C SIMPLIFIED

CONCISE NOTES WITH YOUTUBE INTEGRATION

SATPAL SINGH GILL

BlueRose ONE .com
Stories Matter
NewDelhi • London

BLUEROSE PUBLISHERS
India | U.K.

Copyright © Satpal Singh Gill 2026

All rights reserved by author. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior permission of the author. Although every precaution has been taken to verify the accuracy of the information contained herein, the publisher assumes no responsibility for any errors or omissions. No liability is assumed for damages that may result from the use of information contained within.

BlueRose Publishers takes no responsibility for any damages, losses, or liabilities that may arise from the use or misuse of the information, products, or services provided in this publication.



For permissions requests or inquiries regarding this publication,
please contact:

BLUEROSE PUBLISHERS
www.BlueRoseONE.com
info@bluerosepublishers.com
+91 8882 898 898
+4407342408967

ISBN: 978-93-7310-454-6

Cover Design: Deepak Katheria
Typesetting: Manisha Rawat

First Edition: September 2026

This book is dedicated to my beloved wife,

Mandeep

In the quiet spaces between each line,
your silent strength has shaped what's mine.
In every pause, in every voice,
your faith has guided every choice.

*In every page of **C Simplified**, your presence lives,*
for it is your strength that made this work what it is.

Preface

C programming has been the cornerstone of modern computing and remains one of the most influential languages in the world of programming. Its concepts not only form the foundation for advanced languages, but also nurture logical thinking and problem-solving abilities in learners. As an educator and lifelong learner, I have often observed that students find programming more approachable when complex concepts are explained in a structured, concise, and practice-oriented manner.

This book is designed with that vision. Each chapter presents a short summary of the key concepts, enabling readers to grasp the essentials quickly and effectively. To complement the text, I have integrated QR codes that link directly to my YouTube channel video of that particular topic, where I have explained the same topic through detailed video session. This dual mode of learning - textual and visual - offers students the flexibility to revisit concepts at their own pace and strengthen their understanding.

The book is not only intended for beginners who are taking their first step in C programming, but also for learners who wish to revise the fundamentals in a simplified way. The emphasis is on clarity, practicality, and engagement, so that learning programming becomes a rewarding experience rather than a challenge.

I am grateful to my students, colleagues, and the academic environment at MGM that constantly inspires me to innovate in teaching and learning. It is my hope that this book will serve as a handy guide for students, educators, and self-learners alike.

If this work ignites even a small spark of curiosity and confidence in learners, I will consider my effort truly worthwhile.

Satpal Singh Gill
Assistant Professor,
MGM's College of Computer Science & IT,
Nanded

Contents

Chapter 1: Basics of C 1
Chapter 2: Decision Making 9
Chapter 3: Looping 17
Chapter 4: Arrays 25
Chapter 5: Pointers 28
Chapter 6: Functions 30
Chapter 7: Structure, Union and Enum 37
Chapter 8: Storage Classes 44
Chapter 9: Input-Output in C 48
Chapter 10: Bitwise Operations 53
Chapter 11: Macros, File Inclusion and Conditional Compilation 57
Chapter 12: Miscellaneous 61

Chapter 1
Basics of C

1.1 Variables and Data Types

In C programming, variables are fundamental for storing data. A variable is a data name that may be used to store a data value. Unlike constants that remain unchanged during the execution of a program, a variable may take different values at different times during execution. A variable name can be chosen by the programmer in a meaningful way to reflect the program's function or nature.

Each variable must be declared with a specific data type, such as int for integers, float for floating-point numbers, and char for characters. C language data types categorize the type of values a variable can hold, determining the amount of memory allocated and the operations that can be performed. Following data types are used in general:

int: Stores whole numbers (integers) without decimal points. Can be further modified with short, long, signed, and unsigned to control range and memory usage.

char: Stores single characters, enclosed in single quotes (e.g., 'a', '7', '$'). Occupies one byte of memory and can also represent small integer values.

float: Stores single-precision floating-point numbers (numbers with decimal points).

The data type dictates the size and interpretation of the data stored in the variable. Proper variable declaration and data type selection are essential for writing efficient and accurate C programs.

Watch it on YouTube:

https://youtube.com/shorts/u0VVtMLdjgM



1.2 scanf() function

The scanf function in C is a standard input function used to read formatted input from the standard input stream (stdin), such as the keyboard. It parses the input according to a format string, which specifies the expected data types and formats of the input. scanf then stores the converted values into the variables passed as arguments. Common format specifiers include %d for integers, %f for floating-point numbers, %c for characters and %s for strings.

Example :

scanf("%d",&n);

In the above statement, it reads an integer from the keyboard and stores it in the integer variable n. The %d format specifier indicates that an integer is expected, and &n provides the memory address of n so that scanf can store the input value there.

Watch it on YouTube:

https://youtube.com/shorts/PZnCnSzqrk0



1.3 printf() function

In C language, printf() function is a standard library function, used to print formatted output in many ways to the standard output stdout (which is generally the console screen). It's versatile, allowing you to display various data types like integers, floats, characters, and strings. printf uses format specifiers within a string to control the output's appearance. These specifiers determine how the corresponding arguments are converted and displayed. For example, %d is used for integers, %f for floating-point numbers, %c for characters and %s for strings.

Syntax of printf

printf("formatted string", args...);

formatted_string: It is a string that specifies the data to be printed. It may also contain a format specifier as a placeholder to print the value of any variable or value.

args: These are the variable/values corresponding to the format specifier.

Watch it on YouTube:

https://youtube.com/shorts/BvJ-XmEUN8Y

1.4 Arithmetic operators in C

Arithmetic operators in C are fundamental for performing mathematical calculations. These operators include addition (+), subtraction (-), multiplication (*), division (/), and modulus (%). They are used to manipulate numerical values and variables within a C program. The modulus operator (%) provides the remainder of a division, which is useful in various programming tasks. The order of operations follows the standard mathematical precedence, with parentheses taking the highest priority. Understanding and correctly using these operators is crucial for writing effective C code that involves numerical computation and data processing. Proficiency in these operators lays the foundation for more complex algorithms and programming techniques.

Following is the list of arithmetic operators in C:

+ Adds two operands.

- Subtracts second operand from the first.

* Multiplies both operands.

/ Divides numerator by denominator.

% Modulus Operator gives remainder after dividing first operand with second operand.

Watch it on YouTube:

https://www.youtube.com/shorts/9wv5Gu_vWXw

1.5 Format specifier in C

The format specifier in C is used to tell the compiler about the type of data to be printed or scanned in input and output operations. They always start with a % symbol and are used in the formatted string in functions like printf(), scanf, etc.

The C language provides a number of format specifiers that are associated with the different data types such as %d for int, %f for float, %c for char, etc.

