Master C Language

Basic to Advance

Pawannpreet Singh

Our Website

www.digitalcolleglibrary.com

Scan this QR code to go on website

Digital College Library

@just_c0des_

Content

Content

Chapter – 4 Arrays

Chapter – 6 Structures and Unions

6.1 Introduction

- Structures
- Unions

6.2 Declaring – Defining Structures

- Structure Declaration
- Structure Definitions
- Accessing Structures Members
- Initializing Structure Variables

6.3 Passing Structures to Function

- Passing Structure by Value
- Passing Structure by Reference

6.4 Nested Structures

- Define the Inner Structures
- Define the Outer Structure
- Create and Use Nested Structures

6.5 Accessing Union Members

- Define the Union
- Create a Union Variable
- Access Union Members
- Avoid Conflicting Access

6.6 “typedef” for Structures and Unions

- Define the Structure or Union
- Use “typedef” to create an Alias
- Declare Variables Using the Alias

Chapter – 7 File Handling

1

C Basics

1.1 Introduction about “C”

“Dennis Ritchie” created the C programming language at Bell Labs in the early 1970s, and it is still used today and is one of the most popular and robust languages used in computer programming. Its portability, speed, and clear yet expressive syntax have made it a cornerstone for developing a variety of software applications, operating systems, embedded devices, and more.

C is fundamentally a procedural programming language that places a strong emphasis on organized programming methods. It offers a wide variety of primitive data types, including pointers, characters, integers, and floating-point numbers. A key idea in C is the concept of pointers, which allows for direct memory management and dynamic memory allocation.

The syntax of C is known for its strength and simplicity. The language expresses complex algorithms and logic using a combination of functions and control structures. Functions are programmable building blocks that take arguments, carry out calculations, and return results. The use of conditionals (if, else), loops (for, while), and switches in control structures enables programmers to modify the execution's course in response to various circumstances.

The ability to manage memory in C is both a strength and a limitation. With the explicit memory manipulation provided by pointers in C, it is possible to access low-level hardware resources and use memory effectively. However, with this ability comes the duty of controlling memory allocation and deallocation, which if not managed effectively can result in problems like memory leaks and segmentation errors.

Many functions are available in the wide standard library of C to carry out tasks ranging from input/output operations to mathematical computations and string manipulations. By using already existing functionality, the standard library provides a solid basis for creating more complicated applications.

1.2 First “C” Program

Codes: -

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

Explanation about Codes: -

- **#include <stdio.h>:-** This line includes the standard input-output library “<stdio.h>” in the program. It provides functions like “printf” for input and output operations.

- **int main() { ... }:-** This is the main function, the entry point of the program. Every C program must have a main function. It returns an integer “int” value to the operating system, indicating the program's exit status.
- **printf("Hello, World!\n");:-** This line uses the “printf” function to print the text "Hello, World!" to the console. The “\n” is an escape sequence representing a newline character, which moves the cursor to the next line after printing.
- **return 0;:-** The return statement ends the main function and returns the value 0 to the operating system. A return value of 0 conventionally indicates a successful execution of the program.

How the Flow of First Program Works…….???

- The “#include” directive is used to include the content of the specified file (“stdio.h” in this case) into your program before compilation. This allows you to use the functions and declarations defined in that file.
- The “main” function is where the execution of the program begins. It's followed by a pair of curly braces “{....}”, which enclose the statements that make up the body of the function.
- The “printf” function is used to print formatted text to the console. In this case, it's printing the text "Hello, World!" followed by a newline character.
- The “return 0;” statement ends the “main” function and returns the value 0 to the operating system. This value indicates that the program executed successfully.

1.3 Variables

A variable in C is a specifically defined area of memory where a value is stored. These values can come from of several data types, including characters, integers, and floating-point numbers. You can store and change data using variables, which gives your programs flexibility and dynamic. Let's explore variables in C and look at a few examples of how to use them.

1.3.1 Declaring Variables

Before they can be used, variables must first be defined. In a variable declaration, the data type and name that will be used to identify the variable are both specified. Declaring a variable uses the following syntax in general:

```
data_type variable_name;
```

For example, let's declare a few variables of different data types:

```
int age;

float temperature;

char firstInitial;
```

In this example, we've declared an **integer** variable named “age”, a **floating-point** variable named “temperature”, and a **character** variable named “firstInitial”.

1.3.2 Initializing Variables

Giving a variable its first value at the moment of declaration is referred to as initialization. Variables without initialization have unexpected values, which might result in issues. The following describes how to define and initialize variables:

```
int score = 100;

float pi = 3.14159;

char grade = 'A';
```

In this example, the variables score, pi, and grade are declared and immediately assigned initial values.

1.3.3 Assigning Values

You can change the value of a variable after its initialization using the assignment operator (=). For example:

```
score = 95;

pi = 3.14;

grade = 'B';
```

Example of Variables: -

```
#include <stdio.h>
int main() {
        // Declare a variable named 'age' of type int
            int age;
```

```
    // Assign a value to the 'age' variable
        age = 25;

    // Print the value of the 'age' variable
        printf("My age is: %d\n", age);
    return 0;
}
```

1.4 Data Types

Data types are used in the C programming language to specify the kind of data that a variable can store. They decide how much memory is allotted to a variable and what kinds of values it can hold. To effectively manage various sorts of data, C offers a wide range of data types.

C supports several basic data types, which are classified into three categories: integer, floating-point, and character.

1.4.1 Integer Data Type

To express entire integers without decimal points, integer data types are utilized. Numbers like 0, 1, -3, 42, and other values are stored in integer data types. You may select the best type based on the demands of your program since C offers a variety of integer data

types with varying sizes and ranges. In C, the following integer data types are often used:

- **int : -** The most common integer data type in C. It typically uses 4 bytes of memory and can store a wide range of values, usually from -2147483648 to 2147483647.
- **Short int:-** A smaller integer data type that uses 2 bytes of memory. It has a smaller range compared to “int”, often from -32768 to 32767.
- **Long int:** - A larger integer data type that uses 4 or 8 bytes of memory, depending on the system. It has a broader range than “int”.
- **Long Long int:** - An even larger integer data type that uses at least 8 bytes of memory and has a wider range compared to “long int”.

Here's an **example** demonstrating the usage of integer data types in C:

```
#include <stdio.h>


    int main() {

        int age = 25;

            short int smallNumber = 100;

            long int bigNumber = 1000000;

            long long int hugeNumber = 123456789012345;
```

```
    printf("Age: %d\n", age);
    printf("Small Number: %hd\n", smallNumber);
    printf("Big Number: %ld\n", bigNumber);
    printf("Huge Number: %lld\n", hugeNumber);

    return 0;
}
```

In this example, we've declared and initialized variables using different integer data types. The “%d” format specifier is used for “int”, “%hd” for “short int”, “%ld” for “long int”, and “%lld” for “long long int” when printing their values using “printf”.

1.4.2 Floating-Point Data Type

Real numbers containing both integer and fractional components, as well as numbers with decimal points, are represented using floating-point data types. When you need to conduct computations using decimal values, you utilize floating-point numbers. The two main floating-point data types offered by C are float and double. Now let's explore these data types:

- **float: -** For storing “floating-point” integers with a single degree of accuracy, use the “float” data type. The average memory use is 4 bytes. It has an accuracy of about 7 decimal digits for representing integers.

Example of using the **float** data type:

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

```
int main() {
        float pi = 3.14159;
            printf("Value of pi: %f\n", pi);

    return 0;
}
```

- **double: -** Floating-point values with double precision are stored using the double data type. Typically, 8 bytes are used as memory. It has an accuracy of about 15 decimal digits for representing integers.

Example of using the **double data** type:

```
#include <stdio.h>
    int main() {
            double salary = 55000.75;
                printf("Salary: %lf\n", salary);

    return 0;
}
```

In both examples, the “printf” function is used to print the values of floating-point variables. The format specifiers “%f” and “%lf” are used for float and double variables respectively.

1.4.3 Character Data Type

Letters, numbers, symbols, and control characters are all represented by the character data type "char". letters from the ASCII character set, which includes alphanumeric letters, punctuation marks, and special control characters, are commonly stored in the "char" data type, which has a size of 1 byte.

Here's a breakdown of how the **char** data type is used in C:

Declaration and Initialization: - You can declare and initialize a “char” variable like this:

```
char myChar = 'A';
```

Here is “myChar” variable is declared and assigned the character 'A'.

Character Literals: - Character literals are enclosed in single quotes. For example:

```
char letter = 'B';
```

Escape Sequences: - Escape sequences are unique groups of characters that start with a backslash (\) and stand in for special or unprintable characters. Typical escape patterns include:

- '\n': Newline
- '\t': Tab
- '\r': Carriage return
- '\\': Backslash
- '\'': Single quote

Printing Character: - You can use the “%c” format specifier in the “printf” function to print characters:

```
char myChar = 'Z';
printf("Character: %c\n", myChar);
```

Reading Characters: - You can use the “%c” format specifier with the “scanf” function to read characters from input:

```
char userInput;
printf("Enter a character: ");
scanf("%c", &userInput);
printf("You entered: %c\n", userInput);
```

1.5 Constants

Constants are fixed values that remain the same while a program is being run. They are used to represent things like mathematical constants, configuration parameters, and more that remain constant during the course of the program. By giving meaningful names to quantities that may otherwise appear in your code as "magic numbers," constants make your code easier to understand, and maintain, and make fewer mistakes. Here is how C defines and employs constants:

1.5.1 Defining Constants

Constants in C can be defined using the “#define” preprocessor directive or using the “const” keyword.

Using #define

A constant value is given a symbolic name via the "#define" directive, making it simpler to utilize throughout your program. Its syntax is as follows:

Syntax:

```
#define CONSTANT_NAME value
```

Example:

```
#define PI 3.14159
#define MAX_LENGTH 100
```

Using “const” keyword

The term "const" defines a variable as a constant. As a result, type checking is provided, and the compiler is able to detect accidental changes to the constant value. As for the syntax:

Syntax:

```
const data_type CONSTANT_NAME = value;
```

Example:

```
const int MAX_ATTEMPTS = 3;
const double SPEED_OF_LIGHT = 299792458.0;
```

1.5.2 Using Constants

Once constants are defined, they can be used in your code wherever you would use the corresponding literal values.

