    double expense;
    cout << "Enter your expenses (type -1 to stop):" << endl;

    while (true) {
        cin >> expense;
        if (expense == -1) break; // Stop input when the user enters -1
        expenses.push_back(expense); // Add expenses to the vector
    }
}

int main() {
    vector<double> expenses;

    // Step 1: Allow users to input expenses
    enterExpenses(expenses);

    // Step 2: Calculate the total expenditure
    double total = 0.0;
    for (size_t i = 0; i < expenses.size(); i++) {
        total += expenses[i];
        cout << "Expense on day (" << i + 1 << ") is $" << expenses[i] << endl;
    }

    // Step 3: Calculate the average expenditure
    double average;
    if (expenses.empty()) average = 0.0;
    else average = total / expenses.size();
    cout << "\nTotal expenditure: $" << total << endl;
    cout << "Average daily expenditure: $" << average << endl;

    return 0;
}
```

Exercise 14:

Function	Purpose	Return Type	Key Features	Example
size()	Returns the number of elements currently in the vector.	size_t	Shows the number of elements stored. Does not change the vector contents.	vector<int> v = {1, 2, 3}; cout << "Size: " << v.size(); // Output: Size: 3

resize()	Changes the size of the vector by adding or removing memory locations.	void	Expands or shrinks the vector. Newly added memory locations are initialized to a default value.	vector<int> v = {1, 2, 3};
				v.resize(5);
				cout << v.size(); // Output: 5
				// v = {1, 2, 3, 0, 0}

capacity()	Returns the total capacity (allocated memory) of the vector.	size_t	Shows how much memory has been allocated, which may be larger than size().	vector<int> v;
				v.reserve(10);
				cout << v.capacity(); // Output: 10

reserve()	Allocates memory for the specified number of elements.	void	Pre-allocates memory to avoid frequent reallocations during ***push_back()*** operations.	vector<int> v;
				v.reserve(10);
				cout << v.capacity(); // Output: 10

empty()	Checks if the vector contains any elements.	bool	Returns true if the vector has no elements, otherwise returns false.	vector<int> v;
				cout << v.empty(); // Output: true
				v.push_back(1);
				cout << v.empty(); // Output: false

Exercise 15:

1)

```
vector<int> numbers;
numbers.push_back(10); // Correct: Adds the value 10 to the vector
```

2)

```
vector<int> numbers;
numbers.resize(5); // Correct: Sets the size of the vector to 5
```

3)

```
vector<int> numbers;
numbers.reserve(10); // Correct: Preallocates memory for 10 elements
```

4)

```
vector<int> numbers = {1, 2, 3};
if (numbers.empty() == false) { // Correct: empty() is a member function
cout << “Vector is not empty.”;
}
```

5)

