A Guided Introduction to C++ Programming

From Basic Concepts to Advanced Techniques for College Students

Oladele O. Campbell, PhD

Peter Bake, MSc

Aishatu Aliyu Mohammed, MSc

Adamu Isah Dagah, MSc

September 2024

ISBN: 978-978-989-866-4

Preface

This comprehensive textbook offers a thorough and practical introduction to C++ programming, specifically designed for college computer science students. Starting with the basics, it covers the advantages of high-level programming, the compilation process, and the structure of C++ programs. Readers will learn to develop algorithms and explore two popular program design approaches. Each chapter includes clear objectives, practical examples, and review exercises to reinforce learning. With real-world examples and assignments, students will gain a solid foundation in C++ programming, preparing them for successful careers in software development. Dive into the world of C++ and master the skills needed to excel in the dynamic field of computing.

Oladele Campbell,
20 September 2024

Foreword

I am pleased to introduce you to the fascinating world of C++ programming. This book delves into the intricacies of this high-level language, exploring its advantages, structure, and development process. It is designed for those with little to no prior programming experience. In comparison to others, the book is very rich in content and unique in the simple style of presentation. In it you will find the fundamentals of the language, including:

- What C++ is, its relevance and realities, also its priorities, profits and promises
- Advantages and advances of using C++, a powerful high-level language.
- Basic components and elements of C++ programming
- Control flow statements, and structures and common challenges faced by C++ programmers and how to overcome them
- Data structures, detailed methodologies and discover the inner workings of compilers and their role in translating source code.
- Explore the anatomy of a C++ program, understanding its components and organization with exciting real-life practical examples (to help you learn and apply concepts learned from the book), exercises garnished with comprehensive explanations
- Functions, which are reusable blocks of code
- Gain insights into algorithm development and popular program design approaches.

The book also deals with pointers, which are variables that store memory addresses and object-oriented programming (OOP) concepts, such as classes and objects. By the end of the book, you will be able to write simple C++ programs to solve a variety of problems.

This book is a valuable resource for anyone who wants to learn C++ programming. It is clear, concise, and easy to follow. Whether you are a student, a professional, or simply someone curious about programming, this book is a great place to start. We hope you find this book to be a valuable resource for learning C++ programming. Let's dive in and discover the exciting world of C++! Thank you.

Dr. David Michael
Associate Professor of Telecommunications Engineering
Federal University of Technology, Minna.

Acknowledgments

The completion of this book would not have been possible without the support and contributions of many individuals and organisations. We express our heartfelt gratitude to:

- The Almighty God for the strength, wisdom, and providence that enabled the completion of this work.
- The many generations of students in our C++ programming courses have influenced, inspired, and contributed to the writing of this book.
- Nigeria's Tertiary Education Trust Fund (TETFund) for funding this book's publication.
- The Rector of Niger State Polytechnic, Dr Ibrahim Umar Kontagora, saw the value of the book and gave the final approval needed for submission of the book proposal to TETFund.
- The TETFund Desk officer, Mr Isah Danlami Shehu, encouraged the submission, the patience when the manuscript would have almost been withdrawn in the face of seeming delay, and saw to its eventual publication.
- Mr Mustapha Mahmud, the ASUP chairman, Niger State Polytechnic, was the first peer reviewer of the manuscript and provided useful critique and improvements, and encouragement all the way.
- Associate Professor Marufu Olalere of the National Open University of Nigeria, Abuja, who, on short notice and despite the great inconvenience, reviewed and provided the external peer review needed for submission to TETFund.
- Dr Abel Alade of the Federal Polytechnic, Bida, who also, on short notice, provided the second external peer review.
- Dr Ismaila Sanusi of the University of Eastern Finland facilitated the plagiarism check of the manuscript on TurnItIn.
- Associate Professor David Michael, who read the manuscript and wrote the foreword.
- Mrs Victoria Campbell, (the wife of the first Author), who had earlier published two books for secondary school students as a commercial subject teacher and provided encouragement and support for the accomplishment of this work.
- The management and staff at Ahmadu Bello University Press, Zaria, for a good job in publishing the book.

We are deeply grateful for the support and encouragement we may have received throughout this journey from others we may have missed in the above list.

Oladele Campbell, PhD

Contents

Chapter 1

Introduction to C++ Programming

1.1 Objectives

In this chapter, you will:

- Learn about the advantages of using a high-level language like C++.
- Discover what a compiler is and how it works.
- Learn about the compilation process in C++.
- Examine the anatomy (or structure) of a C++ program.
- Explore how a C++ program is developed and processed.
- Learn what an algorithm is and how to develop algorithms.
- Get familiar with two popular program design approaches.

1.2 What is C++?

C++ is a high-level programming language for developing application and system programs. C++ is an object-oriented programming language, making it possible to develop programs that are essentially objects interacting to provide services or solutions users want. The programming language C is a forerunner to C++, which extends the former. That's why C++ is a superset of C. Both programming languages have some features in common. However, C, a procedural language, is limited compared to C++ which has, in addition to procedural language features, facilities for developing object-oriented programs. Several once or presently-popular programs developed using C++ include Windows XP, Microsoft Office, Google Chrome, Mozilla Firefox, etc.

1.3 Why Program in C++?

Why do we need to learn and use a programming language like C++? At its core, a computer or modern computing devices (like mobile phones, tablets, laptops, etc.) are simply processors with some memory capable of running instructions like "store 5 in memory location 23459". However, a computing device does not understand that instruction directly and, as you will soon observe (or must have realized) in your learning to program, we do not write such instructions. Instead, in C++ programming, the program you write (called source code) is stored in the computer as text files containing C++ codes. Why would we express a program as a text file in a programming language, instead of writing the processor's instructions as in the previous example? The answer is in the advantage derived by doing so.

1.4 Advantages of C++ Program

Some advantages of C++ programs over programs written in other languages are:

- **Conciseness**: High-level programming language like C++ allows us to express common sequences of commands more concisely (i.e., more shortly and smartly).

- **Maintainability**: Modifying code is easier. C++ is an object-oriented language, meaning a program refers to objects that interact to produce a program's behaviour. This adds to the maintainability of code.
- **Portability**: A program written in C++ can be run on different hardware processors with little or no need to modify codes.

1.5 A Research on C++

Are you wondering what you stand to benefit from learning C++? Why is C++ a programming language worth giving your time and attention to? Let us examine the usefulness of learning C++. To start, google the following about C++:

- Its current position in popularity or usage among programming languages.
- Will it be useful in the next 10 years?
- What types of programs are written in C++?
- Salaries of C++ programmers in Nigeria, South Africa, the UK, and the US.
- Apart from those listed above, does C++ have more advantages over other languages?

Write your findings in a term paper of two or three pages.

1.6 Compilation of High-Level Language Source Code

Before we go into the details, remember that C++ is a high-level programming language. Other high-level programs are Java and VB among others. Now, let us look at how things work in high-level programming languages in general.

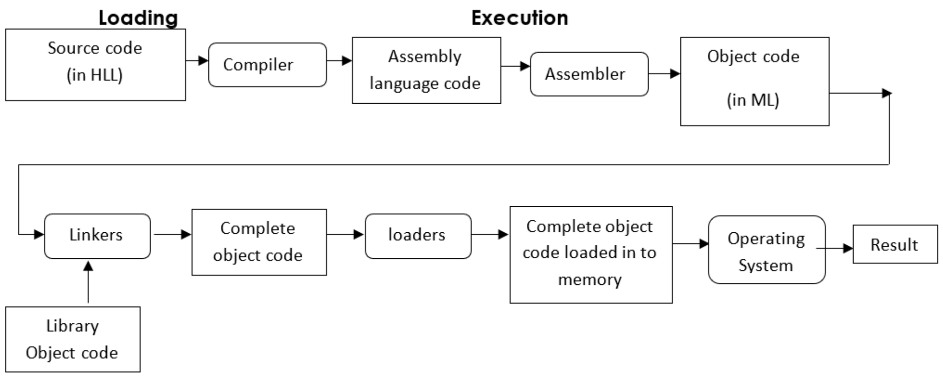

Figure 1.1: Transition of High-Level Language Program

Programs in a high-level language need to be translated into the language of the hardware before they can be executed. This translation begins with the work of a compiler, a program that reads the source code, checks for compliance with the programming language rules (called syntax), and indicates syntax errors (if there are any) in the source program. The programmer corrects such errors and the compiler checks it again and, if there is no error, converts the program into assembly code. The resulting code is passed to another translator program called an assembler, which translates the program into object codes (i.e., the machine language [ML]). Some compilers translate directly the high-level language source code into machine language.

As shown in Figure 1.1, our object code is usually not yet in its final form. The original high-level language code might have required the service of other codes (called libraries) which must be added to the object code. This is done by a program called the linker. The complete object code is then transferred to the loader, another program which transfers the complete object code into the memory. No program can be executed unless it is loaded into the hardware's memory. From there, the program instructions are picked one by one by the operating system and executed to provide the required result.

1.7 C++ Compilation Process

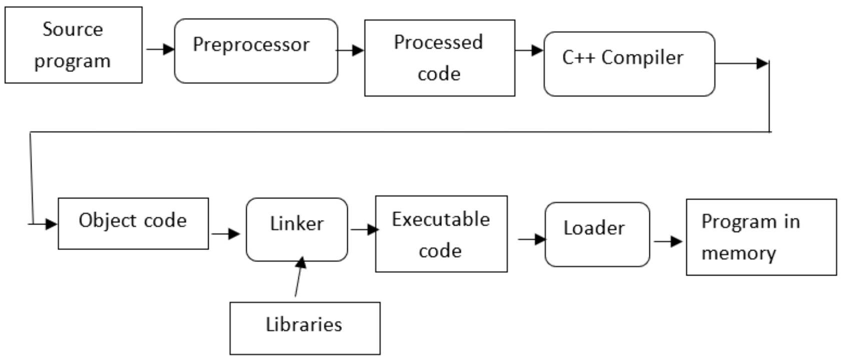

Figure 1.2: Compilation of a C++ program

As you can see in Figure 1.2, C++ adds an extra step to the compilation process we saw in Figure 1.1. The code is run through a program called preprocessor (which is a part of the C++ compiler). The preprocessor applies some modifications to the source code before being fed to the compiler. From there, the compilation process proceeds as stated before in Figure 1.1.

1.8 A Brief History of C++

According to the January 2023 TIOBE index, C++ remains one of the top 10 most popular high-level programming languages, sitting at the third position on the index. It was developed more than 30 years ago by Bjarne Stroustrup at AT&T Bell laboratories in the US. Initially, it was developed as an extension of the C language and was called "C with classes", making C++ a superset of C. However, whereas C is a procedural language, C++ is an object-oriented language. Later in this book, you will learn what makes C++ an object-oriented language. C++ is also a versatile and multi-paradigm language, which means it supports various programming styles, such as procedural, object-oriented, and generic.

C++ is known for its efficiency and performance, and it is often used to create operating systems, embedded systems, and high-performance applications. Many popular software packages, such as Microsoft Windows and Adobe Photoshop, were written in C++.

C++11, the 2011 revision of the language, brought major changes to the language, including support for multithreading, improved support for type inference and move semantics, and the introduction of new features such as lambda expressions and regular expressions.

C++17, C++20, and C++23 were released in 2017, 2020, and 2023 respectively, and continue to introduce new features and improvements to the language, such as template type deduction, improved type checking, and support for more modern programming paradigms.

1.9 A C++ Program

Let us examine a C++ program to see how C++ programs look. Though this example is a simple program, it shows some of what you will find in a bigger program later. You do not need to be concerned with knowing the details of this program's elements now. We will look at them in detail later. You only need to enter this program on a computer or any Android phone having a C++ compiler and see the effect of the program. There are free C++ compilers on the Internet that you can download and install on your PC or phone. Therefore, go ahead, search and get it to run your C++ codes.

Consider this C++ program:

```cpp
#include <iostream>

using namespace std;

```

```
 5 int main()
 6 {
 7     cout << "Hello, world!" << endl;
 8     cout << "My first C++ program." << endl;
 9     cout << "The sum of 2 and 3 = " << 5 << endl;
10     cout << "7 + 8 = " << 7 + 8 << endl;
11     return 0;
12 }
```

Note: The program lines are numbered for identification and explanation purposes. Line numbers are not part of the program. When you enter the program to run it, do not include line numbers.

1.9.1 Sample Output

When you compile and execute this program, the following four lines are displayed on the screen:

```
Hello, world!
My first C++ program.
The sum of 2 and 3 = 5
7 + 8 = 15
```

You can observe from the above that the four lines of output were generated by the statements in lines 7 – 10. The four statements are output statements using `cout`. However, unlike what you know in VB or other languages where keywords such as `PRINT`, and `WRITE` were used as output statements, here `cout` is an identifier. This name `cout` is defined in namespace `std` which is kept in the header file `<iostream>`.

The first line of the program is:

```
#include <iostream>
```

allows us to use the (predefined identifier) `cout` to generate output and the (manipulator) `endl` (another identifier) to generate a new line (i.e., it takes the cursor or insertion point to the next line).

The third lin line:

```
using namespace std;
```

allows you to use `cout` and `endl` without prefix `std::`. It means if you do not include this statement at the beginning of the program, you need to write `cout` as `std::cout` and `endl` as `std::endl` in every instance where you have `cout` and `endl` in your program.

The fifth line:

```
int main()
```

indicates the main portion of every C++ program. Everything starts and ends in this portion. The name `main` refers to a function (a function is simply a small program (subprogram) that performs a task). The word `int` (integer) refers to what value this function returns (return type) at the end of executing it. Here, the function returns an integer value (in this case, 0 is returned to the operating system—see line 11). This signals to the operating system that the function has finished its job.

The sixth line consists of a left brace ({) which marks the beginning of the body of the function `main`. The right brace at the last line of the program matches the left brace and marks the end of the body of the function `main`.

The « used in the lines where `cout` appears is an operator. It is called the stream insertion operator.

Let us consider the output statements in lines 7-10 and see how the output is generated on the screen.

```
cout << "Hello, world!" << endl;
```

As earlier mentioned, `cout` is an identifier in C++ (i.e., a name referring to a value). Its function is to evaluate whatever comes after it and display (insert) such value on the standard output devices – in this case, the device is a monitor. Therefore, the absolute statement: “Hello, world!” evaluates to `Hello, world!` and `Hello, world!` is displayed on the screen. Then meeting the second « and `endl` (which means a new line), the compiler takes the cursor (or insertion point) to the next line. Please, note that "Hello, world!" is a string – a string is anything enclosed in double quotation (" ").

The second output is:

```
cout << "My first C++ program." << endl;
```

It works as the previous statement.

The next output statement is:

```
cout << "The sum of 2 and 3 = " << 5 << endl;
```

The compiler sees the expression "The sum of 2 and 3 =" as a string. Remember anything enclosed in double quotation marks is a string, so the computer evaluates the string to itself, which is (The sum of 2 and 3 =). Then, it encounters the second insertion stream operator («) and which indicates a value to be inserted into the output stream. Here, 5 is an integer and evaluating result into 5. Thus, immediately after displaying the sum of 2 and 3 =, 5 is written. Then the `endl` makes the computer insert a new line; that is, it takes the cursor to the next line.

This is the general syntax for using the standard output stream identifier `cout` in a C++ program:

```
cout << expression or manipulator << expression or manipulator<<...;
```

1.10 Processing a C++ Program

Though we have seen how the compilation of a C++ program works, let us check again with an illustration using the previous C++ program. The steps to process the earlier C++ program (or any other program) are shown in Figure 1.3.

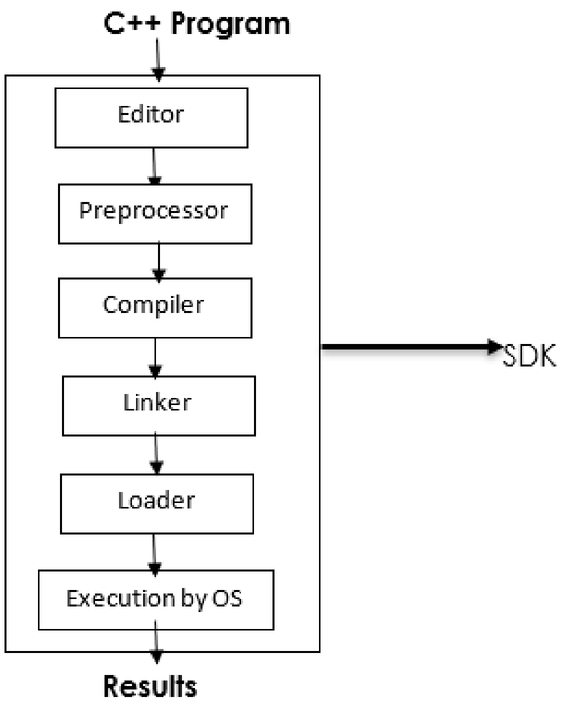

Figure 1.3: Processing a C++ Program

1.10.1 Step 1: Use a Text Editor

Use a text editor (from a C++ Software Development Kit (SDK) or any stand-alone editor) to create (enter/code) a C++ program following the rules (called syntax) of C++. This program is called a source code or source program. The source codes must be saved with a text file name and end with the extension.CPP. For example, if you saved the preceding example program with MyFirstCPPprogram, its complete name is MyFirstCPPprogram.CPP. This step is your job as a programmer. Hence, you must learn, understand and master the rules of C++ programming language to create good working source codes. As shown in Figure 1.3, this is done with SDK by most programmers. Well-known SDKs are Visual C++ and Visual Studio.NET (from Microsoft), C++ Builder (from Borland), and CodeWarriors (from Metrowerks) among others.

1.10.2 Step 2: Preprocessor Directives

Usually at the beginning, the C++ program contains statements starting with the hash symbols (#). Such statements are called preprocessor directives. A preprocessor is a program (usually part of the compiler) that performs the instruction in these statements. The preprocessor is also part of the SDK.

1.10.3 Step 3: Compiler Checks

This is the job of the compiler. Here the program statements are read and checked for compliance with the syntax of C++. Errors are reported with hints of the source and possible steps to enable the programmer to correct the error. After correcting all errors, the compiler translates the source codes into equivalent machine language. This equivalent program in machine language is called object code or object program. Note that the compiler is also part of the SDK.

1.10.4 Step 4: Linker Adds Library Codes

The object code usually requires some other codes. These codes available in the SDK are called library codes. The linker (a program in SDK) takes care of this step. It adds the library codes to your original codes. The resulting program is now executable by the computer. The command that does the linking on Visual C++ and Visual Studio.NET is “build” or “rebuild”; on C++ Builder, it is “Build” or “Make”, and it is “Make” in CodeWarrior.

1.10.5 Step 5: Loader Transfers to Memory

Next, the executable program must be loaded into the memory. From there, the computer picks the instructions one after the other and executes them. The program that does this is called the loader.

1.10.6 Step 6: Execution by Operating System

The final step is to execute the program. This is done by the operating system.

1.11 Programming

This book is all about programming. Programming is the art and science of developing a program for computing devices. Programming is a process of problem-solving. Different techniques are used by programmers. Good problem-solving techniques are nicely outlined and easy to follow. They not only solve the problem but also give insight into how the solution was reached.

To be a good problem solver and a good programmer, you must learn and follow good problem-solving techniques. A common problem-solving technique includes the following steps:

- Analyze a problem (to understand it).
- Outline the problem requirements.
- Identify the input, output, and processing.
- Design a solution usually called an algorithm to solve the problem.

Figure 1.4 below summarizes the programming process you will follow in developing, running, and maintaining a C++ program.

To develop a program to solve a problem, as shown above in Figure 1.4, you start by analyzing the problem. Analyzing means breaking the problem into bits you can process and understand. You then

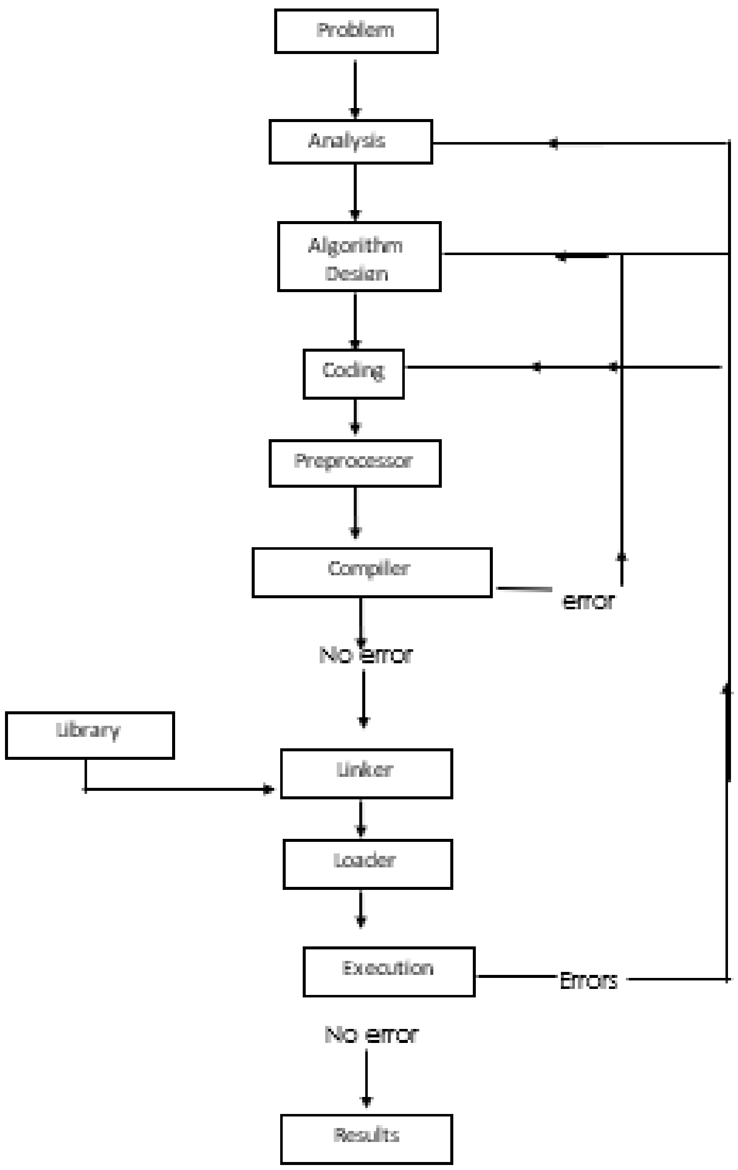

Figure 1.4: Programming Process

design the algorithm, write the program instructions in a programming language, and enter the program into a computer system.