The conversion character defines the data type to be handled, such as integers (e.g., %d), floating-point numbers (e.g., %f, %lf), characters (e.g., %c), and strings (e.g., %s). Additional specifiers can modify the output, controlling things like field width, precision, and padding.

Watch it on YouTube:

https://youtube.com/shorts/Firs-Ox1SYo

1.6 Arithmetic Assignment Operators in C

Arithmetic Assignment Operators in C combine an arithmetic operation with assignment. They provide a shorthand way to modify a variable's

value by performing an operation and storing the result back into the same variable. Examples include += (addition assignment), -= (subtraction assignment), *= (multiplication assignment), /= (division assignment), and %= (modulo assignment). These operators improve code readability and efficiency by reducing redundancy, as they avoid the need to write the variable name twice in a single statement. For instance, x+=5; is equivalent to x=x+5; x-=3 is equivalent to x=x-3 and so on.

Arithmetic Assignment Operators are also known as compound assignment operators. These operators combine an operation and assignment in a single step.

Watch it on YouTube:

https://www.youtube.com/shorts/5QXzepAu5uM

1.7 Types/Categories of operators in C

In C programming, operators are categorized based on the number of operands they operate on. There are 3 categories of operators in C:

Unary operators

Binary operators

Ternary operators

Unary operators work on a single operand (e.g., increment/decrement operators). Binary operators require two operands (e.g., arithmetic operators like addition and subtraction). Ternary operators, specifically the conditional operator (?:), work on three operands. These classifications help define how operators function and how they are used to manipulate data within a C program. Each type has a different precedence and associativity, which impacts the order of operations in an expression.

Watch it on YouTube:

https://youtube.com/shorts/1u1BgrQ78Yg

1.8 Swapping two variables in C

Swapping two variables in C involves exchanging the values stored in them. There are several common methods to achieve this. One approach is using a temporary variable to hold the value of one variable while assigning the other variable's value. Another approach uses arithmetic operations like addition and subtraction. A third method utilizes bitwise XOR operations, which is often considered efficient.

The choice of method can depend on factors like code readability and performance considerations for specific compilers and architectures. While the core concept remains the same, subtle differences might exist based on the compiler and optimization levels used.

The segment of code for swapping (interchanging) two variables is as follows:

temp = first;

first = second;

second = temp;

In the above segment of program, the temp variable is assigned the value of the first variable. Then, the value of the first variable is assigned to the second variable. Finally, the temp (which holds the initial value of first) is assigned to second. This completes the swapping process.

Watch it on YouTube:

https://youtube.com/shorts/_DFHtMMae7c

1.9 C preprocessor directive in C (#include directive)

C preprocessor directives are commands embedded in the C source code that are processed by the C preprocessor before the actual compilation. These directives begin with a '#' symbol and perform tasks like including header files, defining macros, and conditional compilation.

preprocessor directive is used to include the contents of a specified file into the source code before compilation. It allows you to use functions, constants, and variables from external libraries or header files. There are two types:

#include <file_name>

#include "filename"

Here, file inclusion with double quotes (" ") tells the compiler to search for the header file, first in the directory of source file; if not found, then search in system directory, while <> is used for system libraries, which would be searched .only in the system directory.

Watch it on YouTube:

https://youtube.com/shorts/dJ_P7PxXZFM

1.10 Comments in C

In C programming, the comments are human-readable notes in the source code of a C program used to make the program easier to read and understand. They are not a part of the executable program by the compiler or an interpreter.

There are two main types of comments: single-line comments, denoted by //, multi-line comments, enclosed between /* and */. Single-line comments are used for short explanations, while multi-line comments can span multiple lines and are often used for larger blocks of text, like function descriptions or to temporarily disable code segments during debugging.

Proper commenting is essential for collaborative projects and for understanding code years after it's written. Well-commented code saves time and reduces errors by making the purpose and functionality of the code clear to developers.

Watch it on YouTube:

https://youtube.com/shorts/OGbxQEc8ciw



Chapter 2
Decision Making

2.1 If else statement in C

In C, the if-else statement is a fundamental control flow structure. It allows the program to execute different blocks of code based on whether a condition is true or false. The if part contains a condition, and if the condition evaluates to true, the code within the if block is executed. If the condition is false, the code within the else block (if present) is executed.

Syntax of if-else Statement

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

Example of if-else Statement

```
int n = -7;
// If the number is negative
if (n < 0)
        printf("Negative");
// If the number is not negative
else
        printf("Not Negative");
```

Watch it on YouTube:

https://youtube.com/shorts/nVZggSNUh4o

2.2 Relational operators in C

The result that we get after the relational operation is a boolean value, that tells whether the comparison is true or false. Relational operators are mainly used in conditional statements and loops to check the conditions in C programming.

Relational operators in C are used to compare two operands. The standard relational operators include: equal to (==), not equal to (!=), greater than (>), less than (<), greater than or equal to (>=), and less than or equal to (<=). These operators are fundamental for controlling the flow of a program through conditional statements (if, else, if else) and loops (for, while, do-while). Using them, you can evaluate conditions and execute different code blocks accordingly. The use of relational operators forms the basis of many programming tasks, such as sorting and filtering.

Example:

```
if (a > b)
        printf("a is greater than b\n");
else
        printf("a is less than or equal to b\n");
```

Here, '>' is a relational operator used to check whether a is greater than b or not, and accordingly, the result is printed.

Watch it on YouTube:

https://youtube.com/shorts/IGg8GeA7qRs

2.3 Nested if statement

Nested if statements involve placing one if statement inside another. This allows for more complex decision-making logic in code, where the outcome of one condition determines whether to evaluate subsequent conditions.

Nested if statements are useful when you need to test multiple conditions that depend on each other. For example, checking if a number is a 2 digit number or not.

Example:

```
if(num>=10)
{
        if(num<=99)
                printf("\n2 digit number");
        else
                printf("\nNot 2 digit number");
}
else
        printf("\nNot 2 digit number");
```

You can nest as many if statements as you want, but avoid making the code too deep - it can become hard to read. Nested if is often used together with else and else if for more complex decision making.

Watch it on YouTube:

https://youtube.com/shorts/qzrQ_By3C94

2.4 Operator precedence of logical operators

Operator Precedence, also known as operator hierarchy, is a set of rules that controls the order in which operations are performed in an expression without parentheses. It is a fundamental concept in programming languages and is crucial for writing correct and efficient code.

The operator precedence of logical operators dictates the order in which logical operations are evaluated in an expression. Typically, logical operators have a defined hierarchy. For example, 'NOT' usually has the highest precedence, followed by 'AND', and then 'OR' having the lowest. This means 'NOT' operations are performed first, then 'AND' operations, and finally 'OR' operations. Parentheses can override the default precedence, allowing for explicit control over the evaluation order. Understanding this precedence is crucial to ensure that logical expressions are evaluated as intended, avoiding unexpected results in programming and other contexts.