```
vector<int> numbers = {1, 2, 3};
cout << numbers.capacity(); // Correct: capacity() returns a value
```

END OF CHAPTER EXERCISES

Multiple Choice Questions

1. What character is used to terminate a C-string?
   A) \n
   B) \0
   C) \t
   D) \r
2. Which function is used to copy a C-string safely?
   A) strcpy_s
   B) strncpy
   C) strcpy
   D) strncpy_s
   E) B and C
   F) A and D
3. What does the strlen function return?
   A) The size of the array
   B) The number of characters in a string, excluding the null terminator
   C) The number of characters in a string, including the null terminator
   D) The memory address of the string
4. Which of the following is illegal in C++ when working with C-strings?
   A) char str[10] = “Hello”;
   B) strcpy_s(dest, 10, “Hello”);
   C) str1 == str2;
   D) strlen(“Hello”);
5. What will happen if the null character in a C-string is lost?
   A) The program will crash immediately
   B) The string will be read correctly
   C) The program might read beyond the array boundaries
   D) The string will be truncated
6. Which function concatenates two C-strings safely?
   A) strcat
   B) strcat_s
   C) strncat

D) strncat_s
E) A and C
F) B and D

7. What does the strcmp function return if two strings are equal?
A) 1
B) -1
C) The length of the strings
D) 0

8. Which function is used to compare up to a specified number of characters in two C-strings?
A) strcmp
B) strncmp
C) strcpy
D) strncpy

9. What is the purpose of the cin.getline function in C++?
A) To read a single character from the input
B) To read a single word
C) To copy a string
D) To read a line of text from the input

10. Which library must be included to use the string class in C++?
A) <iostream>
B) <string>
C) <cstring>
D) <cctype>

11. What operator is used to concatenate two strings in C++?
A) +
B) *
C) &&
D) ||

12. Which function is used to check if a character is alphanumeric?
A) isalpha
B) isdigit
C) isalnum
D) isprint

13. In example 08, what does the *cleanString* function do in the provided code?
    A) Removes all spaces from a string
    B) Converts all characters to uppercase
    C) Removes all punctuation and converts uppercase letters to lowercase
    D) Reverses the string
14. Which function is used to find the position of a character in a string?
    A) index
    B) search
    C) locate
    D) find
15. Which function is used to replace a character in a string?
    A) change
    B) substitute
    C) replace
    D) swap
16. Which function converts a string to a double?
    A) stol
    B) stoi
    C) stof
    D) stod
17. In example 13, what is the output of the *formatNumber* function for the input 1234567.89?
    A) 1,234,567.89
    B) \$1,234,567.89
    C) 1234567.89
    D) \$1234567.89
18. What is the main advantage of using vectors over arrays in C++?
    A) Faster access to elements
    B) Fixed size
    C) Dynamic size

D) Easier syntax

19. Which function is used to add elements to the end of a vector?
    A) push_back()
    B) add()
    C) insert()
    D) append()
20. What does the size() function of a vector return?
    A) The total capacity of the vector
    B) The number of elements currently stored in the vector
    C) The maximum size the vector can grow to
    D) The amount of memory allocated for the vector
21. Which function can be used to reserve a specific amount of space in a vector?
    A) resize()
    B) reserve()
    C) allocate()
    D) set_size()
22. What happens when you try to access a vector element outside its initialized range?
    A) The program crashes immediately
    B) The element is automatically initialized
    C) Unexpected behavior can result
    D) The vector resizes to accommodate the new element
23. Which function is used to check if a vector is empty?
    A) isEmpty()
    B) clear()
    C) size()
    D) empty()
24. What is the default value for elements in a vector declared with int as the base type and initialized with a specific size?
    A) 1
    B) -1
    C) NULL
    D) 0

25. Which function is used to change the size of a vector?
    A) resize()
    B) set_size()
    C) change_size()
    D) adjust_size()
26. What does the capacity() function of a vector return?
    A) The number of elements currently stored in the vector
    B) The total capacity of the vector
    C) The maximum size the vector can grow to
    D) The amount of memory allocated for the vector
27. Which function can be used to add elements to a vector outside of the initialized range?
    A) push_back()
    B) insert()
    C) add()
    D) set()

True or False Questions

28. C-strings are arrays of characters terminated by the null character \0.
29. The strcpy_s function is unsafe and should be avoided.
30. The strlen function includes the null terminator in its count.
31. C-strings can be directly compared using the == operator.
32. The strcat_s function concatenates two C-strings and ensures the result is null-terminated.
33. If a C-string is declared without specifying its size, the compiler will automatically allocate enough memory to include the null terminator.
34. The strcmp function returns a positive number if the first string is lexicographically greater than the second string.
35. The cin.getline function can be used to read only a single word into a C-string.
36. The strncpy_s function copies a specified number of characters from the source string to the destination string.
37. The ***find*** function returns ***npos*** constant if the character is not

found in the string.
38. The isPalindrome function defined in this chapter modifies the input string.
39. The ***replace*** function can replace multiple characters in a string.
40. The ***stoi*** function converts a string to an integer.
41. Vectors in C++ can dynamically change their size during runtime.
42. The push_back() function adds elements to the beginning of a vector.
43. The size() function returns the total capacity of the vector.
44. The reserve() function initializes the reserved locations in a vector.
45. Accessing a vector element outside its initialized range can result in unexpected behavior.
46. The resize() function can both increase and decrease the size of a vector.
47. The capacity() function returns the number of elements currently stored in the vector.
48. The push_back() function can be used to add elements to a vector **only** if the index is within the initialized range.

Code Writing Questions

49. Write a program that concatenates two C-strings safely.
50. Write a program that asks users to enter any two C-strings and the number of characters they want to compare in the 2 C-strings. The program must print out the comparison result along with an explanation.
51. In exercise 01, you wrote the stringLength function that counts how many characters are in a C-string, excluding the null character. The restriction was the string length could be up to 100 characters. Modify the code of this exercise to find the length of strings with arbitrary lengths.
52. Write a program that creates file paths, for example, C:\Users***userName***\Documents\file.txt. The program must ask a user to enter their user name. Then, the ***userName*** is embedded in

its right place in the file path. All strings implemented in the program must be C-Strings and hence utilize C-Strings member functions.

53. Write a simple game utilizing the string class member functions ***replace*** and ***find***. The game will have a pre-selected sentence like "The quick brown fox jumps over the lazy dog." Then one of the words like ***fox*** must be replaced with underscores _____, and the modified sentence is printed out to the console: "The quick brown _____ jumps over the lazy dog." The user will try to guess the correct word from the context of the sentence. If the guess is correct, the complete sentence will be printed out to the console.

54. Write a program that replaces a selected word with synonyms. The program will:

- Ask the user to enter a sentence and a keyword (the word the user wants to replace with synonyms).
- Try to match the keyword with a predefined set of synonyms.
- If the keyword is found, print out different versions of the sentence using the synonyms.
- If the keyword is not found, display a message saying, "Keyword not found in the sentence."

Sample run1:

```
Enter a sentence: Cats are smart animals.
Enter the keyword to replace: smtr
Keyword not found in the sentence.
```

Sample run2:

```
Enter a sentence: Cats are smart animals.
Enter the keyword to replace: smart
Modified sentence: Cats are intelligent animals.
Modified sentence: Cats are clever animals.
Modified sentence: Cats are bright animals.
```

55. Write a program that allows users to enter multiple full names and store them in a vector. The program must continuously prompt users to enter names until they type “stop.” After entering names, the program must ask users if they want to continue adding more names. If a user chooses to continue, they can enter more names; otherwise, the program will display all the names entered. Additionally, the program should print the current size and capacity of the vector after each set of names is entered. In the end, all entered names must be printed out to the console.

Sample Run:

```
Enter full names (type 'stop' to finish):

Enter a full name: John Smith
Enter a full name: Emily Johnson
Enter a full name: Michael Brown
Enter a full name: stop

Do you still want to add more names? (yes/Yes/no/No):
yes
Current size of the vector: 3
Current capacity of the vector: 3
Enter full names (type 'stop' to finish):

Enter a full name: Sarah Davis
Enter a full name: David Wilson
Enter a full name: Jessica Martinez
Enter a full name: Daniel Anderson
Enter a full name: stop

Do you still want to add more names? (yes/Yes/no/No):
Yes
Current size of the vector: 7
Current capacity of the vector: 9
Enter full names (type 'stop' to finish):

Enter a full name: Laura Thompson
Enter a full name: James Taylor
Enter a full name: Olivia Moore
Enter a full name: stop

Do you still want to add more names? (yes/Yes/no/No):
no
Current size of the vector: 10
Current capacity of the vector: 13

