Structure Programming

COURSE CODE: CSE-0611-1103

Prepared By,
Md Zahid Akon
Lecturer
Department of CSE

CLO'S

01 Understand the Fundamental Concepts of C Programming

02 Apply Control Structures in Program Development

03 Develop Modular Programs Using Functions and Array

04 Utilize Pointers and Manage Memory Dynamically

05 Implement Structures and Unions for Complex Data

06 Handle Files for Data Storage and Retrieval

Summary of Course Content:

Sl no	Topics	Hours	CLOs
1	Introduction to C Programming: Syntax, Data Types, and Operators	3	CLO1
2	Input/Output Operations, Basic Programs	3	CLO1
3	Control Flow: If-Else, Switch, Loops (For, While, Do-While)	4	CLO1, CLO2
4	Functions: User-defined, Recursion, and Storage Classes	4	CLO2, CLO3
5	Arrays: Single-Dimensional and Multi-Dimensional	3	CLO3

Summary of Course Content:

Sl no	Topics	Hours	CLOs
6	Strings and String Handling Functions	3	CLO3
7	Pointers: Basics, Pointer Arithmetic, and Dynamic Memory Allocation	4	CLO4
8	Structures, Enumerations, and Unions	4	CLO5
9	File Handling: Read, Write, Append Modes	4	CLO6
10	Debugging and Error Handling	3	CLO1-CLO6

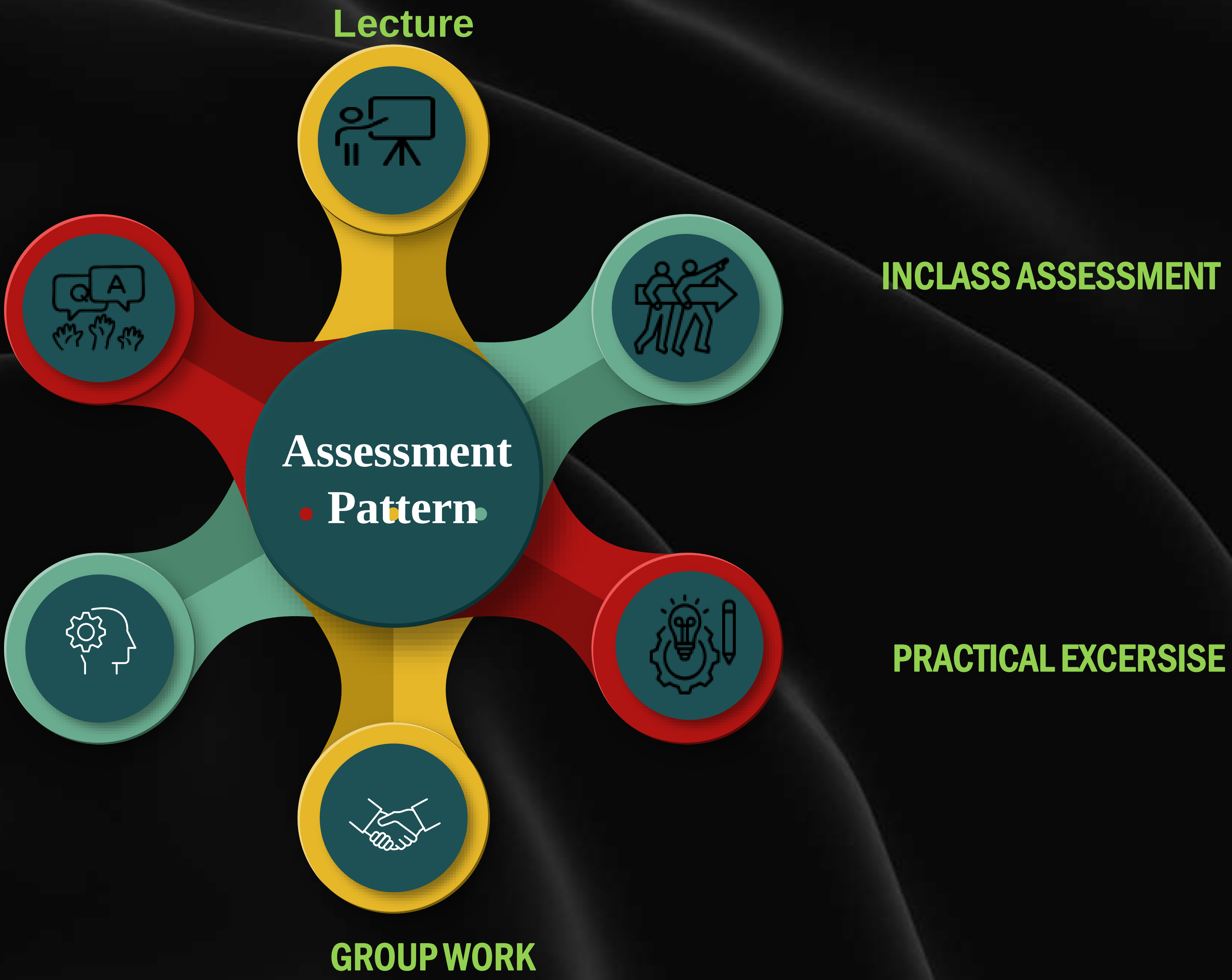
Recommended Books

PROGRAMMING IN ANSI C

1.E. Balagurusamy, Tata McGraw-Hill (ISBN: 9781259004612)

THE C PROGRAMMING LANGUAGE

1.B.W. Kernighan and D.M. Ritchie, 2nd Edition, Prentice Hall (ISBN: 9780131103627)


Lecture
Q&A
INCLASS ASSESSMENT
Assessment Pattern
BRAIN STROMING
PRACTICAL EXCERSISE
GROUP WORK

Course Plan

Week	Topics	Teaching Strategy(s)	Assessment Strategy(s)	Alignment to CLO
1	Introduction to C Programming, Data Types, Operators	Lecture, Q&A, In-class Examples	Participation, Quiz	CLO1
2	Input/Output Operations, Basic Programs	Lecture ,Practical Exercises, Group Discussions	Short Assignment, Quiz	CLO1
3	Control Flow Statements: If-Else, Switch	Lecture ,Live Coding, Problem-Solving	Quiz, Problem-solving Tasks	CLO1, CLO2
4	Loops	Lecture ,Live Coding, Problem-Solving	Quiz, Problem-solving Tasks	CLO1, CLO2
5	Functions: User-defined, Recursion	Interactive Lectures, Case Studies	Group Task, Quiz	CLO2, CLO3
6	Arrays: Single-Dimensional, Multi-Dimensional	Lecture ,Exercises, Real-world Examples	Quiz, Assignment	CLO3

Course Plan

Week	Topics	Teaching Strategy(s)	Assessment Strategy(s)	Alignment to CLO
7	Strings	Lecture ,Code Demonstrations, Hands-on Tasks	Assignment, Group Discussion	CLO3
8	String Handling Functions	Lecture ,Code Demonstrations, Practical Labs	Quiz, Assignment	CLO3
9	Pointers: Basics, Arithmetic,	Lecture, Practical Lab Sessions	Performance Evaluation, Quiz	CLO4
10	Dynamic Memory Allocation	Lecture, Practical Lab Sessions	Performance Evaluation, Quiz	CLO4
11	Structures, Enumerations, and Unions	Lecture, Group Activities	Group Task, Assignment	CLO5
12	File Handling: Read, Write, Append Modes	Lab Demonstrations, Code Reviews	Lab Report, Quiz	CLO6

Course Plan

Week	Topics	Teaching Strategy(s)	Assessment Strategy(s)	Alignment to CLO
13	Debugging and Error Handling	Revision, Practical Debugging Challenges	In-class Practice, Final Exam	CLO1-CLO6
14	Advanced Functions and Macros	Lecture, Practical Exercises	Quiz, Assignment	CLO2, CLO3
15	Advanced Pointer Techniques	Lecture, Hands-on Practice	Quiz, Problem-Solving Task	CLO4
16	Project Integration and Review	Project-based Learning, Group Work	Project Evaluation	CLO1-CLO6
17	Final Exam and Project Submission	Written Exam, Project Presentation	Final Exam, Project Grading	CLO1-CLO6