1.11.1 Analyzing the Problem

Analyzing the problem is the first and most important step. This requires you to do the following:

- Ensure you thoroughly understand the problem. Why waste time solving a problem you do not understand (or have misunderstood!)? Normally, there are words you may not know or it could be that you have forgotten their meaning. In such cases, look up the meanings on the Internet or ask people.
- Understand the problem requirements. You will get this from the description or the specification of the problem. Such requirements are the given input(s), expected output(s), and required processing needed to transform inputs into outputs.
- If the problem is complex, divide the problem into sub-problems and repeat steps 1 and 2 to address each sub-problem. That is called divide and conquer.

1.11.2 Designing a Solution

After analyzing the problem comes designing a solution (called an algorithm). If you are dealing with a complex problem that you have broken into parts and, now, you have sub-problems, you need to design an algorithm for each sub-problem. Once you have designed an algorithm, you need to check for correctness. You can do this by walking through the algorithm with sample data. (i.e., tracing or running the algorithm with the data). At other times, you may need to ascertain the correctness using mathematical analysis.

Once you have a correct algorithm, you can convert the algorithm into code, using a programming language. With a compiler, check for syntax correctness and, if the program is correct, it is converted into a machine language code. Next, the linker adds necessary codes to the object code and that results in a complete executable code.

The final step is to execute the program after the codes have been transferred into the memory. Though the program is given approval by the compiler indicating no syntax errors, it does not guarantee that the program will run correctly. The program may terminate abruptly during execution due to runtime errors such as division by zero. Even if the program terminates normally, it may still generate erroneous results. Such errors are called logical errors. Under these two preceding circumstances of runtime or logical errors, you may have to reexamine the code, the algorithm, or even the problem analysis.

You will benefit from your overall programming experience if you develop the good programming practice of spending enough time to thoroughly analyze (i.e., understand) the program, and design the algorithm before attempting to code the program. Taking this careful approach to programming has several advantages:

- It is much easier to discover errors in programs.
- It is easier to follow and understand by someone else and you when you come back to it several weeks, months, or years.

Next, we explore some examples to demonstrate and make you more familiar with the problem-analysis and algorithm-design techniques.

1.12 Programming Example 1

1.12.1 Design an Algorithm to Find the Perimeter and Area of a Rectangle

Problem Analysis

A rectangle is a four-sided figure with 2 equal lengths and 2 equal breadths as shown below:

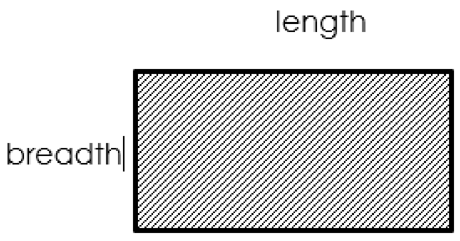

Figure 1.5: Rectangle

Perimeter is the total distance around the rectangle: length + length + breadth + breadth. The area is the shaded space.

Expected Output(s)

Perimeter, area

Given Input(s)

Length, breadth

Required Processing(s)

- Perimeter = length + length + breadth + breadth
- Perimeter = 2xlength + 2xbreadth
- Perimeter = 2x (length + breadth)
- Area = length x breadth

(NOTE: Perimeter, area, length, and breadth are variables, i.e., they will hold any value. A variable is an identifier in an algorithm or a program).

Having understood the problem, developing an algorithm is easier.

Algorithm Design

```
1. Start
2. Get the length of the rectangle into variable length
3. Get the breadth of the rectangle into variable breadth
4. Find the perimeter: perimeter = 2x (length + breadth)
5. Find the area: area = length x breadth
6. Display perimeter and area
7. Stop
```

C++ Code

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

int main()
{
    double length, breadth;

    cout << "Enter the length of the rectangle: ";
    cin >> length;

    cout << "Enter the breadth of the rectangle: ";
    cin >> breadth;

    double perimeter = 2 * (length + breadth);
    double area = length * breadth;

    cout << "The perimeter of the rectangle is: " << perimeter << endl;
    cout << "The area of the rectangle is: " << area << endl;

    return 0;
}
```

Example Output

```
Enter the length of the rectangle: 45
Enter the breadth of the rectangle: 30
The perimeter of the rectangle is: 150
The area of the rectangle is: 1350
```

Explanation

The program prompts the user to enter the length and the breadth of the rectangle using the `cout` statement and stores the input in the `length` and `breadth` variables using the `cin` statement. Then it calculates the perimeter of the rectangle by multiplying the sum of the length and the breadth of the rectangle by 2, and it calculates the area of the rectangle by multiplying the length and the breadth of the rectangle. Finally, the code uses the `cout` statement to display the calculated perimeter and area of the rectangle.

1.12.2 Exercise

Run the program and see how it works.

1.13 Programming Example 2

1.13.1 Design an Algorithm that Calculates Sales Tax and Final Price of an Item Sold in a Supermarket in Nigeria

The sales tax (or Value Added Tax (VAT)) is calculated as follows: 7.5% of the item-price.

Problem Analysis

Expected Output(s)

Sales tax, final price

Given Input(s)

To get the expected outputs, we need these inputs:

- Item price

Required Processing(s)

- Sales tax = 7.5/100 * item price
- Final price = item price + sales tax

Algorithm Design

1. Get the item's price into variable itemPrice
2. Find salesTax: salesTax = 7.5/100 * itemPrice
3. Find the finalPrice: finalPrice = itemPrice + salesTax
4. Display salesTax
5. Display finalPrice
6. Stop

C++ Code

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

int main()
{
    double itemPrice;
    double salesTax;
    double finalPrice;
    const double TAX_RATE = 0.075; // 7.5% sales tax rate

    cout << "Enter the price of the item: ";
    cin >> itemPrice;

    salesTax = itemPrice * TAX_RATE;
    finalPrice = itemPrice + salesTax;

    cout << "The sales tax for this item is: " << fixed << setprecision
       ↪ (2) << salesTax << endl;
    cout << "The final price of this item is: " << fixed <<
       ↪ setprecision(2) << finalPrice << endl;

    return 0;
}
```

Example Output

```
Enter the price of the item: 800
The sales tax for this item is: 60.00
The final price of this item is: 860.00
```

Explanation

This code first declares three variables: `itemPrice`, `salesTax`, and `finalPrice`. The variable `itemPrice` stores the price of the item entered by the user, `salesTax` stores the calculated sales tax, and `finalPrice` stores the final price of the item, which is the sum of the item price and the sales tax. The sales tax rate is stored in a constant variable `TAX_RATE` which is set to 7.5%. The code prompts the user to enter the price of the item using the `cout` statement and stores the input in the `itemPrice` variable using the `cin` statement. Then it calculates the sales tax by multiplying the item price by the tax rate, and it calculates the final price by adding the sales tax to the item price. Finally, the code uses the `cout` statement to display the calculated sales tax and final price of the item.

1.13.2 Exercise

Run the program and see how it works.

1.14 Programming Example 3

1.14.1 Design an Algorithm to Play a Number Guessing Game

The object is to randomly generate an integer greater than or equal to 0 and less than 100. Then, prompt the player (user) to guess the number. If the player guesses the number correctly, output an appropriate message. If the guessed number is less than the random number by the computer, output the message: "Your guess is lower than the number. Guess again." Otherwise, output the message: "Your guess is higher than the number. Guess again". This continues until the user guesses the random number correctly.

Problem Analysis

Generate a random number; suppose `randomNum` stands for a random number, and `guess` for the number entered by the player. After the user has entered a value for the variable `guess`, you can compare this number, as follows, until the user has entered the correct number:

- If `guess < randomNum`
- Print "Your guess is lower than the number. Guess again".
- Otherwise
- Print "Your guess is higher. Guess again".

Algorithm Design

1. Generate `randomNumber`, a number between 0-99
2. Repeat the following steps until the player has guessed the correct number:
3. Prompt the player to enter a value for a guess
4. If (`guess < randomNumber`)
5. Print “Your guess is lower than the number. Guess again!”
6. Otherwise
7. Print “Your guess is higher than the number. Guess again”.
8. Print "Congratulations! You guessed the number correctly!".

Note: The `if` and `repeat` instructions we saw in the last two programming exercises are known as selection and repetition logic structures that enable us to develop powerful algorithms and programs. You will learn how to implement such logic structures in your C++ programs in Chapters 3 and 4. But before that, have a taste of the “fun” of learning to develop codes in C++, here is the C++ program for the above game. Enter it on a system or an Android phone and run it.

C++ Code

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

int main()
{
    srand(time(0));
    int randomNumber = rand() % 100; //generate random number between 0 and 99
    int guess;

    cout << "Welcome to the Number Guessing Game!" << endl;
    cout << "I have generated a random number between 0 and 99." << endl;
    cout << "Try to guess the number: ";

    while (cin >> guess)
    {
        if (guess == randomNumber)
        {
            cout << "Congratulations! You guessed the number correctly! " << endl;
            break;
        }
        else if (guess < randomNumber)
        {
            cout << "Your guess is lower than the number. Guess again: ";
        }
        else
        {
            cout << "Your guess is higher than the number. Guess again: ";
        }
    }
    return 0;
}
```

Example Output

```
Welcome to the Number Guessing Game!
I have generated a random number between 0 and 99.
Try to guess the number: 50
Your guess is higher than the number. Guess again: 25
Your guess is higher than the number. Guess again: 13
Your guess is higher than the number. Guess again: 9
Your guess is higher than the number. Guess again: 5
Your guess is lower than the number. Guess again: 6
Congratulations! You guessed the number correctly!
```

Explanation

The program uses the C++ standard library's `iostream`, `cstdlib`, and `ctime` headers to input and output data, generate a random number, and seed the random number generator with the current time respectively. It creates a variable called `randomNumber` and assigns it a random number between 0 and 99 using the `rand() % 100` function. Then it creates a variable `guess` which is initially empty, the program prompts the user to guess the number, and it uses the `cin » guess` statement to take the user's input and store it in the `guess` variable. The program then uses a `while` loop to keep prompting the user to guess the number until they guess the correct number. If the guess matches the random number, it will output the message "Congratulations! You guessed the number correctly!" Otherwise, if the guess is less than the random number, it will output the message "Your guess is lower than the number. Guess again."

1.15 Programming Methodology

Programming methodologies are principles, techniques, or approaches that programmers follow in the development of programs. Two popular approaches to programming are structured programming and object-oriented programming.

1.15.1 Structured Programming

As earlier mentioned, most programming problems are complex. Dividing such a problem into smaller sub-problems is called structured design. Each sub-problem is analyzed and a solution is obtained to solve the sub-problem. The solution to all the sub-problems is combined to solve the overall problem. The process of implementing a structured design is structured programming. Other names for this programming application are top-down design, stepwise refinement, and modular programming.

1.15.2 Object-Oriented Programming (OOP)

Object-oriented programming (OOP) is a widely-used programming methodology. In OOP, the first step in problem-solving is to identify the components called objects in the problem domain and the interactions between these objects.

For example, suppose you want to write a program that automates the video rental process for a local video store, the two main objects are video and customer. After identifying the objects, the next step is to specify or identify for each object the relevant data (attributes) and possible operations to be performed on, or by, the object. For example, for a video object, the data might include:

- Movie name
- Starring actors
- Producer
- Producing company
- Number of copies in stock

Some of the operations on a video object may include:

- Checking the name of the movie
- Reducing the number of copies in stock by one after a copy is rented
- Increasing the number of copies by one after a customer returns a particular video

Can you think of some data attributes for the customer object and some of its operations?

As you must have realized now, in OOP each object consists of data and operations on that data. An object combines data and operations into a single unit. In OOP, the final program is a collection of interacting objects.

For some problems, the structured approach to program design is very effective while other problems are better addressed by OOP. For example, if a problem requires manipulating a set of numbers with mathematical functions, you might use the structured design approach and outline the steps required to obtain the solution.

1.16 Chapter Quick Reminder

- Computer languages:
  - Machine language: The most basic, consisting of 0s and 1s
  - Assembly language: Uses easy-to-remember instructions (mnemonics)
  - High-level languages: Like C++
- Language processing tools:
  - Assemblers: Translate assembly language to machine language
  - Compilers: Translate high-level language (source code) to machine language (object code)
  - Linkers: Link object code with SDK libraries to form executable code
- C++ program execution steps:
  - Edit
  - Pre-processing
  - Compile
  - Link
  - Load (transfer executable code to main memory)
  - Execute
- Algorithm: A finite step-by-step procedure for problem-solving or task performance
- Problem-solving process:
  - Analyze the problem
  - Design the algorithm
  - Implement the algorithm using a programming language
- Structured programming approach:
  - Easier to understand, debug, and test
  - Divides problem into smaller sub-problems
  - Solves each sub-problem
  - Combines solutions to form the overall solution
- Object-Oriented Programming (OOP):

  - Program is a collection of interacting objects
  - Objects consist of:
    - Data
    - Operations on the data

1.17 Review Exercises with Answers

1.17.1 Why do you need to translate a program written in a high-level language into machine language?

Answer: High-level languages need to be translated into machine language so the computer can execute them.

1.17.2 What are the advantages of employing a programming approach like structured programming over directly writing a program in a high-level language?

Answer: Structured programming provides better organization, readability, and maintainability of code, making it easier to debug and modify.

1.17.3 What is the output of the following C++ program?

```cpp
#include<iostream>
using namespace std;
int main()
{
    cout << "This is exercise B." << endl;
    cout << "In C++, the multiplication symbol is *" << endl;
    cout << "2 + 3 * 5 = " << 2 + 3 * 5 << endl;
    return 0;
}
```

Answer: When this program runs, it will output the following lines on the console (screen):

```
This is exercise B.
In C++, the multiplication symbol is *
2 + 3 * 5 = 17
```

The first `cout` statement outputs the string "This is exercise B." on the console. The second `cout` statement outputs the string "In C++, the multiplication symbol is *" on the console. The third `cout` statement outputs the string "2 + 3 * 5 = " followed by the result of the expression `2 + 3 * 5` which is `2 + 15 = 17`, on the console. The `endl` is used to print a new line after each statement.

Note that in C++ and most programming languages, multiplications and divisions are performed before additions and subtractions in an arithmetic expression, this is known as operator precedence. You will learn more about that in the next chapter.

1.18 Programming Exercises

1.18.1 Write an algorithm using pseudocode for a program that calculates and displays the current balance in a bank account. The program accepts from the user the starting balance, the total naira amount of deposits, and the total dollar amount of withdrawals made.

Solution

Problem Analysis:

- Input = start_balance, deposit, withdrawal
- Output = current_balance
- Processing: current_balance = start_balance + deposit − withdrawal

Algorithm in Pseudocode:

```
1. DISPLAY "Enter starting balance: "
2. INPUT start_balance
3. DISPLAY "Enter total deposits: "
4. INPUT deposit
5. DISPLAY "Enter total withdrawals: "
6. INPUT withdrawal
7. current_balance = start_balance + deposit - withdrawal
8. DISPLAY "Current balance: NGN", current_balance
9. END
```

1.18.2 Design an algorithm to find the weighted average of four test scores. The four test scores and their respective weights are given below:

- testScores1 weight = 0.20
- testScores2 weight = 0.35
- testScores3 weight = 0.15
- testScores4 weight = 0.30

1.18.3 The cost of an international call from New York to Abuja is calculated as follows: a connection fee of $1.99 for the first 3 minutes and $0.45 for each additional minute. Design an algorithm that asks the user to enter the number of minutes the call lasted. The algorithm then uses the number of minutes to calculate the amount due.

1.18.4 You are given a list of students' names and their test scores. Design an algorithm that does the following:

- Calculate the average score.
- Determine and print the names of all the students whose test scores are below the average.
- Print the names of all the students whose test scores are equal to the highest test score.

HINT: You can divide this problem into sub-problems as follows. The first sub-problem determines the average test score. The second sub-problem determines and prints the names of all students whose test scores are below the average. The third sub-problem determines the highest test score. The fourth sub-problem determines and prints the names of all the students who had the highest test scores. The main algorithm combines the solution of all the sub-problems.

Chapter 2

Basic Elements of C++

2.1 Objectives

In this chapter, you will:

- Become familiar with the basic components of a C++ program.
- Explore simple data types.
- Discover how to use arithmetic operators.
- Examine how the computer evaluates arithmetic operators in C++.
- Learn what an assignment statement is and what it does.
- Become familiar with the string data type.
- Discover how to input data into memory with input statements.
- Become familiar with the use of increment and decrement operators.
- Examine ways to output results using output statements.
- Learn how to use preprocessor directives and why they are useful.
- Explore how to properly structure a program including using comments to document a program.
- Learn about file input and output.
- Learn how to write a C++ program.

2.2 Basic Components of a C++ Program

A C++ program contains subprograms called functions. Some of the functions come with the C++ language and they are called predefined or standard functions, while other functions written by programmers are called programmer-defined functions. Each function, whatever its type, accomplishes a task when executed. You will learn more about functions in Chapter 5.

Every C++ program has a function called `main`. This is where everything starts or stops in a C++ program.

Let us consider a simple C++ program:

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

int main()
{
    cout << "Welcome to C++ program" << endl;
    return 0;
}
```

If you run the above program using a C++ compiler on a computer or an Android phone, what will be displayed on the screen? Try it out to confirm your answer.

2.3 Components of the C++ Language

Learning C++ is like learning a foreign language. If someone writes a letter to you in a foreign language, it may be strange to you. You can neither read nor write the language. To read or write sentences in such a language, you will need to learn its alphabet, words, and grammar.

This is true of learning any programming language like C++. To read or write statements in C++, you must learn C++ character sets, words, special symbols, syntax, and semantic rules. You also need to understand the meaning of the basic components of the language. The syntax rules tell you which characters, words, symbols, and statements are legal and which are not, while the semantic rules tell you the meaning of C++ words, symbols, and instructions.

The smallest meaningful unit of a program, written in any programming language, is a token. C++ tokens are divided into special symbols, word symbols, and identifiers.

2.3.1 Special Symbols

- +, -, *, /
- ., ;, ?, ,
- <=, !=, ==, >=

The first row includes symbols for addition, subtraction, multiplication, and division. The second row consists of punctuation marks. In C++, commas are used to separate items in a list. Semicolons are used to end a C++ statement. Note that a blank, which is not shown above, is also a special symbol. You create a blank symbol by pressing the space bar (once) on the keyboard.

The third row consists of the tokens made up of 2 characters which are regarded as a single symbol. No character can come between the two characters, not even a blank.

2.3.2 Word Symbols

Some of the word symbols include the following: `int`, `float`, `double`, `char`, `const`, `void`, `return`, `main`, `using`, `include`, `namespace`, etc.

2.3.3 Identifiers

Another category of token you will find in a C++ program is identifiers. These are names of things that appear in a program such as variables, constants, and functions. Some identifiers are predefined; others are user or programmer-defined. Examples of predefined identifiers are `cout` and `cin`, for generating output and input data on standard output and input devices respectively. All identifiers must obey the C++ rules for forming identifiers.

The rules are:

- Identifiers can be made of only letters, digits, and underscore characters. No other symbols are allowed.
- It must not begin with a digit. Though it can begin with an underscore, it is advisable not to do so because it can lead to error.
- The identifier can be any length. But some compilers have a specific maximum significant number of characters.

Example

The following are legal identifiers in C++:

- `first`
- `conversion`
- `pageRate`
- `convert1`

Some illegal identifiers and why they are wrong are given in Table 2.1.

Identifier	Reason for Invalidity
`1st`	Begins with a digit
`first name`	Contains a space
`first-name`	Contains a hyphen

Table 2.1: Example of Invalid Identifiers in C++

NOTE: C++ is case-sensitive – that is, upper case (capital letters) and lower case (small letters) are considered different. Thus, the identifier `NUMBER` is not the same as the identifier `number`. Similarly, the identifiers `X` and `x` are different.

Therefore, you will agree that a programming language can be defined as a set of rules, symbols, special symbols, word symbols, and identifiers.

2.4 Data Types

Programs are written to manipulate data. Such data are of various forms. A program that deals with student scores will process numbers. A program that sorts a list of names will process text (or strings) data. The program that processes students' scores can add, subtract, or divide the numeric data while the program that sorts names cannot do such arithmetic operations. So, you will see that the type of data determines the operations that the computer can perform on such data.

We can define the data type thus:

- Data type: a set of values together with a set of operations.

C++ data types fall into these 3 classes:

- Simple data type
- Structured data type
- Pointers

2.4.1 Simple Data Types

Simple data types are basic data types from which structured data types are formed. These include:

- Integral: a data type that deals with integers, and numbers without a decimal point.
- Floating point: a data type that deals with decimal numbers.
- Enumeration type: a user or programmer-defined data type.

The integral types are further classified as shown in the table:

Data Type	Storage Size
`int`	4 bytes
`bool`	1 byte
`char`	1 byte

Table 2.2: Integer Data Types and Their Required Storage Sizes

Table 2.2 Integer Data types and their required storage sizes

+

int Data Type

This is used to store integers, i.e., whole numbers without decimal parts. Examples of integers in C++ are -6272, 0, -67, 78, and +763. No commas are allowed in integers; so, you can write 36782 and not 36,782. The latter number will be regarded to be 2 integers 36 and 782 as the comma is known to be a separator.

bool Data Type

This has two possible values: true and false. This enables programs to store logical values and manipulate logical expressions.