Watch it on YouTube:

https://youtube.com/shorts/QYxkZsTn_50

2.5 Conditional operator in C

The conditional operator in C is kind of similar to the if-else statement as it follows the same algorithm as of ‘if’. The conditional operator takes less space and helps to write the if-else statements in the shortest way possible. It is also known as the ternary operator in C as it operates on three operands.

The conditional operator provides a concise way to express conditional assignments. It takes the following form:

condition ? expression1 : expression2

If the condition evaluates to true, expression1 is evaluated; otherwise, expression2 is evaluated. The result of the chosen expression becomes the operator's overall result. This is useful for short, simple conditional assignments or return values, making code more readable compared to longer if-else statements in some instances.

Example:

```
int a,b,c;

a=7;

b=3;

c = a>b ? a : b;
```

In the above segment of code the statement, "c = a>b ? a : b;" is evaluated as follows:

It checks the condition a > b. If the condition is true, the value of a is stored in c. Otherwise, the value of b is stored in c.

Watch it on YouTube:

https://youtube.com/shorts/oRPVOm2a0qU

2.6 switch-case statement in C

In C, switch statement is a conditional statement that allows you to execute different code blocks based on the value of a variable or an expression. It is often used in place of if-else ladder when there are multiple conditions.

The switch-case statement is similar to multiple if-else statements in C. The if-else statement provides two alternative actions to be performed, whereas the switch-case construct is a multi-way branching statement. A switch statement in C simplifies multi-way choices by evaluating a single variable against multiple values, executing specific code based on the match. It allows a single variable or expression to be tested for equality against a list of values.

The parenthesis in front of the switch keyword holds an expression. The expression should evaluate to an integer or a character. Inside the curly brackets after the parenthesis, different possible values of the expression form the case labels.

One or more statements after a colon(:) in front of the case label forms a block to be executed when the expression equals the value of the label.

You can literally translate a switch-case as "in case the expression equals value1, execute the block1", and so on.

C checks the expression with each label value, and executes the block in front of the first match. Each case block has a break as the last statement. The break statement takes the control out of the scope of the switch construct.

You can also define a default case as the last option in the switch construct. The default case block is executed when the expression doesn't match with any of the earlier case values.

Syntax of switch

```
switch (expression)

{

case value1:

// Code_block1

break;

case value2:

// Code_block2

break;

default:

// Default_code_block

}
```

Here,

expression: Expression or variable whose value is tested. It should result in an integer or character value.

case: Each switch case represents a block of code that will be executed with the expression evaluates to its corresponding value. There should be at least one case in the switch.

value1, value2: These are the possible values of the expression for which the corresponding cases are executed. These values must be unique.

break: This is used to exit the switch statement once a case is executed. Without break, the program will continue executing the subsequent cases (this is called "fall through").

default: This block of code is executed if none of the case values match the expression. It is optional to add this block.

Watch it on YouTube:

https://youtube.com/shorts/1HGTgEhZYZI

****

Chapter 3
Looping

3.1 While loop in C

The while loop in C allows a block of code to be executed repeatedly as long as a given condition remains true. It is often used when we want to repeat a block of code till some condition is satisfied. The condition is evaluated before each iteration. The loop continues until the condition becomes false.

The syntax of the while loop is:

```
while (condition)
{
        // the body of the loop
}
```

The while loop evaluates the condition inside the parentheses (). If condition is true, statements inside the body of while loop are executed. Then, condition is evaluated again. The process goes on until condition is evaluated to false. If condition is false, the loop terminates (ends).

It's a fundamental control flow structure for tasks involving repetition where the number of iterations might not be known beforehand. It's a pre-test loop, meaning the condition is checked before the loop body is executed. Proper initialization of variables used in the condition and updating them within the loop body are crucial to prevent infinite loops.

Watch it on YouTube:

https://youtube.com/watch?v=Q9BlM5s36nc&t=26s



3.2 Increment/ decrement operator in C

The increment (++) and decrement (--) operators in C are unary operators for incrementing and decrementing the numeric values by 1, respectively. They are one of the most frequently used operators in programming for looping, array traversal, pointer arithmetic, and many more.

Increment Operator in C

The increment operator (++) is used to increment the value of a variable in an expression by 1. It can be used on variables of the numeric type, such as integer, float, character, pointers, etc.

The Increment Operator in C can be used in two ways, either as a prefix (pre-increment) (eg. ++m) or a postfix (post-increment) (eg. m++). In prefix form, the increment/decrement happens before the variable's value is used in the expression. In postfix form, the increment/decrement happens after the variable's value is used.

Decrement Operator in C

The decrement operator is used to decrement the value of a variable in an expression. In the Pre-Decrement, the value is first decremented and then used inside the expression. Whereas in the Post-Decrement, the value is first used inside the expression and then decremented.

Understanding the difference between prefix and postfix is crucial for predicting the outcome of expressions, especially when these operators are combined with other operators or used within complex statements.

These operators are commonly used in loops and for manipulating counters.

Watch it on YouTube:

https://youtube.com/shorts/6VlatKsUzl4

3.3 Types of while loop

The while loop in C allows a block of code to be executed repeatedly as long as a given condition remains true. It is often used when we want to repeat a block of code till some condition is satisfied.

While loops come in several forms, each with distinct characteristics. Following are the types of while loop:

while

In while loop, a condition is evaluated before processing a body of the loop. If a condition is true then and only then the body of a loop is executed. Condition is written at the top of the loop. This is called as entry-controlled loop.

do-while loop

The do-while loop, executes the body at least once before evaluating the condition. Condition is written at the bottom of the loop. This is called as exit-controlled loop.

while(1)

while(1) is an infinite loop which will run infinite number of times. But, we can use the break statement explicitly to terminate the loop. Interestingly not while(1) but any integer which is non-zero will give a

similar effect as while(1). Therefore, while(1), while(2) or while(-255), all will give the effect of infinite loop only.

Watch it on YouTube:

https://youtube.com/shorts/M83Wk-qn5IE



3.5 Difference between while and do-while loop

while loop and do while loop concepts are fundamental to control flow in programming, allowing developers to create loops that repeat a block of code based on certain conditions. The choice between "while" and "do while" depends on the specific requirements of the program and the desired behavior of the loop.

The key difference between while and do-while loops lies in their execution behavior. A while loop checks the condition before executing the loop body. If the condition is initially false, the body is never executed. While is considered as 'entry-controlled' loop.

Conversely, a do-while loop checks the condition after executing the loop body. This guarantees that the body is executed at least once, regardless of the initial condition. The do-while loop is useful when you need to perform an action at least once, such as prompting a user for input. do-while is considered as 'exit-controlled' loop.