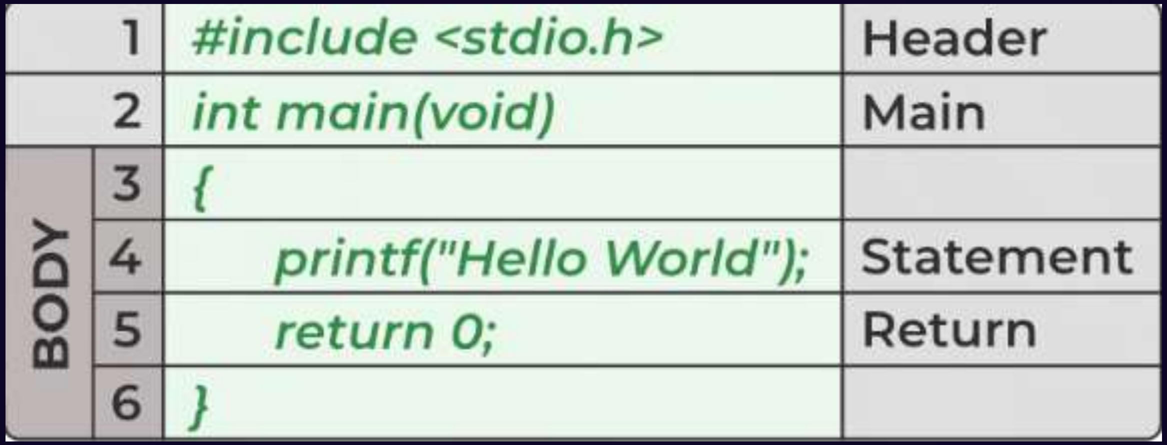
Week 1

Chapter 1

Introduction to C Programming, Data Types, Operators

Structured programming

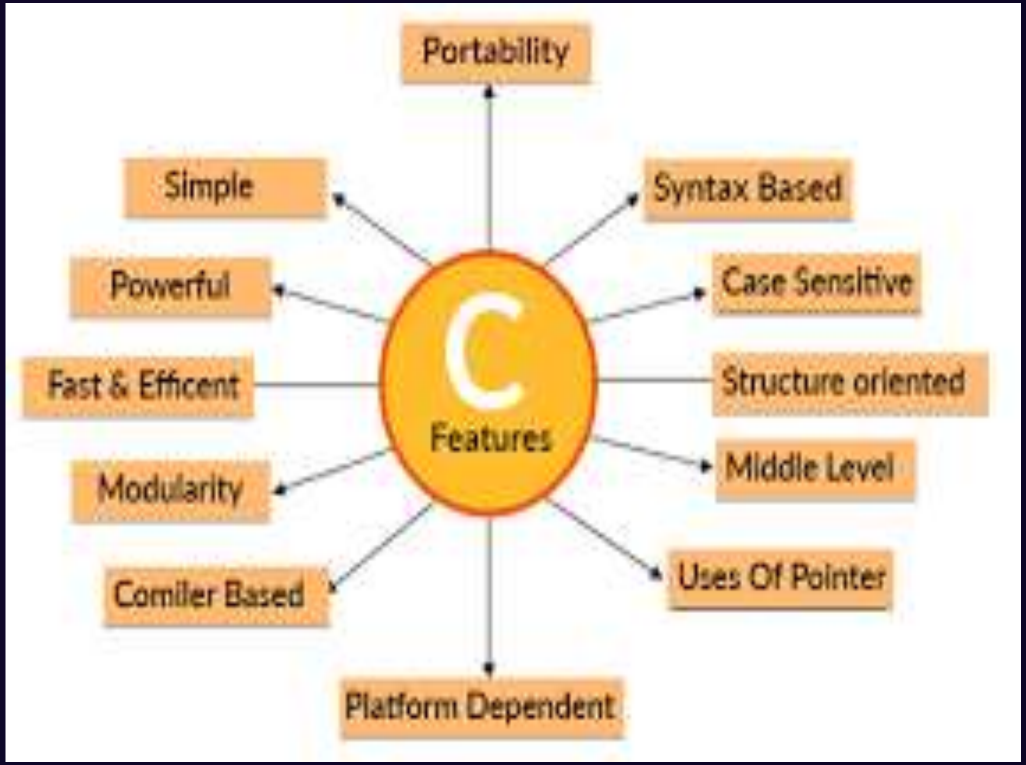
Structured programming (sometimes known as *modular programming*) is a subset of procedural programming that enforces a logical structure on the program being written to make it more efficient and easier to understand and modify. Certain languages such as Ada, Pascal, and dBASE are designed with features that encourage or enforce a logical program structure.



	1	`#include <stdio.h>`	Header
	2	`int main(void)`	Main
BODY	3	`{`	
	4	`    printf("Hello World");`	Statement
	5	`    return 0;`	Return
	6	`}`	

Why is C called a structured programming language?

C is called a structured programming language because to solve a large problem, C programming language divides the problem into smaller modules called functions or procedures each of which handles a particular responsibility. The program which solves the entire problem is a collection of such functions.



My First Program

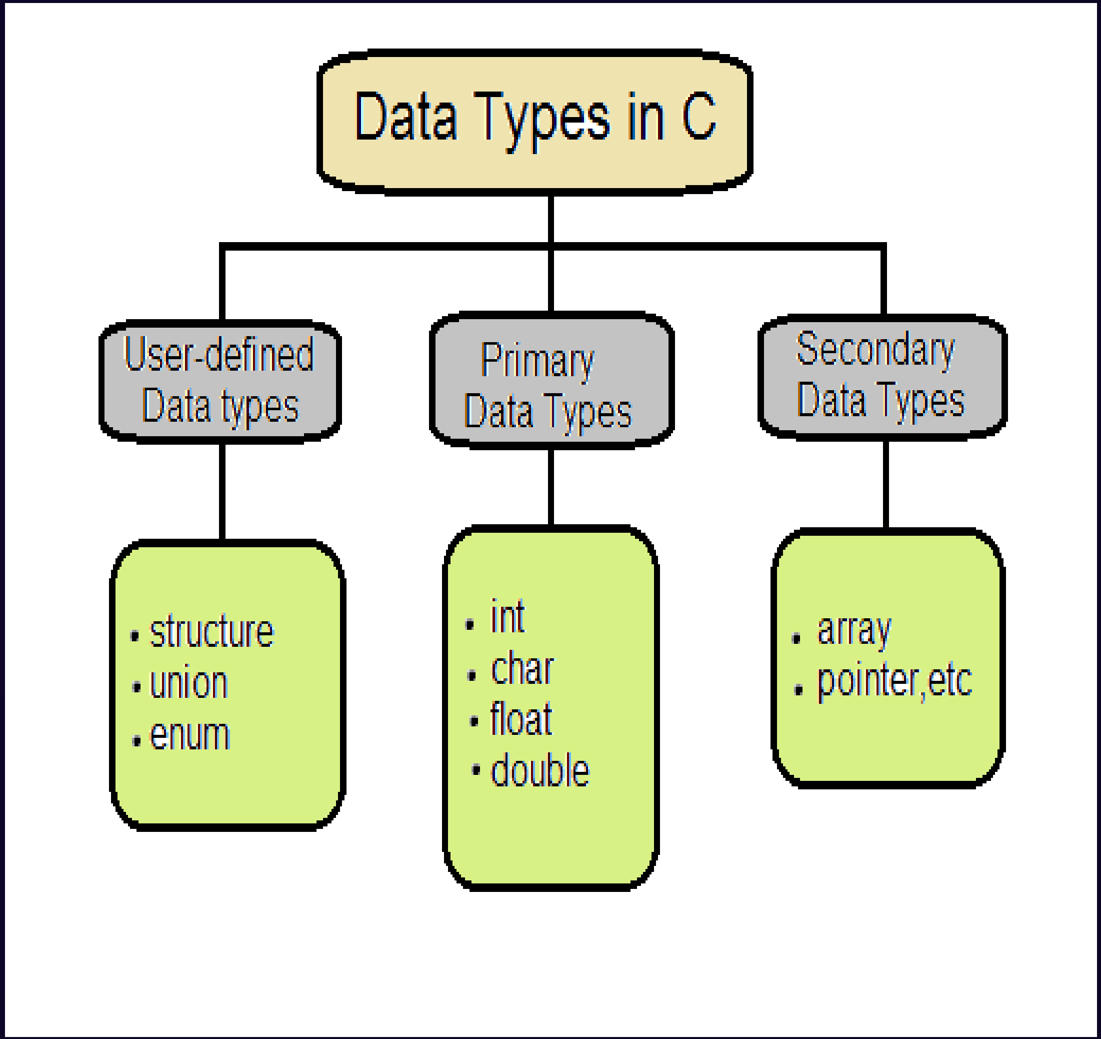
```c
#include <stdio.h>

int main() {

    printf("Hello, World!");

    return 0;
}
```

```
Hello World!
```

Data Types in C

A data type is an attribute associated with a piece of data that tells a computer system how to interpret its value. Understanding data types ensures that data is collected in the preferred format and that the value of each property is as expected

- int: Stores integers (e.g., int age = 25;)
- float: Stores decimal numbers (e.g., float price = 12.99;)
- double: Stores large decimals (e.g., double pi = 3.141592;)
- char: Stores single characters (e.g., char grade = 'A';)



Format specifiers in C

Format specifiers are used in functions like printf and scanf to input or output values of different data types.

Specifier	Data Type	Example
%d	Integer (decimal)	printf("%d", 10);
%f	Floating-point	printf("%.2f", 3.14);
%c	Character	printf("%c", 'A');
%s	String	printf("%s", "Hello");
%lf	Double	printf("%lf", 3.14159);
%x	Hexadecimal integer	printf("%x", 255);
%o	Octal integer	printf("%o", 8);

Variables in C

- A **variable** is a named memory location used to store data.
- Its value can change during program execution.

data_type variable_name = value; // defining single variable

or

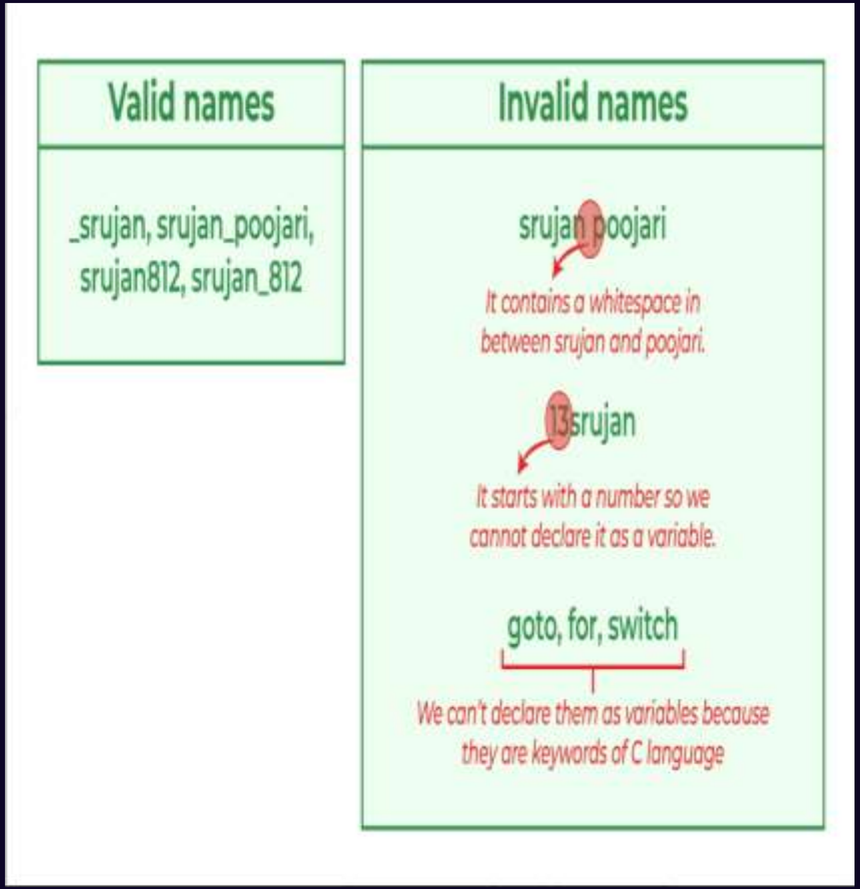
data_type variable_name1, variable_name2; // defining multiple variable

- **data_type:** Type of data that a variable can store.
- **variable_name:** Name of the variable given by the user.
- **value:** value assigned to the variable by the user.