Char Data Type

It is the smallest integral data type. It is used for storing small numbers (-128 to 127). In addition, the char data type is used to represent characters; i.e., letters, digits and special symbols. Examples of values belonging to the char data type are as follows:

- 'A', 'a', 'g', '*', '+', '$', '.', '&'

The char data type allows only the symbol or character to be placed between the single quotation marks. Thus, the value "abc" is not of type char.

Floating-point Data Types

This is used to present real numbers. Examples of a machine's way of representing real numbers in C++ are given in the table below:

Data Type	**Range**
Float	-3.4E-38 to 3.4E+38
Double	-1.7E-308 to 1.7E+308

Table 2.3: Representing real (decimal) numbers in C++

C++ provides three types for manipulating real numbers: float, double and long double. Just like integral data types, floats, double and long differ in the range of values each data type can represent on the computer.

- Float: it represents any decimal number between -3.4E-38 and 3.4E+38. It is allocated four (4) bytes.
- Double: It represents any real numbers between -1.7E-308 and 1.7E+308. It is allocated 8 bytes.

However, the maximum or minimum range of values for both float and double are system-dependent. Float and double also differ in the number of decimal places each allows. Float allows 6 or 7 decimal places while double allows up to 15 decimal places.

2.5 Arithmetic Operators

One of the most-important works the computer does is calculating using integer and floating-point types, following the rules of the arithmetic operations.

There are five arithmetic operators:

- + Addition.
- - Subtraction
- * Multiplication
- / Division
- % Modulus arithmetic

+, -, *, and / work on both integer and floating-point numbers. Modulus (%) works on only integer numbers to give the remainder. That is, 4%5 is 4. On the other hand, / gives the integer quotient as a result of division involving two integers. That is, integer division makes the computer truncate (or remove) any fractional (or decimal) part in the answer, and there is no rounding. Hence, 4/5 results in 0 and not 0.8. However, division involving floating-point numbers results in the floating-point quotient.

Given this arithmetic expression: $A + B$. A and B are operands.

An operator that uses only one operand is called a unary operator while operators that work on two operands are called binary operators.

This implies while %, / and * are binary operators, + and – are both unary and binary operators.

2.6 Examples

Table 2.4 presents examples of arithmetic expressions in C++ and how they are evaluated by the computer.

Expression	Evaluation
2+5	7
13+89	102
34-20	14
45-90	-45
2*7	14
5/2	2
14/7	2
34%5	4
-34%5	-4
34%-5	4
-34%-5	-4
4%6	4

Table 2.4: Examples of arithmetic expressions and their evaluations

Note that, in the divisions, 34/5 and $-34/-5$, the quotients, which are 6, are the same, but the remainders are different. What rules do you notice at work here? When one of the numbers is negative, what happens to the quotient? What happens to the remainder?

Consider and run the program below to verify the evaluation of arithmetic expressions in Table 2.4 and see how arithmetic operators work in a C++ program.

```
#include<iostream>
using namespace std;
int main ( )
{
 cout<<"2+5="<<2+5<<endl;
 cout<<"13+89="<<13+89<<endl;
 cout<<"34-20="<<34-20<<endl;
 cout<<"45-90="<<45-90<<endl;
 cout<<"2*7="<<2*7<<endl;
 cout<<"5/2="<<5/2<<endl;
 cout<<"14/7="<<14/7<<endl;
 cout<<"34%5="<<34%5<<endl;
 cout<<"-34%5="<<-34%5<<endl;
 cout<<"34%-5="<<34%-5<<endl;
 cout<<"-34%-5="<<-34%-5<<endl;
 cout<<"4%6="<<4%6<<endl;
 return 0;
}
```

What output did you get? Are they as in Table 2.4?

Consider and Run the Program Below

Now consider and run the program below that works on floating point numbers. It is important to note the way things work with floating point numbers.

```
#include <iostream>
using namespace std;
int main ()
{
    cout<< "5.0+3.5=" << 5.0 + 3.5 << endl;
    cout<< "3.0+9.4=" << 3.0 + 9.4 << endl;
    cout<< "16.3 - 5.2=" << 16.3 - 5.2 << endl;
    cout<< "4.2 * 2.5=" << 4.2 * 2.5 << endl;
    cout<< "5.0 / 2.0=" << 5.0 / 2.0 << endl;
    cout<< "34.5 / 6.0=" << 34.5 / 6.0 << endl;
    cout<< "34.5 / 6.5=" << 34.5 / 6.5 << endl;
    return 0;
}
```

Sample Run Output

```
5.0 + 3.5 = 8.5
3.0 + 9.4 = 12.4
16.3 - 5.2 = 11.1
4.2 * 2.5 = 10.5
5.0 / 2.0 = 2.5
34.5 / 6.0 = 5.75
34.5 / 6.5 = 5.30769
```

2.7 Operator's Precedence

When there is more than one operator in an arithmetic expression, C++ uses the operator precedence rule to evaluate the expression. For arithmetic operators we have considered so far, see Table 3.3 for their precedence rule.

Level	Operator	Associativity
5	+, -	Left to Right
6	*, /, %	Left to Right

Table 2.5: Operator precedence for arithmetic operators

NOTE: Arithmetic operators are not the only operators we have in C++. These two groups of arithmetic occupy levels 5 and 6. Others occupy levels 1-4 before these arithmetic operators, and others after in levels 7 to 18.

The associativity rule applies when operators in the same level of precedence are present in an expression. From table 3.3, the associativity rule for the levels 5 and 6 arithmetic operations is Left to Right. This means where you have a $+$ and a $-$ in an expression, the evaluation is done from the leftmost operator to the rightmost.

When you enclose an expression in parentheses, this has higher precedence over other precedence groups; i.e., the expression in the parentheses will be evaluated first. However, within the parentheses, the precedence rule in table 3.3 holds.

2.8 Expressions in C++

- Integral expression: where all values are integer.
- Floating-point expression: where all values are floating points.
- Mixed expression: where there is a mix of integral and floating point values.

Expression Type	Description
Integral expression	All values are integer
Floating-point expression	All values are floating points
Mixed expression	Mix of integral and floating point values

Table 2.6: Rules for evaluating expressions in C++

2.9 Type Conversion (Casting)

In the previous example on the mixed expression, we learnt that where an operator has integral and floating-point operands, the integral value is automatically converted to a floating point. This is called implicit type coercion. C++ provides a means for you to explicitly indicate you want to convert a value from one data type to another. This is done by the use of a cast operator. The format for explicit type conversion is stated thus:

```
static_cast<dataType name> (expression)
```

This is how it works: First, the expression inside the parenthesis is evaluated. Its value is then converted to the data type indicated by `<dataType name>`. Note that `static_cast` is a reserved word in C++.

2.9.1 Examples

Expression	Result
static_cast<double>(15)	15.0
static_cast<int>(7.6)	7

Table 2.7: Some examples of explicit type conversions

NOTE: Type Cast operator can also be written thus:

```
double (15) = 15.0
int (7.6) = 7
```

You can also use cast operators to explicitly convert char data value into data and vice versa. To convert char data values into data, C++ uses a collating sequence in the ASCII character set. For example, looking up the ASCII table, you will see that `static_cast<int>('A')` is 65 and `static_cast<int>('8')` is 56. Similarly, `static_cast<char>(65)` is 'A" and `static_cast<char>(56)` is '8'.

2.10 String Type

A string type is a programmer-defined data type for storing strings. A string is a sequence of 0 or more characters enclosed in double quotation marks. A string containing no character is called a null or empty string. The following are examples of string.

- "Bello"
- "Zungeru"
- " "

Every character in a string has a position. The position of the first character is 0, the second is 1, and so on. The length of a string is the number of characters in it. Each space is counted as a character as in this string: "John_Bello"

- position of J is 0
- position of n is 3
- position of first l is 7
- length of the string is = 10

For string: "It is a beautiful day.", the length is 22.

2.11 Input

Data must be loaded into memory before it can be processed. There are two ways you can make the computer do this in your C++ program:

- Instruct the computer to allocate memory space.
- Include statements in the program to put data into the allocated memory

Allocating of memory is done through the use of named constant and variables.

2.11.1 Constants

A named constant is a memory location whose content is not allowed to change during program execution. The C++ syntax for declaring a named constant is:

`const dataType Identifier = value;`

Examples

```
const double CONVERSION = 2.54;
const int NO_OF_STUDENTS = 20;
const char BLANK = 11;
const double PAY_RATE = 15.75;
```

2.11.2 Variables

A variable refers to a memory location whose value or content may change during program execution. The syntax for declaring one variable or multiple variables is:

`dataType identifier, identifier, ......;`

Example

```
double amountDue;
int counter, Age;
string name;
```

2.11.3 Putting data into variable

This can be done through:

- Use of C++ assignment statements
- Use input (read) statements

2.11.4 Assignment Statement

The syntax for writing assignments in C++ is:

```
Variable = Expression;
```

In an assignment statement, the value of an expression should match the data type of the variable. The expression of the right is evaluated and the result is assigned to the variable. We say a variable is initialized the first time a value is assigned to it. In C++, = is called the assignment operator.

Example Given the following variable declarations:

```
int i, j;
double sale;
char first;
string str;
```

The following are valid assignment statements:

```
i = 4;
j = 4 * 5 - 11;
sale = 0.02 * 1000;
first = 'D';
str = "it is a sunny day.";
```

Now, read the program and try to figure out what it does. Then run it to see the output.

```
#include<iostream>
#include<string>
using namespace std;
int main(  )
{
    int i, j;
    double sale;
    char first;
    string str;
    i = 4;
    cout << "i=" << i << endl;
    j = 4 * 5 - 11;
    cout << "j=" << j << endl;
    sale = 0.02 * 1000;
    cout << "sale=" << sale << endl;
    first = 'D';
    cout << "first=" << first << endl;
    str = "it is a sunny day";
    cout << "str=" << str << endl;
    return 0;
}
```

NOTE: Suppose x, y, and z are int variables, the following is a legal statement in C++:

```
x = y = z;
```

Remember, = is an operator and its associativity is from right to left. So, the above statement is executed thus: the value in variable z is assigned to the memory location (or variable) y. Subsequently, the value in y is assigned to x. Therefore, at the end of the execution of the statement, whatever the initial values of the three variables, x, y, and z will contain the same values.

2.12 Input Read Statement

Data can also be put into variables from the standard input device (usually the keyboard). Inputting data into variables from the standard input device is accomplished by the use of `cin` and the operator ≫. The syntax for the input statement is:

```
cin >> variable >> variable . . . . .;
```

In C++, ≫ is called the stream extraction operator. The syntax above shows that we can put data into one or more variables by use of one input statement. That is what the ellipsis (. . . . .) indicates.

Example: Suppose `miles` is a variable of type double. Suppose a user enters 73.65 on the keyboard

```
cin >> miles;
```

The above input statement indicates that 73.65 is loaded into a variable (or memory location) called `miles`.

2.12.1 Example of a program using cin.

```
#include <iostream>
using namespace std;
int main ( )
{
    int feet;
    int inches;
    cout << "Enter two integers separated by spaces";
    cin >> feet >> inches;
    cout << "feet = " << feet << endl;
    cout << "inches = " << inches << endl;
    return 0;
}
```

2.12.2 Another example that shows how to read a string and numeric data:

```
#include<iostream>
using namespace std;
int main ( )
{
    string firstName;        // line1
    string lastName;        // line2
    int age;                      // line3
    double weight;          // line4
    cout << "Enter the first name, last name, age and weight, separated
       ↪ by space" << endl;
    cin >> firstName >> lastName >> age >> weight;
    cout << "Name: " << firstName << " " << lastName << endl;
    cout << "Age: " << age << endl;
    cout << "Weight: " << weight << endl;
    return 0;
}
```

Note the // use in labelling some of the lines in the program. These are called single lines (or on-line comments). Single-line comments in C++ begin with //. Comments are explanations or information that help to document various aspects of a program so that the reader can easily understand how a program works. Comments are ignored by the compiler since they are not executable statements. Another way for inserting comments in C++ is the use of /* or */. For instance, writing this on a line in a program:

```
/* this is a comment in C++ */
```

/* */ is called a multiple-line comment as we can write the previous comment thus:

```
/* this is multiple lines
comment in C++ */
```

2.12.3 Exercise

Consider the following declarations:

```
int firstNum, secondNum;
double z;
char ch;
string name;
```

Also, suppose that the following statements are executed in the order given.

```
firstNum = 4;
secondNum = 2 * firstNum + 6;
z = (firstNum + 1) / 2.0;
ch = 'A';
cin >> secondNum;
cin >> z;
firstNum = 2 * secondNum + static_cast<int>(z);
cin >> name;
secondNum = secondNum + 1;
cin >> ch;
firstNum = firstNum + static_cast<int>(z);
z = firstNum - z;
```

In addition, suppose the input entered from the keyboard is:

`8 16.3 John D.`

You are required to determine the values of the declared variables after the last statement is executed.

2.13 Increment and Decrement Operators

Suppose, `count` is an int variable. The assignment statement

`count = count + 1;`

will increase the value kept in the variable `count` by 1. To execute this assignment, the computer first evaluates the expression on the right. Then, it assigns this value to the variable on the left, which is `count`. The same evaluation applies to `count = count - 1`, but here the value in the variable `count` is decreased by 1. To simplify the writing of these two types of assignment statements for incrementing or decrementing the values of variables, C++ provides the increment `++` and decrement `-` operators. The syntax of the two is given below.

2.13.1 Forms of Increment and Decrement Operators

Both increment and decrement operators have two forms each. The syntax for each of the two forms of these operators is given below:

- Pre-Increment: `++x`
- Post-Increment: `x++`
- Pre-Decrement: `-x`
- Post-Decrement: `x-`

All pre and post-operators increment or decrement a variable by 1. But the difference is in the timing of the incrementing or decrementing. Pre does the increment or decrement so that the new updated value of a variable is what is used in an expression or an assignment while post increment or decrement means the old or current value of the variable is used in an expression or assignment. Thereafter, the variable is incremented or decremented. Let us see some examples to clarify further the way these operators work.

Suppose that `x` and `y` are int variables, consider the following statements:

```
x = 5;
y = ++x;
```

The first statement assigns 5 to variable `x`. In the second statement, first, the value of `x` increments from 5 to 6 by the pre-increment operator. Then, the new value of `x` is assigned to `y`. Therefore, after the execution of the above statements, both `x` and `y` have the value of 6.

Now consider the following statements:

```
x = 5;
y = x++;
```

As before, the first statement assigns 5 to variable `x`. To execute the second statement, first, the value of `x` which is 5 is assigned to `y`. Then, the value `x` is incremented by 1. Therefore, after the execution of the two statements, the value of `x` is 6, and the value of `y` is 5.

2.14 Outputs in C++ codes

Results in the memory can be shown in the standard output device (usually the screen) by the use of an output statement. This is accomplished via the use of `cout` and the operator ≪. The general syntax or format for writing it in C++ is:

```
cout << expression or manipulator << expression or manipulator . . . .
   ↪ .;
```

≪ is called the stream insertion operator. Generating output with `cout` follows two rules:

- The expression is evaluated and its value is printed (or displayed) at the current insertion point on the output device.
- A manipulator is used to format the output. The simplest manipulator is `endl`, which causes the insertion point to move to the beginning of the next line.

Examples of C++ Output

No.	STATEMENT	OUTPUT
1	`cout ≪ 29/4 ≪ endl;`	7
2	`cout ≪ "Hello there" ≪ endl;`	Hello there
3	`cout ≪ 12 ≪ endl;`	12
4	`cout ≪ "4+7" ≪ endl;`	4+7
5	`cout ≪ 4+7 ≪ endl;`	11
6	`cout ≪ "4+7 =" ≪ 4+7 ≪ endl;`	4+7 = 11
7	`cout ≪ "A" ≪ endl;`	A
8	`cout ≪ 2+3*5 ≪ endl;`	17
9	`cout ≪ "Hello\nthere." ≪ endl;`	Hello there.

Table 2.8: Examples of C++ Output

2.14.1 Escape Sequences

In statement 9 in the above table (Table 2.8), the newline character `'\n'` causes the insertion point to move to the beginning of the next line before printing the string `"there"`. As a result, `"Hello"` and `"there"` are printed on separate lines. In C++, the backslash (\) is called the *escape character*, and `'\n'` is the *Newline Escape Sequence*. Table 2.9 presents other commonly used escape sequences in C++.

ESCAPE SEQUENCE	NAME	DESCRIPTION
`\n`	Newline	Moves cursor to the beginning of the next line
`\t`	Tab	Moves the cursor to the next tab stop (8 spaces)
`\b`	Backspace	Moves the cursor one space back (left)
`\r`	Carriage return	Moves the cursor to the beginning of the current line
`\\`	Backslash	Prints a backslash (\)
`\'`	Single quotation	Prints a single quotation mark (')
`\"`	Double quotation	Prints a double quotation mark (")
`\v`	Vertical tab	Moves to the next vertical tab stop
`\a`	Bell (alert)	Generates a beep sound
`\f`	Form feed	Produces a blank page
`\?`	Question mark	Prints a question mark (?)

Table 2.9: Commonly Used C++ Escape Sequences

More Examples on Escape Sequences:

1.
```
cout << "The new line escape sequence is \\n." << endl;
```

generates this output:

```
The newline escape sequence is \n.
```

How? In the string, we have one escape sequence (\\). As you can see in the above Table 2.9, that is, the escape sequence with backlashes. This makes the computer print \. Since what follows \\ is n, this is added to the output so that we now have:

```
The newline escape sequence is \n
```

2.
```
cout << "The tab character is represented as\\'\\t\\'" << endl;
```

At first, this example may be confusing or difficult to evaluate and to know what will be printed. It is simple. Identify the escape sequences in the string one by one and then you will be able to state easily what the output will be.

So, let us identify the escape sequences in the string inside double quotes. Looking from left to right, there are three of them:

- \' generates '
- \\ generates \, (and after inserting the letter t in the output stream),
- \' generates '

Hence, by bringing all the evaluations together, the output will be:

```
The tab character is represented as '\t'
```

2.15 More on Assignment Statements

We have two types of assignment statements in C++:

- Simple assignment statements (the ones we have used in our programs).
- Compound assignment statements: This is written by combining the assignment operator (=) with other arithmetic operators (+,-,*,/,%) so that we have these operators: +=, -=, *=, /=, %=

The syntax for converting from simple to compound assignment statements:

```
variable = variable (arithmetic operator) expression
variable (arithmetic operator) = expression
```

Examples:

Simple assignment	Compound assignment
`x = x * y;`	`x *= y;`
`i = i + 5;`	`i += 5;`
`sum = sum + number;`	`sum += number;`
`x = x / (y + 5);`	`x /= (y + 5);`

Table 2.10: Examples of simple and compound assignment statements

2.16 File Input and Output (I/O)

So far, we have made use of the standard input and output stream variables (i.e., `cin` and `cout`) in our programs. This enabled us to get input from the keyboard and send output to the monitor screen. Often such programs are small. Imagine where we want our program to process or generate a large volume of data. This is where using files on internal or external storage devices becomes a feasible option.

Using files in C++ is similar to using standard input and output devices. You will need to indicate you are using files by including them in your program header file `fstream` (instead of `iostream`). `fstream` contains definitions for `ifstream` (input file stream, like `cin`) and `ofstream` (output file stream, like `cout`).

To use File I/O in your program is a 5-step process:

1. Include the header file `fstream` in the program.
2. Declare file stream variables.
3. Associate the file stream variables with the i/o sources.
4. Use the file stream variables with », « or other i/o functions.
5. Close the files.

Let us see an example of File i/o

STEP 1: Include the header file fstream

Write:

```
    #include <fstream>
```

STEP 2: Declare file stream variables.

e.g.

```
ifstream inData;
ofstream outData;
```

STEP 3: Associate the file stream variables with the i/o source.

The syntax for doing this looks like this:

```
filestream variable.open(SourceName);
```

where SourceName is the location and name of your input and output files. e.g. following the example in step 2 above and given that our files are resident in a drive labelled ‘a’ on the computer, then you write:

```
inData.open("a:\\prog.data");
outData.open("a:\\prog.out");
```

STEP 4: Use the file stream variables with », « or other I/O functions.

e.g.,

```
inData >> payRate;
outData << "The Paycheck is, $" << pay << endl;
```

STEP 5: Close the files

e.g.,

```
inData.close();
outData.close();
```

Closing these files releases them for use in other parts of your program. Additionally, closing an output file ensures all your output is written from the output buffer into the output file.

2.17 Chapter Quick Reminder:

- A C++ program is a collection of functions.
- Every C++ program has a function called `main`.
- In C++, identifiers are names of things.
- A C++ identifier consists of letters, digits and underscores, and must begin with a letter or underscore.
- Reserved words cannot be used as programmer-defined identifiers.
- All reserved words in C++ consist of lowercase (small) letters.
- The modulus operator (%) takes only integer operands.
- Arithmetic expressions are evaluated using the precedence rules and associativity of the arithmetic operators.
- All operands in an integral expression, or integer expression, are integers, and all operands in a floating-point expression are decimal numbers.
- A mixed expression consists of both integers and decimal numbers.
- When evaluating an operator in an expression, an integer is converted to a floating-point (decimal) number, with a decimal part of 0, only if the operator has mixed operands.
- The Cast operator converts values from one data type to another.
- During program execution, the contents of a named constant cannot be changed.
- A named constant is declared using the reserved word `const`.
- A named constant is initialized when it is declared.
- All variables must be declared before they are used.
- C++ does not automatically initialize variables.
- Every variable has a name, a value, a data type and a size.
- When a new value is assigned to a variable, the old value is destroyed.
- Only an assignment statement or an input (read) statement can change the value of a variable.
- In C++, » is called the stream extractor operator.
- Input from the standard input device is accomplished using `cin` and the stream extractor operator ».
- In C++, « is the stream insertion operator.
- The output of the program to the standard output device is accomplished by using `cout` and «.
- Outputting or accessing the value of a variable in a program does not destroy the contents of the variable.
- To use `cin` and `cout`, the program must include the header file `<iostream>` and either include the statement `using namespace std;` or refer to these identifiers as `std::cin` and `std::cout`.
- The manipulator `endl` positions the cursor to the beginning of the next line.
- A stream in C++ is an infinite sequence of characters from a source to a destination.
- An input stream is a stream from a source to a computer.
- An output stream is a stream from the computer to a destination.
- `cin` stands for common input, which is an input stream object, typically initialized to the standard input device, that is the keyboard.