Example:

```
#include <stdio.h>

#define PI 3.14159
        const int MAX_SCORE = 100;

int main() {
    double radius = 5.0;
    double area = PI * radius * radius;

int score = 85;
    if (score > MAX_SCORE) {
        printf("Invalid score\n");
    }
```

```
    return 0;
}
```

Explanation about Codes:

- **#include <stdio.h>:** This line includes the standard input-output library.
- **#define PI 3.14159:** Defines a constant PI with the value 3.14159.
- **const int MAX_SCORE = 100;:** Declares a constant MAX_SCORE with the value 100 using the const keyword.
- **double area = PI * radius * radius;:** Calculates the area of a circle using the constant PI.
- **int score = 85;:** Initializes the score variable.
- **if (score > MAX_SCORE) { ... }:** Compares the score with the MAX_SCORE constant.

1.6 Basic Input/output Function

For communicating with users and showing information on the screen, basic input/output (I/O) operations are necessary. "printf()" for output and "scanf()" for input are two essential I/O routines. These functions allow for communication between the user and the program and are a part of the standard I/O library "stdio.h>". Here is an explanation of these functions' operation with example code:

1.6.1 “printf()” Function

The “printf()” function is used to display output on the screen. It allows you to format and print various types of data, such as text and variables.

Syntax:

```
printf("format_string", argument_list);
```

- **format_string:** This is a string containing placeholders that determine how the output will be formatted.
- **argument_list:** This is a list of values that correspond to the placeholders in the format string.

Example:

```
#include <stdio.h>

int main() {
        int age = 25;
            printf("My age is %d years.\n", age);
    return 0;
}
```

In this example, the “printf()” function is used to display the text "My age is " followed by the value of the age variable. The “%d” placeholder is used to indicate where the integer value should be inserted in the output.

1.6.2 “scanf()” Function

The “scanf()” function is used to read input from the user. It allows you to receive data and store it in variables.

Syntax:

```
scanf("format_string", &variable1, &variable2, ...);
```

- **format_string:** Similar to printf(), this string contains placeholders that specify the data type of the input.
- **&variable1, &variable2, ...:** These are memory addresses where the input data will be stored.

Example:

```
#include <stdio.h>

int main() {
        int age;
        printf("Enter your age: ");
        scanf("%d", &age);
    printf("You entered: %d years.\n", age);
    return 0;
}
```

In this example, the “scanf()” function is used to read an integer input from the user. The “%d” placeholder in the “format_string”

indicates that the input should be treated as an integer. The “&age” argument passes the memory address of the age variable, allowing “scanf()” to store the input value in that variable.

1.7 Operators and Expression

Fundamental ideas like operators and expressions are utilized to carry out calculations and work with data. Expressions are mixtures of values, variables, and operators that produce a single value. Operators are symbols that denote operations. Building complicated calculations and logic into your programs requires a solid understanding of how operators operate and how to construct expressions.

1.7.1 Operators

C provides a variety of operators that can be categorized into several groups:

i. Arithmetic Operator: These operators perform basic arithmetic calculations.

Symbol	**Name**
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus

ii. Relational Operator: These operators compare two values.

Symbol	**Name**
==	Equal to
!=	Not Equal to
<	Less Than
>	Greater Than
<=	Less than or Equal to
>=	Greater than or Equal to

iii. Logical Operator: These operators are used to combine and manipulate logical values (0 for false and 1 for true).

Symbol	**Name**
&&	Logical AND
||	Logical OR
!	Logical NOT

iv. Assignment Operator: These operators assign values to variables.

Symbol	**Name**
=	Assign
+=	Add and Assign
-=	Subtract and Assign
*=	Multiply and Assign
/=	Divide and Assign
%=	Modulus and Assign

v. Increment and Decrement Operator:

Symbol	Name
++	Increment
--	Decrement

1.7.2 Expressions

A set of values, variables, and operators combined to produce a single value is called an expression. Expressions can range in complexity from a single value to a combination of several processes. Assignments, computations, conditions, loops, and other things can all utilize them.

Example: Arithmetic Expression

```
#include <stdio.h>

        int main() {

                int a = 10, b = 5, c = 7;

                int sum = a + b;

                int product = a * c;

                float division = (float)a / b;

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

            printf("Product: %d\n", product);

             printf("Division: %.2f\n", division);

      return 0;
```

```
}
```

In this example, arithmetic expressions are used to calculate the sum, product, and division of variables a, b, and c. The “float)a / b” expression ensures that the division result is calculated as a floating-point value to preserve decimal places.

Example: Logical Expressions

```
#include <stdio.h>


int main() {
    int x = 5, y = 10;
 int result1 = (x > 3) && (y <= 10);
 int result2 = (x == 5) || (y > 12);


    printf("Result 1: %d\n", result1);
    printf("Result 2: %d\n", result2);


 return 0;
}
```

Logical expressions are used to evaluate conditions. In this example, “result1” evaluates to 1 (true) because both conditions are true. “result2” evaluates to 0 (false) because only the second condition is true.

2

Control Statements

2.1 Introduction

Control statements in the C programming language are essential constructs that allow you to dictate the flow of a program. They determine the sequence in which instructions are executed, based on conditions and loops. Control statements help you create flexible and dynamic programs by enabling you to make decisions and repeat actions as needed. In C, there are mainly four types of control statements: conditional statements, switch case statements, loops, and jump statements.

2.2 Conditional Statements

The essential building blocks of the C programming language are conditional statements, which let you modify the course of your program depending on specific circumstances. Based on the values of variables or expressions, these statements let your program to take actions, run specified blocks of code, and handle various circumstances. The if, else, and else constructs are generally used in C to implement conditional statements. The switch statement also offers an alternate method for dealing with many circumstances which we will study in 2.3 section. We'll go in-depth on conditional

statements in this discussion, including their syntax, use, and recommended practices.

2.2.1 if Statement

The simplest conditional statement in C is the "if" statement. If a certain condition is true, you are given the option to run a block of code. If the condition is false, the code block is skipped, and the following statement in the program is executed. The if statement has the following syntax:

```
if (condition) {
        // Code to execute if the condition is true
    }
```

Breakdown of Syntax:

- **condition:** An expression or a condition that evaluates to either true (non-zero) or false (zero).
- **{}:** The curly braces define the code block that gets executed if the condition is true.

Example:

```
    int age = 25;
if (age >= 18) {
        printf("You are an adult.\n");
    }
```

In this example, if the age variable is greater than or equal to 18, the message "You are an adult" will be printed.

2.2.2 else Statement

In order to give an alternate code block to execute when the "if" condition is false, the "else" statement is used in combination with the "if" statement. It enables your software to choose an alternative course based on how the circumstance turns out. The following is the syntax:

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

Example:

```
int age = 15;
    if (age >= 18) {
        printf("You are an adult.\n");
    } else {
        printf("You are a minor.\n");
    }
```

In this example, if the age variable is less than 18, the message "You are a minor" will be printed because the else block is executed.

2.2.3 else-if Statement

A more organized method of handling multiple situations is to utilize the "else if" statement. If the first condition is false, you can chain many else if statements after the first if statement to test further conditions. You may design a sequence of condition tests using this construct. The following is the syntax:

```
if (condition1) {
        // Code to execute if condition1 is true
} else if (condition2) {
        // Code to execute if condition2 is true
} else if (condition3) {
        // Code to execute if condition3 is true
} else {
        // Code to execute if none of the conditions are true
}
```

Example:

```
int score = 75;

    if (score >= 90) {
        printf("A grade.\n");
    } else if (score >= 80) {
        printf("B grade.\n");
```

```
    } else if (score >= 70) {
            printf("C grade.\n");
     } else {
            printf("F grade.\n");
}
```

In this example, the program checks the value of score against multiple conditions and prints the corresponding grade.

2.3 Switch Case Statements

There is also the "switch" statement, which can handle several situations. It's especially helpful when you want to compare the value of a single variable or expression to the values of many other constants. The "switch" statement performs a single evaluation of the expression before comparing the outcome to the constant values specified in "case" labels. Here is how to use it:

```
switch (expression) {
        case constant1:
            // Code to execute if expression matches constant1
            break;
        case constant2:
            // Code to execute if expression matches constant2
            break;
    // Add more cases as needed
```

```
    default:
        // Code to execute if none of the cases match
}
```

Example:

```
    char grade = 'B';

    switch (grade) {
        case 'A':
            printf("Excellent!\n");
                break;
        case 'B':
            printf("Good!\n");
                break;
        case 'C':
            printf("Average.\n");
                break;
        default:
            printf("Not specified.\n");
}
```

In this example, the switch statement checks the value of grade and executes the code block corresponding to the matching case label.

2.4 Loops

Loops are important control structures in the C programming language that let you continually run a piece of code based on a given condition. They provide you the ability to handle arrays or collections of data, automate repetitive activities, and execute actions a certain number of times. The "while" loop, the "for" loop, and the "do-while" loop are the three main forms of loops offered by C. We'll examine each form of loop, its syntax, use cases.

2.4.1 while Loop

The while loop is a fundamental looping construct in C. It repeatedly executes a block of code as long as a specified condition is true. The syntax of the “while” loop is as follows:

```
while (condition) {
        // Code to execute while the condition is true
}
```

Breakdown of Syntax:

- **condition:** A Boolean expression or condition that determines whether the loop should continue executing. If the condition is true, the loop continues; if false, it terminates.
- **{}:** The curly braces define the code block to be executed while the condition is true.

Example:

```
int count = 0;
```

```
while (count < 5) {
        printf("Count is %d\n", count);
    count++; // Increment count to eventually exit the loop
}
```

In this example, the while loop runs as long as count is less than 5. It prints the current value of count and increments it in each iteration.

2.4.2 for Loop

The “for” loop is a versatile and widely used loop in C. It allows you to specify loop initialization, condition, and update expressions all in one line. The syntax of the “for” loop is as follows:

```
for (initialization; condition; update) {
        // Code to execute while the condition is true
    }
```

Breakdown of Syntax:

- **initialization:** An expression or statement that initializes loop control variables or sets initial values.
- **condition:** A Boolean expression that determines whether the loop should continue executing. If the condition is true, the loop continues; if false, it terminates.
- **update:** An expression or statement that updates loop control variables after each iteration.
- **{}:** The curly braces define the code block to be executed while the condition is true.

Example:

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

In this example, the "for" loop initializes "i" to 0, checks if i is less than 5, and increments i by 1 in each iteration.