Watch it on YouTube:

https://youtu.be/hnlNqc0LaG4

3.6 For loop in C

A for loop in C is a control flow statement that allows code to be executed repeatedly based on a counter. We can use for loop to iterate over a sequence of elements or perform a set of tasks a fixed number of times.

The general syntax of for loop varies slightly depending on the programming language, but it typically consists of three main components: initialization, condition, and increment (or decrement).

Syntax:

```
for (initialization; condition; increment/decrement)
{
        // Code to be executed repeatedly
}
```

It typically consists of three parts: initialization, a condition, and an increment/decrement. The initialization sets up the counter, the condition is checked before each iteration, and the increment/decrement changes the counter after each iteration. The loop continues as long as the condition is true. For loops are essential for tasks like iterating through arrays or performing a specific action a fixed number of times.

Example:

```
for(x=1 ; x<=5 ; x=x+1)
{
```

```
        -------

        -------

        -------
}
```

Watch it on YouTube:

https://youtube.com/shorts/XUO0qKrBOy4

3.7 Nested for loop in C

A nested loop means a loop statement inside another loop statement. For a nested loop, the inner loop performs all of its iterations for each iteration of the outer loop.

Nested for loops in C are loops within loops, allowing for iterating over multiple dimensions of data or performing repetitive tasks. The outer loop controls the overall iterations, while the inner loop completes its iterations for each cycle of the outer loop.

Example:

If the outer loop is running from i = 1 to 5 and the inner loop is running from j = 1 to 3, then, for each value of the outer loop variable (i), the inner loop will run with the value of j from 1 to 3.

Syntax:

```
for ( initialization; condition; increment ) // outer loop
{
```

```
        // statement of outer loop
        for ( initialization; condition; increment ) // inner loop
        {
                // statement of inside loop
        }
        // statement of outer loop
}
```

Watch it on YouTube:

https://youtube.com/shorts/_mTilY0gcwQ



3.8 break and continue statement in C

The break and continue statements are the jump statements that are used to skip some statements inside the loop or terminate the loop immediately without checking the test expression. These statements can be used inside any loops such as for, while, do-while or while(1) loops.

break:

The break statement in C is used to terminate from the loop immediately. When a break statement is encountered inside a loop, the loop iteration stops there, and control returns from the loop immediately to the first statement after the loop. Basically, break statements are used in situations when we are not sure about the actual number of iteration for the loop, or we want to terminate the loop based on some condition. break statement is also used in switch statement to terminate the construct after one of the case is executed.

continue:

The continue statement, on the other hand, skips the remaining statements in the current iteration of the loop and proceeds to the next iteration. Both are helpful for controlling the flow of a program and can be used to handle specific conditions or to optimize loop performance.

Watch it on YouTube:

https://youtube.com/shorts/pih-loe_JX4



****

Chapter 4
Arrays

4.1 Arrays in C (short)

An array is a linear data structure that stores a fixed-size sequence of elements of the same data type in contiguous memory locations. Each element can be accessed directly using its index, which allows for efficient retrieval and modification. Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

Array declaration is the process of specifying the type, name, and size of the array. In C, we have to declare the array like any other variable before using it. Declaring an array involves specifying the data type and the number of elements.

Array in C provides random access to its elements, which means that we can access any element of the array by providing the position of the element, called the index.

Arrays can be one-dimensional, multi-dimensional, and are often used in conjunction with loops to iterate over elements and perform operations.

Syntax:

data_type array_name[size];

The above statements create an array with the name array_name, and it can store a specified number of elements (size) of the same data type (data_type).

Example:

// Creates array arr to store 5 integer values.

int arr[5];

Watch it on YouTube:

https://youtube.com/shorts/p8JFjf1tSwQ

https://www.youtube.com/watch?v=fLs8-0V1z04

4.2 Multi-dimensional Array in C

A multi-dimensional array in C can be defined as an array that has more than one dimension. Having more than one dimension means that it can grow in multiple directions. In C, multi-dimensional arrays are essentially arrays of arrays. Some popular multidimensional arrays include 2D arrays which grows in two dimensions, and 3D arrays which grows in three dimensions.

Multi-dimensional arrays are declared using multi-values in brackets.

Example:

int array[3][4];

The above declaration creates a 2D array with 3 rows and 4 columns. Accessing elements is done using indices, like array[1][2]. Understanding how these arrays map to memory is crucial for efficient programming, particularly when dealing with large datasets or image processing, as it influences memory access patterns and performance.

Watch it on YouTube:

https://youtube.com/shorts/s9hBx1mHCDo

4.3 Character Array (string) in C

In C, a string is often represented as a character array. These arrays are null-terminated, meaning a special character '\0' marks the end of the string, allowing functions to determine its length. This representation enables direct memory access and efficient manipulation of text data. Common operations include string length calculation, copying, and concatenation, often performed using functions from the string.h library.

Example:

char greeting[] = "Hello";

Here, greeting is an array of characters with following characters in sequence: {'H', 'e', 'l', 'l', 'o', '\0'}.

Watch it on YouTube:

https://youtube.com/shorts/O7X2HgkfbPc

****

Chapter 5
Pointers

5.1 Pointer in C

A pointer is a variable that stores the memory address of another variable. Instead of holding a direct value, it holds the address where the value is stored in memory. It is the backbone of low-level memory manipulation in C. Accessing the pointer directly will just give us the address that is stored in the pointer.

Pointers allow functions to modify data outside their scope and are essential for working with arrays, strings, and linked lists. Understanding pointers is crucial for mastering C and its capabilities.

A pointer is declared by specifying its name and type, just like simple variable declaration but with an asterisk (*) symbol added before the pointer's name.

Syntax:

data_type* name

Here, data_type defines the type of data that the pointer is pointing to. An integer type pointer can only point to an integer. Similarly, a pointer of float type can point to a floating-point data, and so on.

Example:

int *ptr;

In the above statement, pointer ptr is declared, that can store the address of an integer. It is pronounced as pointer to integer.

Watch it on YouTube:

https://youtube.com/shorts/OTS4a0aTuRk

5.2 Relationship Between Pointers and Arrays in C

In C, there's a strong relationship between pointers and arrays. Whenever an array is created, actually, a pointer is created by the name of array that points to the first element of array. You can access array elements using pointer arithmetic, such as *(array + i) which is equivalent to array[i].

Arrays and pointers are not exactly the same; arrays allocate memory for elements, while pointers store addresses. However, this close relationship offers flexibility. You can pass arrays to functions using pointers, avoiding the need to copy the entire array. Understanding this connection is vital for memory management and efficient C programming.

Watch it on YouTube:

https://youtube.com/shorts/e_LoEVeA99U

****

Chapter 6
Functions

6.1 Functions in C

Functions in C are self-contained blocks of code designed to perform specific tasks. Every C program has at least one function, which is main(). Functions promote code reusability and modularity, making programs easier to understand, maintain, and debug.