- `cout` stands for common output is an output stream object, typically initialized to the standard output device, usually the screen.
- When inputting data into a variable, the extraction operator ≫ skips all the leading whitespace characters.
- The header file `fstream` contains the definition of `ifstream` and `ofstream`.
- For file I/O you must include the header file `fstream` in your program. You must also do the following: declare variables of type `ifstream` for file input and `ofstream` for file output; use the open statement to open input and output files.
- To close a file, as indicated in the `ifstream` variable `inFile`, you use the statement `inFile.close`. To close a file as indicated by the `ofstream` variable `outFile`, you must use the statement `outFile.close`.

Programming Exercises

1. The following programs have syntax errors. Correct them. On each successive line, assume that any preceding error has been corrected.

a.

```
#include <iostream>
const int PRIME = 11,213;
const RATE = 15.6
int main()
{
    int I,x,y,w;
    x = 7;
    y= 3;
    x = x +w;
    PRIME = x + PRIME
    cout<<PRIME<<endl
    wages = RATE * 36.75
    cout << "wages = << wages << endl;
    return 0;
}
```

A corrected version:

```
#include <iostream>
using namespace std;
int main()
{
    float PRIME = 11.213;
    const float RATE = 15.6;
    int x,y,w;
    float wages;
    x = 7;
    y= 3;
    x = x +w;
    PRIME = x + PRIME;
    cout<<PRIME<<endl;
    wages = RATE * 36.75;
    cout << "wages = "<< wages << endl;
    return 0;
}
```

Do you notice any syntax (or logical) error still?

b.

```
const char = BLANK = ' ';
const int ONE 5;
int main ( )
{
    int a,b, cc;
    a = ONE +5;
    b = a + BLANK;
    cc := a + ONE *2;
    a + cc = b;
    one = b + c;
    cout << "a = " << a << ", b = " << b", cc =" << cc <<endC
    return 0;
}
```

2. Write a program that prompts the user to enter five test scores and then print the average score. (Assume that test scores are decimal numbers).

3. Write a program that prints the following pattern on the screen:

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

4. Write a program that prompts the user to input a four-digit positive integer. The program then outputs the digits of the number one digit per line.

5. Write a program that prompts the user to input a number. The program should then output the number and a message, stating whether the number is positive, negative or zero.

6. Write a program that prompts the user to enter 3 numbers. The program then outputs the numbers in ascending order.

7. Write a program that takes as input given lengths expressed in feet and inches. The program should then convert and output the lengths in centimetres. Assume that the given lengths in feet and inches are integers.

Hints: 1 inch = 2.54 metres. 1 foot = 12 inches.

8. Given the input:

```
46 A 49
```

and the C++ code:

```
int x = 10, y = 18;
char z = 'A';
cin >> x >> y >> z;
cout << x << " " << y << " " << z;
```

What is the output?

9. The following program is supposed to read two numbers from a file named `input.dat` and write the sum of the numbers to a file named `output.dat`. However, it fails to do so. Rewrite the program so that it accomplishes what it is intended to do. (Also, include statements to close the files).

```
#include <iostream>
#include <fstream>
using namespace std;
int main ( )
{
    int num1, num2;
    ifstream infile;
    ofstream outfile;
    outfile.open("output.dat");
    infile.open("input.dat");
    infile >> num1 >> num2;
    outfile << "Sum = " << num1 + num2 << endl;
    infile.close();
    outfile.close();
    return 0;
}
```

Chapter 3

Control Structures 1 (Selection)

In this chapter, you will:

- Learn about control structures
- Examine relational and logical operators
- Explore how to form and evaluate logical (also called Boolean) expressions
- Discover how to use the selection structures If, If . . else, and switch in a program.

3.1 Control Structures

Control structures are programming or logical units that determine how instructions are executed in the program. There are three control structures: sequence, selection (or conditional), and repetition (also called, iteration or looping).

- In a sequence, structured program instructions are executed one after the other in the order they appear in the program.
- A selection structure makes program execution go in different paths, depending on a particular condition during program execution. The selection control structure gives the computer the power to handle various scenarios during program execution.
- Repetition structures are used when the computer needs to repeatedly execute or run the same processing instructions on large data. For instance, to process the results and generate a printout of millions of candidates that sat for an exam.

3.2 Relational Operators

Relational operators are used to compare between data values. They are used in forming relational expressions. The result of such expressions is either True (T) or False (F). In C++, false is 0, and true is any number greater than zero (i.e., 1,2, 3.5 etc.). This means you can assign a numeric value to a variable of the type bool.

The following are relational operators in C++:

Operator	**Description**
==	Equality
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
!=	Not equal to

Table 3.1: Relational Operators

3.2.1 Examples

- $4 < 5$ = True (which is 1)

- 9 != 9 = False (which is 0)
- 8.5 < 8 = False
- 9.5 <= 9.2 = False
- 12 >= 12 = True

3.2.2 Programming Exercises:

Given the following program:

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

int main()
{
    int firstNum, secondNum, thirdNum;
    bool result1, result2, result3;

    firstNum = 5;
    secondNum = 2 + 3 * 5;
    thirdNum = firstNum;

    result1 = firstNum > secondNum;
    result2 = thirdNum != secondNum;
    result3 = secondNum >= firstNum;

    cout << result1 << endl;
    cout << result2 << endl;
    cout << result3 << endl;
}
```

What values will be displayed on the screen?

Read the program below and consider how the boolean data type is used. What will be the output? Then run the program and see if you are correct.

```
#include <iostream>
using namespace std;
int main()
{
    int x1 = 10, x2 = 20, m = 2;
    bool b1, b2;
    b1 = x1 == x2; // false
    b2 = x1 < x2; // true
    cout << "b1 is = " << b1 << "\n";
    cout << "b2 is = " << b2 << "\n";
    bool b3 = true;

    if (b3)
        cout << "Yes" << "\n";
    else
        cout << "No" << "\n";

    int x3 = false + 5 * m - b3;
    cout << x3;

    return 0;
}
```

3.3 Logical (Boolean) Operators

Logical operators are like relational operators whose evaluation results in true or false. With logical operation, more powerful comparisons or conditions can be processed by the computer. Such expressions may include arithmetic and relational operators.

The available logical operators in C++ are:

Operator	**Description**
&&	Logical AND
||	Logical OR
!	Logical NOT

Table 3.2: Logical Operators

3.4 Order of Precedence of Operators

Here, by order of precedence, we mean which operations will be performed before another when we have an expression containing operators from different classes of operators. The table below presents the order precedence of the various types of operators you will use in C++ expressions.

Operator	**Precedence**
Postfix	++, –
Unary	+, -, !, ~, ++, –, &, *, (type)
Multiplicative	*, /, %
Additive	+, -
Shift	«, »
Relational	<, <=, >, >=
Equality	==, !=
Bitwise AND	&
Bitwise XOR	^
Bitwise OR	|
Logical AND	&&
Logical OR	||
Conditional	?:
Assignment	=, +=, -=, *=, /=, %=, «=, »=, &=, ^=, |=
Comma	,

Table 3.3: Order of evaluating operators from all classes

3.5 Types of Selection Structures

Different programming problems require particular types of selection structures in your C++ program. The following are three main forms of selection structures:

- One-way Selection
- Two-way Selection
- Multiple-way Selection

3.5.1 One-way Selection

One-way selection (or single alternative) structure has the form:

```
IF condition, Then:
    Execute Step(s) in segment A
End of IF structure
```

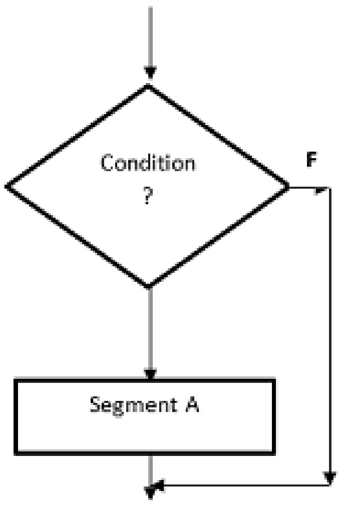

Figure 3.1: One-way Selection Structure

As shown in Figure 3.1, the computer will evaluate a condition. If the condition is true, segment A of steps or instructions is executed. If it is false, the computer will bypass segment A and continue with the step or instruction after segment A.

Note: segment here and in other selection structures refers to a group of one or more instructions that are to be executed as a unit of the program.

3.5.2 Two-way Selection (double alternative)

This structure has the form:

```
If condition, then:
    [Segment A]
Else
    [Segment B]
[End of IF structure]
```

If a condition is true, then step(s) in segment A will be executed; otherwise, step(s) in B will be executed.

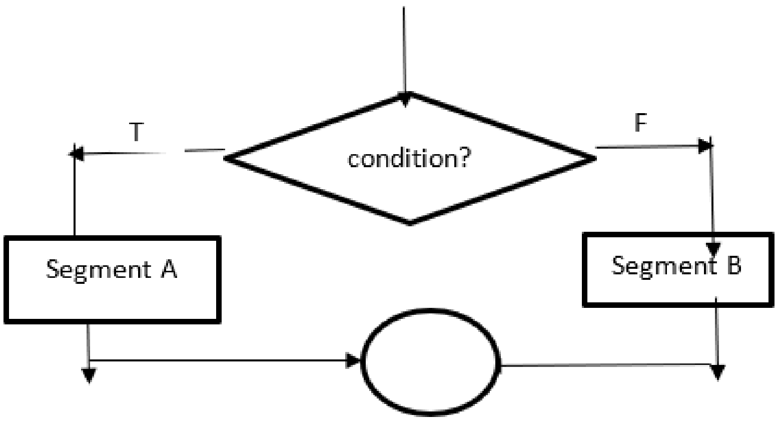

Figure 3.2: Two-way Selection Structure

3.5.3 Multiple-way Selection

has the form:

```
If condition-1, then
    [Segment A1,]
Else, if condition-2, then:
    [Segment A2]
    ...
Else, if condition-m, then;
    [Segment Am]
Else [Segment B]
end of if structure
```

The multiway selection is used to direct program execution where the computer needs to test for two or more conditions and execute some corresponding segments of step(s) when any one of the conditions holds. After executing that segment, program execution continues with the next step or instruction after the multiway selection. Note that (as indicated in Figure 3.3) only one of those conditions can be true when the computer runs a multiway selection. If none of the conditions is true, then the computer will execute the instruction(s) in segment B.

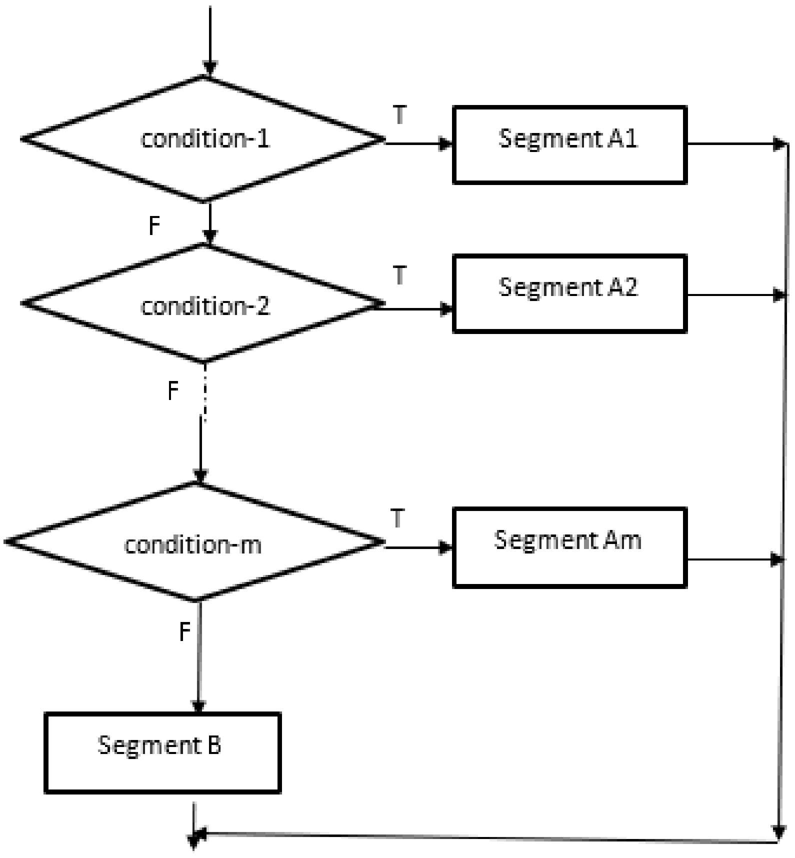

Figure 3.3: Multiple-way Selection Structure

3.5.4 One-way Selection C++ Syntax

The syntax for implementing a one-way selection in C++ is:

```
If (expression)
    Statement(s)
```

Note: Statement(s) refers to one or more executable statements. Remember, each must end with a semicolon. You should enclose a group of statements to be executed with open and close braces ({ }). If not, the computer assumes that the only statement to be executed is the one immediately after the if statement, even when you intended more than one statement to be executed if the expression is true.

Example

```
double score;
string grade;
if (score >= 75)
{
    grade = "AB";
}
```

3.5.5 Two-way Selection C++ Syntax

The syntax for the two-way selection is:

```
If (expression)
    statements(1)
else
    statements(2)
```

Example

```
if (hours > 40.0)
{
    wages = 40.0 * rate + 1.5 * rate * (hours - 40.0);
}
else
{
    wages = hours * rate;
}
```

3.5.6 Multiple-way Selection (Switch) C++ Syntax

This is called a switch in C++. The general syntax for the switch selection is:

```
switch (expression)
{
    case value1:
        Statements1
        break;
    case value2:
        Statements2
        break;
    case valuen:
        Statementsn
        break;
    default:
        Statements
}
```

Note: switch, case, break, and default are reserved words. Each Statements1 .... Statementsn refers to one or more statements. value1 .....valuen are possible values of the expression.

How switch works

First, the value of the expression is evaluated. When the value matches one of the case values (i.e., value1,......,valuen), the statements belonging to that case are executed by the computer until it meets the optional statement break for that case. If no break is used, the computer will execute statements under other subsequent case values until it meets a break or gets to the end of statements under case valuen. Otherwise, if the value of the expression does not match any of value1, ...,valuen, the statement(s) under default will be executed. If a switch structure has no default part, and if the value of the expression does not match any of the case values, the entire switch is skipped.

A break statement causes an immediate exit from the switch structure.

Example

```
switch (grade)
{
    case 'A':
        cout << "The grade is A";
        break;
    case 'B':
        cout << "The grade is B";
        break;
    case 'C':
        cout << "The grade is C";
        break;
    case 'D':
        cout << "The grade is D";
        break;
    case 'F':
        cout << "The grade is F";
        break;
    default:
        cout << "The grade is invalid";
}
```

3.6 Chapter Quick Reminder:

- Control structures alter or determine the flow of control in a program.
- The three control structures are sequence, selection and repetition.
- Selection structures implement decisions in a program.
- The relational operators are: == (equality), < (less than), <= (less than or equal to), > (greater than), >= (greater than or equal to), != (not equal to).
- Including a space between the symbols in relational operators creates syntax errors.
- A logical expression evaluates to 1 (or a non-zero value) or 0. The logical value 1 (or any non-zero value) is treated as true; the logical value 0 is treated as false.
- In C++, int variables can store values of a logical expression.
- In C++, bool variables are used to store the value of a logical expression.
- One-way selection syntax is:

  ```
  if (expression)
      statement(s)
  ```

  If the expression is true, statement(s) is executed; otherwise, the computer executes the statement immediately after statement(s).
- The two-way selection syntax is:

  ```
  if (expression)
      statement(s)1
  else
  ```

```
        statement(s)2
```

If the expression is true, statement(s)1 executes; otherwise, the computer executes the statement(s)2.

- A switch is used to handle multi-way selection.
- The execution of a break in a switch structure immediately makes the computer exit the structure.

3.7 Exercises

1. Correct the following code so that it prints the correct output:

```
if (score >= 60)
    cout << "You pass." << endl;
else;
    cout << "You fail." << endl;
```

2. Write C++ statements that display "Male" if the variable `gender` is 'M', "Female" if the `gender` is 'F', and "Invalid" otherwise.

3. Suppose the input is 5. What is the value of `alpha` after execution of the following C++ code:

```
cin >> alpha;
switch (alpha)
{
    case 1:
    case 2:
        alpha = alpha + 2;
        break;
    case 4:
        alpha++;
    case 5:
        alpha = 2 * alpha;
    case 6:
        alpha = alpha + 5;
        break;
    default:
        alpha--;
}
```

Chapter 4

Control Structures II (Repetition)

In this chapter, you will:

- Learn about repetition (looping) structures
- Explore how to end a loop
- Examine break and continue statements
- Discover how to form and use a nested control structure

4.1 Why is Repetition Needed?

Suppose you want to find the average of five numbers. From what you have learned so far, you can write this code segment:

```
double num1, num2, num3, num4, num5;
double sum, average;
cin >> num1 >> num2 >> num3 >> num4 >> num5;
sum = num1 + num2 + num3 + num4 + num5;
average = sum / 5;
```

But suppose you want to find the average of 100, 1000, or 1,000,000 numbers. How many variables will you need? You would have to declare many variables, and your programs will be very long, depending on the number of variables. You will see that this takes a huge amount of space and time.

This is where repetition comes to ease a difficult or complex situation. What we need is a code with a few statements in which some will be repeatedly executed. To address our previous challenge, we may have something like:

```
1 sum = 0;
2 cin >> num;
3 sum = sum + num;
```

In the above code, statements 2 and 3 will be repeatedly executed to add several numbers.

4.2 Types of Repetition Structures

Three types of repetition structures are available:

- While Looping Structure
- For Looping Structure
- Do While Looping Structure

4.2.1 While Loop Syntax

```
while (expression)
    statement(s)
```

How the While Loop Works

The expression in a while loop is a logical expression that produces a logical value true or false. If it is true, the statement(s) (the body of the loop) consisting of one or more statements is executed. The loop condition - expression - is then re-evaluated. If it is again true, statement(s) is executed again. This continues until the expression is no longer true.

Example

```
i = 0;
while (i <= 20)
{
    cout << i << " ";
    i = i + 5;
}
cout << endl;
```

In the above example, variable `i` is initialized to 0. When the computer evaluates the expression `(i <= 20)`, it finds it to be true and then executes the loop. Two outputs are generated: the value in variable `i`, which is 0, and a space. This implies a space is inserted between one value of `i` and the next. Then `i` is incremented by 5, and the new value of 5 is assigned to `i`. Then the computer evaluates the loop condition and, since it finds it true still, the loop is executed. This looping continues until the looping condition becomes false. Therefore, the output generated will be `0 5 10 ....` Can you complete the output?

4.2.2 Do...While Loop

Syntax for the Do-While Loop

```
do
    statement(s)
while (expression)
```

How the Do-While Loop Works

Statement(s) which consists of one or more statements is executed first, and then the expression is evaluated. Statement(s) is executed as long as the expression is true.

Example

```
i = 0;
do
{
    cout << i << " ";
    i = i + 5;
}
while (i <= 20);
```

4.2.3 For Loop

A loop that executes a known number of times.

Syntax

```
for (initial statement; loop condition; update statement)
    statement(s)
```

How the For Loop Works

The initial statement is executed. The loop condition is executed. If the loop condition is true, then:

- Execute statement(s).
- Execute the update statement.
- Repeat step 2 until the loop condition becomes false.

The initial statement usually initializes a variable.

Example

```
for (i = 0; i < 10; i++)
    cout << i << "\n";
cout << endl;
```

4.3 Break Statements

Used to exit out of a conditional or repetition structure. When the computer meets a break statement in a control structure, execution of the structure terminates, and the computer continues with the first statement after the control structure in a program.

Example

Run this program and see the output it displays on the screen:

```
#include <iostream>
using namespace std;
int main()
{
    int num = 0;
    int sum = 0;
    bool isNegative = false;
    cout << "Please enter any nonnegative number" << endl;
    cin >> num;
    while (!isNegative)
    {
        if (num < 0)
        {
            cout << "Negative number found in the data" << endl;
            isNegative = true;
        }
        else
        {
            sum = sum + num;
            cout << "Please enter next nonnegative number" << endl;
            cin >> num;
        }
    }
    cout << "The sum of the numbers you have entered is " << sum <<
       ↪ endl;
    return 0;
}
```

4.4 Continue Statement

It is used in a while, for, and do-while loop. When the continue statement is executed in a loop, it skips the remaining statements in the loop and proceeds with the next iteration of the loop.

In a while or do-while loop, the logical expression is evaluated immediately after the continue statement. In a for loop, the update statement is executed after the continue statement, and then the loop condition (loop-continue test) executes.

4.5 Chapter Quick Reminder

C++ has three looping structures: while, do-while, and for repetition structures.

- The syntax for the while structure is:

```
while (expression)
    statement(s)
```

- The syntax for do-while is:

```
do
    statement(s)
while (expression)
```

- The syntax of for structure is:

```
for (initial statement; loop condition; update statement)
    statement(s)
```

Both while and for structures are called pre-test loops because the looping condition is tested first, and if it is not true, the body of the loop will not be executed at all.

Do-while is called the post-test looping structure. The body of the loop is executed before the looping condition is tested. So do-while will be executed at least once.

Executing a break statement in the body of a loop immediately terminates the execution of the loop.

Executing a continue statement in the body of a loop makes the computer skip the remaining statement(s) in the body and it continues with the next iteration of the loop.

4.6 Exercises

What is the output of each of the following C++ codes?

1.
```
count = 1;
y = 100;
while (count < 100)
{
    y = y - 1;
    count++;
}
cout << "y = " << y << " and count = " << count << endl;
```

2.
```
num = 5;
while (num > 5)
    num = num + 2;
cout << num << endl;

num = 1;
while (num < 10)
```

```
    {
        cout << num << " ";
        num = num + 2;
    }
    cout << endl;
```

Chapter 5

Simple Data Structures in C++

Objectives

In this chapter, you will:

- Know how to declare, initialize, and use arrays in your program.
- Become familiar with pointers, how to declare and initialize them.
- Understand how to use pointers to access memory.
- Know how pointers are related to arrays and differentiate between them.
- Understand vectors, how to declare and initialize vectors.
- Know how to create a struct, instantiate, and use it in your program.

5.1 Arrays

An array in C++ is a collection of elements of the same type, stored in contiguous memory locations. The elements can be accessed by their index, which starts at 0.

There are several ways to define and initialise arrays in C++.

For example, the following declares an array of integers with 5 elements and initializes it later with five values:

```
int myArray[5];
myArray[0] = 1;
myArray[1] = 2;
myArray[2] = 3;
myArray[3] = 4;
myArray[4] = 5;
```

You can also declare and initialize an array at the same time. Here is an example:

```
int myArray[5] = {1, 2, 3, 4, 5};
```

In this example, `myArray` is an array of 5 integers, with the values 1, 2, 3, 4, 5.

Another way to perform the above is to use a loop. Here is an example:

```
int myArray[5];
for (int i = 0; i < 5; i++) {
    myArray[i] = i + 1;
}
```

The elements of the array can be accessed using the square brackets, like this:

```
cout << myArray[3] << endl; // Outputs 4
```

Here is an example program that uses an array in C++:

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

int main() {
    // Declare an array of integers with 5 elements
    int myArray[5] = {1, 2, 3, 4, 5};

    // Print out the elements of the array
    for (int i = 0; i < 5; i++) {
        cout << "myArray[" << i << "] = " << myArray[i] << endl;
    }

    // Modify the last element of the array
    myArray[4] = 10;

    // Print out the elements of the array again
    cout << "After modification:" << endl;
    for (int i = 0; i < 5; i++) {
        cout << "myArray[" << i << "] = " << myArray[i] << endl;
    }

    return 0;
}
```

When the program is executed, it will output:

```
myArray[0] = 1
myArray[1] = 2
myArray[2] = 3
myArray[3] = 4
myArray[4] = 5
After modification:
myArray[0] = 1
myArray[1] = 2
myArray[2] = 3
myArray[3] = 4
myArray[4] = 10
```

Explanation

In the above program, we first declared an array of integers with 5 elements and initialized it with the values 1, 2, 3, 4, and 5. Then, we used a for loop to print out the elements of the array. Next, we modified the last element of the array (`myArray[4]`) to be 10. Finally, we used another for loop to print out the elements of the array again, to show that the last element has been modified.