2.4.3 do-while Loop

The "do-while" loop is similar to the "while" loop, but it guarantees that the block of code is executed at least once because it checks the condition after the loop body. The syntax of the "do-while" loop is as follows:

```
do {
    // Code to execute at least once
} while (condition);
```

Breakdown of Syntax:

- **{}:** The curly braces define the code block to be executed at least once.
- **condition:** A Boolean expression or condition that determines whether the loop should continue executing after the first iteration. If the condition is true, the loop continues; if false, it terminates.

Example:

```
int num = 0;


do {
    printf("Num is %d\n", num);
    num++; // Increment num to potentially exit the loop
} while (num < 5);
```

In this example, the “do-while” loop always executes at least once because it evaluates the condition after the first iteration.

2.5 Jump Statements

In the C programming language, jump statements are control structures that let you change the way programs typically execute. They enable you to move control between different areas of the code, usually inside of loops, conditional expressions, or functions. The jump statements "goto", "break", "continue", and "return" are examples. They can be effective tools, but they should only be employed sparingly, as wrong use can result in code that is hard to understand and maintain.

Let's explore each of these jump statements in detail:

2.5.1 goto Statement

The “goto” statement is a rarely used and often discouraged feature in C. It allows you to transfer control to a labeled statement within your program. Here's the basic syntax of the “goto” statement:

```
goto label;
        // ...
label:
    // Code to execute after the goto statement
```

Breakdown of Syntax:

- **goto label;:** This statement transfers control to the labeled statement identified by label.
- **label::** The label is a user-defined identifier followed by a colon. It marks the location in the code where the program will jump to when the goto statement is encountered.

Example:

```
#include <stdio.h>
    int main() {
        int i = 0;
    loop_start:
        if (i < 5) {
            printf("i is %d\n", i);
          i++;
```

```
        goto loop_start; // Jump back to the loop_start label
    }
    return 0;
}
```

In this example, the program uses “goto” to create a loop that prints the value of i while incrementing it until it reaches 5.

2.5.2 break Statement

The “break” statement is used to exit a loop or switch statement prematurely. It's a powerful tool for controlling program flow within loops. When a “break” statement is encountered, the program immediately exits the loop or switch, and execution continues with the next statement after the loop or switch.

Using “break” in loop

```
    for (int i = 0; i < 10; i++) {
        if (i == 5) {
            break; // Exit the loop when i is equal to 5
        }
        printf("i is %d\n", i);
    }
```

In this example, the break statement is used to exit the “for” loop when i becomes equal to 5. This prevents further iterations of the loop.

Using “break” in Switch Statement

```
switch (choice) {
    case 1:
        // Code for option 1
        break;
    case 2:
        // Code for option 2
        break;
    default:
        // Code for other options
}
```

In a “switch” statement, break is used to exit the switch block after a specific case is executed. Without “break”, the program would continue executing code in subsequent cases.

2.5.3 continue Statement

The “continue” statement is used within loops to skip the current iteration and proceed to the next one. It is particularly useful when you want to avoid executing certain code for a specific iteration of a loop.

Using “continue” in Loop

```
for (int i = 0; i < 5; i++) {
        if (i == 2) {
```

```
        continue; // Skip the iteration when i is equal to 2
    }
        printf("i is %d\n", i);
}
```

In this example, the continue statement skips the iteration when i is equal to 2, so "i is 2" is not printed, and the loop continues with the next iteration.

2.5.4 return Statement

The “return” statement is used to exit a function and return a value to the caller. It is a fundamental part of C functions and is often used to communicate results or return control to the calling code. The “return” statement can be used with or without a return value, depending on the function's return type.

Using “return” with Value

```
int add(int a, int b) {
        int result = a + b;
        return result; // Returns the sum of a and b
    }
```

In this example, the return statement returns the computed sum of a and b to the caller.

Using “return” without value (Void Functions)

```
void greet() {
        printf("Hello, world!\n");
        return; // No value is returned; it's optional in void functions
}
```

In a void function, you can use return without a value to exit the function. It's optional in void functions, as shown above.

3

Function

3.1 Introduction

In the C programming language, a function is a named chunk of code that can accept input arguments, carry out actions, and, if you want to return a result. It has a specific capability and enables it to be called from other software areas. The separation of concerns is encouraged through functions, which also improve the general structure and manageability of C programs. They are defined using a particular syntax that has a name, a return type, a list of required parameters, and the body of the function, which contains the actual code to be performed. Functions are important tools for code reuse and abstraction in C since they may be called repeatedly from different places in the program.

3.2 Function Declaration - Definition

A function declaration is a statement that provides the compiler with information about a function's name, existence, return type, and list of parameters without supplying the function's implementation. Before the function is ever declared or called, the compiler can evaluate the accuracy of function calls and their compliance with the defined function using this as a prototype or signature. Function

declarations are essential to the correct modularity, type checking, and structuring of the code in C applications.

Syntax of Function Declaration: A function declaration follows a specific syntax:

return_type function_name(parameter1_type, parameter2_type, ...);

- **return_type:** Specifies the data type of the value that the function will return upon execution. Use “void” if the function doesn't return any value.
- **function_name:** A user-defined identifier that serves as the name of the function.
- **parameter1_type, parameter2_type, etc.:** If the function requires parameters, these specify the data types of the input values that the function expects. Parameters are enclosed within parentheses ().

Example:

```
#include <stdio.h>
// Function declaration
int add(int num1, int num2);
int main() {
int result;
int a = 5;
int b = 3;
```

```
    // Calling the add function
    result = add(a, b);

    printf("The sum of %d and %d is %d\n", a, b, result);

    return 0;
}

// Function definition
int add(int num1, int num2) {
    return num1 + num2;
}
```

Explanation about Example:

- We first include the <stdio.h> header for using “printf”.
- We declare the “add” function before the “main” function. The function declaration specifies the function's name “add”, return type “int”, and parameter list “int num1, int num2”.
- Inside the main function, we declare variables ‘result’, ‘a’, and ‘b’, and then call the “add” function to calculate the “sum” of ‘a’ and ‘b’. This function call is possible because we've provided a function declaration for “add”, which tells the compiler about its signature.
- After calling “add”, we print the result using “printf”.

- Below the main function, we define the add function, which takes two integer parameters (num1 and num2) and returns their sum.

3.3 Parameters and Arguments

In C programming, parameters and arguments are fundamental concepts related to functions. They play a crucial role in passing data into functions and enablg code reusability.

3.3.1 Parameters

Within the parentheses of a function's declaration are placeholders and variables known as parameters. When a function is invoked, it provide the input values that it anticipates. The purpose of parameters is to enable the function to operate with certain data supplied during its invocation. Parameters function as local variables inside the scope of the function. The function (the callee) receives information from the calling code (the caller) through parameters.

Syntax for Parameters:

return_type function_name(parameter_type parameter_name1, parameter_type parameter_name2, ...);

- **return_type:** Specifies the data type of the value the function will return, or it can be void if the function doesn't return anything.
- **function_name:** The name of the function.

- **parameter_type:** The data type of the parameter.
- **parameter_name1, parameter_name2, etc.:** The names assigned to the parameters within the function.

Example:

```
#include <stdio.h>

    // Function declaration with parameters
        void greetUser(char name[]);
int main() {
        char userName[] = "Pawan";

// Calling the greetUser function with the userName as an argument
    greetUser(userName);
      return 0;
}

// Function definition that takes a parameter
    void greetUser(char name[]) {
          printf("Hello, %s!\n", name);
        }
```

Explanation about Example:

- We have a function named “greetUser” that takes one parameter, name, which is an array of characters (a string).
- In the main function, we declare a character array “username” and initialize it with the value "Pawan".
- We then call the “greetUser” function and pass “username” as an argument. This means that the value stored in the “username” array is passed to the name parameter of the “greetUser” function.
- Inside the “greetUser” function, we use the name parameter to display a greeting message using “printf”. The “%s” format “specifier” is used to print the value of the name parameter (the user's name).

3.3.2 Arguments

On the reverse end of arguments are the actual values or expressions passed to a function during a function call. The parameters of the function are initialized using these values. A function is called by passing its arguments, and these arguments must have the same data types and be passed in the same sequence as the parameters listed in the function's declaration.

Syntax for Arguments:

```
result = function_name(argument1, argument2, ...);
```

- **result:** This is where the result of the function call, if it returns a value, is stored.
- **function_name:** The name of the function being called.

- **argument1, argument2, etc.:** The actual values or expressions that correspond to the parameters of the function.

Example:

```
#include <stdio.h>
    // Function declaration
        int add(int num1, int num2);
            int main() {
                int result;
            int a = 5;
            int b = 3;

    // Calling the add function with arguments
        result = add(a, b);
            printf("The sum of %d and %d is %d\n", a, b, result);
     return 0;
    }

    // Function definition
     int add(int num1, int num2) {
                return num1 + num2;
            }
```

Explanation about Example:

- We have a function called add that takes two integer parameters, “num1 and num2”.
- Inside the main function, we declare two integer variables, ‘a’ and ‘b’, and initialize them with values 5 and 3, respectively.
- To calculate the sum of ‘a’ and ‘b’, we call the add function with these variables as arguments:

  result = add(a, b);

  Here, ‘a’ and ‘b’ are the arguments passed to the add function.
- The “add” function receives these arguments, performs the addition operation, and returns the result.
- Finally, in the “printf” statement, we use the values of a, b, and result to display the sum in the output.

3.4 Return Value

A "return value" is a piece of data that a function can deliver to the caller once it has finished its job. It enables functions to return information or outcomes to the caller code. The function's definition and implementation both include a "return" statement that defines the return value.

Syntax for Returning a Value:

```
return_type function_name(parameters) {
        // Function body
   return expression; // Return a value of the specified data type
}
```

- **return_type:** Indicates the data type of the value that the function will return, such as “int”, float, or any other valid C data type. If the function doesn't return a value, you use void as the return type.
- **function_name:** The name of the function.
- **parameters:** Any input values or parameters that the function might require.
- **expression:** The value that the function sends back as its result. This expression should match the specified return type.