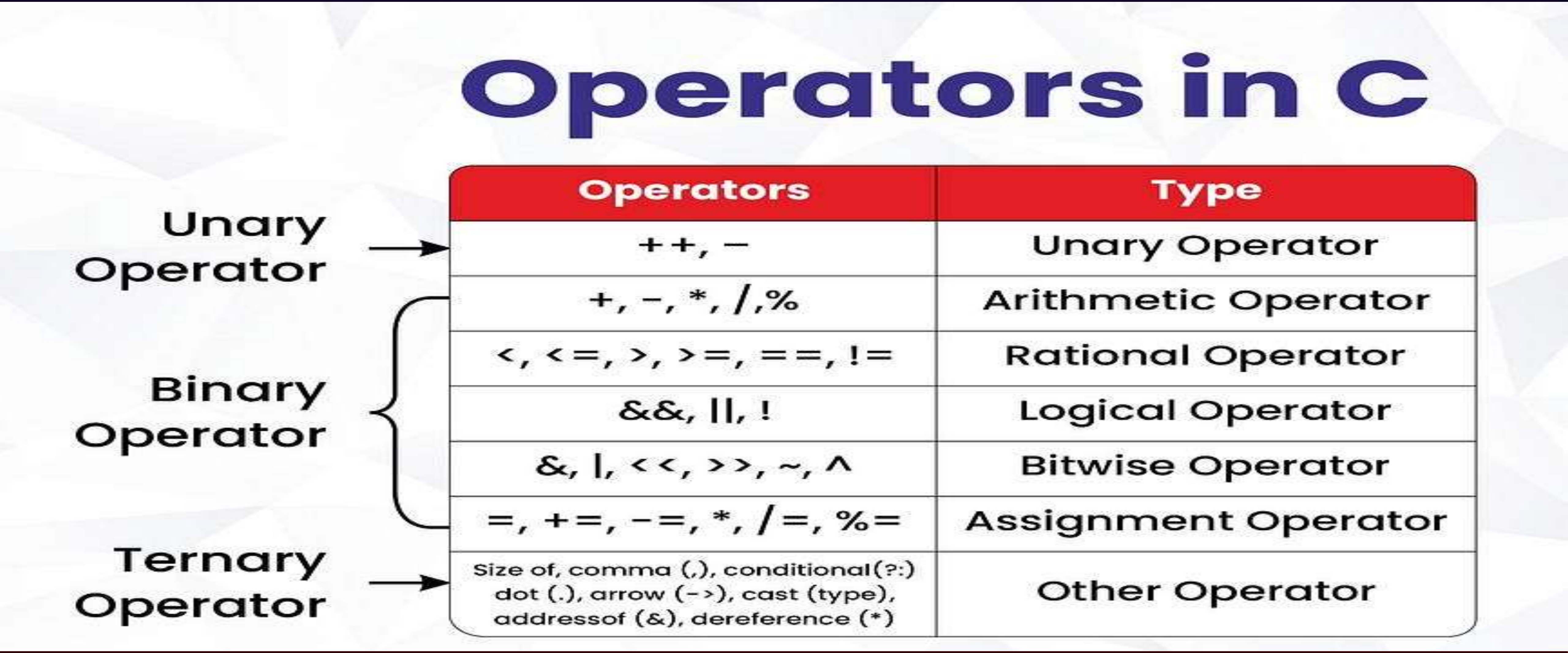
```
int a=10;          //Variable initialization
a=10;               //assignment
```

Identifiers In C

- An identifier can include letters (a-z or A-Z), and digits (0-9).
- An identifier cannot include special characters except the '_' underscore.
- Spaces are not allowed while naming an identifier.
- An identifier can only begin with an underscore or letters.
- We cannot name identifiers the same as keywords because they are reserved words to perform a specific task. For example, printf, scanf, int, char, struct, etc. If we use a keyword's name as an identifier the compiler will throw an error.
- The identifier must be **unique** in its namespace.
- C language is case-sensitive so, 'name' and 'NAME' are different identifiers.


Valid names
_srujan, srujan_poojari, srujan812, srujan_812
Invalid names
srujan poojari
It contains a whitespace in between srujan and poojari.
13srujan
It starts with a number so we cannot declare it as a variable.
goto, for, switch
We can't declare them as variables because they are keywords of C language

Operators


Unary Operator
Binary Operator
Ternary Operator

Operators in C

Operators	Type
++, –	Unary Operator
+, –, *, /,%	Arithmetic Operator
<, <=, >, >=, ==, !=	Rational Operator
&&, ||, !	Logical Operator
&, |, <<, >>, ~, ^	Bitwise Operator
=, +=, –=, *, /=, %=	Assignment Operator
Size of, comma (,), conditional(?:) dot (.), arrow (->), cast (type), addressof (&), dereference (*)	Other Operator

Week 2

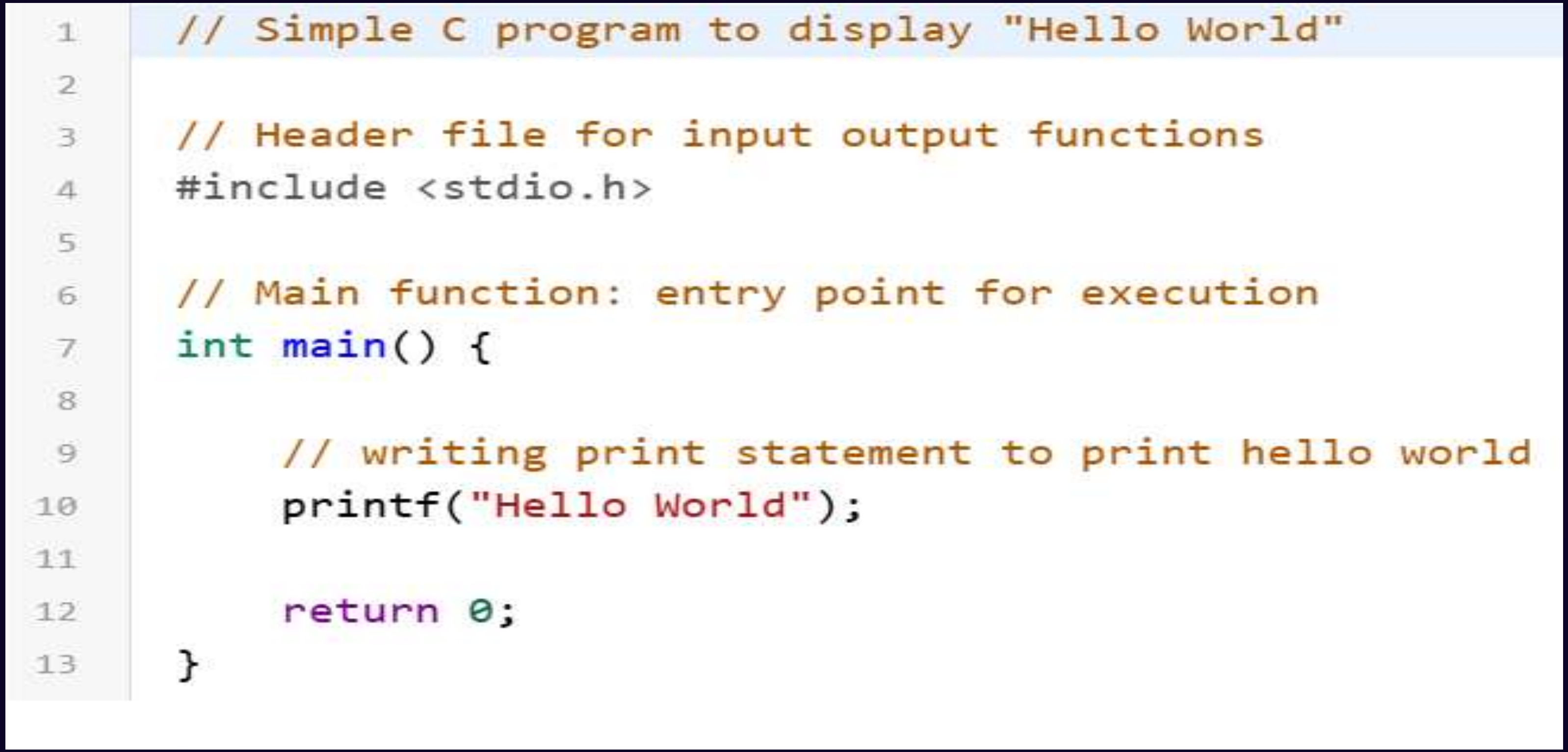
Input/Output Operations, Basic Programs

Basic Input and Output in C

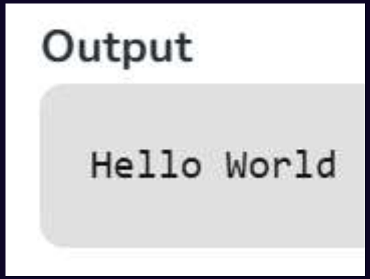
C language has standard libraries that allow input and output in a program. The **stdio.h** or **standard input output library** in C that has methods for input and output

scanf()	printf()
The scanf() method, in C, reads the value from the console as per the type specified and store it in the given address. Syntax: **scanf("%X", &variableOfXType);**	The printf() method, in C, prints the value passed as the parameter to it, on the console screen. Syntax: **printf("%X", variableOfXType);**

Simple C program to display "Hello World"



```c
// Simple C program to display "Hello World"

// Header file for input output functions
#include <stdio.h>

// Main function: entry point for execution
int main() {

    // writing print statement to print hello world
    printf("Hello World");

    return 0;
}
```



Output

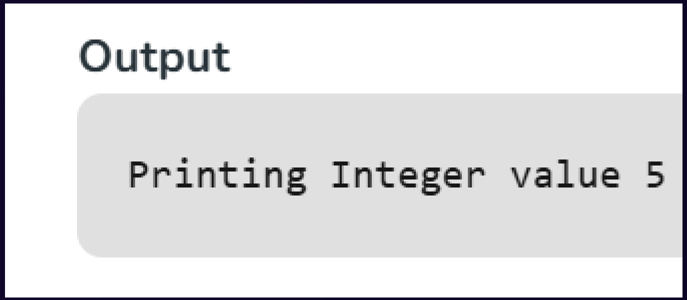
```
Hello World
```

Explanation:

- **#include <stdio.h>** – This line includes the standard input-output library in the program.
- **int main()** – The main function where the execution of the program begins.
- **printf(“Hello, World!\n”);** – This function call prints “Hello, World!” followed by a new line.
- **return 0;** -This statement indicates that the program ended successfully.