5.2 Pointers

A pointer is a variable that stores the memory address of another variable. In C++, a pointer to a specific data type is declared with the `*` operator. For example, the following declares a pointer to an integer:

```cpp
int *myPointer;
```

You can also initialize a pointer to point to a specific variable, thus:

```cpp
int myVariable = 5;
int *myPointer = &myVariable;
```

In the above, &myVariable is the memory address of myVariable, and myPointer is set to point to it. To access the value stored at that memory address, we use the * operator, like this:

```cpp
int value = *myPointer;
```

Pointers can also be used to manipulate arrays. For example, the following code sets the first element of myArray to 10 using a pointer:

```cpp
myPointer = &myArray[0];
*myPointer = 10;
```

Worked Examples on Pointers

Exercise 1

Declare an integer variable called x and set it equal to 5. Declare a pointer variable called ptr that points to x. Use the pointer to change the value of x to 10. Print out the value of x to check that it has been updated correctly.

```cpp
int x = 5;
int* ptr = &x;
*ptr = 10;
cout << x << endl;
```

Exercise 2

Declare two integer variables called a and b. Declare a pointer variable called ptr that points to a. Use the pointer to swap the values of a and b. Print out the values of a and b to check that they have been swapped correctly.

```cpp
int a = 5, b = 10;
int* ptr = &a;
swap(ptr, &b);
cout << a << " " << b << endl;
```

Exercise 3

Declare an array of integers called numbers with 5 elements. Declare a pointer variable called ptr that points to the first element of the array. Use a loop to print out each element of the array using the pointer.

```cpp
int numbers[5] = {1, 2, 3, 4, 5};
int* ptr = numbers;
for (int i = 0; i < 5; i++) {
    cout << *ptr << endl;
    ptr++;
}
```

Exercise 4

Create a struct called Person with two members: name (a string) and age (an integer). Create an instance of the struct called person1. Declare a pointer variable called ptr that points to person1. Use the pointer to access and modify the member variables of the struct. Print out the values of the member variables to check they were modified correctly.

5.3 Vectors

Vectors: Vectors are a dynamic array implementation in C++ Standard Template Library (STL). They are like arrays but have the ability to change size during runtime.

A vector is a container in the Standard Template Library (STL) of C++ that holds a collection of elements. It is like an array, but with additional functionality such as dynamic resizing when a program is running and the ability to insert and remove elements. In other words, vectors are a dynamic array implementation in C++.

To declare a vector in C++, you use the keyword `vector` followed by the type of element that the vector will hold, enclosed in angle brackets. For example:

```cpp
vector<int> myVector;
```

The above statement declares a vector of integers called `myVector`.

To initialize a vector, you can use any of the following methods:

```cpp
vector<int> myVector;
// method 1: using push_back() function
myVector.push_back(1);
myVector.push_back(2);
myVector.push_back(3);

// method 2: using assignment operator
vector<int> myVector = {1, 2, 3};

// method 3: using initializer list
vector<int> myVector {1, 2, 3};
```

Any of the above methods will initialize a vector variable called `myVector` with 1, 2, and 3.

Here are some examples of how to use vectors in C++:

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

int main() {
    vector<int> myVector = {1, 2, 3, 4, 5};

    // Accessing elements of a vector
    cout << "The first element is: " << myVector[0] << endl;

    // Adding elements to a vector
    myVector.push_back(6);
    cout << "The new last element is: " << myVector.back() << endl;

    // Removing elements from a vector
    myVector.pop_back();
    cout << "The new last element is: " << myVector.back() << endl;

    // Iterating over the elements of a vector
    cout << "The elements of the vector are: ";
    for (auto i : myVector)
        cout << i << " ";
    cout << endl;

    // You can also use Iterators
    cout << "The elements of the vector are: ";
    for (auto it = myVector.begin(); it != myVector.end(); ++it)
        cout << *it << " ";
    cout << endl;

    return 0;
}
```

Run the above C++ code. What output do you see on your screen? It would look like this:

```
The first element is: 1
The new last element is: 6
The new last element is: 5
The elements of the vector are: 1 2 3 4 5
The elements of the vector are: 1 2 3 4 5
```

Exercises

Write a program that:

1. Creates a vector of 10 integers and initializes it with the first 10 even numbers.
2. Creates a vector of strings and adds 5 names. It prints out the names in reverse order.
3. Creates a vector of 10 float numbers and finds the sum of all the elements.

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

int main() {
    // Exercise 1
    vector<int> evenNumbers(10);
    for(int i = 0; i < 10; i++) evenNumbers[i] = 2 * i;

    // Exercise 2
    vector<string> names = {"John", "Mike", "Emily", "Jessica", "Adam"
       ↪ };
    cout << "The names in reverse order are: ";
    for(auto it = names.rbegin(); it != names.rend(); it++) cout << *it
       ↪  << " ";
    cout << endl;

    // Exercise 3
    vector<float> floatNumbers(10);
    float sum = 0;
    for(int i = 0; i < 10; i++) {
        floatNumbers[i] = i * 0.1;
        sum += floatNumbers[i];
    }
    cout << "Sum of all elements: " << sum << endl;
    return 0;
}
```

Explanation

The above code is a C++ program that demonstrates the usage of vectors.

In the main function, the program first creates a vector called `evenNumbers` with a size of 10. It uses a for loop to initialize the vector with the first 10 even numbers (0, 2, 4, 6, 8, 10, 12, 14, 16, 18).

Then, it creates a vector called `names` and initializes it with 5 names using an initializer list. Then it uses an iterator to print out the names in reverse order.

Next, it creates a vector called `floatNumbers` with a size of 10. It uses a for loop to initialize the vector with the first 10 float numbers (0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9). Finally, it uses a for loop to find the sum of all the elements in the vector and prints the result.

At the end of the code, there is a return statement that returns 0, indicating that the program has been executed successfully.

Run the above program to see what output it displays on your screen.

5.4 Structs

In C++, a struct is a user-defined data type that can store a collection of different data types in a single unit. It is similar to a class, but the members of a struct have public visibility by default, whereas members of a class have private visibility by default.

To create a struct, you use the keyword `struct` followed by the name of the struct and a set of curly braces. Inside the curly braces, you can define the members of the struct, which can be any valid C++ data type. Here is an example of a struct called `Person` that has three members: a string for the name, an int for the age, and a char for the gender:

```
struct Person {
    string name;
    int age;
    char gender;
};
```

To instantiate a struct, you create a variable of the struct type and assign it values for its members. You can use the dot operator (`.`) to access the members of a struct. Here is an example of instantiating a struct called `Person` and initializing its members:

```
Person myPerson;
myPerson.name = "John Audu";
myPerson.age = 30;
myPerson.gender = 'M';
```

You can also initialize the struct members when you create the struct object.

```
Person myPerson = {"John Audu", 30, 'M'};
```

You can also create an array of structs and access its elements using array notation.

```
Person people[3] = {
    {"John Audu", 30, 'M'},
    {"Bola Bello", 25, 'F'},
    {"Ngozi Smith", 35, 'M'}
};
cout << "Name of the first person is: " << people[0].name << endl;
```

It is also possible to create a vector of structs and use it in the same way as an array of structs.

```
vector<Person> people = {
    {"John Audu", 30, 'M'},
    {"Bola Bello", 25, 'F'},
    {"Ngozi Smith", 35, 'M'}
};
cout << "Name of the first person is: " << people[0].name << endl;
```

You can also create a struct inside another struct.

```cpp
#include <iostream>

using namespace std;

int main() {
    struct Address {
        string street;
        string city;
        string state;
    };

    struct Person {
        string name;
        int age;
        char gender;
        Address address;
    };

    Person myPerson = {"John Audu", 30, 'M', {"123 Maizube St", "Minna"
        ↪ , "Niger State"}};
    cout << "Address of " << myPerson.name << " is " << myPerson.
        ↪ address.street << endl;

    return 0;
}
```

Exercise

Write a program that uses a struct variable and displays the following about a student: name = Amina, age = 23, and grade = 80.

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

struct Student {
    string name;
    int age;
    int grade;
};

int main() {
    Student myStudent;
    myStudent.name = "Amina";
    myStudent.age = 23;
    myStudent.grade = 80;

    cout << "Student's name: " << myStudent.name << endl;
    cout << "Student's age: " << myStudent.age << endl;
    cout << "Student's grade: " << myStudent.grade << endl;
    return 0;
}
```

Explanation

In the above program, we first defined a struct called `Student` which contains three members: a string for the name, an int for the age, and another int for the grade. Then in the main function, we create an instance of struct `Student` called `myStudent`, initializing its members with the name, age, and grade of the student.

We use the dot operator (`.`) to access the members of the struct, that is, we assign values `Amina`, 23, and 80, respectively. Then we print out the student's name, age, and grade using `cout`.

At the end, the program returns 0, indicating that the program has executed successfully.

Run the program and confirm its output.

5.5 Map

A map is a data structure that stores key-value pairs, where the key is used to access the value. In C++, the standard template library (STL) provides a map container class that can be used to implement a map.

To declare a map, you need to specify the types of the key and value. Here is an example of how to declare a map that stores strings as keys and integers as values:

```
std::map<std::string, int> myMap;
```

Example

```
#include <iostream>
#include <map>

using namespace std;

int main() {
    map<string, int> myMap;
    myMap.insert(make_pair("hello", 1));
    myMap.insert(make_pair("world", 2));
    myMap["goodbye"] = 3;

    int value = myMap["hello"];
    cout << "The value associated with the key 'hello' is: " << value
        ↪ << endl;

    value = myMap.at("goodbye");
    cout << "The value associated with the key 'goodbye' is: " << value
        ↪  << endl;

    auto it = myMap.find("world");
    if (it != myMap.end()) {
        value = it->second;
        cout << "The value associated with the key 'world' is: " <<
            ↪ value << endl;
    }
    return 0;
}
```

Explanation

This is a C++ program that demonstrates the use of the map container from the STL (Standard Template Library). The map container is a data structure that stores elements in the form of key-value pairs, where the key is used to identify an element and the value is the actual data stored for that element.

The program starts by including the necessary headers, `iostream` for input and output operations and `map` for the map container. The program then defines a `main()` function, which is the entry point of the program.

In the `main()` function, a map container called `myMap` is declared with the key type as `string` and the value type as `int`. The `myMap` is then populated with elements by using the `insert()` method to add key-value pairs to the map. The `insert()` method takes a `pair<string, int>` object as an argument, which is created using the `make_pair()` function. The `myMap` can also be populated by using the subscript operator `[]`.

The program then demonstrates three ways to access the value associated with a key in the map. The first is by using the subscript operator `[]`, the second is by using the `at()` method, and the third is by using the `find()` method. The `find()` method returns an iterator to the element with the specified key, or to the end of the map if the key is not found.

The program ends by returning 0 from the `main()` function, indicating a successful execution.

5.6 Chapter Quick Reminder

- Arrays in C++ are collections of elements of the same type, stored in contiguous memory locations.
- Pointers are variables that store the memory address of another variable.
- Vectors are dynamic arrays that can change size during runtime.
- Structs are user-defined data types that can store a collection of different data types.
- Maps are data structures that store key-value pairs.

5.7 Exercises

1. Write a program that declares an array of 10 integers and initializes it with the first 10 prime numbers.
2. Write a program that declares a pointer to an integer and uses it to modify the value of the integer.
3. Write a program that creates a vector of 5 strings and prints them in reverse order.
4. Write a program that defines a struct called `Car` with members `make`, `model`, and `year`. Create an instance of the struct and print its members.
5. Write a program that creates a map with string keys and integer values. Populate the map with 5 key-value pairs and print them.

Chapter 6

Functions

Objectives

In this chapter, you will:

- Know functions
- Understand how to declare a function in your C++ program
- Know how to define a function in C++
- Understand how to make your program call a function
- Become familiar with function arguments and parameters
- Understand how functions receive inputs: by value, reference, and pointers
- Learn about the function return value
- Understand function overloading
- Learn what recursion is and how to use it in C++

6.1 What is a Function in C++?

In C++, a function is a block of code that performs a specific task. It can take zero or more arguments as input and can return a value or not, depending on the function's design. Functions are useful for organizing and reusing code, as they allow you to define a set of actions once and then call them from multiple locations in your program.

A function has a name, a return type, and a list of parameters. The name is used to identify the function, and the return type specifies the type of value that the function returns. The parameters are variables that are passed to the function when it is called.

Here is an example of a simple function in C++:

```cpp
int add(int x, int y) {
    return x + y;
}
```

This function has the name `add`, a return type of `int`, and two parameters: `x` and `y`. It adds the two parameters together and returns the result.

To call this function, you would use its name followed by the values of the parameters in parentheses:

```cpp
int result = add(3, 4);
```

This would call the `add` function with the values 3 and 4 for the parameters `x` and `y`, respectively. The function would then return the result, which would be assigned to the variable `result`. What value do you think `add` will return?

6.2 Declaring a Function

In C++, you can declare a function using the following syntax:

```
return_type function_name(parameter1_type parameter1_name,
    ↪ parameter2_type parameter2_name, ...);
```

The function declaration in the above syntax does not include the function body. Only that first line in bold starting with `return_type` refers to the function declaration. It is also called the function header.

For example, the following is a function that takes two integers as arguments and returns their sum:

```
int add(int x, int y);
```

You can also specify a default value for one or more parameters using the `=` operator. For example:

```
int add(int x, int y = 0);
```

This function can be called with one or two arguments. If you call it with one argument, `y` will be set to 0 by default.

You can also declare a function prototype, which tells the compiler about the function's return type, name, and parameters (that is, the function header), but does not provide a function body. This is useful when you want to use the function before you define it. For example:

```
int add(int x, int y); // This is a function prototype.

int main() {
    int result = add(5, 7);
    // ...
}

int add(int x, int y) {
    return x + y;
}
```

In this case, the function prototype for `add()` appears before `main()`, so the compiler knows that `add()` is a function that takes two integers as arguments and returns an integer. The full function definition appears after the function `main()`.

6.3 Defining a Function

To define a function in C++, you use the following syntax:

```
return_type function_name(parameter_list) {
    // function body
}
```

In the above syntax:

- The `return_type` is the data type of the value that the function returns. If the function does not return a value, you can use the `void` keyword as the return type.
- The `function_name` is the name you give to the function. It should be a descriptive name that reflects the purpose of the function.
- The `parameter_list` is a comma-separated list of variables that the function takes as input. Each parameter consists of a data type and a parameter name.

Here's an example of a function that takes two integers as input and returns their sum:

```cpp
int add(int x, int y) {
    return x + y;
}
```

To call a function in C++, you simply use its name followed by a list of arguments enclosed in parentheses. For example, to call the `add` function defined above, you would write:

```cpp
int result = add(3, 4);
```

This would assign the value 7 to the variable `result`.

Functions can also be defined inside a class in C++, in which case they are called member functions. You will learn about classes in the next chapter. Member functions have access to the member variables and other member functions of the class, and can be called using the dot notation:

```cpp
class MyClass {
public:
    int add(int x, int y) {
        return x + y;
    }
};

int main() {
    MyClass obj;
    int result = obj.add(2, 3);
    return 0;
}
```

6.4 Calling a Function

To call a function in C++, you simply use the function name followed by a set of parentheses, like this:

```cpp
function_name();
```

If the function takes arguments, you can pass them as values or variables inside the parentheses, separated by commas as shown below:

```cpp
function_name(arg1, arg2, arg3);
```

You can call a function from anywhere in your code, including inside other functions. Here is an example of calling a function inside another function:

```cpp
#include <iostream>

void printMessage() {
    std::cout << "Hello, world!" << std::endl;
}

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

The above example defines a function called `printMessage` that prints a message to the console screen. The `main` function calls `printMessage`, causing it to be executed. When the program runs, it will output `"Hello, world!"`.

6.5 Arguments and Parameters

In C++, function arguments are the values passed to a function when called in a program, and function parameters are the variables that receive these values when the function is defined.

Here's an example of a simple C++ function that takes two arguments and returns their sum:

```cpp
int add(int a, int b) {
    return a + b;
}
```

In this example, `a` and `b` are the function parameters, and when the function is called with arguments, for example, `add(3, 4)`, the values of 3 and 4 are assigned to the parameters `a` and `b` respectively.

6.6 How Functions Receive Inputs: By Value, Reference, and Pointers

In C++, inputs to functions called arguments can be passed to a function in several ways, including by value, by reference, and by a pointer.

6.6.1 Passing by Value

```cpp
#include <iostream>

void printValue(int x) {
    std::cout << "The value is: " << x << std::endl;
}

int main() {
    int a = 5;
    printValue(a);
    return 0;
}
```

6.6.2 Passing by Reference

```cpp
#include <iostream>

void doubleValue(int& x) {
    x *= 2;
}

int main() {
    int a = 5;
    doubleValue(a);
    std::cout << "The value is: " << a << std::endl;
    return 0;
}
```

6.6.3 Passing by Pointer

```cpp
#include <iostream>

void tripleValue(int* x) {
    *x *= 3;
}
```

```cpp

int main() {
    int a = 5;
    tripleValue(&a);
    std::cout << "The value is: " << a << std::endl;
    return 0;
}
```

Exercises

1. Create a function that takes two integers as arguments and returns their sum.
2. Create a function that takes a pointer to an array of integers and its size, and prints the elements of the array.
3. Create a function that takes a reference to a string, and appends a given suffix to it.

Answers

Exercise 1

```cpp
#include <iostream>

int addNumbers(int x, int y) {
    return x + y;
}

int main() {
    int a = 5, b = 7;
    std::cout << "The sum is: " << addNumbers(a, b) << std::endl;
    return 0;
}
```

Output:

```
The sum is: 12
```

Exercise 2

```cpp
#include <iostream>

void printArray(int* arr, int size) {
    for (int i = 0; i < size; i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;
}

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int size = sizeof(arr) / sizeof(arr[0]);
    printArray(arr, size);
    return 0;
}
```

Exercise 3

```cpp
#include <iostream>
#include <string>

void addSuffix(std::string& str, std::string suffix) {
    str += suffix;
}

int main() {
    std::string name = "John";
    addSuffix(name, " Audu");
    std::cout << "Name: " << name << std::endl;
    return 0;
}
```

Output:

```
Name: John Audu
```

6.7 Function Return Value

In C++, a function can return a value to the calling code by specifying the return type followed by the function name and then using the `return` keyword followed by the value or expression to be returned. For example, a function that returns an integer might be defined like this:

```cpp
int add(int x, int y) {
    return x + y;
}
```

This function can be called by other code, and the value it returns can be assigned to a variable or used in an expression:

```cpp
int result = add(3, 4);
std::cout << "The result is " << result << std::endl;
```

A function that returns no value can be declared with the `void` return type. In this case, the return statement can be omitted or replaced with `return;`.

```cpp
void print_hello() {
    std::cout << "Hello World" << std::endl;
}
```

This function can be called like this:

```cpp
print_hello();
```

6.8 Function Overloading

Function overloading in C++ allows multiple functions with the same name to be defined in the same scope (i.e., within a program), as long as they have different parameter lists. This allows for more readable and flexible code, as the same function name can be used for multiple related tasks.