When the algorithm of a certain problem involves long and complex logic, it is broken into smaller, independent and reusable blocks. These small blocks of code are known by different names in different programming languages such as a module, a subroutine, a function or a method.

A function in C typically consists of a return type, a name, a list of parameters, and a body of code enclosed in curly braces. Functions can be called from other parts of the program, and they can return a value or perform actions without returning a value (void). Proper use of functions is crucial for writing efficient and organized C programs.

Watch it on YouTube:

https://youtube.com/shorts/KfxgNJczj2k



6.2 Basics of functions in C

In programming, functions are fundamental building blocks that encapsulate a specific task or set of operations. Understanding the terms related to functions is crucial for effective programming.

Function Declaration (or Function Prototype):

A function declaration informs the compiler about a function's existence, its name, the type of value it will return (return type), and the types and order of the values it expects as input (parameters). It acts as a promise to the compiler that the function will be defined later in the code.

Function Definition:

The function definition provides the actual implementation or body of the function. It includes the set of statements that will be executed when the function is called. The definition must match the declaration in terms of return type, name, and parameters.

Function Call:

A function call is the act of executing the code within a function. When a function is called, control of the program flow transfers to the function's body, and the statements within that body are executed. After the function completes its task (and potentially returns a value), control returns to the point from where the function was called.

Watch it on YouTube:

https://youtube.com/shorts/afi7nBbYQko



6.3 String handling functions

String handling functions are pre-defined routines in programming languages that allow for the manipulation and processing of strings, which are sequences of characters. These functions provide efficient ways to perform common operations on text data.

6.3.1 strlen function in C

The strlen function in C is a standard library function used to determine the length of a null-terminated string. It counts the number of characters in a string until it encounters a null character (‘\0’). The function takes a single argument, a pointer to a character array (string), and returns an integer representing the length of the string. It’s also important to note that strlen does not include the null terminator in the length.

Example:

char str[] = “Hello”;

l = strlen(str);

printf(“\nLength of string = %d”,l);

The above segment of code will print Lenth of string = 5, because str contains a string that contains 5 characters.

Watch it on YouTube:

https://youtube.com/shorts/s5tR4kgqAUU

6.3.2 strcmp function in C

The strcmp function in C is used to compare two strings. It takes two arguments, which are pointers to the strings to be compared. It returns an integer value that indicates the relationship between the strings: a negative value if the first string is smaller than the second, zero if the strings are equal, and a positive value if the first string is greater than the second. The comparison is based on the ASCII values of the characters in the strings. This function is case-sensitive and crucial for various string manipulation tasks in C programming, such as sorting and searching.

Example:

char stra[] = “abcdefg”;

char strb[] = “abfrt”;

int p = strcmp(stra,strb);

printf(“\np = %d”,p);

The above segment of code prints -3 as the value of p, which is the difference between ‘c’ (ASCII-67) and ‘f’ (ASCII-70), those are the third position characters in both the strings. (First two characters of the string are same)

Watch it on YouTube:

https://youtube.com/shorts/TGe9o2Xu3zk



6.3.3 strcpy function in C

The strcpy function in C is used to copy a string from a source location to a destination location. It takes two arguments: a pointer to the destination buffer and a pointer to the source string. strcpy copies characters from the source string to the destination buffer until it encounters a null terminator ('\0'). A crucial thing to remember is that strcpy doesn't perform bounds checking; this means that if the source string is larger than the destination buffer, it can lead to buffer overflows, a common source of security vulnerabilities. For safer alternatives, strncpy or other safer string copy functions should be used, which allow specifying the maximum number of characters to copy, mitigating the risk of overflows.

Example:

char stra[] = "abc";

```
char strb[] = "xyz";

strcpy(stra,strb);

printf("\nstra = %s",stra);

printf("\nstrb = %s",strb);
```

The above segment of code will copy the contents of strb string in the existing contents of stra string and prints the following output on the screen:

```
stra = xyz

strb = xyz
```

Watch it on YouTube:

https://youtube.com/shorts/LJjgVIOddxg

6.3.4 strcat function in C

The strcat function in C is used to concatenate two strings. It appends a copy of the source string to the end of the destination string. The destination string must have enough allocated space to hold the combined string, including the null terminator. It's important to be aware of potential buffer overflows if the destination string isn't large enough, which can lead to security vulnerabilities. Strcat takes two arguments: the destination string and the source string.

Example:

```
char stra[] = "abc";

char strb[] = "xyz";

strcat(stra,strb);
```

printf("\nstra = %s",stra);

printf("\nstrb = %s",strb);

The above segment of code will add the contents of strb string in front of the existing contents of stra string and prints the following output on the screen:

stra = abcxyz

strb = xyz

Watch it on YouTube:

https://youtube.com/shorts/Rgm1RQw_VNU

6.4 Recursion in C

Recursion is the process of defining something in terms of itself. In C, we know that a function can call other functions. It is even possible for the function to call itself. These types of construct are termed as recursive functions.

Recursion is a programming technique where a function calls itself repeatedly until a specific base condition is met. It is the technique of making a function, call itself. This technique provides a way to break complicated problems down into simple problems which are easier to solve.

A function that performs such self-calling behaviour is known as a recursive function, and each instance of the function calling itself is called a recursive call.

A recursive function is a function that solves a problem by calling itself on a smaller subproblem. It typically consists of two main parts:

Base Case – the condition under which the recursion stops.

Recursive Case – where the function, calls itself with modified arguments, gradually approaching the base case.

Watch it on YouTube:

https://youtube.com/shorts/cRqFQKa13Co



https://youtu.be/aC-H7dTG7Xg



****

Chapter 7
Structure, Union and Enum

7.1 Structure in C

Structures in C are user-defined data types that group related variables of different data types under a single name. This allows for organizing data logically. Structures are defined using the struct keyword, followed by a tag name and a list of members (variables) enclosed in curly braces. When a struct type is declared, no storage or memory is allocated. To allocate memory of a given structure type and work with it, we need to create variables.

Structures allow you to organize data in a clean and logical way, making your programs easier to manage and scale. Whether you're building a student record system or designing a billing application, understanding structures in C programming is a must.

Structures in C, let you combine different types of variables under one name. Instead of creating separate variables for related data (like a student's name, age, and marks), you can group them together in a structure.

This makes your code cleaner, more organized, and easier to manage - especially when working with large datasets or records. Each variable inside a structure is called a member, and they can be of different data-type.

Example:

```
struct employee
{
        int empno;
        char nm[20];
        float salary;
};
```

struct employee e1;

Watch it on YouTube:

https://youtube.com/shorts/TOTssmRu9XI

7.2 Nested structure in C

A nested structure in C is a structure within a structure. One structure can be declared inside another structure. In the same way, structure members are declared inside a structure.