You have entered the following names:
John Smith
Emily Johnson
Michael Brown
Sarah Davis
David Wilson
Jessica Martinez
Daniel Anderson
Laura Thompson
James Taylor
Olivia Moore
```

56. Write a program to read large text from the console and use a vector to save the text word by word separately. The vector will expand dynamically to save all words in the text. The easiest way to split the input text into separate words is by using the ***stringstream*** class from the <sstream> library. You can create a stringstream object like this: ***stringstream objectName(input);***. The ***objectName*** can be any variable name you choose, and ***input*** is the variable that holds the entire input text. Then, you can use the statement ***objectName >> word*** to extract words from the ***objectName*** into the ***word*** variable. The program must print the list of all words extracted from the input text.

PROJECTS

Project 1: Bubble Sort Algorithm

In Chapter 6, example 11, the bubble sort algorithm was implemented using arrays. Reimplement bubble sort using vectors.

Project 2: The Advantage of Using the `reserve()` Function

Objective: Write a program to test how the ***`reserve()`*** member function can enhance the overall performance of programs.

Instructions:

1. **Program Structure**:
    - Your program must have two blocks of code:
        - One block using the ***`reserve()`*** function.
        - One block without using the ***`reserve()`*** function.
2. **Time Measurement**:

- Measure the time before and after running each block.
- Calculate the execution time for each block by subtracting the start time from the end time.

3. **Implementation**:
    - Inside each block, write a loop to add items to a vector.
    - To get reasonable results, test your code with a large number of items added to the vector, such as 1,000,000, 10,000,000, and 100,000,000.

Hints:

- In C++, you can measure the system time using the ***`now()`*** member function of the ***`high_resolution_clock`*** class from the ***`<chrono>`*** library. ***now()*** function is better than ***time()*** from the *<ctime>* library because it can measure time with very fine granularity (nanoseconds, microseconds, or milliseconds depending on the system).
- The line ***auto variableName = high_resolution_clock::now();*** captures the current time.
- The ***auto*** keyword is used to declare the variable type because the type returned by ***now()*** is complex (***std::chrono::time_point***). Using ***auto*** allows the system to deduce the variable type automatically.
- More details about using <chrono> and its classes can be found at: https://en.cppreference.com/w/cpp/chrono

CHAPTER 9

Recursive Thinking in C++

HUSSAM GHUNAIM

Learning Objectives

Type your learning objectives here.

- Understands the recursion mechanism
- Write void recursive functions
- Write recursive functions that return values

INTRODUCING RECURSION

A recursive function, in simple words, is a function that calls itself. Initially, this might sound weird, but later, we will explore some scenarios where implementing recursive functions can be the best solution.

Example 01

To clarify the idea of recursive functions, let us start by comparing an iterative code to a recursive code. When you run the given code, you will get a countdown from 3 to 0 twice. One output comes from the iterative code, lines 12 and 13. The second output comes from calling the *countDown(3)* function, line 16. As you can see, the output is identical; however, the implementation is entirely different. The iterative code is straightforward and self-explanatory. The recursive code, however, needs a careful explanation. First of all, when we say a function is allowed to call itself, this implies a possibility for an infinite loop unless we add some code to terminate the recursive calls at some point.

```
1   #include <iostream>
2   using namespace std;
3
4   void countDown(int num) {
5       cout << num << endl;
6       if (num > 0)
7           countDown(num - 1);
8   }
9
10  int main() {
11      cout << "Iterative count down code" << endl;
12      for (int i = 3; i > -1; i--)
13          cout << i << endl;
14
15      cout << "Recursive count down code" << endl;
16      countDown(3);
17
18      return 0;
19  }
```

In line 16, when the *countDown(3)* is called, the value 3 will be passed to the function's formal parameter in line 4. Next, line 5 prints the current value of the ***num*** variable, which is 3, and the *if*-statement in line 6 checks the condition ***num > 0***. At this time, this condition returns **true** and causes a recursive call to the *countDown* function with the updated argument ***num – 1***, which is ***3 – 1 = 2***. This time, 2 will be printed to the console, and the *if*-statement condition still returns **true**, causing a new recursive call *countdown (2 -1)*. This pattern will keep repeating until the *if*-statement condition returns **false**, terminating the recursive calls.

To better understand example 01, let us visualize how recursion works. Figure 01 shows what happens when a function calls itself recursively. Every time a function calls itself recursively, a new frame for the function call will be created on the call stack in memory (RAM), exactly similar to a function that calls another function, except here, the function being called is itself. This is a

primary reason why recursion often consumes more memory than iterative code. However, it provides effective solutions to some problems, which you will discover shortly.

Starting from the *main()* function, when *countDown(3)* is called, the system will invoke the function in memory and move the execution control to it. The first call will pass 3 as an argument. Then, inside the function, the code will run normally, that is, printing the current value of the argument ***num*** and checking the *if* condition ***num > 0***, which returns **true**. Therefore, *countDown* function will be called again with the updated argument *(3 – 1)*. Now, the execution will move over to the second copy of the *countDown(2)* function with the argument ***num*** value as 2. Execution will continue as normal until hitting the *if* condition again. This pattern will continue until the *if* condition returns **false** when the recursion terminates.

At this point, when the last created copy of the *countDown(0)* completes its execution, it will be terminated and removed from memory, as we expect it would happen with every function when it ends its execution. The execution control will return to the previous copy of the function, letting it continue its execution as well. This pattern will continue until the execution control returns to where it started in the *main()* function. Now, the *main()* function can continue its execution if there is any more code left before the *main()* function returns and is terminated and removed from memory as well.

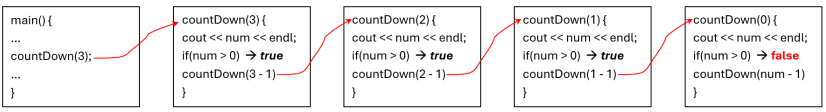


Figure 01: The recursive calls for the countDown() function.

Stacks to Manage Recursion

As you have already seen so far, recursion can create many copies of the recursive function. This can lead to a runtime error called

a **Stack Overflow**, which occurs when all available memory for running a program is used up. But what are stacks? Stacks are a data structure used to organize data by controlling how data must be added and removed from the stack. Figure 02 shows what stacks look like. A stack can be thought of as a single-opening container, meaning there is only one opening for both adding and removing data. Each time a new recursive call is made, a new frame is added to the top of the stack. When the execution reaches the last recursive call, which is now at the top of the stack, that frame is removed, effectively freeing memory. This process continues until the first frame added to the stack is removed. Thus, stacks are known as Last-In, First-Out (**LIFO**) data structures.

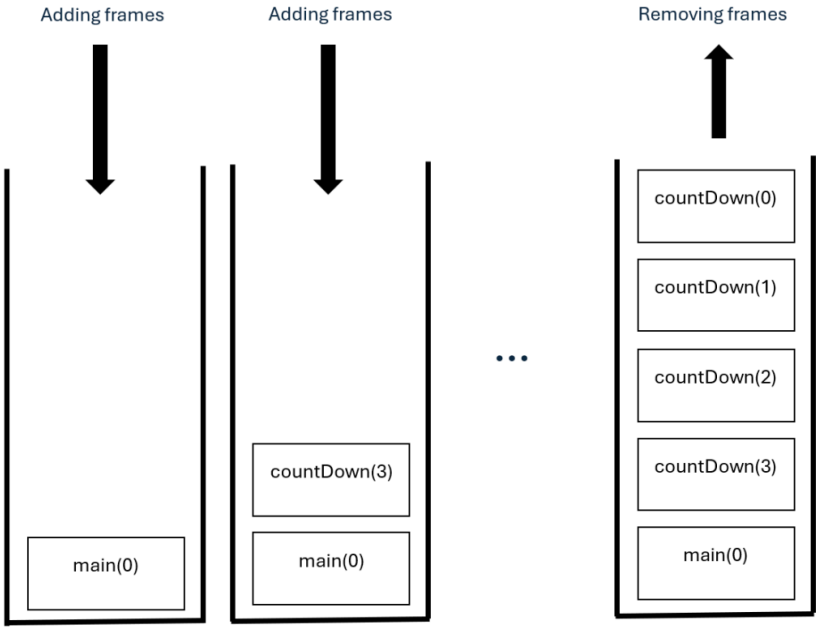


Figure 02: Managing recursion using stacks.

Example 02

Run the given code and comment on the results. Figure 3 illustrates the execution flow. The program starts executing at the *main()* function, where the *triangle(rows)* recursive function is called for the first time on line 16. The execution control then moves to the first copy of the *triangle(rows)* function created in RAM. At this point, as the variable ***rows*** value is 3, the *if*-statement condition in the function returns **false**, causing the second recursive call to be invoked on line 6. Note that the remaining code in lines 7 and 8 will not be executed until the execution control returns to this particular copy of the *triangle(rows)* function. The execution control then moves to the second copy of the recursive function, where the argument ***rows*** is 2. The *if*-statement condition still returns **false**, and the third recursive call is invoked with an argument value of 1. In the final recursive call, the argument ***rows*** is 0, which causes the *if*-statement condition to return **true**, terminating this particular copy of the recursive function. The execution control then returns to the previous copy of the *triangle(rows)* function, resuming from where it left off—specifically, the *for* loop in line 7. As the *for* loop executes, the ***rows*** value is 1, so one star is printed to the console. When this function finishes execution and is terminated, the execution control returns to the previous recursive call, where the ***rows*** value is 2, and the *for* loop prints two stars. This process continues recursively, with the number of stars printed corresponding to the ***rows*** value. The execution control eventually returns to the *main()* function. If any code remains in the *main()* function, it will resume execution until the end of the *main()* function body, at which point the program terminates.