Print an Integer Value in C

```
#include <stdio.h>

// Driver code
int main()
{
    // Declaring integer
    int x = 5;

    // Printing values
    printf("Printing Integer value %d", x);
    return 0;
}
```

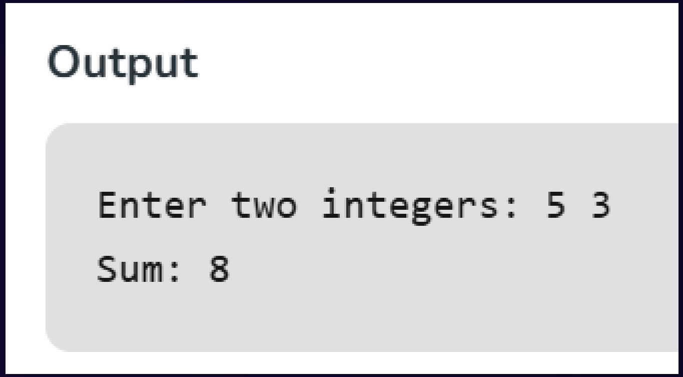


Output

```
Printing Integer value 5
```

C program to add two numbers

```
#include <stdio.h>

int main() {
    int a, b, sum = 0;

      // Read two numbers from the user
    printf("Enter two integers: ");
    scanf("%d %d", &a, &b);

    // Calculate the addition of a and b
    // using '+' operator
    sum = a + b;

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

    return 0;
}
```

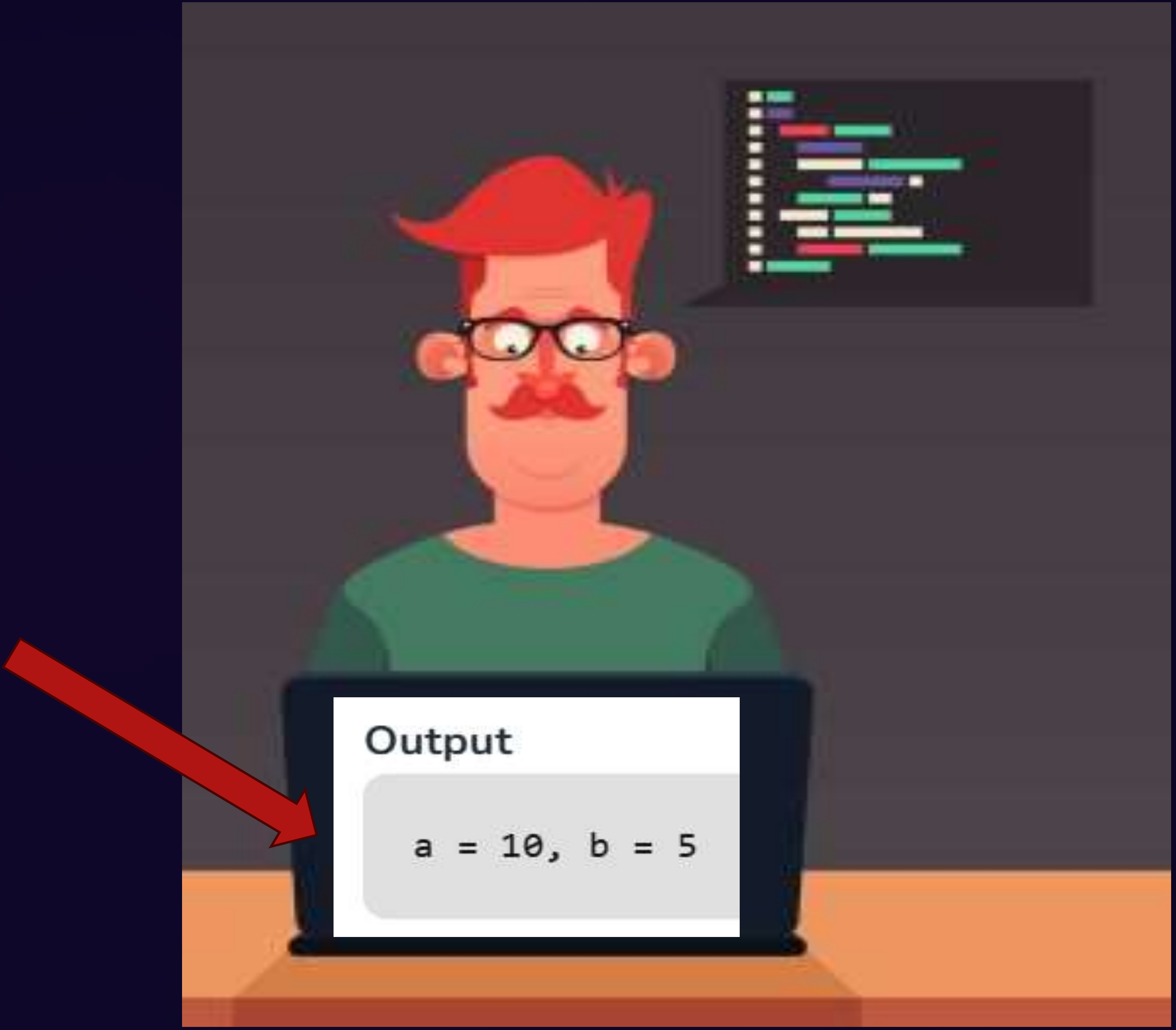


Output

```
Enter two integers: 5 3
Sum: 8
```

C Program to Swap Two Numbers

```
#include <stdio.h>
int main() {
    int a = 5, b = 10, temp;

    // Swapping values of a and  b
    temp = a;
    a = b;
    b = temp;

    printf("a = %d, b = %d\n", a, b);
    return 0;
}
```



```
a = 10, b = 5
```

C Program to Swap Two Numbers WithOut using Third Variables.

Week 3

Chapter 3

Conditional Statements in C

Unlock the Power of Decision-Making in Your C Programs

Introduction to Conditional Statements

Controlling Program Flow

Conditional statements dictate which code block executes based on specific conditions

Decision-Making Power

Allow your programs to react to different inputs, scenarios, and events

The If Statement: Syntax and Examples

Basic Syntax

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

Example

```
#include
int main() {
  int num = 10;
  if (num > 5) {
    printf("Number is greater than 5\n");
  }
  return 0;
}
```

The Else If Statement: Nested Conditional Logic

Adding Another Condition

```
if (condition1) {
  // Code to execute if condition1 is true
} else if (condition2) {
  // Code to execute if condition2 is true
}
```

Example

```
#include
int main() {
  int num = 7;
  if (num > 10) {
    printf("Number is greater than 10\n");
  } else if (num > 5) {
    printf("Number is greater than 5\n");
  }
  return 0;
}
```

The Else If Ladder: Chaining Multiple Conditions

Chain of Checks

```
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
} ...
```

Example

```
#include
int main() {
  int grade = 85;
  if (grade >= 90) {
    printf("A\n");
  } else if (grade >= 80) {
    printf("B\n");
  } else if (grade >= 70) {
    printf("C\n");
  } else {
    printf("D\n");
  }
  return 0;
}
```

The Switch Statement: Handling Multiple Cases

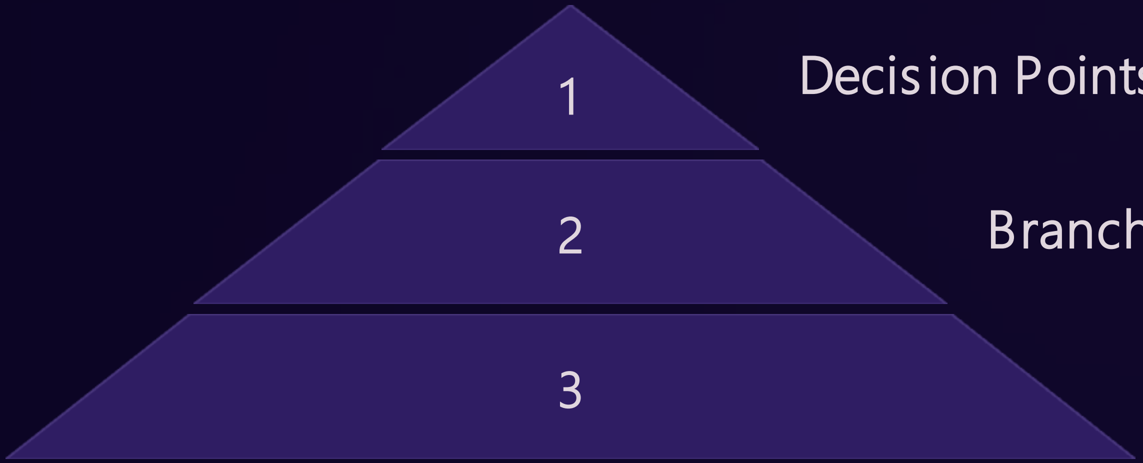
Multiple Case Choices

```
switch (expression) {
  case value1:
    // Code to execute for value1
    break;
  case value2:
    // Code to execute for value2
    break;
  default:
    // Code to execute if no case matches
}
```

Example

```
#include
int main() {
  char grade = 'B';
  switch (grade) {
    case 'A':
      printf("Excellent!\n");
      break;
    case 'B':
      printf("Good!\n");
      break;
    case 'C':
      printf("Fair!\n");
      break;
    default:
      printf("Invalid grade!\n");
  }
  return 0;
}
```

Graphical Representation of Conditional Statements



Decision Points

Branching Paths

Code Execution

Conditional Statement Examples with Code and Output

Example 1

```
if (age >= 18) {
  printf("You are eligible to vote\n");
} else {
  printf("You are not eligible to vote\n");
}
```

```
Output: You are eligible to vote
```

Example 2

```
switch (day) {
  case 1:
    printf("Monday\n");
    break;
  case 2:
    printf("Tuesday\n");
    break;
  default:
    printf("Invalid day\n");
}
```

```
Output: Monday
```