When one of the elements in the definition of a struct type is of another struct type, then we call it a nested structure in C. Nested structures are defined when one of the elements of a struct type is itself a composite representation of one or more types.

This allows for organizing data in a hierarchical manner, representing complex relationships between different data elements. This approach enhances code readability and modularity, enabling better management of related data. For instance, a structure representing an 'employee' might contain a nested structure for 'address', encapsulating street, city, and postal code. Accessing members of nested structures requires using the dot operator (.) multiple times. This design pattern promotes data abstraction and simplifies data manipulation in C programs.

Syntax:

A general syntax of a nested structure is as follows :

```
struct struct1
{
        type var1;
```

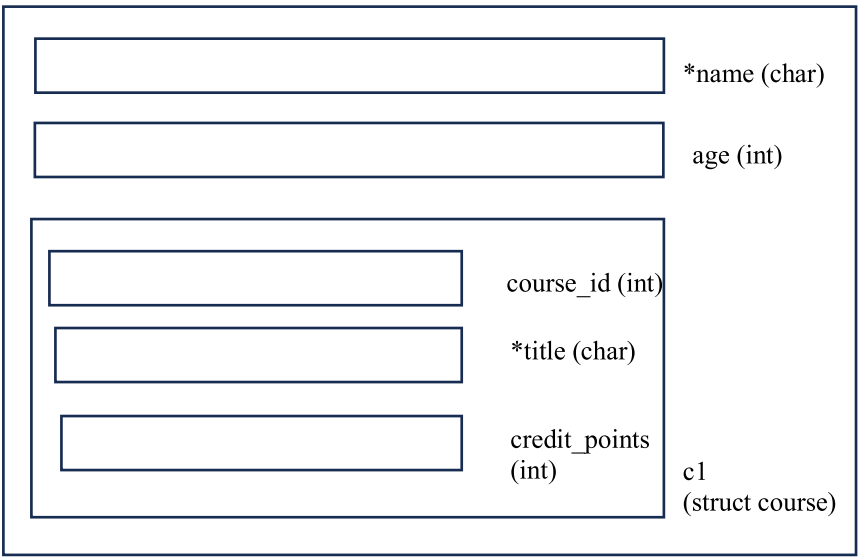
```
        type var2;
        struct struct2 strvar;
};
```

Example:

We can think of nested structures in the following situation. If we want to define a struct type representing a "student" with "name" and "age" as its elements and another struct type called "course" that is characterized by "course ID", "title", and "credit points". Here, the "student" structure has an inner course structure.

```
struct course
{
        int course_id;
        char *title;
        int credit_points;
};
struct student
{
        char *name;
        int age;
        struct course c1;
};
struct student s1;
```

The ‘s1’ variable of ‘struct student’ data-type will be stored in the memory as follows :



s1 (struct student)

Watch it on YouTube:

https://youtu.be/JmTHRMtykS4

7.3 Structure and function in C

Structures in C are user-defined data types that group variables of different data types under a single name. They're fundamental for organizing related data. Functions in C are blocks of code designed to perform specific tasks, promoting code reusability and modularity.

Just as we can pass arguments of primary data types, a variable of struct data type can also be passed to a function. You can also pass structures using call by value and call by reference methods. A function in C may also return a struct data type.

Watch it on YouTube:

https://youtu.be/KkhH4RjprXA

7.4 Structure and pointer in C

In C programming, structures and pointers are fundamental concepts often used together. A structure is a user-defined data type that groups variables of different data types under a single name. Pointers, on the other hand, are variables that store the memory address of another variable.

You can use pointers with structures to make your code more efficient, especially when passing structures to functions or changing their values.

To use a pointer to a structure, just add the * symbol, like you would with other data types. To access its members, you must use the -> operator instead of the dot . operator.

Watch it on YouTube:

https://youtube.com/shorts/9w33SKExAG4

7.5 Union in C (User-defined Data Type) in C

In C, union is a user-defined data type that can contain elements of the different data types just like structure. Unlike a structure, which allocates enough space to hold all its members, a union allocates enough space for

its largest member. This means, all members share the same memory. Only one member of a union can be active at any given time.

The value of a union variable can be accessed using the operator. A value can be assigned to the union variable using the assignment.

In a union, all the variables share the same memory, so only one variable can store a value at a time.

Watch it on YouTube:

https://youtube.com/shorts/mCczQE_bugk

7.6 Enumerated data type in C

An enumerated data type (or enum) is a user-defined data type that assigns meaningful names to a fixed set of integral constants, making code more readable and maintainable by replacing raw numbers with descriptive labels.

Each member of an enum is assigned an integer value, starting from 0 by default, but these values can be explicitly specified. Enums are commonly used to represent states, options, or any set of related values. Using enums makes the code easier to understand compared to using raw integer constants directly. The compiler also automatically assigns integer values to enum members if not specified, minimizing errors.

Using the enum data type in C language makes any program much easier to maintain as well as understand. We define enums using the keyword 'enum'.

Watch it on YouTube:

https://youtube.com/shorts/JP3pS4dcB6w

****

Chapter 8
Storage Classes

8.1 Storage class in C

Storage classes in C define the scope, lifetime, visibility, and memory location of variables. They determine how a variable is allocated and accessed throughout the program, ensuring efficient memory usage and control over variable behaviour.

A storage class represents the visibility and a location of a variable. It tells from what part of code we can access a variable. A storage class in C is used to describe the following things:

The variable scope.

The location where the variable will be stored.

The initialized value of a variable.

A lifetime of a variable.

Common storage classes include auto, static, extern, and register.

Watch it on YouTube:

https://youtube.com/shorts/ruMnrHz-OhI



8.2 Automatic storage class in C (auto)

In C, the auto keyword specifies the automatic storage class for a variable. This is the default storage class for variables declared inside a function or a block. It means the variable is created when the block is entered and

destroyed when the block is exited. The auto keyword is rarely explicitly used because it's the default; variables declared inside a function are auto by default. The automatic storage class allocates memory on the stack. This is in contrast to other storage classes like static, which persist throughout the program's execution, or extern, which indicates a variable defined elsewhere.

Watch it on YouTube:

https://youtube.com/shorts/OTJEtMm9EY4

8.3 static storage class in C

This storage class is used to declare static variables that have the property of preserving their value even after they are out of their scope! Hence, static variables preserve the value of their last use in their scope. It is capable of returning a value even when control is transferred to the function call. The keyword static retains the variable's value between function calls, giving it a lifetime throughout the program's execution, unlike non-static local variables that are created and destroyed with each function call.

Watch it on YouTube:

https://youtube.com/shorts/rrgCc0AUPn0

8.4 Register storage class in C

In C, the register storage class specifier is a hint to the compiler to store a variable in a CPU register for faster access. However, modern compilers often optimize code so effectively that the register keyword is largely ignored, and the compiler decides the best location for a variable. Using register may also be restricted depending on the C standard version. It's primarily used for variables that are frequently accessed, like loop counters, to potentially improve performance. The actual impact of register is compiler and architecture-dependent, and it is generally not recommended to rely on it.