Example:

```
#include <stdio.h>

    // Function declaration

        int add(int num1, int num2);

            int main() {

                    int result;

                int a = 5;

                int b = 3;


     // Calling the add function and storing the result

                result = add(a, b);


     // Display the result

            printf("The sum of %d and %d is %d\n", a, b, result);
```

```
    return 0;
}

// Function definition
    int add(int num1, int num2) {
            int sum = num1 + num2;

// Returning the sum as the result
    return sum;
}
```

Explanation about Example:

- We have a function called add that takes two integer parameters, “num1 and num2”.
- Inside the main function, we call the add function with arguments ‘a and b’, which are ‘5’ and ‘3’, respectively.
- The add function calculates the sum of “num1” and “num2” and stores it in the sum variable.
- The return sum; statement sends the value of sum back to the caller (the main function in this case)
- In the main function, we store the returned value in the result variable and then display it using “printf”. The output will show "The sum of 5 and 3 is 8."
-

3.5 Function Call

The process of calling or running a function is known as a function call. When you call a function, you're telling the computer to run the code contained inside while optionally handing it some input (arguments) and getting a return value. Here is a code sample to help you understand function calls quickly:

Syntax for Function Call:

```
return_type result_variable = function_name(arguments);
```

- **return_type:** This is the data type of the value that the function is expected to return. Use void if the function doesn't return a value.
- **result_variable:** An optional variable to store the value returned by the function.
- **function_name:** The name of the function you want to call.
- **arguments:** Any values or expressions enclosed in parentheses () that you want to pass as input to the function. If the function doesn't require any input, you can leave the parentheses empty.

Example:

```
#include <stdio.h>

// Function prototype
int add(int num1, int num2);
```

```
int main() {
    int result;
    int a = 5;
    int b = 3;

// Calling the add function and storing the result in 'result' variable
    result = add(a, b);

  // Display the result
    printf("The sum of %d and %d is %d\n", a, b, result);
  return 0;
}

  // Function definition
    int add(int num1, int num2) {
        int sum = num1 + num2;

     // Returning the sum as the result
    return sum;
}
```

Explanation about Example:

- We have a function prototype for “add” at the top, which informs the compiler about the function's signature.
- Inside the “main” function, we declare two integer variables, a and b, and initialize them with values 5 and 3, respectively.
- We call the “add” function with the arguments a and b and store the result in the “result” variable:

```
result = add(a, b);
```

- The add function performs the addition operation with the provided arguments and returns the result.
- Finally, in the “printf” statement, we use the values of a, b, and “result” to display the sum in the output.

3.6 Scope and Lifetime

Variable accessibility and durability are discussed by the terms scope and lifetime, respectively. understanding variable behavior and memory management depends on learning these ideas.

3.6.1 Scope

Scope defines the region or portion of a program where a variable can be accessed or used. In C, there are three primary types of scope:

- **Block Scope:** Variables declared within a block of code, such as within a function, are said to have block scope. They are only accessible within that block and any nested blocks.

- **Function Scope:** Variables declared as function parameters or at the top of a function have function scope. They are accessible throughout the function but not outside of it.
- **File Scope (Global Scope):** Variables declared outside of any function, typically at the top of a source file, have file scope. They are accessible from anywhere within the source file where they are declared.

3.6.2 Lifetime

Lifetime refers to the duration during which a variable exists and retains its value. In C, there are two primary types of variable lifetime:

- **Automatic Lifetime:** Variables with automatic storage duration have a lifetime limited to the block in which they are declared. They are created when execution enters the block and destroyed when execution exits the block.
- **Static Lifetime:** Variables with static storage duration have a lifetime that extends throughout the entire program's execution. They are initialized only once, and their values persist across function calls.

Example:

```
#include <stdio.h>


int globalVariable = 10; // Global variable with file scope
```

```
void functionExample() {
    int localVariable = 20; // Local variable with function scope
    printf("Inside function: globalVariable = %d, localVariable = %d\n", globalVariable, localVariable);
}

int main() {
    int blockVariable = 30; // Local variable with block scope
    printf("Inside main: globalVariable = %d, blockVariable = %d\n", globalVariable, blockVariable);

    functionExample(); // Call the function

// Attempting to access "localVariable" here would result in a compilation error as it's out of scope
    return 0;
}
```

Explanation about Example:

- "globalVariable" has file scope and static lifetime, so it can be accessed from both main and "functionExample'.
- "blockVariable" has block scope and automatic lifetime, so it can only be accessed within the main "function".
- "localVariable" has function scope and automatic lifetime, so it's only accessible within the "functionExample" function.

4

Arrays

4.1 Introduction

An array is a group of identically typed items that are kept in close order to one another in memory. Access to these items is made possible via an array's index or position. When you declare an array in C, you must specify how many items it will include; this size cannot be modified at a later time. For storing and managing collections of data, such as numbers, characters, or other forms of values, arrays are frequently employed. A C array's first member normally has an index of 0, and its last element typically has an index that is one lower than the array's size.

4.2 Array Declaration – Initialization

Arrays are essential data structures used to store collections of elements of the same data type. The declaration and initialization of arrays are fundamental concepts in C, and they play a crucial role in how you work with data efficiently.

4.2.1 Array Declaration

To declare an array in C, you need to specify its data type and provide a name for the array, followed by square brackets containing

the array's size. For example, to declare an integer array that can store five elements, you would write:

```
int myArray[5];
```

This line of code declares an integer array named “myArray” with a size of five. The size indicates how many elements the array can hold, and it must be a positive integer.

4.2.2 Array Initialization

Initialization involves assigning initial values to the array elements. You can initialize an array at the time of declaration in several ways:

Explicit Initialization: You can explicitly specify the initial values for each element in the array using curly braces “{}” and providing the values separated by commas. For example:

```
int myArray[5] = {10, 20, 30, 40, 50};
```

This initializes “myArray” with the values 10, 20, 30, 40, and 50, respectively.

Partial Initialization: If you don't provide enough values to initialize all elements, the remaining elements will be automatically initialized to zero for numeric types. For example:

```
int myArray[5] = {10, 20};
```

In this case, myArray will be initialized as {10, 20, 0, 0, 0}.

Omitting Size: You can omit the array size during initialization, and the compiler will automatically determine the size based on the number of elements you provide:

```
int myArray[] = {1, 2, 3, 4, 5}; // Size is automatically set to 5
```

Example:

```
#include <stdio.h>

int main() {
    // Declare and initialize an integer array with explicit values
    int myArray1[5] = {10, 20, 30, 40, 50};

    // Declare and initialize a character array with partial values
    char myArray2[5] = {'A', 'B'};

    // Declare and initialize a float array without specifying size
    float myArray3[] = {1.1, 2.2, 3.3, 4.4, 5.5};

    // Output the elements of the arrays
    printf("myArray1: ");
    for (int i = 0; i < 5; i++) {
        printf("%d ", myArray1[i]);
```

```
    }
    printf("\n");

    printf("myArray2: ");
    for (int i = 0; i < 5; i++) {
        printf("%c ", myArray2[i]);
    }
    printf("\n");

    printf("myArray3: ");
    for (int i = 0; i < sizeof(myArray3) / sizeof(myArray3[0]); i++) {
        printf("%.1f ", myArray3[i]);
    }
    printf("\n");

    return 0;
}
```

- “myArray1” is an integer array explicitly initialized with values 10, 20, 30, 40, and 50.
- “myArray2” is a character array with partial initialization. It has values 'A' and 'B', and the remaining elements are automatically initialized to null characters ('\0').

- “myArray3” is a float array without a specified size. The compiler automatically determines the size based on the number of elements provided.

Important (Arrays):

Array indices in C start from 0, so the first element is accessed using “myArray[0]”, the second with “myArray[1]”, and so on. Arrays are contiguous blocks of memory, meaning elements are stored one after another in memory, allowing for efficient access and manipulation. Arrays have a fixed size once declared, and this size cannot be changed during runtime. You can use loops and other programming constructs to work with array elements, making it a versatile tool for handling data in C.

4.3 Array Indexing and Bounds

Accessing elements in C arrays is a fundamental operation, and it involves using the array's index or position to retrieve or modify specific elements within the array. Here's a detailed explanation of how to access elements in C arrays:

Array Index

The first element of an array is at index 0, the second member is at index 1, and so on since array indexing starts at 0. You must first supply the array name, followed by square brackets holding the element's index, in order to access an element in an array. For example, if you have the integer array “myArray”:

```
int myArray[5] = {10, 20, 30, 40, 50};
```

You can access its elements as follows:

myArray[0] retrieves the first element, which is 10.

myArray[1] retrieves the second element, which is 20.

myArray[2] retrieves the third element, which is 30.

myArray[3] retrieves the fourth element, which is 40.

myArray[4] retrieves the fifth element, which is 50.

Array Bounds

It's important to remember that accessing items outside of an array's boundaries (with an index larger than or equal to the array size or less than 0) may result in unpredictable behavior and memory corruption. You are responsible for making sure that the index you use is inside the array's valid range.

Example: Here's an example program that demonstrates how to access elements in an array:

```
#include <stdio.h>


int main() {
   int myArray[5] = {10, 20, 30, 40, 50};

   // Access and print individual elements
   printf("Element at index 0: %d\n", myArray[0]);
```

```
    printf("Element at index 2: %d\n", myArray[2]);

    // Modify an element
    myArray[1] = 99;
    printf("Modified element at index 1: %d\n", myArray[1]);

    return 0;
}
```

In this example, we access elements using their respective indices and print their values. We also demonstrate how to modify an element by assigning a new value to it.

4.4 Iterating Through Arrays

Iterating through arrays in C is a fundamental task, and it involves systematically visiting each element of an array for various operations, such as printing, summing, searching, or modifying elements. Here are the steps to iterate through arrays in C:

Declare and Initialize the Array

Begin by declaring an array of a specific data type and, optionally, initializing it with values. For example:

```
int myArray[5] = {10, 20, 30, 40, 50};
```

It's essential to know the size of the array, either by specifying it explicitly or using the “sizeof” operator. The size is crucial for setting the loop termination condition and preventing access outside the bounds of the array.

```
int arraySize = sizeof(myArray) / sizeof(myArray[0]);
```

Here, “arraySize” is set to 5, which is the number of elements in “myArray”.

Use a Loop

Typically, a “for” or “while” loop is used to iterate through the array. The loop variable represents the current index or position within the array. For example, to print all elements in `myArray`, you can use a `for` loop:

```
for (int i = 0; i < arraySize; i++) {
    printf("Element at index %d: %d\n", i, myArray[i]);
}
```

In this loop, “I” starts at 0 and iterates up to “arraySize – 1”, accessing each element in the array.

Perform Desired Operation

Inside the loop, perform the desired operation on each element. In the example above, we're printing the elements. You can replace the “printf” statement with any operation you want to perform on the elements.

Terminate the Loop

Ensure that the loop terminates when the index reaches the last valid position in the array. In this case, the condition “i < arraySize” ensures the loop stops after processing all elements.