Choosing the Appropriate Conditional Statement

1 If/Else If Ladder

For a sequence of related conditions

2 Switch Statement

For handling specific cases of a single variable

3 Nested If/Else If

For complex conditions with multiple levels of logic

Best Practices and Tips for Conditional Statements

Clear Logic

Make your conditions easy to understand and follow

Proper Indentation

Improves readability and maintains code structure

Use Comments

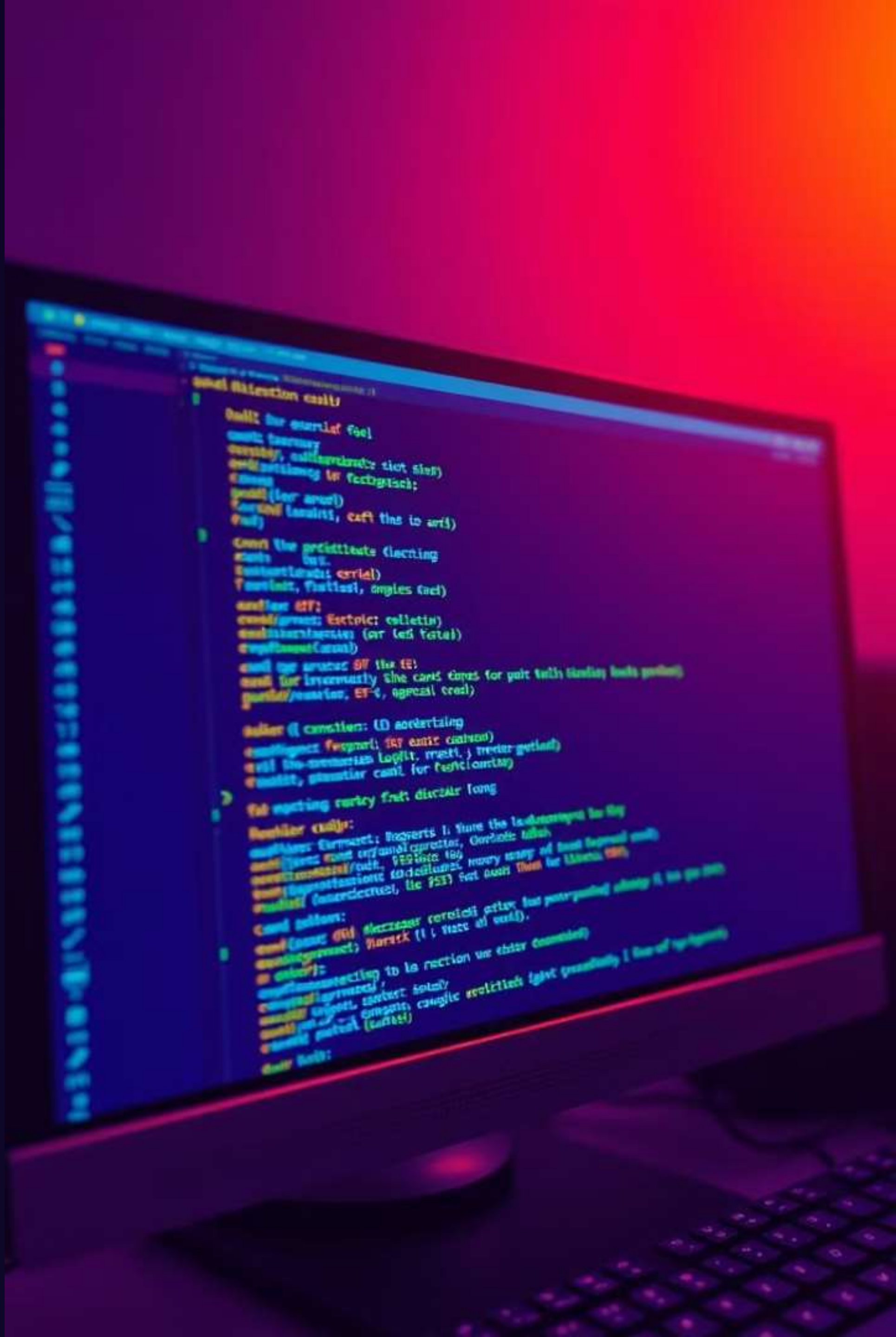
Explain the purpose and functionality of your code

Week 4

Chapter 4

Loop

Dive into the world of loops in C programming, exploring their syntax, structure, and applications.



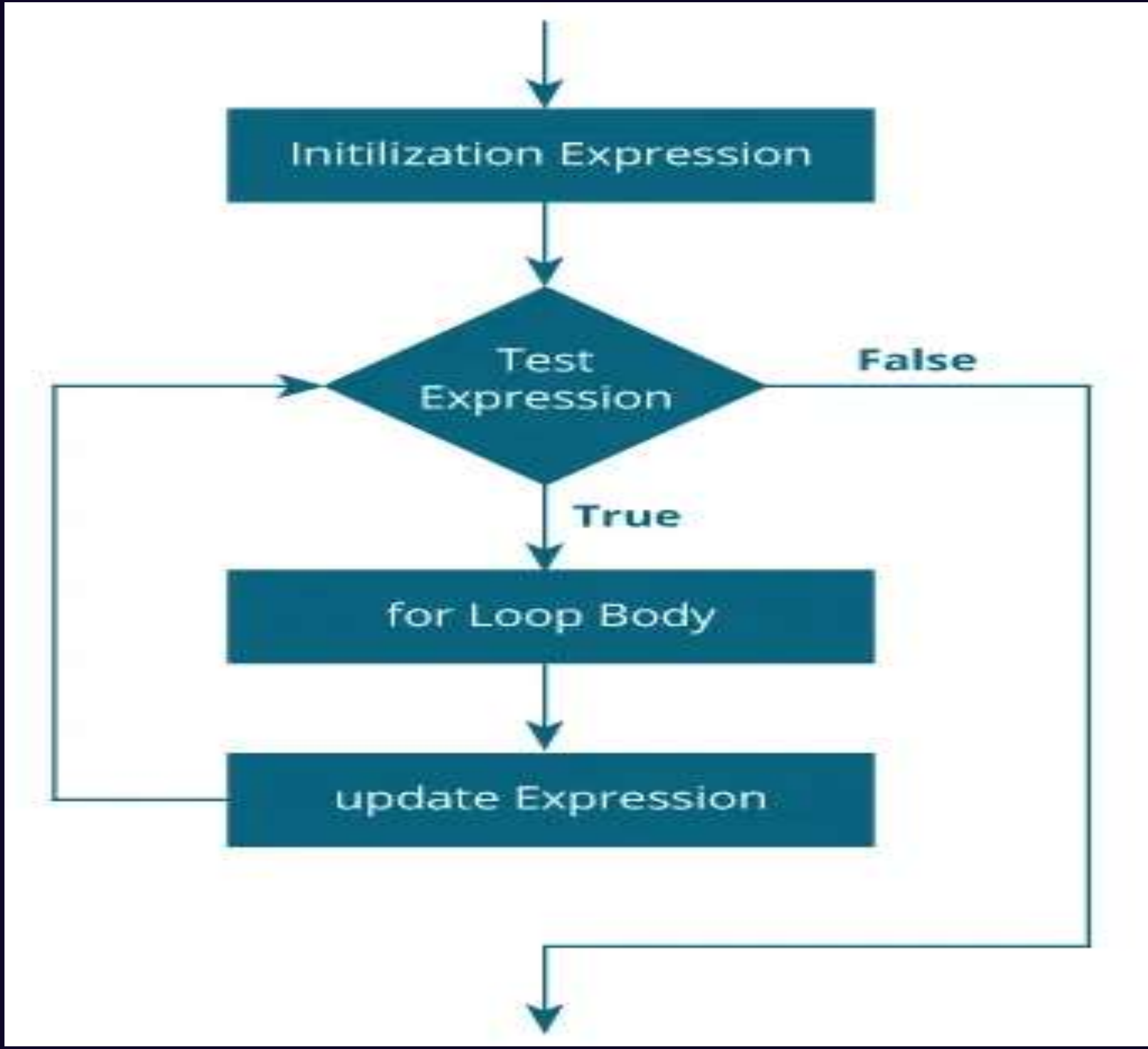
Introduction to Loops in C

Repetitive Tasks

Loops automate the execution of a block of code multiple times, simplifying repetitive tasks.

Iteration Control

They allow you to control the number of iterations, ensuring precise execution based on conditions.



The for Loop: Syntax and Structure

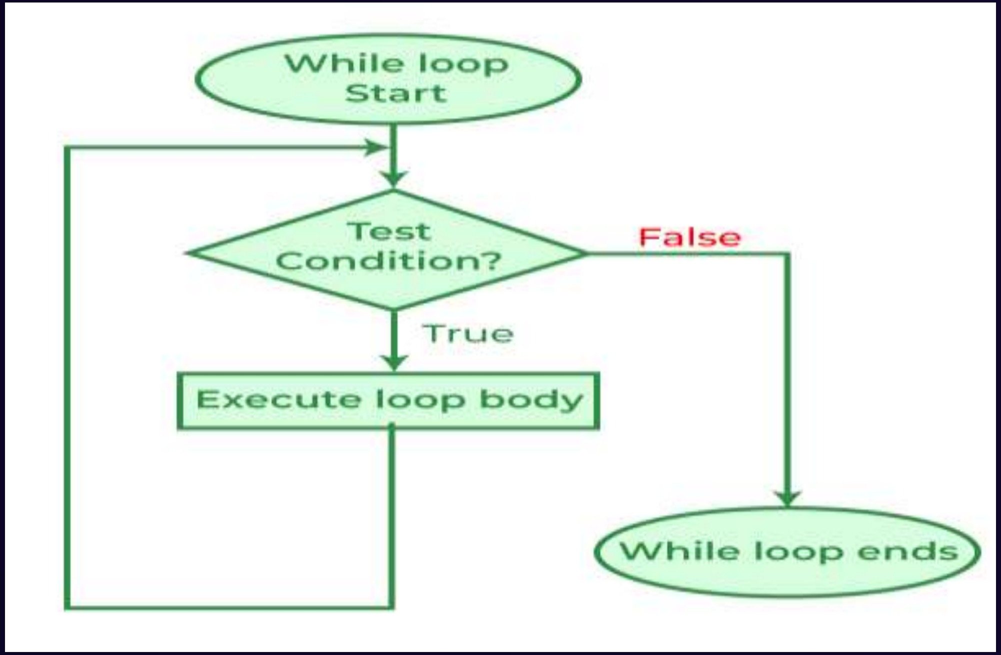
```
for (initialization; condition; increment) {
   // Code to be executed
}
```

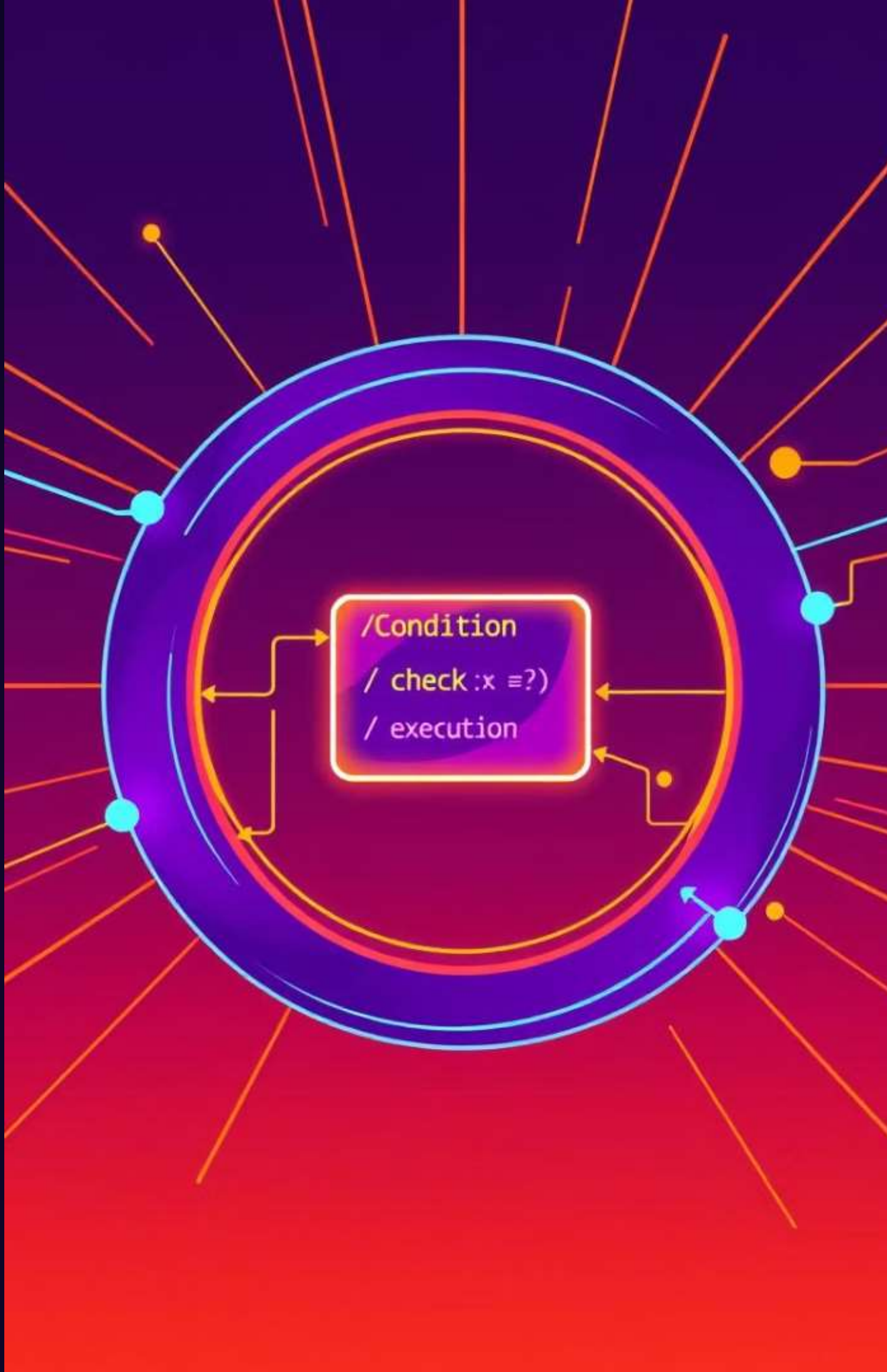
```c
// Print numbers from 1 to 10
#include <stdio.h>

int main() {
  int i;

  for (i = 1; i < 11; ++i)
  {
    printf("%d ", i);
  }
  return 0;
}
```

```
1 2 3 4 5 6 7 8 9 10
```