Watch it on YouTube:

https://youtube.com/shorts/6L7ojyilNWM



8.5 External storage class in C

In C, the extern storage class is used to declare a variable or function that is defined elsewhere in the program. It informs the compiler that the variable or function is defined in another file or later in the same file. extern declarations don't allocate storage; they simply tell the compiler about the variable's or function's type and name.

Extern storage class is used when we have global functions or variables which are shared between two or more files.

Keyword extern is used to declaring a global variable or function in another file to provide the reference of variable or function which have been already defined in the original file. The variables defined using an extern keyword are called as global variables. These variables are accessible throughout the program.

Watch it on YouTube:

https://youtube.com/shorts/HIh8IBr5FgU



****

Chapter 9
Input-Output in C

9.1 getch function in C

C programming offers various ways to handle user input. One important function is getch().

The getch() function reads a single character from the keyboard. This function comes from the conio.h. It's a form of input, as it receives data from the user. The program does not display the input character on screen. The function does not need the Enter key to read input.

It's often used for creating simple interactive console applications or games where immediate keyboard input is desired.

Watch it on YouTube:

https://youtube.com/shorts/CidUqeEIdWY



9.2 Difference between getch, getche and getchar functions

The functions getch(), getche(), and getchar() are all used to read a single character from the console, but they differ in how they handle input and output. getch() reads a character without echoing it to the screen, making it suitable for password input. getche() is similar to getch() but echoes the character to the screen. getchar() reads a character from the standard input, which is typically the keyboard, but it waits for the Enter key to be pressed; meaning the input is buffered. Therefore, getchar() is different because it reads buffered input, unlike the other two that read unbuffered input.

Watch it on YouTube:

https://youtube.com/shorts/LtEhK3aLbPQ

9.3 gets function in C

The C library gets() function is a standard library function used to read a line from the standard input stream (stdin) and store it into the string pointed to by str. Unlike scanf(), gets() reads a whole line of text, stops reading when Enter key is pressed, and does not discard the newline character. The newline character is then replaced by a null terminator, and the resulting string is stored in str.

Watch it on YouTube:

https://www.youtube.com/shorts/-tZJkOSaj1U

9.4 sscanf function in C

In C, the sscanf() function can be used to read formatted data from a string rather than a standard input or keyboard. It works similar to scanf but instead of reading from standard input (like the keyboard), it reads from a character array (a string) that you provide. The function parses the string according to a format string, extracting values and storing them in variables. It's a handy tool for parsing data from strings, converting strings to numbers, and extracting specific parts of a string based on the format

specifiers used. Properly using format specifiers like %d for integers, %f for floats, and %s for strings is essential for successful parsing.

Watch it on YouTube:

https://youtube.com/shorts/kuKD-OFdY20

9.5 sprintf function in C

The sprintf function in C is used for formatted output, similar to printf, but instead of printing to the console, it writes the formatted output to a character string. It takes a character buffer, a format string, and a variable number of arguments as input. The format string specifies how the subsequent arguments should be formatted and inserted into the string. sprintf is commonly used for creating strings from numerical values, concatenating strings, and formatting data for display or storage. However, it's important to use sprintf carefully, as it's prone to buffer overflows if the output string is not large enough to accommodate the formatted data.

Watch it on YouTube:

https://youtube.com/shorts/XzkS2jZh-Xo

9.6 Formatted and Unformatted functions in C

Formatted input/output functions in C, like printf and scanf, use format specifiers to control the way data is displayed or read. They offer flexibility in handling various data types.

Unformatted functions, such as getchar and putchar, deal with single characters or raw data without format specifications. These are simpler and faster for basic input/output operations.

Understanding the differences helps in choosing the appropriate functions for specific tasks, balancing between ease of use and performance.

Watch it on YouTube:

https://youtube.com/shorts/0di0xTijItM

9.7 File opening modes in C

File opening modes in C determine how a file is accessed. Common modes include 'r' (read), 'w' (write, overwriting existing content), 'a' (append, writing to the end of the file), 'r+' (read and write), 'w+' (read and write, truncating or creating), and 'a+' (read and append). Binary modes like 'rb', 'wb', 'ab', 'rb+', 'wb+', and 'ab+' are also available, treating the file as a sequence of bytes and preventing character conversions. The chosen mode dictates whether you can read, write, or both, and how the file is handled if it already exists or doesn't exist.

Watch it on YouTube:

https://youtube.com/shorts/75iwr8hS8aI

9.8 File opening and closing in C

In C, file operations involve opening a file using functions like fopen(), performing read/write operations, and closing the file with fclose(). It is crucial to close files after use, to release system resources and prevent data corruption. Files can be opened in various modes (read, write, append, etc.) specified in the fopen() call. Improper file handling, such as failing to close files or not checking for errors during file operations, can lead to resource leaks and program instability. File operations in C demand careful attention to detail for robust and reliable applications.

Watch it on YouTube:

https://youtube.com/shorts/cEjkRZnPWYU

Chapter 10
Bitwise Operations

10.1 Bitwise operators in C

Bitwise operators in C allow manipulation of individual bits within integer data types. These operators include AND (&), OR (|), XOR (^), NOT (~) (also called as one's complement operator), left shift (<<), and right shift (>>). They are crucial for tasks like setting or clearing specific bits, creating bit masks, and efficiently packing or unpacking data. Their usage is particularly prevalent in low-level programming, embedded systems, and areas requiring fine-grained control over data representation. Understanding bitwise operators is fundamental to optimizing code and working with hardware at a fundamental level.

Watch it on YouTube:

https://youtube.com/shorts/xSPvlQ9IGH8

10.2 One's complement operator in C

The one's complement operator in C, denoted by the tilde (~), is a unary operator that performs a bitwise NOT operation. It inverts each bit of an integer operand. If a bit is 0, it becomes 1, and if a bit is 1, it becomes 0. This operation is commonly used in tasks like inverting bit patterns, creating bit masks, or performing certain low-level manipulations of data. The one's complement operator operates on integers and the result is also an integer.

Watch it on YouTube:

https://youtube.com/shorts/gM91ul66sRE

10.3 Left shift and Right shift operators in C

In C, the left shift (<<) and right shift (>>) operators perform bitwise shifting operations on integer operands. The left shift operator shifts the bits of the first operand to the left by the number of positions specified by the second operand, effectively multiplying the number by a power of 2. The right shift operator shifts the bits of the first operand to the right. For unsigned integers, this involves shifting in zeros from the left. For signed integers, the behaviour can be implementation-defined; it may involve either sign extension (shifting in copies of the sign bit) or zero-fill. These operators are often used for efficient multiplication, division, and bit manipulation tasks.