```
1   #include <iostream>
2   using namespace std;
3
4   void triangle(int rows) {
5       if (rows == 0) return;
6       triangle(rows - 1);
7       for (int j = 0; j < rows; j++) cout << "*";
8       cout << endl;
9   }
10
11  int main() {
12      int rows;
13      cout << "Enter the number of rows: ";
14      cin >> rows;
15
16      triangle(rows);
17      return 0;
18  }
```

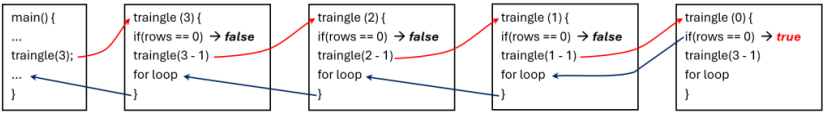


Figure 03: The execution flow for the triangle() function recursive calls.

Exercise 01

Write a recursive function ***void printChars(char ch, int times)*** that prints the character passed as the ***ch*** argument a number of times equal to the value of the ***times*** argument. In the ***main*** function, ask the user to input the character they want printed and the number of times they want it printed.

Exercise 02

Modify example 02 to draw the triangle upside down. Explain how the change you made gets the desired output.

Exercise 03

Write the recursive function ***void printNumbers(int n, int& sum)*** that prints out numbers from 1 up to a positive number n, along with their sum. Sample run:

```
Enter a number: 5
The sum of 1 2 3 4 5  is 15
```

RECURSIVE FUNCTIONS THAT RETURN VALUES

Recursive functions can be written to solve many problems. Therefore, recursive functions can either be void or return values of any type, such as int, double, string, etc. Writing void and non-void recursive functions is very similar. The only difference that we need to pay extra attention to is only the last returned value is usually the expected final value.

Example 03

The provided code contains a recursive function that calculates the factorial of any positive number. Line 12 demonstrates how calling a recursive function is similar to calling an iterative function. The string "The factorial of " is first printed to the console, followed by the value stored in the variable ***num***. Then, the string " is: " is printed, and the recursive function is called. At this point, execution control moves to the factorial function defined on line 4.

Figure 4 illustrates the execution flow. When the if-condition returns false, the recursive function is called again with the updated argument (n – 1). This pattern continues until the base case is reached, where the if-condition returns true, and the function returns the value 1 to the previous recursive call. At this point, the calculation can be completed and returned to the previous recursive call. This pattern continues until the first recursive call returns the final result to the main function.

```cpp
#include <iostream>
using namespace std;

int factorial(int n) {
    if (n == 0) return 1; // Base case: factorial of 0 is 1
    else return n * factorial(n - 1);
}