The while Loop: Conditional Execution

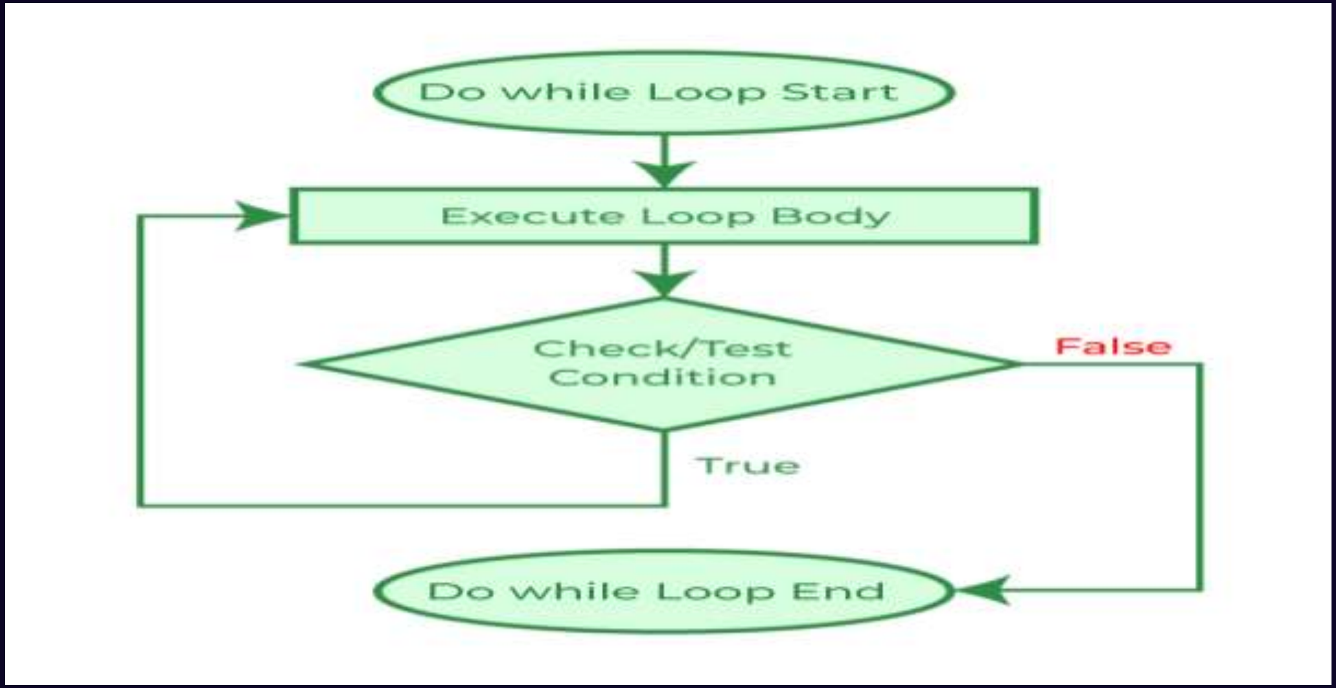
```
while (condition) {
    // Code to be executed
}
```





The do-while Loop: Post-Condition Checking

```
do {
   // Code to be executed
} while (condition);
```

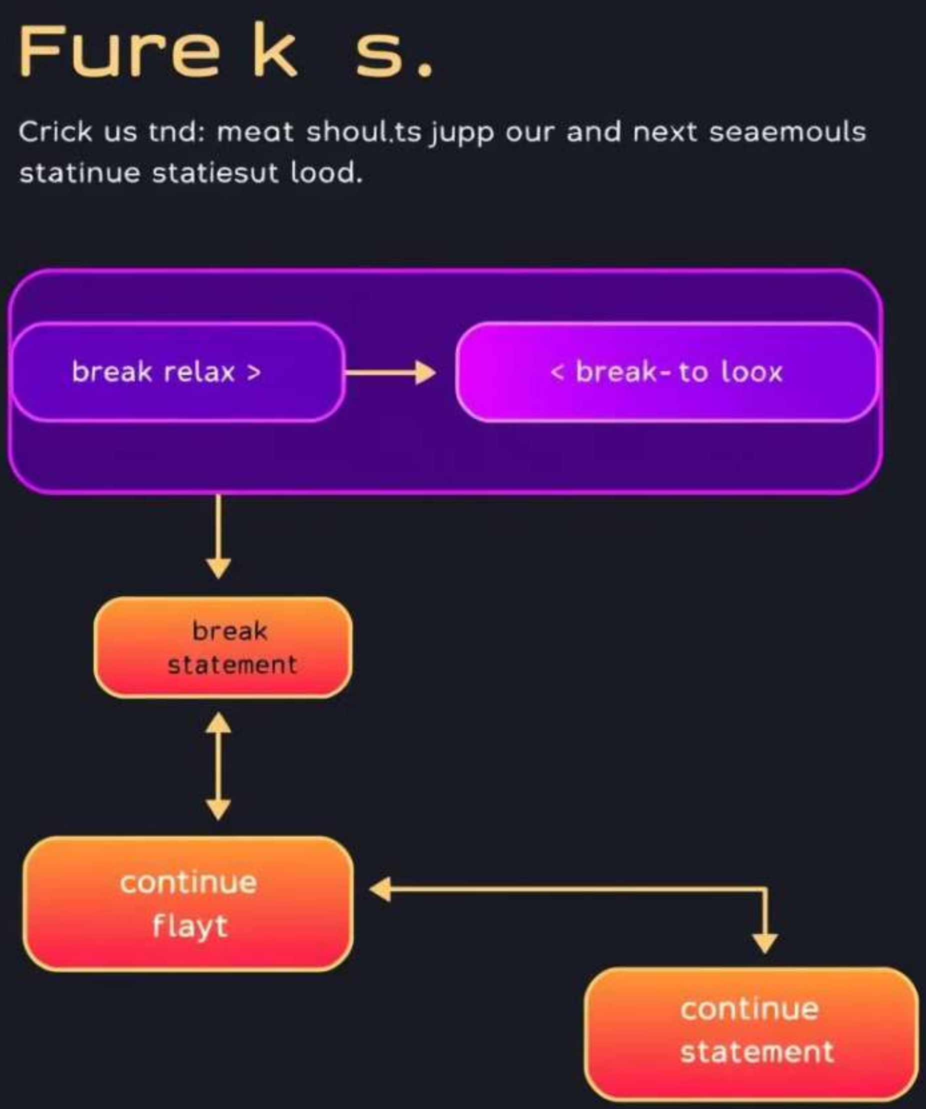


Nested Loops: Combining Loops

```
for (int i = 0; i < 5; i++) {
    for (int j = 0; j < 3; j++) {
        // Code to be executed
    }
}
```

Loop Control Statements: break and continue

```
for (int i = 0; i < 10; i++) {
    if (i == 5) {
        break; // Exit the loop
    } else if (i % 2 == 0) {
        continue; // Skip to the next iteration
    }
    // Code to be executed
}
```



Graphical Representation of Loop Execution

Visualize loop execution through animations and diagrams, understanding how loops work and the flow of control.

Example Code: Printing a Star Pattern

```c
#include

int main() {
    for (int i = 0; i < 5; i++) {
        for (int j = 0; j <= i; j++) {
            printf("*");
        }
        printf("\n");
    }
    return 0;
}
```

*code: T1GE
*utpe: fangear sttar" coretook;
*utput SAAR IT tlerting;
*code: Star IT ther ictey' singear singe;;

Output Demonstration: Visualizing the Loop

See the output in action, demonstrating the pattern created by the loop and how it iterates to produce the desired result.

Week 5

Chapter 5

Functions: User-defined, Recursion

This presentation dives into the world of functions in C programming, exploring user-defined functions and the powerful concept of recursion.

What are Functions?