Example:

Here's a complete example that demonstrates iterating through an array and finding the sum of its elements:

```
#include <stdio.h>


int main() {
    int myArray[5] = {10, 20, 30, 40, 50};
    int arraySize = sizeof(myArray) / sizeof(myArray[0]);
    int sum = 0;

    for (int i = 0; i < arraySize; i++) {
        sum += myArray[i];
    }

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

    return 0;
```

```
}
```

This program iterates through “myArray”, adds up its elements, and prints the sum.

By following these steps, you can efficiently iterate through arrays in C and perform various operations on their elements. The loop provides a structured way to access and manipulate array data.

4.5 Multi-Dimensional Array

An array of arrays represents what a multi-dimensional array is. It is helpful for jobs involving tables, grids, or matrices since it enables you to arrange data in a matrix- or grid-like form. Two-dimensional arrays are the most common type of multi-dimensional array, however, there are other types that have additional dimensions.

Declaring a Multi-Dimensional Array

To declare a multi-dimensional array in C, you specify the data type, the array name, and the dimensions within square brackets. For example, a two-dimensional integer array can be declared as:

```
int matrix[3][4];
```

This declares a 3x4 matrix, meaning it has 3 rows and 4 columns.

Initializing a Multi-Dimensional Array

You can initialize multi-dimensional arrays at the time of declaration by providing values within nested curly braces. For instance:

```
int matrix[3][4] = {
                    {1, 2, 3, 4},
                    {5, 6, 7, 8},
                    {9, 10, 11, 12}
                   };
```

This initializes a 3x4 matrix with specific values.

Accessing Elements in a Multi-Dimensional Array

To access elements in a multi-dimensional array, you use multiple indices. For a two-dimensional array, you use two indices—one for the row and another for the column. For example:

```
int element = matrix[1][2];
```

// Accesses the element in the second row and third column (value: 7)

Iterating Through a Multi-Dimensional Array

When iterating through a multi-dimensional array, you typically use nested loops. For example, to print all elements in a 3x4 matrix:

```
for (int row = 0; row < 3; row++) {
        for (int col = 0; col < 4; col++) {
```

```
        printf("%d ", matrix[row][col]);
    }
    printf("\n");
}
```

This nested `for` loop first iterates through each row and then within each row, it iterates through the columns, printing each element.

Multi-Dimensional Arrays with More Dimensions

You can have multi-dimensional arrays with more than two dimensions. For instance, a three-dimensional array could be used to represent a cube of data:

```
int cube[2][3][4]; // 2x3x4 three-dimensional array
```

Example:

```
#include <stdio.h>
    int main() {
        // Declare and initialize a 2D integer array (3x3 matrix)
        int matrix[3][3] = {
                {1, 2, 3},
                {4, 5, 6},
                {7, 8, 9}
            };
```

```
    // Access and print elements in the 2D array
    printf("Matrix:\n");
    for (int row = 0; row < 3; row++) {
        for (int col = 0; col < 3; col++) {
            printf("%d ", matrix[row][col]);
        }
        printf("\n"); // Move to the next row
    }

    return 0;
}
```

Explanation about Codes:

- We declare a 2D integer array named “matrix” with dimensions 3x3 (3 rows and 3 columns).
- We initialize the “matrix” with values. This creates a simple 3x3 matrix with numbers from 1 to 9.
- We use nested for loops to iterate through the rows and columns of the matrix. The outer loop “row” iterates through the rows, and the inner loop “col” iterates through the columns.
- Inside the nested loops, we use “matrix[row][col]” to access and print each element of the matrix.

- When you run this program, it will produce the following output:

Matrix:

1 2 3

4 5 6

7 8 9

4.6 Passing Array to Function

You can apply arrays in C in a modular and reusable fashion by passing an array to a function. By passing a pointer to the array, which enables the function to access and alter the original array, you may send arrays to functions by reference. Here are the steps and an example for passing an array to a function in C:

Declare the Function Signature

Declare the function that will receive the array as an argument. The function signature should include the array's data type and a pointer to the array. You can also specify the array size or omit it.

```
void myFunction(int arr[], int size);
```

Define the Function

Define the function as you would any other function. The parameter “arr[“] represents a pointer to the array, and size represents the

number of elements in the array. You can now work with the array inside the function.

```
void myFunction(int arr[], int size) {
        // Access and manipulate elements of 'arr' here
}
```

Call the Function

In your main program, call the function and pass the array as an argument. Make sure to provide the correct array and size. Any changes made to the array inside the function will affect the original array because you are working with the same memory location.

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

 myFunction(myArray, size);

// 'myArray' now contains any modifications made in 'myFunction'

 return 0;
}
```

Example:

```
#include <stdio.h>


// Function to double each element in the array
    void doubleArray(int arr[], int size) {
                    for (int i = 0; i < size; i++) {
                            arr[i] *= 2;
                        }
                }

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

        printf("Original Array: ");
              for (int i = 0; i < size; i++) {
                      printf("%d ", myArray[i]);
                     }
   doubleArray(myArray, size);
            printf("\nModified Array: ");
                for (int i = 0; i < size; i++) {
```

```
            printf("%d ", myArray[i]);
        }
    return 0;
}
```

In this example, we pass the “myArray” to the “doubleArray” function, which doubles each element. After calling the function, the modified array is printed, demonstrating that changes made within the function are reflected in the original array.

4.7 String Array

In its basic form, a string is a character array that is ended by the null character (\0). By creating an array of characters that may carry several strings and where each member is a separate string, you can build a string array. Here is an example and explanation for it:

Declare the String Array

To declare a string array, you define an array of character arrays. Each element of the outer array is a character array that can hold a string. The size of the character arrays should be sufficient to store the longest string you expect to store.

```
char strArray[3][20];
```

This declares a string array with 3 elements, each capable of holding a string up to 19 characters long.

Assign Values to the String Array

You can assign values to the elements of the string array either during declaration or later in your program. Here's an example of assigning values during declaration:

```
char strArray[3][20] = {
                        "Hello,",
                        "World!",
                        "C Programming"
                       };
```

Access and Manipulate Strings

You can access and manipulate individual strings within the string array using array indices and string manipulation functions from the “<string.h>” library. Here's an example of accessing and printing the strings:

```
#include <stdio.h>
#include <string.h>
  int main() {
        char strArray[3][20] = {
                                "Hello,",
                                "World!",
                                "C Programming"
                               };
```

```
    for (int i = 0; i < 3; i++) {
        printf("String %d: %s\n", i + 1, strArray[i]);
    }

    // Modify a string
    strcpy(strArray[1], "Universe!");

    printf("Modified String 2: %s\n", strArray[1]);

    return 0;
}
```

In this example, we print the strings stored in the strArray, then use strcpy to modify the second string.

Output:

```
String 1: Hello,
String 2: World!
String 3: C Programming
Modified String 2: Universe!
```

4.8 Array of Pointer

An array of pointers is an array whose elements each point to a different variable or data structure that represents a value. The ability to generate arrays of various data kinds and dynamically allocate RAM for each member makes this a potent feature. I'll give an illustration of how to utilize an array of pointers below, along with some explanation:

Declare the Array of Pointers

Declare an array of pointers by specifying the data type followed by an asterisk (*) to indicate that it's a pointer. For example, to create an array of integer pointers:

```
int* intArray[5]; // Declare an array of 5 integer pointers
```

Allocate Memory of each Element

For each element in the array of pointers, allocate memory dynamically using functions like “malloc()” or “calloc()”. Here's an example of allocating memory for each element in a loop:

```
#include <stdio.h>
#include <stdlib.h>
    int main() {
        int* intArray[5];
            for (int i = 0; i < 5; i++) {
// Allocate memory for an integer
                intArray[i] = (int*)malloc(sizeof(int));
```

```
        *intArray[i] = i * 10; // Store a value in the allocated memory
    }

    // Access and print the values stored in the array
        for (int i = 0; i < 5; i++) {
                printf("intArray[%d] = %d\n", i, *intArray[i]);
        }
    return 0;
}
```

In this example, we allocate memory for each element in the "intArray" and store different integer values.

Deallocate Memory

Don't forget to free the allocated memory using "free()" to prevent memory leaks:

```
    for (int i = 0; i < 5; i++) {
            free(intArray[i]); // Free the allocated memory
    }
```

- An array of pointers allows you to have flexibility in managing different data types and dynamically allocate memory as needed.

- **In the example,** we allocated memory for integers, but you can use the same approach for other data types like double, char, or even user-defined structures.
- Remember to free the allocated memory to avoid memory leaks when you're done using it.

5

Pointers

5.1 Introduction

A pointer is a variable that stores the memory address of another variable in the C programming language. By referring to the region in memory where the real data is kept, pointers are used to alter and access data indirectly. They are a core component of C and are frequently used for operations like dynamic memory allocation, the efficient manipulation of data in memory, and the creation of data structures like arrays and linked lists. Pointers may be used to execute operations like dereferencing (accessing the value at the pointed memory location) and pointer arithmetic (manipulating the memory address). Pointers are defined using an asterisk (*). For effective low-level memory management and C programming, an understanding of pointers is essential.

5.2 Declaration – Initialization Pointers

A very important concept in the C programming language that enables programmers to effectively interact with memory locations is declaring and initializing pointers. In order to provide indirect access to the data, pointers are variables that contain the memory

address of other variables or data structures. Here is an explanation and an example:

5.2.1 Declaration of Pointers

To declare a pointer, you need to specify the data type it points to, followed by an asterisk (*), and then the pointer variable name. For example, if you want to declare a pointer to an integer, you would do it like this:

```
int *ptr; // Declaring a pointer to an integer
```

This declares a pointer variable “ptr” that can hold the memory address of an integer.

5.2.2 Initialization of Pointers

Pointers should be initialized before they are used. Initialization assigns the memory address of an existing variable to the pointer variable. You can initialize a pointer in several ways:

Direct Initialization

You can directly assign the address of a variable to a pointer. For example:

```
int x = 42;

int *ptr = &x; // Initializing ptr with the address of x
```

Here, “&x” retrieves the address of the integer variable x, and that address is stored in the pointer “ptr”.

Dynamic Memory Allocation

Pointers are often used to allocate memory dynamically using functions like “malloc”, “calloc”, or “realloc”. For example:

```
int *arr = (int *)malloc(5 * sizeof(int));
```

This allocates memory for an array of 5 integers and initializes the “arr” pointer to point to the first element of this allocated memory.

Null Initialization

You can initialize a pointer to “NULL” if it doesn't currently point to any valid memory address. This is a good practice to avoid using uninitialized pointers:

```
int *ptr = NULL;
```

Later, you can assign it a valid memory address as needed.

Initialization with an Existing Pointer

You can initialize one pointer with the value of another pointer of the same type:

```
int *ptr1;

int *ptr2 = ptr1; // Initialize ptr2 with the value of ptr1
```

Both “ptr1” and “ptr2” will now point to the same memory address.

Example:

```
#include <stdio.h>

   int main() {

         int number = 42;   // Declare an integer variable 'number' and   initialize it with 42.

        int  *pointer;        // Declare an integer pointer variable 'pointer'.

   pointer = &number;   // Assign the address of 'number' to the pointer.


   printf("Value of 'number': %d\n", number);        // Output: Value of 'number': 42

   printf("Address of 'number': %p\n", &number);        // Output: Address of 'number': (some hexadecimal address)

   printf("Value of 'pointer': %p\n", pointer);      // Output: Value of 'pointer': (same hexadecimal address as &number)

   printf("Value at the address pointed by 'pointer': %d\n", *pointer);
// Output: Value at the address pointed by 'pointer': 42


   // Let's change the value through the pointer

                  *pointer = 99;

   printf("Value of 'number' after modification: %d\n", number);

                // Output: Value of 'number' after modification: 99
```

```
    return 0;
}
```

Explanation about Example:

- We declare an integer variable called number and initialize it with the value 42.
- We declare an integer pointer variable called pointer. Pointers are declared using the * symbol.
- We assign the address of the number variable to the pointer using the “&” operator. This means pointer now "points" to the memory location where number is stored.
- We use “printf” to print the value of number, the address of number, the value of pointer (which will be the address of number), and the value pointed to by pointer (which should be the same as the value of number).
- We change the value of number indirectly through the pointer by using the * operator to access the value at the memory address it points to.
- Finally, we print the modified value of number to confirm that it was indeed changed through the pointer.
- This example demonstrates the basic concepts of declaring, initializing, and using pointers to access and modify data indirectly in C.

This example demonstrates the basic concepts of declaring, initializing, and using pointers to access and modify data indirectly in C.

5.3 Pointer Arithmetic

In pointer arithmetic, a pointer is moved to a different address in memory by adding or removing an integer value from/to it. The amount of bytes the pointer moves forward or backward depends on the size of the data type it is pointing to. When iterating across arrays or dynamically allocated memory, this is especially helpful.

Example:

Suppose you have an integer array and a pointer to its first element:

```
int arr[] = {10, 20, 30, 40, 50};
int *ptr = &arr[0];
```

You can perform pointer arithmetic like this

```
ptr++; // Move the pointer to the next integer element
printf("%d\n", *ptr); // This will print 20

ptr += 2; // Move the pointer two integers ahead
printf("%d\n", *ptr); // This will print 40

ptr--; // Move the pointer back by one integer
printf("%d\n", *ptr); // This will print 30
```

In this example, “ptr++” advances the pointer to the next integer (4 bytes ahead), and “ptr += 2” moves it two integers (8 bytes) ahead. Conversely, “ptr--“ moves the pointer back by one integer. Pointer

arithmetic simplifies array traversal, indexing, and data manipulation in C. However, it's essential to be cautious to prevent accessing memory locations outside the allocated space, which can lead to undefined behavior.

5.4 Accessing Values via Pointers

The dereference operator (*) in C is used to access a value via a pointer. It enables you to access the value kept at the memory address that the pointer is pointing to. This is especially helpful if you wish to change or operate indirectly with data.

Example:

Suppose you have an integer variable and a pointer to it:

```
int num = 42;

int *ptr = &num;
```

You can access the value of “num” via the pointer “ptr” using the dereference operator:

```
int value = *ptr; // Dereference the pointer to retrieve the value
```

Now, the variable value holds the value 42, which is the same as the value of num. You can also modify the value through the pointer:

```
*ptr = 10; // Modify the value of 'num’ indirectly through 'ptr'
```

After this operation, both “*ptr” and “num” will have a value of 10.

Accessing values via pointers is crucial when working with data structures like arrays, linked lists, or when dynamically allocating memory. It provides a way to interact with data indirectly, making it easier to manipulate and manage memory efficiently in C programs.

5.5 Passing Pointers to Function

In C, you effectively grant a function access to the memory region where the data is stored when you send it a pointer. This means that any modifications made to the data inside the function will also impact the data outside the function. It's an effective technique for changing data in a function without having to return values.

Example:

Suppose you have a function that increments an integer value through a pointer:

```
#include <stdio.h>

        void incrementByOne(int *numPtr) {
                (*numPtr)++;
            }

int main() {
        int num = 5;
            printf("Before: %d\n", num);
```

```
    incrementByOne(&num); // Pass the address of 'num'

    printf("After: %d\n", num); // 'num' is now 6

    return 0;
}
```

In this example, the “incrementByOne” function takes a pointer to an integer (int *numPtr) as an argument. When you call the function with “incrementByOne(&num)”, you pass the address of the “num” variable. Inside the function, “(*numPtr)++” increments the value at that memory location, effectively changing the value of “num” outside the function.

5.6 Memory Management

Programming in C requires memory management since it involves dynamically allocating and releasing memory for data. For managing memory, C offers a number of methods and procedures, principally utilizing the "malloc", "calloc", "realloc", and "free" functions. Here is an explanation and an example:

5.6.1 Memory Allocation

malloc: Allocates a specified amount of memory and returns a pointer to the first byte. For example:

```
int *arr = (int *)malloc(5 * sizeof(int));
```

calloc: Allocates memory for an array and initializes it to zero. For example:

```
int *arr = (int *)calloc(5, sizeof(int));
```

5.6.2 Memory Reallocation

realloc: Changes the size of a previously allocated memory block. It may move the block to a new location. For example:

```
arr = (int *)realloc(arr, 10 * sizeof(int));
```

5.6.3 Memory Deallocation

free: Releases memory allocated with malloc, calloc, or realloc. For example:

```
free(arr);
```

Example:

```
#include <stdio.h>

#include <stdlib.h>


int main() {

   int *arr = (int *)malloc(5 * sizeof(int));


   if (arr == NULL) {
```

```
        printf("Memory allocation failed\n");
        return 1;
    }

    for (int i = 0; i < 5; i++) {
        arr[i] = i * 10;
    }

    for (int i = 0; i < 5; i++) {
        printf("%d ", arr[i]);
    }

    free(arr); // Release allocated memory

    return 0;
}
```

In this example, “malloc” is used to allocate memory for an integer array, and free is used to release that memory when it's no longer needed. Proper memory management is essential to avoid memory leaks and ensure efficient memory utilization in C programs.

5.7 Pointers to Pointers

Pointers to pointers, often known as "double pointers" or "pointer-to-pointer," are an important concept in the C programming language. You can access or indirectly alter pointers by using them to hold the address of another pointer. When passing pointers by reference or modifying them dynamically, this approach is extremely helpful. A further asterisk (*) is added to the pointer declaration to indicate that it is a reference to another pointer. You may operate with pointers indirectly since it simply starts a chain of indirection.

Example:

Consider a scenario where you want to swap two integer values using pointers to pointers:

```
#include <stdio.h>


    void swap(int **ptr1, int **ptr2) {
            int *temp = *ptr1;
                *ptr1 = *ptr2;
                *ptr2 = temp;
            }


      int main() {
          int num1 = 5, num2 = 10;
              int *ptr1 = &num1;
```

```
    int *ptr2 = &num2;

    printf("Before swap: num1 = %d, num2 = %d\n", *ptr1, *ptr2);

    swap(&ptr1, &ptr2);

    printf("After swap: num1 = %d, num2 = %d\n", *ptr1, *ptr2);

    return 0;
}
```

In this example, swap takes two pointers to pointers as arguments. It swaps the values pointed to by these pointers, effectively swapping the values of “num1” and “num2” outside the function.

6

Structures and Unions

6.1 Introduction

The fundamental data types in the C programming language that let you create unique composite data structures are structures and unions. They make it simpler to work with complicated data in a program by combining several variables of different data kinds under a single name. Unions and structures both provide this function, although they both have unique characteristics and applications.

Structures

In C, a user-defined data type called a structure is able to store variables of several data types. Under a single term, it is used to represent a group of connected data objects. The following is the syntax for defining a structure:

```
struct structure_name {
        data_type member1;
        data_type member2;
    // …
};
```

Unions

In C, a user-defined data type called a structure is able to store variables of several data types. Under a single term, it is used to represent a group of connected data objects. The following is the syntax for defining a structure:

```
union union_name {
        data_type member1;
        data_type member2;
    // …
};
```

6.2 Declaring – Defining Structures

For organizing and maintaining complicated data structures, the C programming language requires declaring and creating structures. It is simpler to work with related data in your programs when you can combine variables of various data kinds under a single name thanks to structures. With examples, we'll go over how to declare and define structures in this guide.

Step – 1 Structure Declaration

In order to create a structure in C, you must first define its members and supply the structure tag (name). The structure declaration serves as a guide for implementing that structure type's variables in your program. The syntax for declaring a structure is as follows:

```
struct structure_name {
        data_type member1;
        data_type member2;
    // ...
};
```

- **struct:** This keyword indicates the beginning of the structure declaration.
- **structure_name:** You provide a unique name for your structure. It's customary to use meaningful names that describe the data the structure will hold.
- **data_type member1, data_type member2, etc.:** These are the individual members of the structure. Each member has a name and a data type.

Step – 2 Structure Definitions

A structure can be declared, and then its variables can be defined. The structural variables' initial values are set here, together with the memory that will be allocated for them. A structural variable is defined as follows:

```
struct structure_name variable_name;
```

- **struct structure_name:** This specifies the structure type.
- **variable_name:** You provide a name for the variable of that structure type.

Step – 3 Accessing Structures Members

Once you have defined a structure variable, you can access its individual members using the dot (.) operator. Here's the syntax:

variable_name.member_name

- **variable_name:** This is the name of the structure variable.
- **member_name:** This is the name of the specific member you want to access.

Step – 4 Initializing Structure Variables

You can initialize structure variables at the time of declaration. This sets the initial values for the structure members. Here's how you can do it:

struct structure_name variable_name = {value1, value2, ...};

- **value1, value2, etc.:** These are the values you provide for the structure members in the order they are declared in the structure.

Example: Declaring and Defining a Structures

Let's create a straightforward example to show the declaration and definition of a structure. Let's say we want to use the x and y coordinates to represent a point in a 2D coordinate system. For this, a structure can be used.

Step – 1 Structure Declaration

```
struct Point {
    int x;
    int y;
};
```

Here, we've declared a structure named Point with two integer members, x and y.

Step – 2 Structure Definitions

Now, let's define a variable of the Point structure type:

```
struct Point p1;
```

We've defined a Point variable named p1.

Step – 3 Accessing Structures Members

We can access and modify the members of p1 using the dot operator:

```
p1.x = 5;
p1.y = 10;
```

Now, p1 holds the values x = 5 and y = 10.

Step – 4 Initializing Structure Variable

You can initialize a structure variable at the time of declaration:

```
struct Point p2 = {3, 7};
```

This initializes p2 with x = 3 and y = 7.

Now, you have two Point variables, p1 and p2, representing different points in the 2D coordinate system.