int main() {
    int num;
    cout << "Enter a positive number: "; cin >> num;
    cout << "The factorial of " << num << " is: " << factorial(num) << endl;
    return 0;
}
```

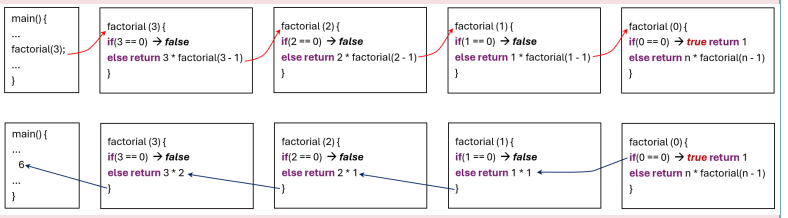


Figure 04: The execution flow for the recursive function factorial.

Recursion is a powerful programming technique used to solve problems that can be broken down into smaller, similar subproblems. By calling the same function within itself, recursion allows for elegant solutions to complex problems. In Chapter 6, we discussed the binary search algorithm. Binary search works by repeatedly dividing the array into halves, thereby reducing the total size of the array. By continuing this process, the search problem is minimized to the smallest possible array, which contains only one item. This makes implementing the binary search algorithm recursively ideal.

Examples 04

Figure 05 from Chapter 6 is presented again here for convenience. The recursive ***binarySearch*** function takes four arguments: a sorted

array, the starting and ending indices of the search range, and the target value to find. The initial search begins with a starting index of 0 and an ending index equal to the array's size minus 1.

Each time *binarySearch* is called, line 7 checks if the starting index has exceeded the ending index. If this condition is **true**, it indicates that the entire search range has been examined without finding the target. In this case, line 7 returns -1, signaling that the target is not present. The *main* function then uses this return value to display an appropriate message to the console, as seen in lines 26 to 29. If the condition in line 7 is **false**, the search continues.

Line 9, calculate the ***middle*** index that will be used in the following *if..else* statement. Note that sometimes, the calculated ***middle*** index has a fraction. Of course, this is illegal and cannot be used as an index. Therefore, the ***middle*** variable is declared as an int, which discards any fraction (if any) in the ***middle*** variable value.

Line 12 compares the target value with the value stored in the ***middle*** index. In the provided example, the target is 29, and the ***middle*** index initially points to 22. Since these values are different, the else condition is evaluated. The function then makes a recursive call, either on line 15 or line 18, depending on whether the target is smaller or larger than the ***middle*** value. If the target is smaller, the search range is narrowed to the lower half of the array by adjusting the ending index. Conversely, if the target is larger, the search range is narrowed to the upper half by adjusting the starting index.

This process repeats with each recursive call, effectively halving the search range until either the target is found or the condition in line 7 becomes **true**, terminating the recursion.

```cpp
#include <iostream>
using namespace std;
const int MAX_Array_Size = 10;

// Recursive function for binary search
int binarySearch(int arr[], int start, int end, int target) {
    if (start > end) return -1;  // Base case: target not found

    int middle = (start + end) / 2; // Calculate the middle index

    // Check if the target is present at the middle
    if (arr[middle] == target) return middle;

    // If target is smaller than middle value, target can only be in the lower subarray
    else if (arr[middle] > target) return binarySearch(arr, start, middle - 1, target);

    // Otherwise, the target can only be in the upper subarray
    else return binarySearch(arr, middle + 1, end, target);
}

int main() {
    int arr[MAX_Array_Size] = { -11, 0, 5, 12, 22, 29, 30, 44, 51, 219 }; //Sorted array
    int target = 29;
    int result = binarySearch(arr, 0, MAX_Array_Size - 1, target);//Call the recursive function

    if (result == -1)
        cout << "Element " << target << " not found in the array" << endl;
    else
        cout << "Element " << target << " found at index: " << result << endl;

    return 0;
}
```

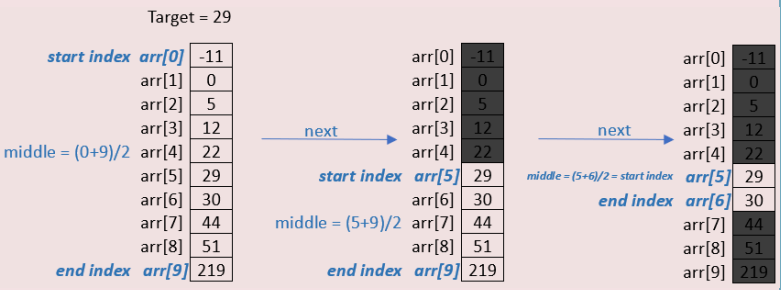


*Figure 05 (**from chapter 6**): Binary search example.*

Exercise 04

Write a program that reverses the user's input. As we do not know the size of the input in advance, you must use a vector to store the input as characters using the cin.get() member function. Reversing the order of the characters is handled by the recursive function ***void***

reverseChars(vector<char>& chars, int start, int end). Reading the input and printing out the reversed input must be done in the *main* function.

Exercise 05

Write a program that counts how many times a character occurs in a given text. The program must have the recursive function *int countChar(const string& str, char target)* to achieve this task. The *main* function will get the text from the user and print out the result.

Hint: It is helpful to utilize the string class member functions like *substr()*, *empty()*, and others as needed.

Exercise 06

Write a program that calculates the length of a given string. The program must have the recursive function *stringLength* that takes a parameter of type **string** and returns an integer. The *stringLength* returns 0 if the string is empty.

Exercise 07

Determine the output for each of the following recursive functions for n = 4:

```
  1)
int mountain(int n) {
   if (n <= 1) return n * 2;
   else return mountain(n - 1) + mountain(n - 2);
}
```

```
2)
int stream(int n) {
   if (n == 0) return 3;
   else return stream(n - 1) - n;
}

3)
int valley(int n) {
   if (n <= 0) return 1;
   else return valley(n - 1) * (n + 1);
}
```

WRAP UP

This chapter introduces the concept of recursion in programming, where a function calls itself. It begins by comparing an iterative code with a recursive one, highlighting the need for a base case to prevent infinite loops in recursion. The chapter explains how recursive calls create new frames on the call stack, leading to potentially higher memory consumption compared to iteration and the risk of stack overflow. The Last-In, First-Out (LIFO) nature of the stack in managing recursive calls is also discussed. The chapter then provides examples of both void and value-returning recursive functions.

ANSWER KEY

Exercise 01:

```
#include <iostream>
using namespace std;

void printChars(char ch, int times) {
    if (times == 0) return;

    cout << ch << " ";
    printChars(ch, times - 1);
}

int main() {
    char character;
    int times;

    cout << "Enter a character: "; cin >> character;
    cout << "Enter how many times to print the character: "; cin >> times;

    cout << "Here is the character printed " << times << " times: ";
    printChars(character, times);
    cout << endl;
    return 0;
}
```

Exercise 02:

You only need to place the recursive call, ***triangle(rows – 1),*** after the ***for*** loop, that is, after printing the stars. In this case, the first recursive call prints a number of stars equal to the passed argument ***rows*** first, and then calls the ***triangle(rows – 1)***. The scored recursive call prints the number of stars equal to ***rows -1***, and so on.

Exercise 03:

```cpp
#include <iostream>
using namespace std;

void printNumbers(int n, int& sum) {
    if (n == 0) return;  // Base case: stop when n is 0
    printNumbers(n - 1, sum);
    sum += n;
    cout << n << ", ";
}

int main() {
    int n, sum = 0;
    cout << "Enter a number: "; cin >> n;
    cout << "The sum of ";
    printNumbers(n, sum);
    cout << "is " << sum << endl;
    return 0;
}
```

Exercise 04:

```cpp
#include <iostream>
#include <vector>
using namespace std;

// Recursive function to reverse a vector of characters
void reverseChars(vector<char>& chars, int start, int end) {
    if (start >= end) return; //Base case: end of recursion

    // swapping characters
    char temp = chars[start];
    chars[start] = chars[end];
    chars[end] = temp;
    reverseChars(chars, start + 1, end - 1);
}

int main() {
    vector<char> charVector;
    char c = '\0';

    cout << "Enter the text to be reversed (press Enter to finish): ";
    while (c != '\n') {// Read characters until new line character is read
        charVector.push_back(c);
        cin.get(c);
    }

    reverseChars(charVector, 0, charVector.size() - 1);
    cout << "Reversed characters: ";
    for (char c : charVector) cout << c;

    return 0;
}
```

Exercise 05:

```cpp
#include <iostream>
#include <string>
using namespace std;

// Recursive function to count occurrences of a character in a string
int countChar(const string& str, char target) {
    if (str.empty()) return 0; // Base case: string is empty

    if (str[0] == target)  //Check first character and count recursively
        return 1 + countChar(str.substr(1), target);
    else
        return countChar(str.substr(1), target);
}

int main() {
    string str;
    char target;
    cout << "Enter a text: "; getline(cin, str);
    cout << "Enter the target character: "; cin >> target;

    cout << "\nThe character '" << target << "' appears ";
    cout << countChar(str, target) << " times in the text.\n";

    return 0;
}
```

Exercise 06:

```cpp
#include <iostream>
#include <string>
using namespace std;

int stringLength(const string& str) {
    // Base case: If the string is empty, the length is 0
    if (str.empty()) return 0;

    // Recursive step: Return 1 (for the first character) + the length of the rest of the string
    return 1 + stringLength(str.substr(1));//str.substr(1) returns the string without the first char
}