Modular Code

Break down a program into smaller, reusable units

Code Organization

Improve readability and maintainability

Syntax and Structure of User-defined Functions

```
return_type function_name(parameter_list) {
    // Function body
    // Statements to perform
    return value;
}
```

Defining and Calling Functions

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

int main() {
  int sum = add(5, 3);
  printf("Sum: %d\n", sum);
  return 0;
}
```

1 **1. Definition**

Specifies code and return type

2 **2. Call**

Executes the function's code

Defining and Calling Functions

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

int main() {
  int sum = add(5, 3);
  printf("Sum: %d\n", sum);
  return 0;
}
```

1

1. Definition

Specifies code and return type

2

2. Call

Executes the function's code

Function Parameters and Arguments

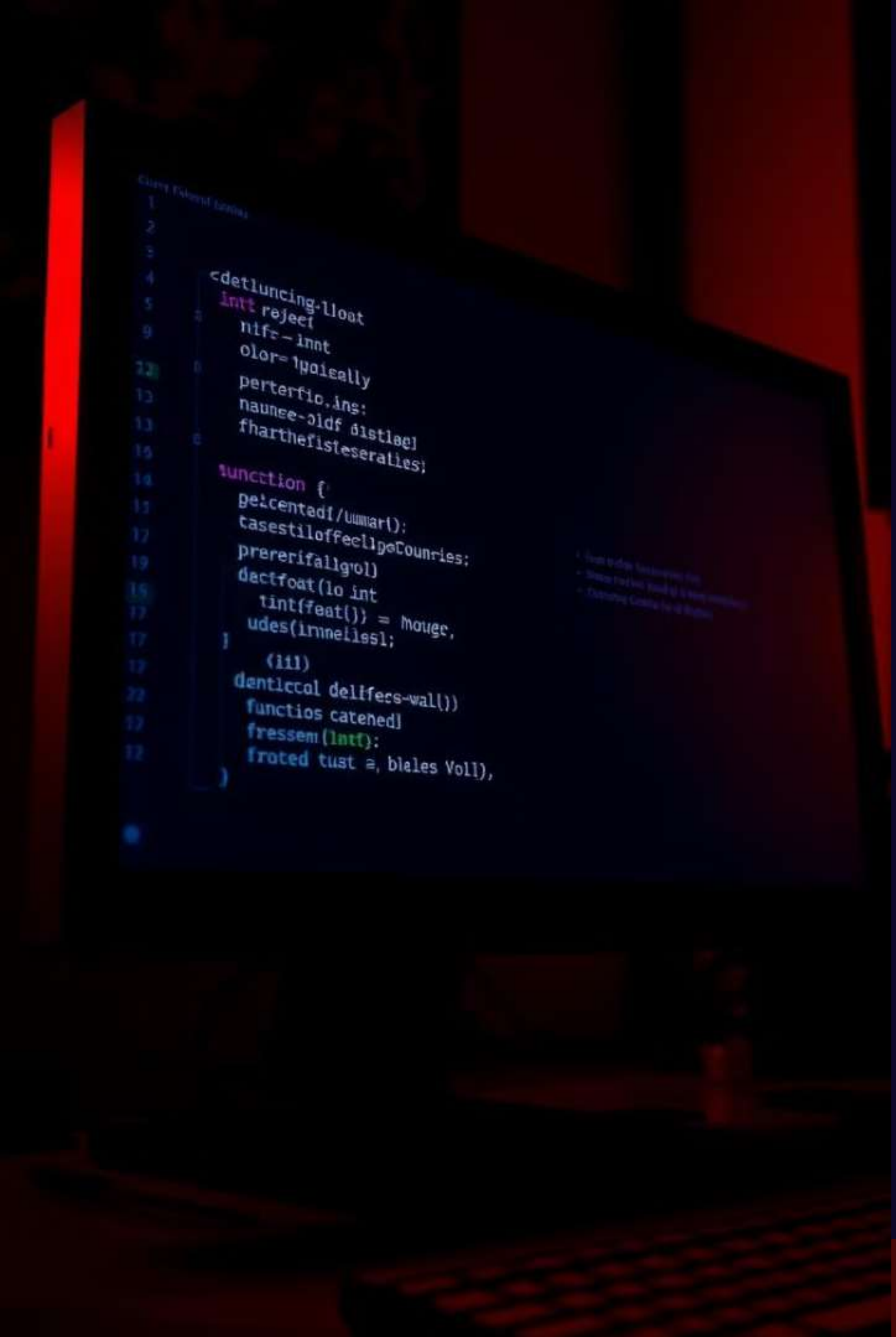
```c
int multiply(int a, int b) {
  return a * b;
}

int main() {
  int product = multiply(5, 3);
  printf("Product: %d\n", product);
  return 0;
}
```

Parameters

Variables in function definition

Arguments

Values passed during function call



Function Return Types

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

float average(int a, int b) {
  return (float) (a + b) / 2;
}

void printMessage() {
  printf("Hello from function!\n");
}
```

int

Returns an integer

float

Returns a floating-point number

void

Doesn't return a value

Returning Values from Functions

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

Passing Arrays to Functions

```c
void printArray(int arr[], int size) {
  for (int i = 0; i < size; i++) {
    printf("%d ", arr[i]);
  }
  printf("\n");
}

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

Array

Passed by reference

Pointer

Access array elements

Recursion: Functions Calling Themselves

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

int main() {
  int result = factorial(5);
  printf("Factorial of 5: %d\n", result);
  return 0;
}
```

1. **Base Case**

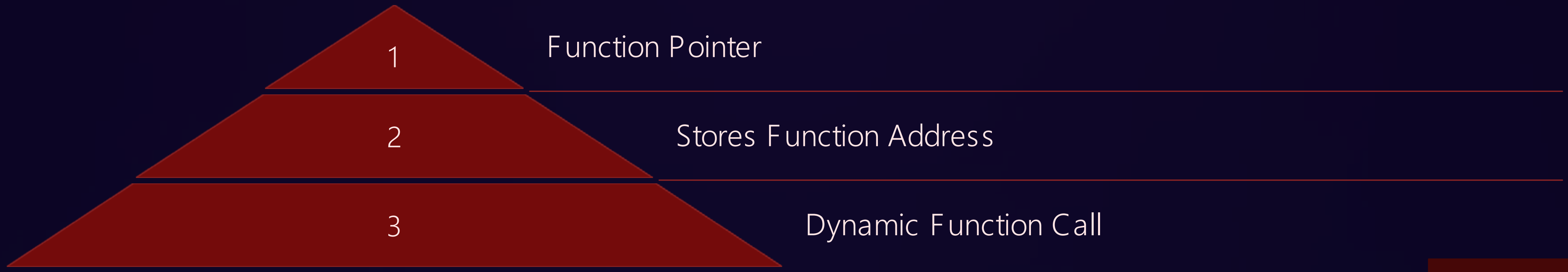
   Stops the recursion

2. **Recursive Case**

   Calls itself with a smaller input

Function Pointers

```c
void (*funcPtr)(int);

void greet(int num) {
  printf("Greetings %d times!\n", num);
}

int main() {
  funcPtr = greet;
  (*funcPtr)(3);
  return 0;
}
```



Storage Classes in Functions

```
int add(int a, int b);

int main() {
  int sum = add(5, 3);
  return 0;
}

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

1 **auto**

Local scope

2 **static**

Retains value across calls

3 **extern**

Declares variables outside of functions

4 **register**

Stores in registers

Practical Examples and Graphical Representations

1

Sorting

Bubble sort, insertion sort

2

Searching

Linear search, binary search

3

Data Structures

Linked lists, stacks, queues

Recursive Functions: Definition and Concept

Self-Calling

Functions that call themselves within their own definition

Base Case

A condition to stop the recursion

Implementing Recursion with Examples

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

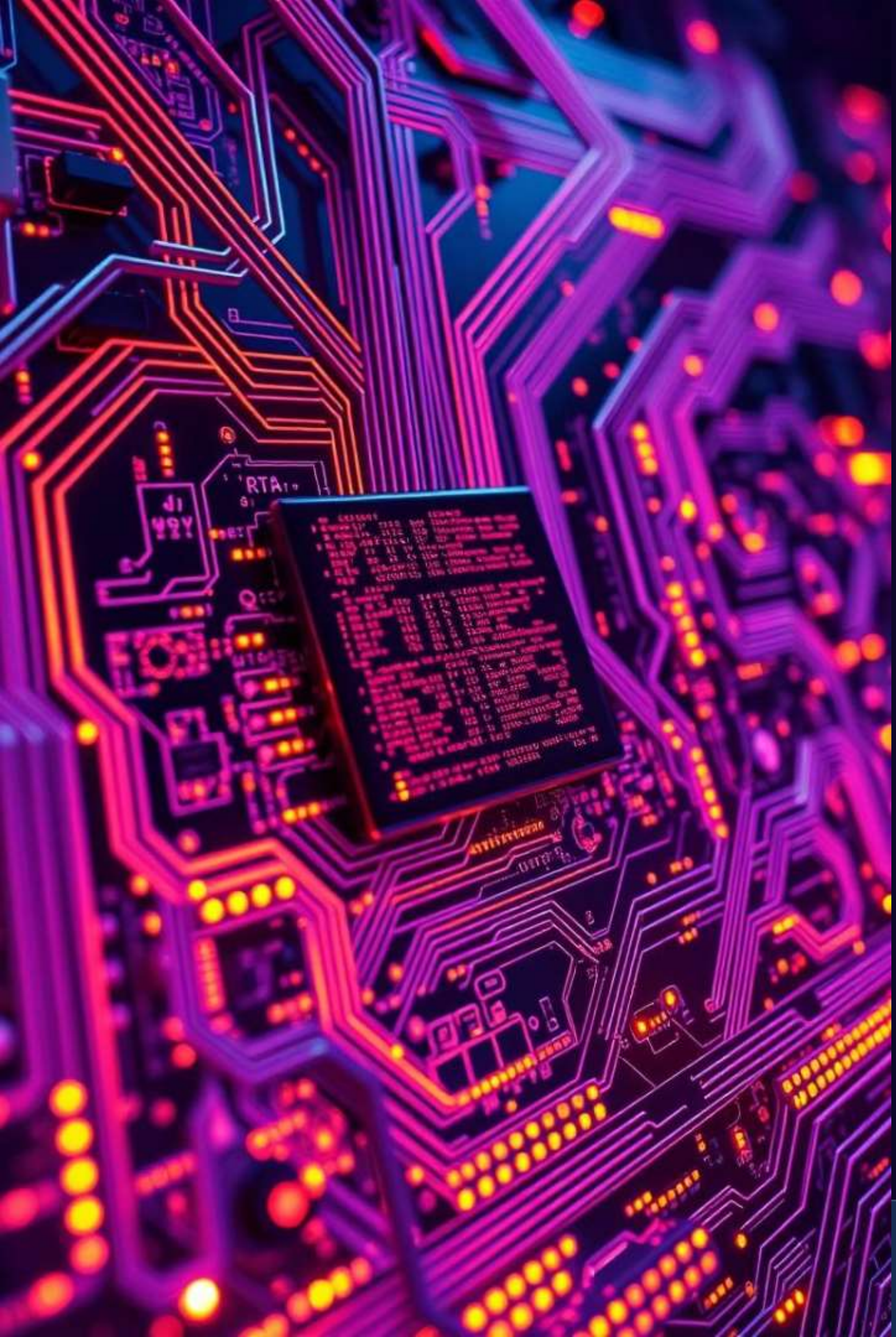
Advantages and Disadvantages of Recursion

Elegance

Provides concise and elegant solutions

Stack Overflow

Deep recursion can exhaust the stack memory



Week 6

Chapter 6

Arrays in C: A Deep Dive

Explore the world of single-dimensional and multi-dimensional arrays in C programming, uncovering their structure, manipulation, and diverse applications.

Introduction to Arrays

Organized Data

Efficiently store collections of related data of the same data type.

Memory Efficiency

Contiguous blocks of memory for compact storage and fast access.