Watch it on YouTube:

https://youtube.com/shorts/_dTsnV-htcY

10.4 Bitwise AND operator

The bitwise AND operator performs a logical AND operation on each corresponding bit of two integer operands. If both bits are 1, the resulting bit is 1; otherwise, it's 0. This operator is commonly used for tasks like

checking if a specific bit is set (equal to 1), masking specific bits to isolate certain values, or clearing specific bits. It is a fundamental operation in low-level programming, hardware interfacing, and data manipulation in various contexts, including network programming and data compression.

Watch it on YouTube:

https://youtube.com/shorts/bJ05wG1Srs4

10.5 Bitwise OR operator in C

The bitwise OR operator in C, represented by the symbol |, performs a logical OR operation on each corresponding bit of two integer operands. If either bit in the corresponding positions of the operands is 1, the resulting bit in the output is 1; otherwise, it's 0. This operator is often used for setting specific bits within a variable, combining flags, or manipulating individual bits in low-level programming tasks. The result of the operation is also an integer.

Watch it on YouTube:

https://youtube.com/shorts/b03O2MLCroQ

10.6 Bitwise XOR operator in C

The bitwise XOR operator in C, denoted by ^, performs an exclusive OR operation on the corresponding bits of two integer operands. The result of

the XOR operation for each bit position is 1 if the bits are different and 0 if they are the same. This operator is commonly used for tasks like toggling specific bits, swapping values without using a temporary variable, and in cryptography. Understanding XOR is crucial for low-level programming, data manipulation, and specific algorithms. It's a fundamental building block in many computational processes.

Watch it on YouTube:

https://youtube.com/shorts/2jkjA2Mld8o

****

Chapter 11
Macros, File Inclusion and Conditional Compilation

11.1 Macros expansion in C

Macro expansion in C is the process where the preprocessor replaces macro names with their defined values. This substitution happens before compilation. Macros can be simple text replacements or more complex, taking arguments like functions. They enhance code reusability and readability, but can sometimes lead to unexpected behaviour if not used carefully, especially with side effects. The preprocessor directives such as #define are used to define macros.

Example:

#define MAX 10

This defines the macro MAX as 10. It means, before giving the program to compiler, the C preprocessor should replace the word MAX with 10.

Watch it on YouTube:

https://youtube.com/shorts/EV2XGe-yDPQ

11.2 Macros with arguments

Function-like macros can take arguments, just like true functions. To define a macro that uses arguments, you insert parameters between the pair of parentheses in the macro definition that make the macro function-

like. The parameters must be valid C identifiers, separated by commas and optionally whitespace.

To invoke a macro that takes arguments, you write the name of the macro followed by a list of actual arguments in parentheses, separated by commas. The invocation of the macro need not be restricted to a single logical line—it can cross as many lines in the source file as you wish. The number of arguments you give must match the number of parameters in the macro definition. When the macro is expanded, each use of a parameter in its body is replaced by the tokens of the corresponding argument. (The macro body is not required to use all of the parameters.)

Example:

```
#define MIN(X, Y)  ((X) < (Y) ? (X) : (Y))
```

Here, a macro MIN is defined that computes the minimum (lowest) of two numeric values.

Watch it on YouTube:

https://youtube.com/shorts/FQ_B2vhiW0A



11.3 File inclusion in C

File inclusion in C allows you to incorporate the contents of another file (e.g. functions defined in another file) into your current source file. It allows you to use functions, constants, and variables from external libraries or header files.

This is primarily achieved using the #include preprocessor directive. There are two main ways to use #include: with angle brackets (<>) for system headers and with double quotes ("") for user-defined or local headers. When using angle brackets, the preprocessor typically searches

the standard include paths. When using double quotes, the preprocessor typically searches the current directory first, and then the standard include paths.

Watch it on YouTube:

https://youtube.com/shorts/9RElOSROEbU

11.4 Conditional compilation in C

Conditional compilation in C allows parts of the code to be included or excluded during the compilation process based on preprocessor directives. These directives, such as #ifdef, #ifndef, #if, #elif, #else, and #endif, evaluate conditions at compile time. This mechanism is useful for platform-specific code, debugging, and feature toggles. By using these directives, developers can customize the build process, create different versions of the same code, and manage dependencies effectively.

The #if is a directive preprocessor that evaluates the expression or condition. If the condition is true then this directory executes the code otherwise #elseif or #endif code is executed.

Syntax:

The general form of #if is as follows :

```
#if constant_expression
        statement sequence
#else
        statement sequence
#endif
```

The #else statement defines a block of code that runs when the condition in an if statement is false. This works similarly to the other keyword in C.

#elif stands for "else-if" and creates an if-else-if compilation chain. The #elif statement is used within an #if statement. This includes a section of text that is determined if its condition is True.

Watch it on YouTube:

https://youtube.com/shorts/ihU2JKz_PiM

****

Chapter 12
Miscellaneous

12.1 typedef statement in C

The typedef statement in C is used to create a new name (an alias) for an existing data type. This can improve code readability and portability by allowing you to use more descriptive names for types, or to abstract away implementation details. For example, you can use typedef to create a new name for a structure, a primitive data type, or a pointer. This can make it easier to understand the purpose of a variable and to modify the code later without having to change every instance of the original type. Essentially, typedef does not create a new type; it just introduces a synonym for an existing type.

Watch it on YouTube:

https://youtu.be/24-iR_R3vgw



12.2 Command Line Arguments in C

In C programming, command-line arguments allow you to pass information to a program when you run it from the command line. The main function has two parameters specifically for this: argc (argument count), which is the number of arguments, and argv (argument vector), which is an array of strings representing the arguments themselves. The first element of argv (argv[0]) is usually the program's name. After that till argv[argc-1] every element is command line arguments.

Watch it on YouTube:

https://youtube.com/shorts/rooMiYkatJE

==*=*=*=*=*=*=*=

Beyond the Semicolon

Learning C is not the end of your programming journey—it is only the beginning. Every accomplished software engineer once started with simple programs, small mistakes, and an eagerness to learn. A program may begin with a single line of code, but innovation begins with a single idea.

As you turn the last page of this book, remember that programming is a skill refined through continuous practice. Let the concepts presented here serve as your foundation, and let your curiosity guide you toward new technologies, challenges, and opportunities. The most valuable program you will ever write is the one that teaches you something new.

May this book serve not only as a guide to C Programming but also as a reminder that persistence, curiosity, and lifelong learning are the true keys to success in the world of technology.

To every learner who reads this book: may your errors become lessons, your challenges become opportunities, and your code become a reflection of your creativity, logic, and determination.

The semicolon may mark the end of a statement, but learning never ends.

Keep exploring. Keep creating. Keep coding ...