int main() {
    string str;
    cout << "Enter a string to calculate its length: "; getline(cin, str);
    cout << "The string length is " << stringLength(str) << endl;
    return 0;
}
```

Exercise 07:

1)

- `mountain(4) = mountain(3) + mountain(2)`
- `mountain(3) = mountain(2) + mountain(1)`

- `mountain(2)` = `mountain(1)` + `mountain(0)`
- `mountain(1)` = 1 * 2 = 2
- `mountain(0)` = 0 * 2 = 0
- `mountain(2)` = 2 + 0 = 2
- `mountain(3)` = 2 + 2 = 4
- `mountain(4)` = 4 + 2 = 6

Therefore, `mountain(4)` returns **6**.

2)

1. `stream(4)` calls `stream(3)` and subtracts 4: `stream(3) - 4`
2. `stream(3)` calls `stream(2)` and subtracts 3: `stream(2) - 3`
3. `stream(2)` calls `stream(1)` and subtracts 2: `stream(1) - 2`
4. `stream(1)` calls `stream(0)` and subtracts 1: `stream(0) - 1`
5. `stream(0)` returns 3 (base case).

Now, let's work our way back up:

- `stream(1)` = `stream(0)` – 1 = 3 – 1 = 2
- `stream(2)` = `stream(1)` – 2 = 2 – 2 = 0
- `stream(3)` = `stream(2)` – 3 = 0 – 3 = -3
- `stream(4)` = `stream(3)` – 4 = -3 – 4 = -7

Therefore, `stream(4)` returns **-7**.

3)

1. `valley(4)` calls `valley(3)` and multiplies the result by (4 + 1) = 5: `valley(3) * 5`
2. `valley(3)` calls `valley(2)` and multiplies the result by (3 + 1) = 4: `valley(2) * 4`
3. `valley(2)` calls `valley(1)` and multiplies the result by (2 + 1) = 3: `valley(1) * 3`
4. `valley(1)` calls `valley(0)` and multiplies the result by (1 + 1) = 2: `valley(0) * 2`
5. `valley(0)` returns 1 (base case).

Now, let's work our way back up:

- `valley(1)` = `valley(0)` * 2 = 1 * 2 = 2
- `valley(2)` = `valley(1)` * 3 = 2 * 3 = 6
- `valley(3)` = `valley(2)` * 4 = 6 * 4 = 24
- `valley(4)` = `valley(3)` * 5 = 24 * 5 = 120

Therefore, `valley(4)` returns **120**.

END OF CHAPTER EXERCISES

Multiple Choice Questions:

1) What is the defining characteristic of a recursive function?
   a) It uses loops.
   b) It calls itself.
   c) It returns a value.
   d) It takes no arguments.
2) What is the purpose of the base case in a recursive function?
   a) To decrease memory usage.
   b) To prevent infinite loops.

c) To print the output of the results.
d) To perform calculations.

3) What data structure is used to manage recursive function calls?
a) Queue
b) Vector
c) Stack
d) Array

4) What does the term “LIFO” stand for?
a) Last-In, First-Out
b) Linear-In, First-Out
c) List-In, First-Out
d) None of the above

5) What happens when a recursive function lacks a proper base case?
a) The program runs faster.
b) A stack overflow error occurs.
c) The program outputs incorrect results.
d) The program terminates normally.

6) In the ***countDown*** function, what is the base case?
a) num > 0
b) num == 3
c) num < 0
d) num value is 0

7) In the ***triangle*** function, where is the recursive call made?
a) Before the for loop.
b) Inside the for loop.
c) After the for loop.
d) Inside the main function.

8) What is the output of ***triangle(2)***?
a) *
b) **
c) *
**
d) **

*

9) When a recursive function completes, what happens to its stack frame?

a) It remains in memory.
b) It is moved to another stack.
c) It is removed from the stack.
d) It is copied to another location.

10) In the discussed ***factorial*** function in this chapter, what is the base case?

a) n > 0
b) n == 0
c) n == 1
d) return n * factorial(n – 1);

11) The ***binarySearch*** function in the text is designed to work with which type of array?

a) Unsorted array
b) Array with duplicate elements only
c) Array containing only positive numbers
d) Sorted array

12) What does the ***substr(1)*** method do when used with a string?

a) Returns the first character of the string.
b) Returns the entire string.
c) Returns the string without the last character.
d) Returns a substring starting from index 1 to the end.

13) In the ***reverseChars*** function, what is the base case that stops the recursion?

a) start >= end
b) start > end
c) start < end
d) start == end

14) According to the text, the binary search algorithm is ideally implemented recursively because it:

a) Repeatedly divides the problem into smaller, similar subproblems.

b) Always processes the entire array.
c) Requires complex loop structures.
d) Can only handle integer arrays.

True/False Questions:

15) A recursive function can call itself directly or indirectly.
16) Iterative solutions always consume less memory than recursive solutions.
17) Stacks use a FIFO (First-In, First-Out) data structure.
18) A stack overflow error occurs when all available memory for running a program is used up.
19) The base case in a recursive function determines when the recursion stops.
20) Recursive functions are always less efficient than iterative functions.
21) Recursive functions can return values.
22) Recursive functions can only return integer values.
23) Every recursive function must have a base case to prevent infinite recursion.
24) Calling a recursive function from the main function is fundamentally different from calling an iterative function.
25) In a non-void recursive function, the first returned value is always the final result.
26) The binary search algorithm works efficiently on unsorted arrays.
27) The ***binarySearch*** function reduces the search space by half in each recursive call.
28) The ***substr()*** method in C++ string class modifies the original string.
29) The recursive ***stringLength*** function calculates the length of a string by iteratively traversing through its characters.

Coding Questions:

30) Write a recursive function that calculates the sum of all digits of a given integer. ***Hint***: To isolate individual digits, consider using the modulo 10 of the given integer. To get rid of the least

significant digit, consider dividing the number by 10 and save the result as an integer.
31) Write a recursive function ***int power(int base, int exponent)*** to calculate the power of a given base raised to a given exponent. Negative numbers are not allowed.
32) Write a recursive function ***bool isPalindrome(string str)*** to check if a given string is a palindrome (reads the same forwards and backward). For simplicity, the code needs to test strings without any modification, like removing spaces or other punctuation marks. For example, "Madam, I'm Adam" will return 'not a palindrome', however, the "madamimadam" will return 'a palindrome'. You can test other simpler strings, like racecar, abcde edcba, and so on.
33) Write a recursive function ***void printArrayReverse(int arr[], int size)*** to print the elements of an array in reverse order.
34) Write a recursive function ***int countOccurrences(int arr[], int size, int target)*** to count the number of times a given integer target appears in an array.
35) Write a recursive function called ***countEven*** that takes a vector of integers as input and returns the number of even integers in the vector.
36) Modify the previous question to count the number of odd integers in the vector.
You only need to change the condition that tests for even numbers to odd numbers. A possible change will be like this:
if(nums[index] % 2 == 2) change to if(nums[index] != 2)
37) Write a recursive function ***int countCharOccurrences(string str, char target, int index)*** to count the number of times a given character target appears in a string.
38) Write a recursive function called ***sumRange*** that takes two integer arguments, ***start*** and ***end***, and returns the sum of all integers in that range (inclusive).
39) Write a recursive function called ***printReverse*** that takes a string as input and prints the string in reverse order to the console.

PROJECTS

Project 01: Recursive Number Guessing Game

Objective: To design and implement a number-guessing game in C++ that utilizes recursion to handle the core guessing logic. This project will test your understanding of recursive functions, including defining base cases and recursive steps, managing function parameters, and controlling program flow using recursion.

Requirements:

1. **Game Setup:**
    - The program should randomly generate a secret integer within a defined range, like between 1 and 100. But the range can be anything else if you want.
    - The program should welcome the player and inform him/her about the selected range of the secret number.
2. **Recursive Guessing Function:**
    - You **must** implement a recursive function to handle the player's guessing process. This function should:
        - Take four parameters: the secret number, the lower bound of the possible range, the upper bound of the possible range, and the counter variable to count the number of player guesses.
        - Prompt the player to enter their guess

within the current bounds.

- **Base Cases:**
    - If the player's guess is outside the current lower and upper bounds, inform him/her of the invalid guess and make a recursive call to the same function, prompting for input again within the valid bounds.
    - If the player's guess is equal to the secret number, congratulate him/her and display the number of guesses it took. This will be one of the stopping conditions for the recursion.
- **Recursive Steps:**
    - If the player's guess is lower than the secret number, inform him/her it's "Too low!" and make a recursive call to the same function. The lower bound for the next recursive call should be updated to be one greater than the player's current guess.
    - If the player's guess is higher than the secret number, inform them it's "Too high!" and make a recursive call to the same function. The upper bound for the next recursive call should be updated to be one less than the player's current guess.

3. **Game Control in `main` Function:**
    - The **`main`** function should:
        - Seed the random number generator to ensure different secret numbers each time the program runs.
        - Define the initial range for the secret number and generate the secret number.
        - Welcome the player and start the guessing process by making the initial call to your recursive guessing function.
        - After the player guesses correctly, ask if they want to play again by entering 'y' or 'n'.
        - Implement a loop that allows the player to play multiple rounds of the game if they choose to. For each new game, a new secret number should be generated, and the guess counter should be reset.
        - Display a thank you message when the player decides to quit.

Grading Criteria:

- Correct implementation of the recursive guessing function with appropriate base cases and recursive steps.
- Proper handling of the game setup in the **`main`** function, including random number generation and the play-again loop.
- Accurate tracking and display of the number of guesses.

- Clear and user-friendly output and prompts.
- Adherence to the requirement of using recursion for the guessing logic.

This project will challenge you to apply the concept of recursion to solve a problem in a structured and interactive way. Good luck!

Sample Run:

```
Welcome to the Recursive Number Guessing Game!
I'm thinking of a number between 1 and 100.
Enter your guess (between 1 and 100): 43
Too high! Try again.
Enter your guess (between 1 and 42): 15
Too low! Try again.
Enter your guess (between 16 and 42): 30
Too low! Try again.
Enter your guess (between 31 and 42): 40
Too high! Try again.
Enter your guess (between 31 and 39): 35
Too high! Try again.
Enter your guess (between 31 and 34): 33
Too high! Try again.
Enter your guess (between 31 and 32): 32
Too high! Try again.
Enter your guess (between 31 and 31): 31
Congratulations! You guessed the number 31 correctly in 8 guesses!
Do you want to play again? (y/n): y

Welcome to the Recursive Number Guessing Game!
I'm thinking of a number between 1 and 100.
Enter your guess (between 1 and 100): 101
Invalid guess. Please stay within the bounds.
Enter your guess (between 1 and 100): 50
Too high! Try again.
Enter your guess (between 1 and 49): 45
Too high! Try again.
Enter your guess (between 1 and 44): 20
Too high! Try again.
Enter your guess (between 1 and 19): 5
Too high! Try again.
Enter your guess (between 1 and 4): -5
Invalid guess. Please stay within the bounds.
Enter your guess (between 1 and 4): 2
Too low! Try again.
Enter your guess (between 3 and 4): 3
Too low! Try again.
Enter your guess (between 4 and 4): 4
Congratulations! You guessed the number 4 correctly in 9 guesses!
Do you want to play again? (y/n): n

Thanks for playing!
```

Project 02: Recursive Insertion Sort Implementation

The goal of this project is to help you understand and implement the recursive insertion sort algorithm. You will write a program that sorts an array of integers using a recursive approach and inserts elements into their correct positions.

Instructions:

1. Understand the Problem:
   - You need to sort an array of integers using the recursive insertion sort algorithm.
   - The algorithm will sort the array by recursively sorting the first n−1 elements and then inserting the last element into its correct position.
   - You may search the web for tutorials if you need more details on how this algorithm works.
2. Write the Code:
   - Create a function insertElement that inserts the last element into its correct position in a sorted subarray.
   - Create a recursive function recursiveInsertionSort, that sorts the array by recursively sorting the first n−1 elements and then using insertElement to insert the last element.
   - Write a main function to test your implementation with an example array.
3. Submission:
   - Submit your source code file.
   - Ensure your code is well-commented and follows good coding practices.
   - Include a brief explanation of how your code works and any challenges you faced during implementation.
4. Evaluation Criteria:
   - Correctness: The program correctly sorts the array using recursive insertion sort.
   - Code Quality: The code is well-organized, readable, and follows good coding practices.
   - Comments: The code is well-commented, explaining the

logic and steps clearly.
• Explanation: The explanation of the code and challenges faced is clear and concise.
Tips:
• Test your code with different arrays to ensure it works correctly.
• Pay attention to the base case in the recursive function to avoid infinite recursion.
• Make sure to handle edge cases, such as an empty array or an array with one element.

Appendix A: C++ Keywords/ Reserved words

HUSSAM GHUNAIM

C++ has a set of keywords that are used by the language system. Programmers should not use these reserved words as variable names or function names or anything else besides their original meaning. (***source***: Keywords (C++) | Microsoft Learn)

STANDARD C++ KEYWORDS

`alignas`
`alignof`
`and`
`and_eq`
`asm`
`auto`
`bitand`
`bitor`
`bool`
`break`
`case`

catch
char
char8_t
char16_t
char32_t
class
compl
concept
const
const_cast
consteval
constexpr
constinit
continue
co_await
co_return
co_yield
decltype
default
delete
do
double
dynamic_cast
else
enum
explicit
export
extern
false
float
for

friend
goto
if
inline
int
long
mutable
namespace
new
noexcept
not
not_eq
nullptr
operator
or
or_eq
private
protected
public
register
reinterpret_cast
requires
return
short
signed
sizeof
static
static_assert
static_cast
struct
switch

template
this
thread_local
throw
true
try
typedef
typeid
typename
union
unsigned
using declaration
using directive
virtual
void
volatile
wchar_t
while
xor [b]
xor_eq

Appendix B: C++ Operators, Precedence, and Associativity Rules

HUSSAM GHUNAIM

C++ has many operators, precedence, and associativity rules. Below is a short list. (***source***: C++ built-in operators, precedence, and associativity | Microsoft Learn)

Operator Description	Operator	Alternative
Group 1 precedence, no associativity		
Scope resolution	`::`	
Group 2 precedence, left to right associativity		
Function call	`()`	
Postfix increment	`++`	
Postfix decrement	`--`	
Static type conversion	`static_cast`	
Group 3 precedence, right to left associativity		
Size of object or type	`sizeof`	
Prefix increment	`++`	
Prefix decrement	`--`	
Logical not	`!`	`not`
Unary negation	`-`	
Unary plus	`+`	
Address-of	`&`	
Indirection	`*`	
Group 5 precedence, left to right associativity		
Multiplication	`*`	
Division	`/`	
Modulus	`%`	
Group 6 precedence, left to right associativity		
Addition	`+`	

Operator Description	Operator	Alternative
Subtraction	-	
Group 8 precedence, left to right associativity		
Less than	<	
Greater than	>	
Less than or equal to	<=	
Greater than or equal to	>=	
Group 9 precedence, left to right associativity		
Equality	==	
Inequality	!=	`not_eq`
Group 13 precedence, left to right associativity		
Logical AND	&&	`and`
Group 14 precedence, left to right associativity		
Logical OR	||	`or`
Group 15 precedence, right to left associativity		
Conditional	? :	
Assignment	=	
Multiplication assignment	*=	
Division assignment	/=	
Modulus assignment	%=	
Addition assignment	+=	
Subtraction assignment	-=	
Group 16 precedence, left to right associativity		
Comma	,	