Here is an example of function overloading in C++:

```cpp
#include <iostream>

void print(int x) {
    std::cout << x << std::endl;
}
```

```

void print(double x) {
    std::cout << x << std::endl;
}

int main() {
    print(3);
    print(3.14);
    return 0;
}
```

This code defines two functions named `print`, one that takes an `int` parameter and one that takes a `double` parameter. When the `print` function is called with an `int` or a `double` argument, the appropriate version of the function is called. The output of this program will be:

```
3
3.14
```

Exercises

1. Define three functions named `print`: one that takes an `int` parameter, one that takes a `double` parameter, and one that takes a `string` parameter. Call each of these functions in `main` with appropriate arguments.
2. Define two functions named `add`: one that takes two `int` parameters and returns the sum, and one that takes two `double` parameters and returns the sum. Call each of these functions in `main` with appropriate arguments.

Answers

Exercise 1

```cpp
#include <iostream>
#include <string>

void print(int x) {
    std::cout << x << std::endl;
}

void print(double x) {
    std::cout << x << std::endl;
}

void print(std::string s) {
    std::cout << s << std::endl;
}

int main() {
    print(3);
    print(3.14);
    print("Hello, World!");
    return 0;
}
```

This code defines three functions named `print`, one that takes an `int` parameter, one that takes a `double` parameter, and one that takes a `string` parameter. When the `print` function is called with an `int`, `double`, or `string` argument, the appropriate version of the function is called. The output of this program will be:

```
3
3.14
Hello, World!
```

Exercise 2

```cpp
#include <iostream>

int add(int a, int b) {
    return a + b;
}

double add(double a, double b) {
    return a + b;
}

int main() {
    int x = 3, y = 4;
    double a = 2.5, b = 3.14;
    std::cout << "The sum of " << x << " and " << y << " is " << add(x,
        ↪ y) << std::endl;
    std::cout << "The sum of " << a << " and " << b << " is " << add(a,
        ↪ b) << std::endl;
    return 0;
}
```

This code defines two functions named `add`, one that takes two `int` parameters and returns the sum, and one that takes two `double` parameters and returns the sum. When the `add` function is called with two `int` or `double` arguments, the appropriate version of the function is called and the returned value is used as the argument of the `cout` statement. The output of this program will be:

```
The sum of 3 and 4 is 7
The sum of 2.5 and 3.14 is 5.64
```

6.9 Recursion in C++

Recursion is a technique in computer programming where a function calls itself to solve a problem. The idea is to divide a complex problem into smaller, simpler subproblems that can be solved using the same function. The function keeps calling itself until it reaches a base case, which is a condition where the problem can be solved without recursion.

Here is an example of a recursive function in C++ that calculates the factorial of a given number:

```cpp
#include <iostream>

int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main() {
    int num = 5;
    std::cout << "The factorial of " << num << " is " << factorial(num)
        ↪ << std::endl;
    return 0;
}
```

This code defines a function named `factorial` that takes an `int` parameter `n`. The function uses a base case of `n == 0` and returns 1. If `n` is not 0, it calls itself with `n-1` and multiplies the returned value with `n` and returns that. This function will continue calling itself until it reaches the base case, and the final returned value will be the factorial of the number passed as an argument. The output of this program will be:

```
The factorial of 5 is 120
```

When using recursion, it's important to ensure that the base case is reached, otherwise the function will keep calling itself indefinitely, resulting in an infinite loop and eventually a stack overflow. It's also important to keep the problem size shrinking in each recursive call, otherwise, the program may not terminate.

It's important to note that recursion is not always the best solution to a problem; in some cases, iteration may be more efficient and easier to understand.

Worked Examples

Example 1

Develop a program using C++ to find the weighted average of four test scores. The four test scores and their respective weights are given below:

- `testScores1` weight = 0.20
- `testScores2` weight = 0.35
- `testScores3` weight = 0.15
- `testScores4` weight = 0.30

Here is an algorithm:

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

double weightedAverage(double testScores1, double testScores2, double
    ↪ testScores3, double testScores4) {
    double weight1 = 0.20;
    double weight2 = 0.35;
    double weight3 = 0.15;
    double weight4 = 0.30;

    double weightedSum = testScores1 * weight1 + testScores2 * weight2
        ↪ + testScores3 * weight3 + testScores4 * weight4;
    double totalWeight = weight1 + weight2 + weight3 + weight4;

    return weightedSum / totalWeight;
}

int main() {
    double testScores1, testScores2, testScores3, testScores4;

    cout << "Enter test score 1: ";
    cin >> testScores1;
    cout << "Enter test score 2: ";
    cin >> testScores2;
    cout << "Enter test score 3: ";
    cin >> testScores3;
    cout << "Enter test score 4: ";
    cin >> testScores4;

    double weightedAvg = weightedAverage(testScores1, testScores2,
       ↪ testScores3, testScores4);
    cout << "Weighted average: " << weightedAvg << endl;

    return 0;
}
```

Explanation

This program takes user input for the four test scores and calls the function `weightedAverage()` to calculate the weighted average of the scores. The function takes in four test scores as arguments and returns the weighted average based on the given weights. The `main` function prompts the user to input the test scores and then displays the final result. Here is an example output:

```
Enter test score 1: 67
Enter test score 2: 78
Enter test score 3: 90
Enter test score 4: 75
Weighted average: 76.7
```

Example 2

You are given a list of students' names and their test scores. Develop a program in C++ that does the following:

1. Calculate the average score.
2. Determine and print the names of all the students whose test scores are below the average.
3. Print the names of all the students whose test scores are equal to the highest test score.

Divide this program into functions as follows. The first function determines the average test score. The second function determines and prints the names of all students whose test scores are below the average. The third function determines the highest test score. The fourth function determines and prints the names of all the students who had the highest test score. The main algorithm combines the solution of all the functions.

Here is a sample C++ program. Note: this is just an example; the program can be written in many other ways.

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

struct Student {
    string name;
    int testScore;
};

// Function to calculate the average test score
double averageScore(const vector<Student>& students) {
    int sum = 0;
    for (const auto& student : students) {
        sum += student.testScore;
    }
    return (double)sum / students.size();
}

// Function to determine and print the names of students below average
```

```
void printBelowAverage(const vector<Student>& students, double average)
    ↪  {
    cout << "Students below average score:" << endl;
    for (const auto& student : students) {
        if (student.testScore < average) {
            cout << student.name << endl;
        }
    }
}

// Function to determine the highest test score
int highestScore(const vector<Student>& students) {
    int highest = INT_MIN;
    for (const auto& student : students) {
        highest = max(highest, student.testScore);
    }
    return highest;
}

// Function to determine and print the names of students with the
    ↪ highest score
void printHighestScore(const vector<Student>& students, int highest) {
    cout << "Students with the highest test score:" << endl;
    for (const auto& student : students) {
        if (student.testScore == highest) {
            cout << student.name << endl;
        }
    }
}

int main() {
    vector<Student> students = {
        {"Alice", 80},
        {"Bob", 70},
        {"Charlie", 90},
        {"David", 85},
        {"Eve", 75}
    };

    double average = averageScore(students);
    cout << "Average test score: " << average << endl;
    printBelowAverage(students, average);

    int highest = highestScore(students);
    printHighestScore(students, highest);

    return 0;
}
```

Explanation

This program defines a `Student` struct to store the name and test score of each student. It then defines four functions: `averageScore`, `printBelowAverage`, `highestScore`, and `printHighestScore`. The `main` function uses these functions to calculate the average test score, print the names of students whose scores are below average, determine the highest test score, and print the names of students who had the highest test score. Below is the output when you run the program:

```
Average test score: 80
```

```
Students below average score:
Bob
Eve
Students with the highest test score:
Charlie
```

6.10 Chapter Quick Reminder

- A function in C++ is a block of code that can be reused multiple times in a program.
- A function definition includes the function name, its parameters, and the code that it executes.
- A function call is used to execute the code within a function by providing the required arguments.
- Arguments are the values passed to a function when it is called, while parameters are the variables that receive the arguments within the function definition.
- A return value is the output of a function, which can be used as an input for another function or assigned to a variable.
- Inputs to a function can be passed in various ways such as by value, reference, and pointer.
- Function overloading allows for multiple functions to have the same name but with different parameters.
- Recursion is a technique where a function calls itself, allowing for the solution of certain problems to be broken down into smaller subproblems.

6.11 Review Exercises

1. What is the purpose of a function in C++?
   (a) To re-use code multiple times in a program (correct answer)
   (b) To create new variables
   (c) To print text
   (d) To create new classes
2. How are arguments passed to a function?
   (a) By reference
   (b) By pointer
   (c) By value
   (d) All of the above (correct answer)
3. What is the difference between a parameter and an argument?
   (a) Parameters are variables that receive the arguments within the function definition, while arguments are the values passed to a function when it is called. (correct answer)
   (b) Parameters are the values passed to a function when it is called, while arguments are the variables that receive the arguments within the function definition.
   (c) Parameters and arguments are the same thing.
   (d) None of the above
4. What is the purpose of a return value in a function?
   (a) To indicate the end of the function
   (b) To output text

   (c) To provide an output that can be used as input for another function or assigned to a variable (correct answer)
   (d) To create new variables

5. Can a function have multiple return statements?
   (a) True (correct answer)
   (b) False

6. What is function overloading in C++?
   (a) A technique where a function calls itself
   (b) Multiple functions can have the same name but with different parameters (correct answer)
   (c) A way to pass arguments to a function
   (d) A way to assign values to variables

7. Can you use recursion to solve any problem?
   (a) True
   (b) False (correct answer)

8. How is recursion used in a function?
   (a) By calling other functions
   (b) By calling itself (correct answer)
   (c) By passing arguments
   (d) By using loops

9. When calling a function, what is the difference between passing an argument by value and by reference?
   (a) Pass by value creates a new copy of the argument, while pass by reference uses the original argument (correct answer)
   (b) Pass by value uses the original argument, while pass by reference creates a new copy of the argument
   (c) There is no difference
   (d) None of the above

10. What is the purpose of function overloading?
   (a) To pass arguments to a function
   (b) To create new variables
   (c) To allow multiple functions to have the same name but with different parameters (correct answer)
   (d) To use recursion

6.12 Programming Exercises

1. Create a function that takes two integers as parameters, adds them together, and returns the result. Call the function with different sets of numbers and print the result.
2. Create a function that takes a string as an argument and returns the number of characters in the string. Test the function by passing different strings as arguments.
3. Create a function that takes an array of integers as a parameter, finds the largest value in the array, and returns the result. Test the function by passing different arrays as arguments.

4. Create a function that takes a string and a character as arguments and returns the number of occurrences of that character in the string. Test the function by passing different strings and characters as arguments.
5. Create a function that takes a number as an argument and returns the factorial of that number. Test the function by passing different numbers as arguments.
6. Create a function that takes two arguments, a string and a character, and replace all occurrences of the character in the string with '*'. Test the function by passing different strings and characters as arguments.
7. Create two functions with the same name that take different types of arguments (e.g., one takes an integer, the other takes a string). Test the function by passing different arguments and observing the output.
8. Create a function that uses recursion to calculate the nth Fibonacci number. Test the function by passing different values of n as arguments.
9. Create a function that takes an array of integers as an argument and uses recursion to sort the array in ascending order.
10. Create a function that takes a number as an argument and uses recursion to calculate the sum of all numbers from 1 to that number.

Chapter 7

Object-Oriented Programming Concepts

Objectives

In this chapter, you will

- become familiar with a class in C++ programming
- define the concept of objects in C++ programming
- examine the specification of a class
- understand how to access class members
- know how to write a C++ program with class
- become familiar with the uses of constructors
- Examine the differences between the default, parameterized and copy construction
- know how a class can improve your code by implementing a simple stack in two ways: first, using a structure and functions, and then using a class
- Learn the OO concepts of inheritance, encapsulation and polymorphism

7.1 Definition of a Class

Class is a group of objects that share common properties and relationships. In C++, a class is a new data type that contains member variables and member functions that operates on the variables. A class is defined with the keyword `class`. It allows the data to be hidden, if necessary, from external use. When we define a class, we are creating a new abstract data type that can be treated like any other built-in data type or A class in C++ is the building block, that leads to Object-Oriented programming. It is a user-defined data type, which holds its data members and member functions, which can be accessed and used by creating an instance of that class. A C++ class is like a blueprint for an object. For Example: Consider the Class of Cars. There may be many cars with different names and brands but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage range etc. So here, Car is the class and wheels, speed limits, and mileage are their properties. A Class is a user-defined datatype which has data members and member functions. Data members are the data variables and member functions are the functions used to manipulate these variables together these data members and member functions define the properties and behaviour of the objects in a Class. In the above example of class Car, the data member will be speed limit, mileage etc. and member functions can be: applying brakes, increasing speed etc. Generally, a class specification has two parts:

- Class declaration
- Class function definition

The class declaration describes the type and scope of its members. The class function definition describes how the class functions are implemented.

```
class class-name
{
private:
    variable declarations;
    function declaration;
public:
    variable declarations;
    function declaration;
};
```

The members that have been declared private can be accessed only from within the class. On the other hand, public members can be accessed from outside the class also. Data hiding is the key feature of OOP. The use of keywords private is optional by default, the members of a class are private. The variables declared inside the class are known as data members and the functions are known as member functions. Only the member functions can have access to the private data members and private functions. However, the public members can be accessed from outside the class. The binding of data and functions together into a single class-type variable is referred to as encapsulation.

```


class item
{
    int number;
    float cost;
public:
    void getdata (int a, float b);
    void putdata (void);
};
```

The class `item` contains two data members and two function members, the data members are private by default while both the functions are public by declaration. The function `getdata()` can be used to assign values to the member variables `number` and `cost`, and `putdata()` for displaying their values. These functions provide only access to the data members from outside the class.

7.2 Objects

Objects are the basic run-time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program must handle. They may also represent user-defined data such as vectors, time, and lists. A programming problem is analyzed in terms of objects and the nature of communication between them. Program objects should be chosen such that they match closely with real-world objects.

7.2.1 Creating Objects

Once a class has been declared we can create variables of that type by using the class name.

```
item x;
```

creates a variable `x` of a type `item`. In C++, the class variables are known as objects. Therefore, `x` is called an object of a type `item`.

```
item x, y, z;
```

is also possible.

```
class item
{
    -----------
    -----------
    -----------
};
item x, y, z;
```

would create the objects `x`, `y`, and `z` of a type `item`.

Note: An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e., an object is created) memory is allocated.

```
class className{
    Access specifier:    //can be private, public and protected
    Data members:        //Variables to be used
    Member Functions(){} //methods to access data member
}; // class name ends with a semicolon
```

7.2.2 Declaring Objects

When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects. Here is the syntax for doing that: :

```
ClassName ObjectName;
```

Examine the following exercises to see how you can create a class and objects

7.2.3 Create a Class

To create a class, use the `class` keyword:

```
class MyClass {             // The class
  public:                   // Access specifier
    int myNum;             // Attribute (int variable)
    string myString;        // Attribute (string variable)
};
```

Example explained The `class` keyword is used to create a class called `MyClass`. The `public` keyword is an access specifier, which specifies that members (attributes and methods) of the class are accessible from outside the class. Inside the class, there is an integer variable `myNum` and a string variable `myString`. When variables are declared within a class, they are called attributes. At last, end the class definition with a semicolon.

7.2.4 Create an Object

In C++, an object is created from a class. We have already created the class named `MyClass`, so now we can use this to create objects. To create an object of `MyClass`, specify the class name, followed by the object name. To access the class attributes (`myNum` and `myString`), use the dot syntax (.) on the object:

```
class MyClass {        // The class
  public:             // Access specifier
    int myNum;        // Attribute (int variable)
    string myString;  // Attribute (string variable)
};

int main() {
  MyClass myObj;   // Create an object of MyClass

  // Access attributes and set values
  myObj.myNum = 15;
  myObj.myString = "Some text";

  // Print attribute values
  cout << myObj.myNum << "\n";
  cout << myObj.myString;
  return 0;
}
```

The program below shows how you can create a Class and Objects

```
#include<iostream>
using namespace std;
class programming
{
private:
    int variable;
public:
    void input_value()
    {
        cout << "In function input_value, Enter an integer\n";
        cin >> variable;
    }
    void output_value()
    {
        cout << "Variable entered is ";
        cout << variable << "\n";
    }
};
main()
{
    programming object;
    object.input_value();
    object.output_value();
    //object.variable; will produce an error because the variable is
       ↪ private
    return 0;
}
```

7.3 Accessing Class Members

The private data of a class can be accessed only through the member functions of that class. For example on page 93, where we created a class `item`, function `main()` cannot contain statements that access `number` and `cost` directly.

`object name.function-name(actual arguments);`

Example:

```
x.getdata(100,75.5);
```

It assigns a value of 100 to `number`, and 75.5 to `cost` of the object `x` by implementing the `getdata()` function. Similarly, the statement

```
x.putdata(); //would display the values of data members.
```

```
x.number = 100; //  is illegal.
```

Although `x` is an object of the type `item` to which `number` belongs, the `number` can be accessed only through a member function and not by the object directly.

Example:

```
class xyz
{
    int x;
    int y;
public:
    int z;
};
---------
----------
xyz p;
p.x =0;     //error,  x is private
p.z=10; // ok, z is public
```

Note: The data members and member functions of a class can be accessed using the dot('.') operator with the object. For example, if the name of an object is `obj` and you want to access the member function with the name `printName()` then you will have to write `obj.printName()`.

7.3.1 Accessing Data Members

The public data members are also accessed in the same way as above. However, the private data members are not allowed to be accessed directly by the object. Accessing a data member depends solely on the access control of that data member. This access control is given by Access modifiers in C++. There are three access modifiers: `public`, `private` and `protected`

```
// C++ program to demonstrate accessing of data member

#include<iostream>
using namespace std;
class Geeks
{
    public:         //Access specifier

    //Data Members
    string geekname;

    //Member Function()
    void printname()
    {
        cout << "Geekname is: " << geekname << endl;
    }
};
int main() {

    //declare an object of class Geeks
    Geeks obj1, obj2;

    //accessing data member
    obj1.geekname = "Danjuma";
    obj2.geekname = "Comfort";
    // accessing member function

    obj1.printname();
    obj2.printname();
    return 0;
}
```

Explanation This above C++ code defines a class `Geeks` with a single public member, a string `geekname`. The class also has a single public member function `printname()`, which prints the value of the `geekname` member variable to the console. In the `main()` function, the code declares two objects of the class `Geeks` named `obj1` and `obj2`. It then assigns values to the `geekname` member variable of each object, "Danjuma" to `obj1` and "Comfort" to `obj2`. Then, the code calls the `printname()` function on each object, which prints the value of the `geekname` variable to the console, resulting in the output: "Geekname is: Danjuma" "Geekname is: Comfort"

In summary, this code demonstrates the use of classes and objects in C++. It defines a class `Geeks` with a single public member variable `geekname` and a single public member function `printname()`. The `main()` function creates two objects of the class, assigns values to their `geekname` member variable, and then calls the `printname()` function on each object to display the value of the `geekname` variable on the console.

7.4 Member Function

Methods are functions that belong to a class. Member can be defined in two places

- Outside the class definition
- Inside the class function

7.4.1 Outside the Class Definition

Member functions declared inside a class must be defined separately outside the class. Their definitions are very much like normal functions. An important difference between a member function and a normal function is that a member function incorporates a membership identity in the label in the header. The 'label' tells the compiler which class the function belongs to.

```
return type class-name::function-name(argument declaration )
{
    function-body
}
```

The membership label `class-name::` tells the compiler that the function `function-name` belongs to the class `class-name`. That is the scope of the function is restricted to the `class-name` specified in the header line. The `::` symbol is called the scope resolution operator.

Example:

```
void item:: getdata (int a, float b )
{
    number=a;
    cost=b;
}
void item:: putdata ( void)
{
    cout<<"number=:"<<number<<endl; cout<<"cost="<<cost<<endl;
}
```

The member functions have some special characteristics that are often used in program development.

- Several different classes can use the same function name. The "membership label" will resolve their scope, and member functions can access the private data of the class. A non-member function can't do so.
- A member function can call another member function directly, without using the dot operator.

To define a member function outside the class definition we have to use the scope resolution (`::`) operator along with the class name and function name.

```


//C++ program to demonstrate function declaration outside class

#include <iostream>
using namespace std;
class Geeks
{
    public:
```

```
    string geekname;
    int id;

    //printname is not defined inside the class definition
    void printname();

    //printed is defined inside the class definition
    void printed()
    {
        cout << "Geek id is: "<< id;
    }
};

//Definition of printname using scope resolution operator::
void Geeks::printname()
{
    cout << "Geekname is: "<< geekname;
}
int main() {
    Geeks obj1;
    obj1.geekname = "Mohammed";
    obj1.id=15;

    //call printname()
    obj1.printname();
    cout << endl;

    //call printed()
    obj1.printed();
    return 0;
}
```

Explanation

The above C++ code defines a class `Geeks` with two public member variables: a string `geekname` and an integer `id`. The class also has two public member functions: `printname()` and `printed()`. In the class definition, `printname()` is declared but not defined, while `printed()` is both declared and defined within the class definition. In the `main()` function, the code declares an object `obj1` of the class `Geeks`, and assigns values to the `geekname` and `id` member variables of the object. Then, the code calls the `printname()` function on the object `obj1`, which is defined outside of the class definition using the scope resolution operator `::`. This function prints the value of the `geekname` variable to the console, resulting in the output "Geekname is: Mohammed". Then, the code calls the `printed()` function on the object `obj1`, which is defined inside the class definition, This function prints the value of the `id` variable to the console, resulting in the output "Geek id is: 15". In summary, this code demonstrates the use of classes, objects, and member functions in C++. It defines a class `Geeks` with two public member variables `geekname` and `id` and two public member functions `printname()` and `printed()`. The `main()` function creates an object `obj1` of the class, assigns values to its member variables and then calls the `printname()` and `printed()` functions on the object to display the value of the `geekname` and `id` variable on the console. It also shows the difference between the member function defined inside the class definition and the member function defined outside the class definition using scope resolution operator.