```
printf("Point p1: (%d, %d)\n", p1.x, p1.y); // Output: Point p1: (5, 10)

printf("Point p2: (%d, %d)\n", p2.x, p2.y); // Output: Point p2: (3, 7)
```

In this example, we declared and defined a structure Point to represent 2D coordinates, created structure variables ‘p1’ and ‘p2’, and accessed their members to store and retrieve values.

6.3 Passing Structures to Functions

Structures can be sent by value or by reference to functions. Providing a structure by value involves making a duplicate of the complete thing, whereas providing a structure by reference involves sending a pointer to its position in memory. Each technique has benefits and applications. The processes for sending structures to functions using both ways will be covered in this guide along with examples.

6.2.1 Passing Structure by Value

Making a duplicate of the structure and giving it to the function is how structures are sent by value. Because this approach is simple and logical, it could use up more memory and perform less effectively when dealing with complex structures.

Step – 1 Define the Structure

First, define the structure that you want to pass by value. Let's use a Person structure as an example:

```
struct Person {
        char name[50];
        int age;
      };
```

Step – 2 Declare and Define Function

Declare and define a function that takes the structure by value as an argument. Here's an example function that prints information about a person:

```
void printPerson(struct Person p) {
        printf("Name: %s\n", p.name);
        printf("Age: %d\n", p.age);
}
```

Step – 3 Call the Function

Now, you can create a Person variable and call the “printPerson” function to print its details:

```
int main() {
        struct Person person1;
        strcpy(person1.name, "Pawan");
            person1.age = 30;

    printPerson(person1);

    return 0;
}
```

In this example, we declare and define a Person structure, create a Person variable “person1”, and pass it by value to the “printPerson” function. The function prints the details of “person1”.

6.2.2 Passing Structure by Reference

When sending a structure by reference, the function receives a pointer to the structure's memory location. This technique uses less memory and enables the function to change the original structure.

Step – 1 Define the Structure

Define the structure, just as we did in the previous example:

```
struct Person {
        char name[50];
            int age;
        };
```

Step – 2 Declare and Define Function

Declare and define a function that takes a pointer to the structure as an argument. Here's an example function that increments a person's age:

```
void incrementAge(struct Person *p) {
                    (p->age)++;
                }
```

Step – 3 Call the Function

Create a Person variable, call the “incrementAge” function, and then print the modified age:

```
int main() {
        struct Person person1;
          strcpy(person1.name, "Bob");
```

```
    person1.age = 25;
    incrementAge(&person1);
    printf("New Age: %d\n", person1.age);

    return 0;
}
```

In this example, we declare and define a Person structure, create a Person variable “person1”, pass it by reference (using &person1) to the “incrementAge” function, and then print the modified age.

6.4 Nested Structures

By adding one or more structures as members within another structure in C, you can create complicated data structures. This makes it possible for you to organize and arrange data that is hierarchical or connected. An overview of working with nested structures is provided below, along with an example:

Step – 1 Define the Inner Structures

First, define the individual structures that you want to nest within the outer structure. These inner structures can have their own members and data types.

```
struct Date {
        int day;
        int month;
        int year;
       };

struct Student {
        char name[50];
         int roll_number;
         struct Date birth_date;
       };
```

In this example, we've defined two structures: Date to represent a date and Student to represent a student with their name, roll number, and birth date.

Step – 2 Define the Outer Structure

Next, define the outer structure that will contain instances of the inner structures. Declare the inner structures as members of the outer structure.

```
struct University {
        char name[100];
        int established_year;
```

```
    struct Student students[100];
};
```

Here, we've defined a University structure with name, established year, and an array of Student structures as its members.

Step – 3 Create and Use Nested Structures

Now, you can create instances of the outer structure and populate them with data, including instances of the inner structures.

```
int main() {
    struct University myUniversity;
    strcpy(myUniversity.name, "Example University");
myUniversity.established_year = 1990;

// Populating student data
strcpy(myUniversity.students[0].name, "Pawan");
myUniversity.students[0].roll_number = 101;
myUniversity.students[0].birth_date.day = 15;
myUniversity.students[0].birth_date.month = 5;
myUniversity.students[0].birth_date.year = 2000;

// Accessing nested structure members
printf("University Name: %s\n", myUniversity.name);
```

```
    printf("Established Year: %d\n", myUniversity.established_year);
    printf("Student Name: %s\n", myUniversity.students[0].name);
 printf("Roll Number: %d\n", myUniversity.students[0].roll_number);
printf("Birth Date: %d/%d/%d\n",
myUniversity.students[0].birth_date.day,
myUniversity.students[0].birth_date.month,
myUniversity.students[0].birth_date.year);

    return 0;
}
```

In this example, we create an instance of the University structure and populate its members, including the students array, which contains instances of the Student structure. We access and print nested structure members using the dot operator.

6.5 Accessing Union Members

Just as with structures, you can access union members by using the dot (.) operator. Unions and structures differ fundamentally in that only one member of a union may be accessed at a time since all members share the same memory location. Here is an instruction on how to contact union members, along with an example:

Step – 1 Define the Union

First, define the union that contains the members you want to access. The union declaration specifies the data types of its members.

```
union MyUnion {
    int integer_member;
    double double_member;
    char string_member[20];
};
```

Here, we've defined a union named “MyUnion” with three members: an integer, a double, and a character array.

Step – 2 Create a Union Variable

Next, create a variable of the union type and assign values to one of its members. Only one member should be accessed and modified at a time, as they all share the same memory space.

```
union MyUnion myVar;
myVar.integer_member = 42;
```

In this example, we've created a union variable myVar and assigned a value of 42 to its “integer_member”.

Step – 3 Access Union Members

To access a specific union member, use the dot operator followed by the member's name.

```
printf("Integer Member: %d\n", myVar.integer_member);
```

Here, we access and print the value of the “integer_member” of the “myVar” union variable.

Step – 4 Avoid Conflicting Access

Remember that you should only access one member of the union at a time. Accessing a different member will read the same memory location, potentially leading to incorrect results. For example, accessing “double_member” or “string_member” after assigning a value to “integer_member” may yield unpredictable results.

```
myVar.double_member = 3.14159;

printf("Double Member: %lf\n", myVar.double_member);
```

In this case, we access the double_member after assigning a value to integer_member.

Example:

```
#include <stdio.h>

union MyUnion {

int integer_member;

double double_member;
```

```
        char string_member[20];
    };

int main() {
    union MyUnion myVar;
        myVar.integer_member = 42;

        printf("Integer Member: %d\n", myVar.integer_member);

    myVar.double_member = 3.14159;
        printf("Double Member: %lf\n", myVar.double_member);

    strcpy(myVar.string_member, "Hello, Union!");
        printf("String Member: %s\n", myVar.string_member);

    return 0;
}
```

In this example, we construct a union variable called “myVar”, give distinct members of its values, and then access each member individually. When working with unions, use caution to make sure you have access to the right member according to how it was allotted because improper access may result in unexpected behavior or mistakes.

6.6 “typedef” for Structures and Unions

Structures and unions can all have new names or aliases created for them using the “typedef” keyword. You may improve code readability and simplify variable declarations with it. Typedef makes it simpler to build complicated data types and declare variables of those kinds when combined with structures and unions. Here is a guide with examples of how to use “typedef” with structures and unions:

Step – 1 Define the Structure or Union

First, define the structure or union as you normally would. Let's create a simple structure and a union as examples:

```
// Define a structure
    struct Student {
        char name[50];
        int roll_number;
    };

// Define a union
    union Color {
        char rgb[3];
        int hex;
    };
```

In this example, we have a structure Student with two members (name and roll_number) and a union Color with two members (rgb and hex).

Step – 2 Use "typedef" to Create an Alias

Next, use "typedef" to create an alias for the structure or union. You provide the new name you want to use for the data type, followed by the keyword "typedef" and the original data type.

```
// Create an alias for the structure
        typedef struct Student Student;

// Create an alias for the union
        typedef union Color Color;
```

Now, you can use Student and Color as shorthand names for the "struct" Student and union Color data types, respectively.

Step – 3 Declare Variables Using the Alias

You can now declare variables using the alias instead of the full data type name, making your code more concise and readable.

```
int main() {
  // Declare a variable of the structure using the alias
      Student student1;
        strcpy(student1.name, "Alice");
```

```
    student1.roll_number = 101;


  // Declare a variable of the union using the alias
        Color color1;
              color1.hex = 0xFF0000;
    return 0;
}
```

In this example, we declare a Student variable as student1 and a Color variable as color1, both using the aliases created with "typedef".

Example: Combining "typedef" and Structures

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


 // Define a structure
          struct Employee {
                char name[50];
                int employee_id;
               double salary;
            };
```

```
// Create aliases using typedef
        typedef struct Employee Employee;


int main() {
   // Declare Employee variables using the aliases
   Employee employee1;
   strcpy(employee1.name, "John Doe");
   employee1.employee_id = 1001;
   employee1.salary = 55000.0;

   // Display employee information
   printf("Employee Name: %s\n", employee1.name);
   printf("Employee ID: %d\n", employee1.employee_id);
   printf("Salary: %.2lf\n", employee1.salary);

   return 0;
}
```

In this example, we define a structure Employee and create an alias Employee using “typedef”. We then declare an Employee variable, employee1, and use it to store and display employee information.

7

File Handling

7.1 Introduction

Working with data in different applications requires the use of handling files, which is a key feature of the C programming language. It gives the program the ability to change file contents as well as read and write data to files. This feature is important for C programmers since it is required for operations like data storage, retrieval, and manipulation.

You must include the standard I/O library, "stdio.h>," which offers functions and data types for interacting with files, in order to conduct file handling operations in C. The FILE structure, which represents a file and holds details like its name, status, and location, is the main data structure used for handling files. The following functions are often used for file operations: "fopen()," "fclose()," "fread()," "fwrite()," "fscanf()," and "fprintf()."

The multiple file modes that C offers, such as read ("r"), write ("w"), append ("a"), binary ("b"), and others, specify how the file is accessed and opened. When interacting with files, error management is essential since operations might fail for a variety of reasons, such as file not found errors or permissions problems. Robust file handling is ensured by checking the return values of file

functions and employing error-handling methods like "perror()" and "feof()".

7.2 Opening and Closing File

Opening and closing files in the C programming language is important for reading from and writing to files.

7.2.1 Opening a File

To open a file, you can use the “fopen()” function. This function takes two arguments: the name of the file you want to open and the mode in which you want to open it (read, write, append, etc.). For example, to open a file named "example.txt" for reading:

```
filePointer = fopen("example.txt", "r");
```

If the file doesn't exist, “fopen()” will return NULL.

7.2.2 Closing a File

Always close the file when you're done using it to free up system resources and ensure data integrity. Use the “fclose()” function:

```
fclose(filePointer);
```

Example:

```
#include <stdio.h>


        int main() {
                    FILE *filePointer;
                   char ch;


   // Open the file for reading
              filePointer = fopen("example.txt", "r");


  // Check if the file was opened successfully
            if (filePointer == NULL) {
                      printf("File could not be opened.\n");
                               return 1; // Exit with an error code
                    }


  // Read and print the contents of the file
              while ((ch = fgetc(filePointer)) != EOF) {
                                                        putchar(ch);
                                                    }
```

```
    // Close the file
        if (fclose(filePointer) == 0) {
                printf("\nFile closed successfully.\n");
            } else {
                printf("\nFile could not be closed.\n");
            return 1; // Exit with an error code
        }
    return 0; // Exit successfully
}
```

In this example, we open "example.txt" for reading, read its contents character by character, and then close the file properly. Always remember to handle potential errors gracefully for robust file handling in C.

7.3 Reading from Files

In the C programming language, you can read from files using the standard input/output functions. To read from a file, follow these steps:

Include Necessary Headers: First, include the necessary headers at the top of your C program to work with files.

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

Declare a File Pointer: Declare a file pointer variable to represent the file you want to read from.

```
FILE *filePointer;
```

Open the File: Use the “fopen()” function to open the file in the desired mode (e.g., "r" for reading). Check if the file opened successfully.

```
filePointer = fopen("example.txt", "r");
    if (filePointer == NULL) {
        printf("Error opening the file.\n");
        return 1;
    }
```

Read Data: Use functions like “fscanf()” or “fgets()” to read data from the file. For example:

```
char buffer[100];
while (fgets(buffer, sizeof(buffer), filePointer) != NULL) {
    printf("%s", buffer); // print or process the data as needed
}
```

Close the File: After you've finished reading from the file, close it using “fclose()”.

```
fclose(filePointer);
```

Example:

```
#include <stdio.h>

int main() {
    FILE *filePointer;
    char buffer[100];

    filePointer = fopen("example.txt", "r");
    if (filePointer == NULL) {
        printf("Error opening the file.\n");
        return 1;
    }

    while (fgets(buffer, sizeof(buffer), filePointer) != NULL) {
        printf("%s", buffer);
    }

    fclose(filePointer);
    return 0;
}
```

This program opens the file "example.txt" in read mode, reads its contents line by line using “fgets()”, and prints each line to the console. Make sure to handle errors and close the file when you're done to ensure proper file handling in your C programs.

7.4 Writing to Files

In the C programming language, you can write data to files using standard input/output functions. Here's a brief guide on how to write to files in C, along with an example:

Include Necessary Headers: Begin by including the necessary header files, typically <stdio.h> for file operations.

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

Declare a File Pointer: Declare a FILE pointer variable to represent the file you want to write to.

```
FILE *filePointer;
```

Open the File: Use the “fopen()” function to open the file in the desired mode, such as "w" for write mode. If the file does not exist, it will be created. Check for successful file opening.

```
filePointer = fopen("example.txt", "w");
                    if (filePointer == NULL) {
printf("Error opening the file.\n");
```

```
    return 1;
}
```

Write Data: Use functions like “fprintf()” or “fputs()” to write data to the file. For example:

```
fprintf(filePointer, "Hello, world!\n");
fputs("This is a line written to the file.\n", filePointer);
```

Close the File: After you've finished writing, close the file using “fclose()”.

```
fclose(filePointer);
```

Example:

```
#include <stdio.h>

int main() {
    FILE *filePointer;

    filePointer = fopen("example.txt", "w");
    if (filePointer == NULL) {
        printf("Error opening the file.\n");
        return 1;
```

```
    }


    fprintf(filePointer, "Hello, world!\n");
    fputs("This is a line written to the file.\n", filePointer);


    fclose(filePointer);
    return 0;
}
```

This program creates (or overwrites if it already exists) a file named "example.txt," writes some text to it using “fprintf()” and “fputs()”, and then closes the file. Make sure to handle errors and close the file properly to ensure successful file writing in C programs.

7.5 Error Handling

Programming in C must include error handling to ensure that your code responds gracefully to unexpected problems and failures. When managing errors in C, it's common practice to use methods that return error codes or establish specialized variables like “errno” to indicate the type of problem. Here is a quick overview of C's error handling along with an example of it:

Error Codes: Most C functions that can encounter errors return an integer error code. A return value of 0 typically indicates success, while non-zero values represent various error conditions.

errno Variable: The “errno” variable stores the error code associated with the most recent error. It's important to check and handle “errno” when a function reports an error.

perror() Function: The “perror()” function is used to print a human-readable error message corresponding to the value of “errno”.

```
if (someFunction() == -1) {
    perror("Error occurred");
}
```

Custom Error Handling: You can implement custom error handling by checking the return values of functions and taking appropriate actions based on the error codes.

```
FILE *filePointer = fopen("example.txt", "r");
if (filePointer == NULL) {
    fprintf(stderr, "Error opening the file: %s\n", strerror(errno));
    return 1;
}
```

Example:

Here's an example that demonstrates error handling when opening a file:

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

int main() {
    FILE *filePointer = fopen("nonexistent_file.txt", "r");
    if (filePointer == NULL) {
        fprintf(stderr, "Error opening the file: %s\n", strerror(errno));
        return 1;
    }

    // Continue with file operations if successfully opened
    fclose(filePointer);
    return 0;
}
```

In this example, if the file "nonexistent_file.txt" doesn't exist or there's another error during file opening, the program will print an error message including a description of the error obtained from “strerror(errno)” and return a non-zero exit code to indicate failure. Proper error handling is essential to make your C programs more robust and user-friendly.

7.6 Text Files VS Binary Files

In C programming, binary files and text files are two common types of files used to store and manipulate data. They have distinct characteristics and use cases:

7.6.1 Text Files

Human-Readable: Text files store data in plain text format, making them human-readable. They typically contain characters, such as letters, digits, and symbols, and are encoded using ASCII or Unicode.

Line-Based: Text files are often organized into lines, separated by newline characters ('\n').

Examples:

Source code files, configuration files, log files, and documents (e.g., .txt, .c, .html, .csv).

```
#include <stdio.h>


int main() {
        FILE *textFile = fopen("example.txt", "w");
    if (textFile == NULL) {
            printf("Error opening the text file.\n");
     return 1;
}
```

```
        fprintf(textFile, "Hello, world!\n");
            fclose(textFile);
    return 0;
}
```

7.6.2 Binary Files

Non-Human-Readable: Binary files store data in a format not directly readable by humans. They can contain any binary data, including text, images, audio, or serialized objects.

Arbitrary Structure: Binary files can have arbitrary structures depending on the data they represent, making them versatile for various data types.

Examples:

Image files (e.g., .jpg, .png), audio files (e.g., .mp3, .wav), and compiled program files (e.g., .exe, .o).

```
#include <stdio.h>


int main() {
        FILE *binaryFile = fopen("example.bin", "wb");
    if (binaryFile == NULL) {
            printf("Error opening the binary file.\n");
```

```
        return 1;
    }

    int data[] = {42, 123, 987};
    fwrite(data, sizeof(int), sizeof(data) / sizeof(int), binaryFile);
    fclose(binaryFile);
    return 0;
}
```

When working with text files in C, you often use functions like “fprintf()” and “fscanf()” for reading and writing. For binary files, you use “fwrite()” and “fread()” to read and write binary data.

Important

The choice between text and binary files depends on the data's nature. Text files are suitable for human-readable, structured data, while binary files are more versatile for arbitrary data types but are not human-readable. Properly choosing between these file types ensure efficient and accurate data storage and retrieval in C programs.

7.7 File Position and Seeking

In C programming, file position and seeking are essential concepts for reading from and writing to files. They allow you to navigate and

manipulate the current position within a file. Here's a brief explanation with an example:

File Position: File position refers to the current location or offset within a file where the next read or write operation will occur. Initially, when you open a file, the file position is set to the beginning (offset 0).

File Seeking: File seeking involves changing the file position to a specific offset within the file. This is done using the “fseek()” function.

Example:

```
#include <stdio.h>


    int main() {
            FILE *filePointer = fopen("example.txt", "r");
        if (filePointer == NULL) {
                printf("Error opening the file.\n");
          return 1;
        }
```

```
    // Move the file position to offset 5 from the beginning
        fseek(filePointer, 5, SEEK_SET);

        char buffer[100];
                if (fgets(buffer, sizeof(buffer), filePointer) != NULL) {
            printf("Data at offset 5: %s", buffer);
         }

    // Move the file position 2 bytes backward from the current position
            fseek(filePointer, -2, SEEK_CUR);

    if (fgets(buffer, sizeof(buffer), filePointer) != NULL) {
        printf("Data after seeking back 2 bytes: %s", buffer);
    }

    fclose(filePointer);
    return 0;
}
```

In this example,

- We open the file "example.txt" in read mode.
- Use “fseek()” to move the file position to an offset of 5 bytes from the beginning of the file.
- Read and print the data at the new position.
- Use “fseek()” again to move the file position 2 bytes backward from the current position.
- Read and print the data after seeking back 2 bytes.

As a result, you can control where you read or write data within a file by manipulating the file position with “fseek()”. The second argument of “fseek()” is the offset, and the third argument specifies the reference point for seeking (“SEEK_SET” for the beginning, “SEEK_CUR” for the current position, and “SEEK_END” for the end of the file). Proper file positioning is crucial for efficient and accurate file I/O operations in C.