Note that all the member functions defined inside the class definition are by default inline, but you can also make any non-class function inline by using the keyword `inline` with them. Inline functions are actual functions, which are copied everywhere during compilation, like pre-processor macro, so the overhead of function calls is reduced.

7.4.2 Inside the Class Definition

Another method of defining a member function is to replace the function declaration with the actual function definition inside the class.

Example:

```cpp
class item
{
    int number; float cost;
public:
    void getdata (int a, float b);
    void putdata(void)
    {
        cout<<number<<endl; cout<<cost<<endl;
    }
};
```

A C++ Program With Class

```cpp
#include<iostream>
using namespace std;
class item
{
    int number;
    float cost;
public:
    void getdata ( int a, float b);
    void putdata (void)
    {
        cout<<"number:"<<number<<endl; cout<<"cost :"<<cost<<endl;
    }
};
void item::getdata (int a, float b)
{
    number=a;
    cost=b;
}
int main ()
{
    item x;
    cout<<"\nobject x"<<endl;
    x.getdata( 100,299.95);
    x.putdata();
    item y;
    cout<<"\n object y"<<endl;
    y.getdata(200,175.5);
    y.putdata();
    return 0;
}
```

Output of the program

```
object x
number:100
cost :299.95

 object y
number:200
cost :175.5
```

Explanation of the above program: The first line `#include<iostream>` includes the standard input/output library which provides the `cout` and `cin` functions for printing to the console and reading from the console, respectively.

The `using namespace std;` makes all the standard library functions and objects available without the `std::` prefix.

The next block of code defines a class `item` with two private members: an integer `number` and a float `cost`.

The class has two public member functions: `getdata()` and `putdata()`. The `getdata()` function takes two arguments, an integer and a float, and assigns them to the private members `number` and `cost`. The `putdata()` function displays the values of the private members `number` and `cost` on the console.

The `void item::getdata (int a, float b)` is called method definition, it defines the `getdata()` method for the `item` class. The `item::` prefix indicates that the method is a member of the `item` class.

The `int main()` function is the main entry point of the program.

In the main function, it creates an object `x` of the `item` class.

The next line `x.getdata( 100,299.95);` calls the `getdata()` method of the `x` object and passes the values 100 and 299.95 to it.

The next line `x.putdata();` calls the `putdata()` method of the `x` object and displays the values of its `number` and `cost` members on the console.

Similarly, the program creates an object `y` of the `item` class.

The next line `y.getdata(200,175.5);` calls the `getdata()` method of the `y` object and passes the values 200 and 175.5 to it.

The next line `y.putdata();` calls the `putdata()` method of the `y` object and displays the values of its `number` and `cost` members on the console.

The last line `return 0;` indicates that the `main()` function has been completed successfully.

In summary, the program demonstrates the use of classes and objects in C++. It defines a class `item` with private members `number` and `cost`, and public member functions `getdata()` and `putdata()`. The `main()` function creates two objects of the class, assigns values to their members, and displays the values on the console.

7.5 Private member functions

Although it is a normal practice to place all the data items in a private section and all the functions in public, some situations may require containing functions to be hidden from outside calls. Tasks such as deleting an account in a customer file or providing an increment to an employee are events of serious consequences and therefore the functions handling such tasks should have restricted access. We can place these functions in the private section. A private member function can only be called by another function that is a member of its class. Even an object cannot invoke a private function using the dot operator.

Example

```
class sample
{
    int m;
    void read (void);
    void write (void);
};
```

if `s` is an object of `sample`, then

`s.read(); //is illegal. However, the function read() can be called by the function update ( ) to update the value of m.`

```
void sample:: update(void)
{
    read( );
}
```

Example code:

```
#include<iostream>
using namespace std;
class part
{
private:
    int modelnum, partnum;
    float cost;
public:
    void setpart ( int mn, int pn, float c)
    {
        modelnum=mn;
        partnum=pn;
        cost=c;
    }
    void showpart ( )
    {
        cout<<"model:"<<modelnum<<endl;
        cout<<"num:"<< partnum <<endl;
        cout<<"cost:"<<"$"<<cost;
    }
};
int main()
{
    part p1,p2;
    p1.setpart(644, 73, 217.55);
    p2.setpart(567, 89, 789.55);
    p1.showpart();
    p1.showpart();
    return 0;
}
```

Explanation of the above code: This C++ code defines a class `part` which has three private members: an integer `modelnum`, an integer `partnum`, and a float `cost`. The class has two public member functions: `setpart()` and `showpart()`. The `setpart()` function takes three arguments: an integer `mn`, an integer `pn`, and a float `c`. It assigns these values to the private members `modelnum`, `partnum`, and `cost`, respectively. The `showpart()` function displays the values of the private members `modelnum`, `partnum`, and `cost` on the console. The output will be in the format "model: X", "num: Y", "cost: $Z" where X, Y, and Z are the values of `modelnum`, `partnum`, and `cost`, respectively. The `int main()` function is the main entry point of the program. The program creates two objects of the class `part` named `p1` and `p2`. The next lines `p1.setpart(644, 73, 217.55);` and `p2.setpart(567, 89, 789.55);` calls the `setpart()` method of `p1` and `p2` objects and passes the values 644, 73 and 217.55 to `p1` and 567, 89 and 789.55 to `p2`. The next lines `p1.showpart();` and `p1.showpart();` calls the `showpart()` method of the `p1` and `p2` objects and displays the values of their `modelnum`, `partnum`, and `cost` members on the console. The last line `return 0;` indicates that the `main()` function has completed successfully. In summary, this program demonstrates the use of classes and objects in C++. It defines a class `part` with private members `modelnum`, `partnum`, and `cost`, and public member functions `setpart()` and `showpart()`. The `main()` function creates two objects of the class, assigns values to their members, and displays the values on the console.

7.6 Constructor

Constructors are special class members which are called by the compiler every time an object of that class is instantiated. Constructors have the same name as the class and may be defined inside or outside the class definition.

A constructor is a special member function whose task is to initialize the objects of its class. It is special because its name is the same as the class name. The constructor is invoked whenever an object of its associated class is created. It is called a constructor because it constructs the values of data members of the class.

A constructor is declared and defined as follows:

```cpp
// class with a constructor
class integer
{
    int m, n;
public:
    integer(void); // constructor declared
    // ...
};

integer::integer(void)
{
    m = 0;
    n = 0;
}
```

When a class contains a constructor like the one defined above, it is guaranteed that an object created by the class will be initialized automatically.

For example:

```cpp
integer int1;   // object int1 created
```

This declaration not only creates the object `int1` of type `integer` but also initializes its data members `m` and `n` to zero.

Examine the following example, it explains the concept of a constructor:

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

class Rectangle {
public:
    // Constructor
    Rectangle(int width, int height) {
        this->width = width;
        this->height = height;
    }
    // Other member functions
    int getArea() {
        return width * height;
    }
    int getPerimeter() {
        return 2 * (width + height);
    }
    void setWidth(int width) {
        this->width = width;
    }
    void setHeight(int height) {
        this->height = height;
    }

private:
    int width;
    int height;
};

int main() {
    Rectangle rect(3, 4);  // create a rectangle object and call the
        ↪ constructor
    int area = rect.getArea();  // call the getArea() member function
    int perimeter = rect.getPerimeter(); // call the getPerimeter()
        ↪ member function
    cout << "Area: " << area << endl;
    cout << "Perimeter: " << perimeter << endl;
    rect.setWidth(5); // set width to 5
    rect.setHeight(6); // set height to 6
    area = rect.getArea(); // call the getArea() member function again
    perimeter = rect.getPerimeter(); // call the getPerimeter() member
        ↪ function again
    cout << "New Area: " << area << endl;
    cout << "New Perimeter: " << perimeter << endl;
    return 0;
}
```

Explanation: This C++ code defines a class called `Rectangle` which represents a rectangle shape. The class has the following features:

- The class has a constructor that takes two integer arguments, `width` and `height`, and assigns them to the private member variables `width` and `height`.
- The class has two member functions `getArea()` and `getPerimeter()` that return the area and perimeter of the rectangle respectively.
- The class has two member functions `setWidth()` and `setHeight()` that allow the user to change the width and height of the rectangle respectively.
- All the member variables and member functions are defined as private, meaning that they can only be accessed within the class.

The `main()` function creates an object of the `Rectangle` class and calls the constructor to initialize the object's width and height to 3 and 4 respectively. Then, it calls the `getArea()` and `getPerimeter()` member functions to get the area and perimeter of the rectangle and prints them. The width and height of the rectangle are then set to 5 and 6 respectively using the `setWidth()` and `setHeight()` member functions. The new area and perimeter of the rectangle are then calculated and printed.

7.7 Types of constructors

- Default constructor
- Parameterized constructor
- Copy constructor

7.7.1 Default Constructor

A constructor that accepts no parameter is called the default constructor. The default constructor for class `A` is `A::A()`. If no such constructor is defined, then the compiler supplies a default constructor. Therefore, a statement such as:

```
A a; // invokes the default constructor of the compiler to create the
    ↪ object "a".
```

The constructor functions have some characteristics:

- They should be declared in the public section.
- They are invoked automatically when the objects are created.
- They don't have return types, not even `void` and therefore they cannot return values.
- They cannot be inherited, though a derived class can call the base class constructor.
- Like other C++ functions, they can have default arguments.
- Constructors can't be virtual.
- An object with a constructor can't be used as a member of a union.

Example of default constructor:

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

class MyClass {
private:
    int x;

public:
    // Default constructor
    MyClass() : x(0) {
        cout << "Default constructor called." << endl;
    }

    // Other member functions
    void printValue() {
        cout << "Value of x: " << x << endl;
    }
};

int main() {
    MyClass obj1; // Default constructor called
    obj1.printValue();
    return 0;
}
```

Explanation: In this example, the class `MyClass` has only one constructor, a default constructor. The default constructor is defined with no parameters and it initializes the private member variable `x` with the value 0. It also prints a message to indicate that the default constructor has been called.

The `main()` function creates one object of the class `MyClass` using the default constructor. Then it calls the `printValue()` member function to display the value of the private member variable `x` for the object.

The output will be:

```
Default constructor called.
Value of x: 0
```

7.7.2 Parameterized Constructor

The constructors that can take arguments are called parameterized constructors. Using parameterized constructor, we can initialize the various data elements of different objects with different values when they are created.

Example:

```cpp
class integer
{
    int m, n;
public:
    integer(int x, int y);
    // ...
};

integer::integer(int x, int y)
{
    m = x;
    n = y;
}
```

The argument can be passed to the constructor by calling the constructor implicitly.

```cpp
integer int1 = integer(0, 100); // explicit call
integer int1(0, 100); // implicit call
```

Example:

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

class MyClass {
private:
    int x;

public:
    // Parameterized constructor
    MyClass(int val) {
        x = val;
        cout << "Parameterized constructor called." << endl;
    }

    // Other member functions
    void printValue() {
        cout << "Value of x: " << x << endl;
    }
};

int main() {
    MyClass obj1(5); // Parameterized constructor called
    obj1.printValue();
    return 0;
}
```

Explanation: In this example, the class `MyClass` has only one constructor, a parameterized constructor. The parameterized constructor is defined with one integer parameter, `val`, and it initializes the private member variable `x` with the value passed as an argument. It also prints a message to indicate that the parameterized constructor has been called.

The `main()` function creates one object of the class `MyClass` using the parameterized constructor, passing the value 5 as an argument. Then it calls the `printValue()` member function to display the value of the private member variable `x` for the object.

The output will be:

```
Parameterized constructor called.
Value of x: 5
```

7.8 Constructor Overloading

When more than one constructor function is defined in a class, then it is called constructor overloading or the use of multiple constructors in a class. It is used to increase the flexibility of a class by having more constructors for a single class. Overloading constructors in C++ programming gives us more than one way to initialize objects in a class.

To illustrate the constructor overloading, let us take the following examples:

```cpp
#include <iostream>
#include <conio.h>
using namespace std;

class student
{
    int roll;
    char name[30];
public:
    student(int x, char y[])    // parameterized constructor
    {
        roll = x;
        strcpy(name, y);
    }
    student()        // normal constructor
    {
        roll = 100;
        strcpy(name, "y");
    }
    void input_data()
    {
        cout << "\n Enter roll no :"; cin >> roll;
        cout << "\n Enter name :"; cin >> name;
    }
    void show_data()
    {
        cout << "\n Roll no :" << roll;
        cout << "\n Name :" << name;
    }
};

int main()
{
    student s(10, "z");
    s.show_data();
    getch();
    return 0;
}
```

7.9 Copy Constructor

A copy constructor is used to declare and initialize an object from another object.

Example:

```cpp
#include <iostream.h>
```

```cpp
class code
{
    int id;
public:
    code() {} // constructor
    code(int a) { id = a; } // constructor
    code(code &x)
    {
        id = x.id;
    }
    void display()
    {
        cout << id;
    }
};

int main()
{
    code A(100);
    code B(A);
    code C = A;
    code D;
    D = A;
    cout << "\n id of A :"; A.display();
    cout << "\n id of B :"; B.display();
    cout << "\n id of C:"; C.display();
    cout << "\n id of D:"; D.display();
}
```

Study the program below and see how it demonstrates default and parameterized constructors:

```cpp
// C++ program to demonstrate constructors

#include <bits/stdc++.h>
using namespace std;

class Geeks
{
public:
    int id;

    // Default Constructor
    Geeks()
    {
        cout << "Default Constructor called" << endl;
        id = -1;
    }

    // Parameterized Constructor
    Geeks(int x)
    {
        cout << "Parameterized Constructor called" << endl;
        id = x;
    }
};

int main()
{
    // obj1 will call Default Constructor
    Geeks obj1;
    cout << "Geek id is: " << obj1.id << endl;

    // obj2 will call Parameterized Constructor
    Geeks obj2(21);
    cout << "Geek id is: " << obj2.id << endl;
    return 0;
}
```

7.10 Destructors

Destructor is another special member function that is called by the compiler when the scope of the object ends.

```
// C++ program to explain destructors

#include <bits/stdc++.h>
using namespace std;

class Geeks
{
public:
    int id;

    // Definition for Destructor
    ~Geeks()
    {
        cout << "Destructor called for id: " << id << endl;
    }
};

int main()
{
    Geeks obj1;
    obj1.id = 7;
    int i = 0;
    while (i < 5)
    {
        Geeks obj2;
        obj2.id = i;
        i++;
    } // Scope for obj2 ends here

    return 0;
} // Scope for obj1 ends here
```

7.11 Inline Function

To eliminate the cost of calls to small functions, C++ proposes a new feature called inline function. An inline function is a function that is expanded in line when it is invoked. That is, the compiler replaces the function call with the corresponding function code.

The inline functions are defined as follows:

```
inline function-header
{
    function body;
}
```

Example:

```
inline double cube(double a)
{
    return (a * a * a);
}
```

The above inline function can be invoked by statements like:

```
c = cube(3.0);
d = cube(2.5 + 1.5);
```

Remember that the `inline` keyword merely sends a request, not a command to the compiler. The compiler may ignore this request if the function definition is too long or too complicated and compile the function as a normal function.

Some of the situations where inline expansion may not work are:

1. For functions returning values if a loop, a switch, or a go-to exists.
2. For functions not returning values, if a return statement exists.
3. If functions contain static variables.
4. If inline functions are recursive.

Example:

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

inline float mul(float x, float y)
{
    return (x * y);
}

inline double div(double p, double q)
{
    return (p / q);
}

int main()
{
    float a = 12.345;
    float b = 9.82;
    cout << "Given " << a << " and " << b << ", the result from
       ↪ function mul = " << mul(a, b) << endl;
    cout << "Given " << a << " and " << b << ", the result from
       ↪ function div = " << div(a, b) << endl;
    return 0;
}
```

So far, we have learned about variables, functions, and other simple control structures in the previous chapters. Before we end this chapter, you'll see how a class can improve your code by implementing a simple stack two ways: first, using a struct variable and functions, and then using a class.

7.12 Stacks

A stack is a way for storing data. Data can be put in the stack using a push operation. The pop operation removes the data. Data is stored in last-in-first-out (LIFO) order. You can think of a stack as a stack of papers. When you perform a push operation, you put a new paper on top of the stack. You can push as many times as you want. Each time the new data goes on top of the stack. You get data out of a stack using the pop operation, which takes the top paper off the stack and gives it to the caller. We are going to write a class which illustrates how the stack works.

7.12.1 Designing a Stack

You start a stack design by designing the data structure. This structure will need a place to put the data (called data) and a count of the number of items currently pushed on the stack (called count):

```
const int STACK_SIZE = 100; // Maximum size of a stack

// The stack itself
struct stack {
    int count; // Number of items in the stack
    int data[STACK_SIZE]; // The items themselves
};
```

Next, you need to create the routines to handle the push and pop operations. The push function stores the item on the stack and then increases the data count.

```
inline void stack_push(struct stack &the_stack, const int item)
{
    the_stack.data[the_stack.count] = item;
    ++the_stack.count;
}
```

Note: This version of the program does not check for stack overflow or other error conditions. Popping simply removes the top item and decreases the number of items in the stack.

```
inline int stack_pop(struct stack &the_stack)
{
    // Stack goes down by one
    --the_stack.count;
    // Then we return the top value
    return (the_stack.data[the_stack.count]);
}
```

There is one item you've overlooked: initializing the stack. You see you must set up the stack before you can use it. Keeping with the spirit of putting everything in a stack_xxxx routine gets the stack_init function.

```
inline void stack_init(struct stack &the_stack)
{
    the_stack.count = 0; // Zero the stack
}
```

Notice that you don't need to zero the data field in the stack, since the elements of data are overwritten by the push operation. You are now finished. To use the stack, you declare it with a struct statement. Next, you must make sure that you initialize it, and then you can push and pop to your heart's content (or at least within the limits of the stack).

```
struct stack a_stack; // Declare the stack

stack_init(a_stack); // Initialize the stack
// Stack is ready for use
```

Below is a complete implementation of the stack using the struct and a short test routine:

```cpp
#include <stdlib.h>
#include <iostream>
using namespace std;

const int STACK_SIZE = 100; // Maximum size of a stack

// The stack itself
struct stack {
    int count; // Number of items in the stack
    int data[STACK_SIZE]; // The items themselves
};

// Initiate the stack
inline void stack_init(struct stack &the_stack)
{
    the_stack.count = 0; // Zero the stack
}

// Push operation
inline void stack_push(struct stack &the_stack, const int item)
{
    the_stack.data[the_stack.count] = item;
    ++the_stack.count;
}

// Pop operation
inline int stack_pop(struct stack &the_stack)
{
    // Stack goes down by one
    --the_stack.count;

    // Then we return the top value
    return (the_stack.data[the_stack.count]);
}

// A short routine to test the stack
int main()
{
    struct stack a_stack; // Stack we want to use

    stack_init(a_stack);

    // Push three values on the stack
    stack_push(a_stack, 1);
    stack_push(a_stack, 2);
    stack_push(a_stack, 3);

    // Pop the items from the stack
    cout << "Expect a 3 ->" << stack_pop(a_stack) << '\n';
    cout << "Expect a 2 ->" << stack_pop(a_stack) << '\n';
    cout << "Expect a 1 ->" << stack_pop(a_stack) << '\n';
    return (0);
}
```

Run the above program and see if it is working as expected.

The C++ class does not only hold data like a struct, it also holds a set of functions for manipulating the data and access protection. Turning the struct stack above into a class you get the following:

```
class stack {
private:
    int count; // Number of items in the stack
    int data[STACK_SIZE]; // The items themselves
public:
    // Initialize the stack
    void init(void);

    // Push an item on the stack
    void push(const int item);

    // Pop an item from the stack
    int pop(void);
};
```

Let's go into this class declaration in more detail. The beginning looks much like a struct definition but you make use of the word `class` instead of `struct`.

```
class stack {
private:
    int count; // Number of items in the stack
    int data[STACK_SIZE]; // The items themselves
```

The above code snippet declares two fields: `count` and `data`. In a class, these items are not called fields; they are called member variables. The keyword `private` indicates the access privileges associated with these two member variables. There are three levels of access privileges: `public`, `private`, and `protected`. Class members, both data and functions, marked `private` cannot be used outside the class. They can be accessed only by functions within the class. The opposite of `private` is `public`, which indicates members that anyone can access. Finally, `protected` is like `private` except that it allows access by derived classes. Derived classes are classes created by you.

You've finished defining the data for this class. Now you need to define the functions that manipulate the data.

```
public:
    // Initialize the stack
    void init(void);

    // Push an item on the stack
    void push(const int item);

    // Pop an item from the stack
    int pop(void);
};
```

This section starts with the keyword `public`. This tells C++ that you want all these member functions to be available to the outside. In this case, you just define the function prototypes. The code for the function will be defined later. Next comes the body of the `init` function. Since this function belongs to the `stack` class, you prefix the name of the procedure with `stack::`

The definition of the `init` function looks like this:

```
inline void stack::init(void)
{
    count = 0; // Zero the stack
}
```

This procedure zeroes the stack's count. In the struct version of the `stack_init` function, you must pass the stack in as a parameter. Since this function is part of the `stack` class, that's unnecessary. This also means that you can access the member variables directly. In other words, instead of having to say `the_stack.count`, you just say `count`.

The functions `push` and `pop` are implemented similarly.

```
inline void stack::push(const int item)
{
    data[count] = item;
    ++count;
}

inline int stack::pop(void)
{
    // Stack goes down by one
    --count;

    // Then we return the top value
    return (data[count]);
}
```

The `stack` class is now complete. All you have to do is use it.

7.12.2 Using a Class

```
#include <stdlib.h>
#include <iostream>
using namespace std;

const int STACK_SIZE = 100; // Maximum size of a stack

// The stack itself
class stack {
private:
    int count; // Number of items in the stack
    int data[STACK_SIZE]; // The items themselves
public:
    // Initialize the stack
    void init(void);

    // Push an item on the stack
    void push(const int item);

    // Pop an item from the stack
    int pop(void);
};

inline void stack::init(void)
{
    count = 0; // Zero the stack
}

inline void stack::push(const int item)
{
    data[count] = item;
    ++count;
}

inline int stack::pop(void)
{
    // Stack goes down by one
    --count;

    // Then we return the top value
    return (data[count]);
}

// A short routine to test the stack
int main()
{
    stack a_stack; // Stack we want to use

    a_stack.init();

    // Push three values on the stack
    a_stack.push(1);
    a_stack.push(2);
    a_stack.push(3);

    // Pop the items from the stack
    cout << "Expect a 3 ->" << a_stack.pop() << '\n';
    cout << "Expect a 2 ->" << a_stack.pop() << '\n';
    cout << "Expect a 1 ->" << a_stack.pop() << '\n';

    return (0);
}
```

Run the above program and see if it is working as expected.

7.13 Friend Classes

In C++, a friend class is a class that has been granted access to the private and protected members of another class. This means that a friend class can access and manipulate the members of the class it is befriended with, even if those members are declared private or protected. For Example:

```
class item {
 private:
 int data;
  friend class setof_items;
};

class set_of_items {
 // ...
};
```

`setof_items` is a friend class to class `item`. Here is another example.

```
class A {
private:
    int x;
public:
    A() { x = 0; }
    friend class B;
};

class B {
public:
    void setX(A &a, int val) { a.x = val; }
    int getX(A &a) { return a.x; }
};
```

In this example, class `A` has a private member variable `x` and class `B` is declared as a friend of class `A`. This means that class `B` has access to the private member variable `x` of class `A`. The class `B` has two member functions, `setX` and `getX`, which can manipulate the private member variable `x` of class `A`.

You can also make a friend function in C++ which is a non-member function that has been granted access to the private and protected members of a class. Here is an example of a friend function:

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

class A {
private:
    int x;
public:
    A() { x = 0; }
    friend int multiply(A &a);
};

int multiply(A &a) {
    return a.x * 2;
}

int main() {
    A obj;
    cout << "Result: " << multiply(obj) << endl;
    return 0;
}
```

In this example, the function `multiply` is a friend function of class `A`, which means it can access the private member variable `x` of class `A`. The function `multiply` takes an object of class `A` as an input and returns the value of `x` multiplied by 2.

It's worth noting that friend classes and functions are generally considered to be a code smell, as they can make code more difficult to understand and maintain. They are used in situations where it is deemed necessary to grant another class or function access to the private or protected members of a class, but in general, it's best to minimize the use of friend classes and functions.

What will be the output of the above program?

7.14 Polymorphism

Polymorphism means one name with many forms. In C++, polymorphism is a mechanism in which the same function, name, and operator can be used to operate on different input parameters or we can define polymorphism as the ability of a message to be displayed in more than one form. A real-life example of polymorphism is a person who at the same time can have different characteristics. Like a man at the same time is a father, a husband, and an employee. So, the same person possesses different behaviours in different situations. This is called polymorphism. Polymorphism is considered one of the important features of Object-Oriented Programming.

7.14.1 Types of Polymorphism

Polymorphism is categorised into two types.

- Compile time Polymorphism
- Runtime Polymorphism

Compiled-time Polymorphism

In compile-time polymorphism, a function is called at the time of program compilation. We call this type of polymorphism as early binding or Static binding. Function overloading and operator overloading is the type of Compile time polymorphism.

Function Overloading Function overloading means one function can perform many tasks. In C++, a single function is used to perform many tasks with the same name and different types of arguments. In the function overloading function will call at the time of program compilation. It is an example of compile-time polymorphism.

In the below example, A function `ADD()` is used to perform two tasks. The two `ADD`s would be to add two integer values and add two strings (concatenate). Readability of the program increases by function overloading. It is achieved by using the same name for the same action.

Source Code:-

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

class Addition {
public:
    int ADD(int X, int Y) { // Function with parameter
        return X + Y;     // this function is performing the addition
           ↪ of two Integer values
    }
    int ADD() {     // Function with the same name but without
         ↪ parameter
        string a = "HELLO";
        string b = "SAM";   // in this function concatenation is
           ↪ performed
        string c = a + b;
        cout << c << endl;
    }
};

int main(void) {
    Addition obj;   // Object is created
    cout << obj.ADD(128, 15) << endl; // first method is called
    obj.ADD();  // second method is called
    return 0;
}
```

In the above example code, we use the function `ADD()` to perform two different tasks which is a property of polymorphism.

Operator Overloading Operator overloading means defining additional tasks to operators without changing their actual meaning. We do this by using the operator function. The purpose of operator overloading is to provide a special meaning to the user-defined data types. The advantage of Operator overloading is to perform different operations on the same operand. **Source Code**:

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

class A
{
    string x;
public:
    A(){}
    A(string i)
    {
        x = i;
    }
    void operator+(A);
    void display();
};

void A::operator+(A a)
{
    string m = x + a.x;
    cout << "The result of the addition of two objects is : " << m;
}

int main()
{
    A a1("Welcome");
    A a2("back");
    a1 + a2;
    return 0;
}
```

Runtime Polymorphism

In a Runtime polymorphism, functions are called at the time of the program execution. Hence, it is known as late binding or dynamic binding. Function overriding is a part of runtime polymorphism. In function overriding, more than one method has the same name with different types of the parameter list. It is achieved by using virtual functions and pointers. It provides slow execution as it is known at the run time. Thus, It is more flexible as all the things are executed at the run time.

Function overriding In function overriding, we give the new definition to the base class function in the derived class. At that time, we can say the base function has been overridden. It can be only possible in the 'derived class'. In function overriding, we have two definitions of the same function, one in the superclass and one in the derived class. The decision about which function definition requires calling happens at runtime. That is the reason we call it 'Runtime polymorphism'.

7.15 Inheritance

Inheritance is a mechanism in object-oriented programming (OOP) that allows a new class to inherit properties and methods from an existing class. The existing class is called the base class or parent class and the new class is called the derived class or child class.

The derived class can use the properties and methods of the base class as if they were its own, and it can also add new properties and methods or override existing ones. This allows for code reuse and the ability to model real-world relationships between classes.

For example, let's say we have a base class called `Animals` that contains properties and methods that are common to all animals, such as the `eat()` method and the `move()` method.

```
class Animals {
    public:
        void eat();
        void move();
};
```

We can then create a derived class called `Mammals` that inherits from the `Animals` class. The `Mammals` class can use the properties and methods of the `Animals` class as if they were its own, and it can also add new properties and methods or override existing ones.

```
class Mammals: public Animals {
    public:
        void giveBirth();
};
```

In this example, the class `Mammals` inherits all the properties and methods from the class `Animals`, and it can also have its own properties and methods, in this case the method `giveBirth`.

Note that inheritance is represented in C++ using the `:` operator in the class definition.

7.16 Encapsulation

Encapsulation is a mechanism in object-oriented programming (OOP) that allows you to hide the implementation details of a class from the user. It is a technique for protecting the data and methods of a class from external access and modification, and it promotes the principle of data hiding.

In C++, encapsulation is achieved by using the access modifiers `public`, `private`, and `protected` in the class definition. The `public` access modifier allows members of the class to be accessed from anywhere, while the `private` access modifier makes the members of the class only accessible within the class itself.

For example, let's say we have a class called `BankAccount` that has a private member variable called `balance` that stores the current balance of the account. We also have a public method called `deposit()` that allows the user to deposit money into the account, and a public method called `withdraw()` that allows the user to withdraw money from the account.

```
class BankAccount {
    private:
        double balance;
    public:
        void deposit(double amount);
        void withdraw(double amount);
};
```

In this example, the class `BankAccount` encapsulates the data (`balance`) and behaviour (`deposit` and `withdraw`) related to the bank account in one place. The user can deposit and withdraw money, but the user can't access or modify the balance directly. This ensures data integrity and security by hiding the implementation details of the class from the user.

Encapsulation is an important principle in OOP as it allows you to change the implementation of the class without affecting the user of the class and it also makes the class more secure, maintainable and reusable.

```
#include <iostream>
using namespace std;
class Animal {
    public:
void function(){
cout<<"Eating..."<<endl;
    }
};
class Man: public Animal
{
 public:
 void function()
    {
       cout<<"Walking ..."<<endl;
    }
};
int main(void) {
 Animal A =Animal();
   A.function();   //parent class object
   Man m = Man();
   m.function();   // child class object

   return 0;
}
```

Explanation The above code is an example of inheritance and polymorphism. The code defines a base class `Animal` with a member function `function()` that outputs, `"Eating..."`. Then it creates a derived class `Man` that inherits from the `Animal` class and overrides the `function()` member function to output `"Walking..."` instead.

In the main function, the code creates an object of the base class `Animal` and an object of the derived class `Man`, and calls the `function()` member function on both.

Since the derived class `Man` has overridden the `function()` member function, when it is called on the `Man` object it will output `"Walking..."`, while when it is called on the `Animal` object it will output `"Eating..."`. This is an example of polymorphism.

7.17 Virtual Function

A virtual function is declared by the keyword `virtual`. The return type of virtual function may be `int`, `float`, or `void`. A virtual function is a member function in the base class. We can redefine it in a derived class. It is part of run time polymorphism. The declaration of the virtual function must be in the base class by using the keyword `virtual`. A virtual function is not static.

The virtual function helps to tell the compiler to perform dynamic binding or late binding on the function. If it is necessary to use a single pointer to refer to all the different classes' objects. This is because we will have to create a pointer to the base class that refers to all the derived objects. But, when the base class pointer contains the derived class address, the object always executes the base class function. To resolve this problem, we use the virtual function. When we declare a virtual function, the compiler determines which function to invoke at runtime. Let's see the below example to understand how the program execution happens without a virtual function and with a virtual function.

```
//without virtual Function

#include <iostream>
using namespace std;
class Add
{
   int x=5, y=20;
    public:
    void display()  //overridden function
    {
        cout << "Value of x is : " << x+y<<endl;
    }
};
class Subtract: public Add
{
    int y = 10,z=30;
    public:
    void display()  //overridden function
    {
        cout << "Value of y is : " <<y-z<<endl;
    }
};
int main()
{
    Add *m;     //base class pointer .it can only access the base class
       ↪  members
    Subtract s;     // making object of the derived class
    m = &s;
   m->display();      // Accessing the function by using the base class
      ↪  pointer
    return 0;
}
```

7.17.1 Example

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

class AA {
public:
    void print() const;
    int sum();
    AA(int a, int b);
private:
    int x;
    int y;
};

void AA::print() const {
    cout << "x: " << x << " y: " << y << endl;
}

int AA::sum() {
    return x + y;
}

AA::AA(int a, int b) {
    x = a;
    y = b;
}

int main() {
    AA obj(10,20);
    obj.print();
    cout<<"Sum : "<<obj.sum()<<endl;
    return 0;
}
```

Explanation of above code: This C++ code defines a class called `AA` with three member functions: `print()`, `sum()`, and a constructor. It also has two private member variables `x` and `y`.

The `print()` member function is a constant function that prints out the values of the private member variables `x` and `y` using the `cout` statement.

The `sum()` member function returns the sum of the values of the private member variables `x` and `y`.

The constructor is a member function with the same name as the class, and it initializes the private member variables `x` and `y` with the values passed as arguments (`a` and `b`) when an object of the class is created.

In the main function, an object `obj` of the class `AA` is created with `x = 10` and `y = 20`. Then the `print()` function is called on this object, which will output the values of `x` and `y` on the console. Finally, the `sum()` function is called on this object which will output the sum of `x` and `y` on the console.

Overall, this class is a simple example of encapsulation. The class encapsulates the data (`x` and `y`) and behavior (printing and summing) related to the data in one place, which makes the code more maintainable and reusable.

7.18 Chapter Quick Reminder

- A class is a group of objects sharing common data (properties) and operations (functions).
- Components of a class are called members.
- Class members are accessed by name.
- In C++, `class` is a reserved word and cannot be used as an identifier.
- Class members are classified into:
  - Private (not accessible outside the class)
  - Public (accessible outside the class)
  - Protected
- By default, all class members are private.
- Public members are declared using the access specifier `public` followed by a colon (`:`).
- A class member can be:
  - A variable
  - A function (declared using a function prototype)
- Class variables are also called class objects, class instances, or objects.
- A class has two parts: class declaration and function definition.
- Objects are basic runtime entities in an object-oriented program.
- Private data of a class can only be accessed through its member functions.
- Methods are functions belonging to a class.
- Member functions can be defined:
  - Inside the class function
  - Outside the class definition
- Private member functions are hidden from other functions, except those of its class.
- Constructors:
  - Special member functions that initialize class objects
  - Types: default, parametrized, and copy constructors
  - Default constructor accepts no parameters
  - Parametrized constructor accepts parameters
  - Copy constructor declares and initializes an object from another object
  - Execute when a class object enters its scope
- Destructors:
  - Execute when a class object goes out of scope
  - A class can have only one destructor
- A class can have multiple constructors.
- Inheritance:

  - Allows a class to inherit properties and methods from another class
  - Enables code reuse and organized class hierarchy
- Encapsulation:
  - Hides implementation details of a class
  - Provides a public interface for accessing properties and methods
- Polymorphism:
  - Allows a single function or operator to work with multiple data types
  - Implemented through virtual functions and function overloading

Exercise Questions

True or False

- All member variables of a class must be of the same type.
- All member functions of a class must be public.
- A class can have more than one constructor.
- A class can have more than one destructor.
- Both constructors and destructors can have parameters.

Find what is wrong in the syntax of each of the following class definitions

```cpp
class AA {
public:
    void print() const;
    int sum();
    AA();
private:
    int x;
    int y;
};
```

```cpp
class BB {
    int one;
    int two;
public:
    bool equal() const;
    print();
    BB(int, int);
}
```

```cpp
class CC {
public;
    void set(int, int);
    void print() const;
    CC();
    CC(int, int);
private:
    int u;
    Int v;
};
```

Consider the following class definition

```cpp
class DD {
public:
    DD();  // Line 1
    DD(int);  // Line 2
    DD(int, int);  // Line 3
    DD(double, int);  // Line 4
private:
    int u;
    double v;
};
```

a. Give the line number of the constructor that is executed in each of the declarations below

- `DD one;`
- `DD two(5, 6);`
- `DD three(3.5, 8);`

b. Write the definition of the constructor in Line 2 so that the private member variable `u` is initialized according to the value of the parameter, and the private member `v` is initialized to 0.

c. Write the definition of the constructor in Line 1 so that the private member variables are initialized to 0.

Consider the following class definition

```cpp
class testClass {
public:
    int sum();
    // Returns the sum of the private member variables
    void print() const;
    // Prints the values of the private member variables
    testClass();
    // default constructor
    // initializes private member variables to 0
    testClass(int a, int b);
    // constructor with parameters
    // Initializes the private member variables to the values
    // specified by the parameters
    // postcondition: x=a; y = b;
private:
    int x;
    int y;
};
```

Write the definitions of the member functions as described in the definition of the class `testClass`.

Write a test program to test the various operations of the class `testClass`.

Write a C++ program to declare a member function and attempt any operation within it.

Write a C++ program by using member functions outside the body of the class to find the area of a rectangle.

Write a program that converts a number entered in Roman numerals to a decimal number. Your program should consist of a class, say, `RomanNum`. An object of type `RomanNum` should do the following:

- Store the number as a Roman numeral.
- Convert and store the number as a decimal number.
- Print the number as a Roman numeral or decimal number as requested by the user.
- Test your program using the following Roman numerals: MCXIV, CCCLIX, MDCLXVI.

Answers to Some Review Questions

Question 6: Write a C++ program by using member functions outside the body of the class to find the area of a rectangle.

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

class Rectangle {
    private:
        int length;
        int width;
    public:
        void setLength(int l);
        void setWidth(int w);
        int getArea();
};

void Rectangle::setLength(int l) {
    length = l;
}

void Rectangle::setWidth(int w) {
    width = w;
}

int Rectangle::getArea() {
    return length * width;
}

int main() {
    Rectangle rect;
    rect.setLength(5);
    rect.setWidth(10);
    cout << "Area of rectangle: " << rect.getArea() << endl;
    return 0;
}
```

Explanation

The above program uses a class called `Rectangle` to represent a rectangle object. The class has private member variables for the length and width of the rectangle, and public member functions to set the length and width, and to calculate and return the area of the rectangle. The member functions are defined outside of the class body. In the main function, an object of the `Rectangle` class is created, and its length and width are set using the `setLength` and `setWidth` member functions. The area of the rectangle is then calculated and printed using the `getArea` member function.

Question 8: Solution

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

class RomanNum {
    private:
        string roman;
        int decimal;
        map<char, int> roman_to_decimal = {{'I', 1}, {'V', 5}, {'X',
            ↪ 10}, {'L', 50}, {'C', 100}, {'D', 500}, {'M', 1000}};

    public:
        RomanNum(string r) {
            roman = r;
            decimal = convertToDecimal();
        }

        int convertToDecimal() {
            int result = 0;
            for (int i = 0; i < roman.size(); i++) {
                if (i > 0 && roman_to_decimal[roman[i]] >
                   ↪ roman_to_decimal[roman[i - 1]]) {
                    result += roman_to_decimal[roman[i]] - 2 *
                        ↪ roman_to_decimal[roman[i - 1]];
                } else {
                    result += roman_to_decimal[roman[i]];
                }
            }
            return result;
        }

        void printRoman() {
            cout << roman << endl;
        }

        void printDecimal() {
            cout << decimal << endl;
        }
};

int main() {
    RomanNum rn1("MCXIV");
    cout << "rn1 as Roman numeral: ";
    rn1.printRoman();
    cout << "rn1 as decimal: ";
    rn1.printDecimal();

    RomanNum rn2("CCCLIX");
    cout << "rn2 as Roman numeral: ";
    rn2.printRoman();
    cout << "rn2 as decimal: ";
    rn2.printDecimal();

    RomanNum rn3("MDCLXVI");
    cout << "rn3 as Roman numeral: ";
    rn3.printRoman();
    cout << "rn3 as decimal: ";
    rn3.printDecimal();

    return 0;
}
```

Bibliography

Bancila, M. (2017). *Modern C++ Programming Cookbook*. Packt Publishing.

Dmitrović, S. (2020). *Modern C++ for Absolute Beginners: A Friendly Introduction*. Packt Publishing.

Farrell, J. (2014). *Object-Oriented Programming Using C++* (4th ed.). Cengage Learning.

Grimes, R. (2017). *Beginning C++ Programming*. Wrox.

Lospinoso, J. (2019). *C++ Crash Course: A Fast-Paced Introduction*. Addison-Wesley Professional.

Malik, D. S. (2010). *C++ Programming: Program Design Including Data Structures* (5th ed.). Cengage Learning.

Malik, D. S. (2017). *C++ Programming: From Problem Analysis to Program Design*. Cengage Learning.

Pandey, H. M. (2017). *Object-Oriented Programming C++ Simplified*. BPB Publications.

Procode Publishing. (2019). *The C++ Programming Language* (5th ed.). Addison-Wesley Professional.

Sahay, S. (2012). *Object Oriented Programming with C++* (2nd ed.). Pearson Education.

Stroustrup, B. (2013). *The C++ Programming Language* (4th ed.). Addison-Wesley Professional.

Stroustrup, B. (2014). *Programming: Principles and Practice Using C++*. Addison-Wesley Professional.

Zak, D. (2016). *An Introduction to Programming with C++*. Oxford University Press.

About the Authors

DR. OLADELE O. CAMPBELL is a distinguished computer science educator with nearly three decades of experience, currently lecturing at Niger State Polytechnic, Zungeru, Nigeria. He holds a PhD in Computer Science Education from the University of South Africa and both an MSc and BSc in Computer Science from the University of Ibadan. Dr. Campbell has mentored numerous students and staff, with expertise in Intelligent Tutoring Systems, software engineering, and programming education. He is a member of Nigeria Computer Society and a Senior Member of the Association for Computing Machinery (ACM) and has contributed to top-tier international journals and conferences. Dr. Campbell has reviewed for several prestigious journals and conferences. Married with four children, he balances his professional commitments with family life and pastoral duties in a local church. His book on C++ programming is a comprehensive guide for beginner and intermediate programmers.

Connect with Dr. Campbell:

- LinkedIn (Oladele Campbell, PhD)
- Twitter (@DeleCampbell)
- Email: 51898772@mylife.unisa.ac.za
- WhatsApp: +2349028482222

PETER BAKE holds a National Diploma, Higher National Diploma, Postgraduate Diploma, MSc, and BSc in Computer Science from various institutions. He is currently a lecturer at Niger State Polytechnic, Zungeru, and is pursuing a PhD in Computer Science at Nasarawa State University. His research focuses on Human-Computer Interaction.

Connect with Peter:

- Email: bakepeters@gmail.com
- WhatsApp: 07063887148
- LinkedIn: [Peter Bake]

AISHATU ALIYU MOHAMMED is an Assistant Chief Technologist in the Computer Science Department at Niger State Polytechnic, Zungeru. She holds an HND in Computer Science from Kaduna Polytechnic and an M.Tech in Computer Science from the Federal University of Technology, Minna. Aishatu has been conducting and supervising students' labs and training them in practical programming for over seven years.

Connect with Aishatu:

- Email: aishatu.aliyu@example.com
- WhatsApp: 08061570914
- LinkedIn: [Aishatu Aliyu Mohammed]

ISAH ADAMU DAGAH holds a Bachelor of Engineering and a Master of Engineering in Computer Engineering from the Federal University of Technology, Minna. He is currently pursuing his PhD in Computer Engineering. Isah began his career as a Field Support Engineer and is now a Principal Lecturer and Director of ICT at Niger State Polytechnic, Zungeru. His research areas include digital image processing, IoT, machine learning, and cloud computing. He is a member of the Nigeria Computer Society, Nigeria Society of Engineers, and the Association for Computing Machinery.

Connect with Isah:

- ✉Email: dagahtech@gmail.com
- WhatsApp: 08036025393
- LinkedIn: [https://www.linkedin.com/in/adamu-isah-dagahbbb14676]