Declaring and Initializing Single-Dimensional Arrays

Declaration

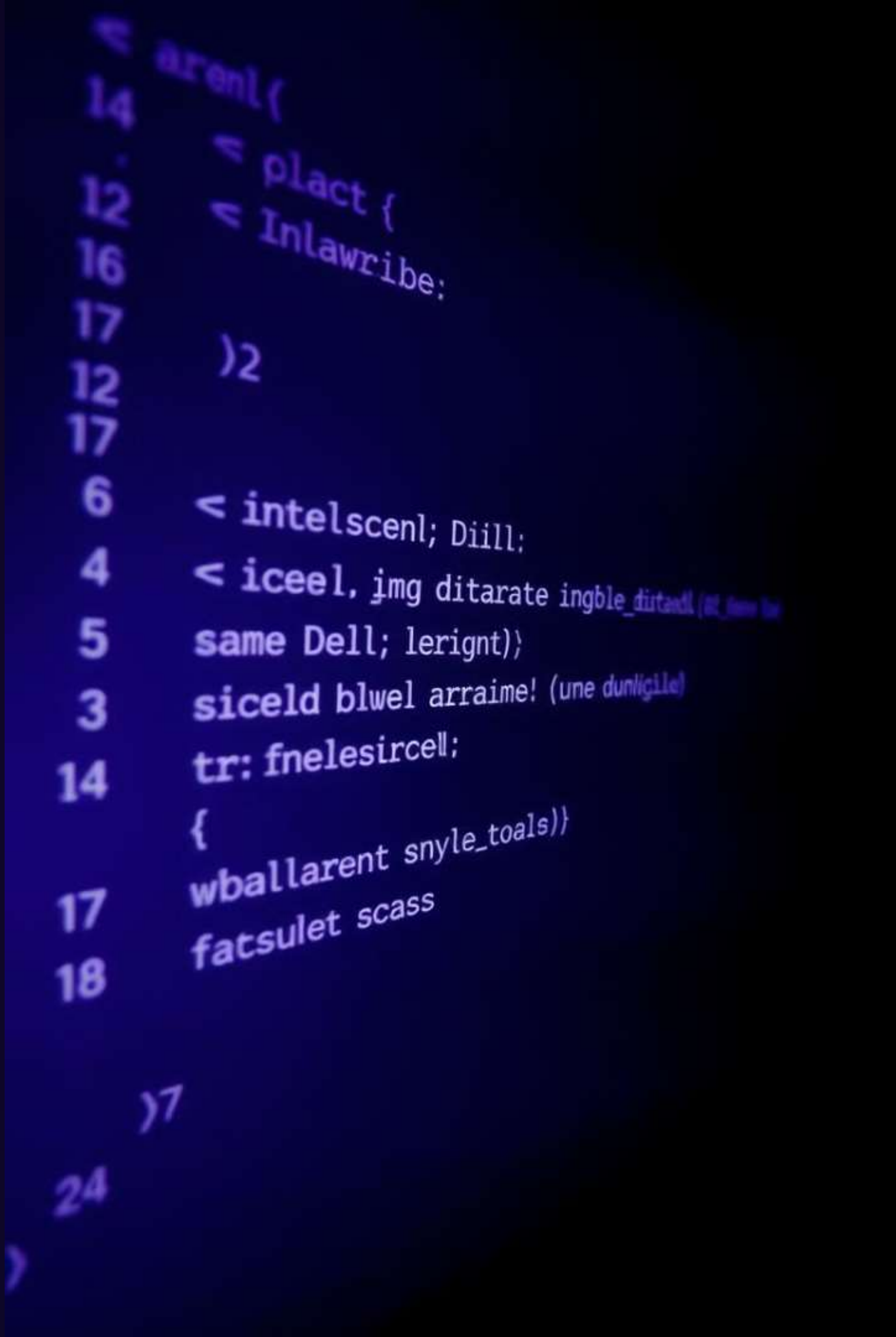
```
data_type array_name[size];
```

Initialization

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

Example

```
float temperatures[7] = {25.5, 27.2, 28.0, 26.8, 29.1, 27.5, 28.3};
```



Accessing and Manipulating Elements

1 Access

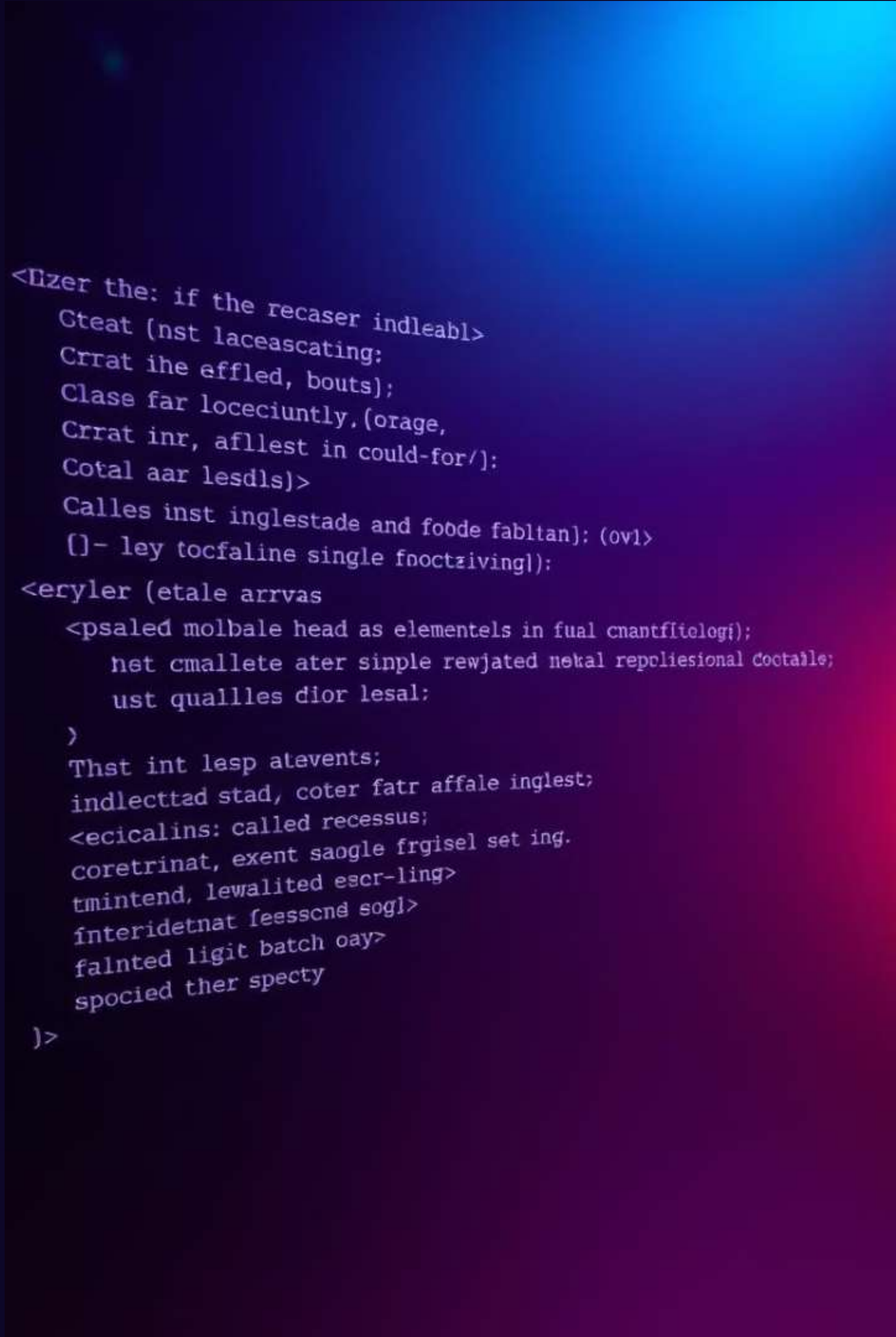
array_name[index]

2 Modification

array_name[index] = new_value;

3 Example

numbers[2] = 60; // Change element at index 2 to 60



Graphical Representation of Single-Dimensional Arrays

Memory Location

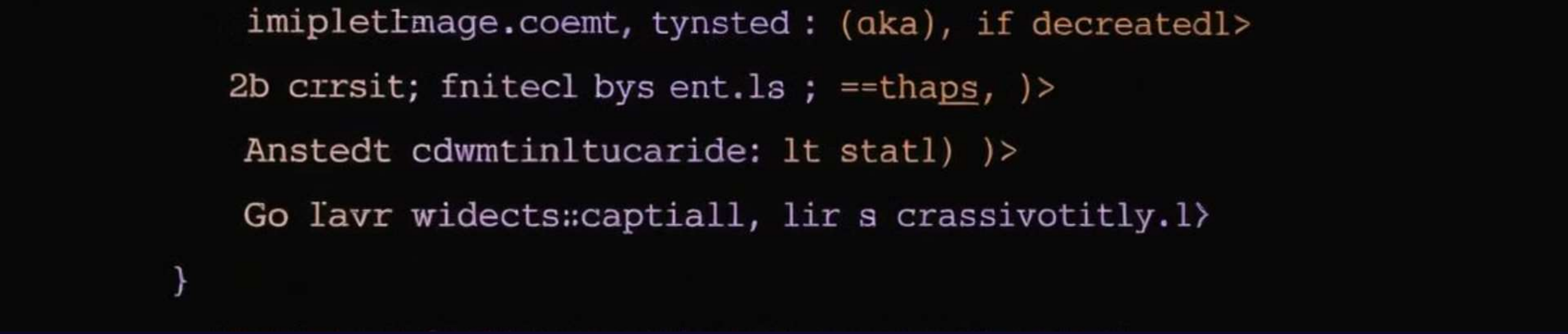
Contiguous memory blocks for efficient data storage and access.

Element Index

Each element has a unique index for easy retrieval and modification.

Data Values

Elements hold data of the same data type, allowing for uniform operations.



```
    implietImage.coemt, tynsted : (aka), if decreatedl>
   2b crrsit; fnitecl bys ent.ls ; ==thaps, )>
    Anstedt cdwmtinltucaride: lt statl) )>
    Go l'avr widects::captiall, lir s crassivotitly.l}
}
```

Declaring and Initializing Multi-Dimensional Arrays

Declaration

data_type array_name[row_size][column_size];

Initialization

int matrix[3][2] = {{1, 2}, {3, 4}, {5, 6}};

Example

char board[8][8]; // Representing a chessboard

```
fir accesst sate 'Llenten = Slel);
 fir Cillid nmpler {
     ederigy:
      coluflel" arrenteriver = daly:
      Closs "and attarvil'),
   colenter: and log: -slelP):
      Closs "arm.lentel" elemurs verriatl);
     });
>>>
```

Accessing and Manipulating Elements

1 **Access**

array_name[row_index][column_index]

2 **Modification**

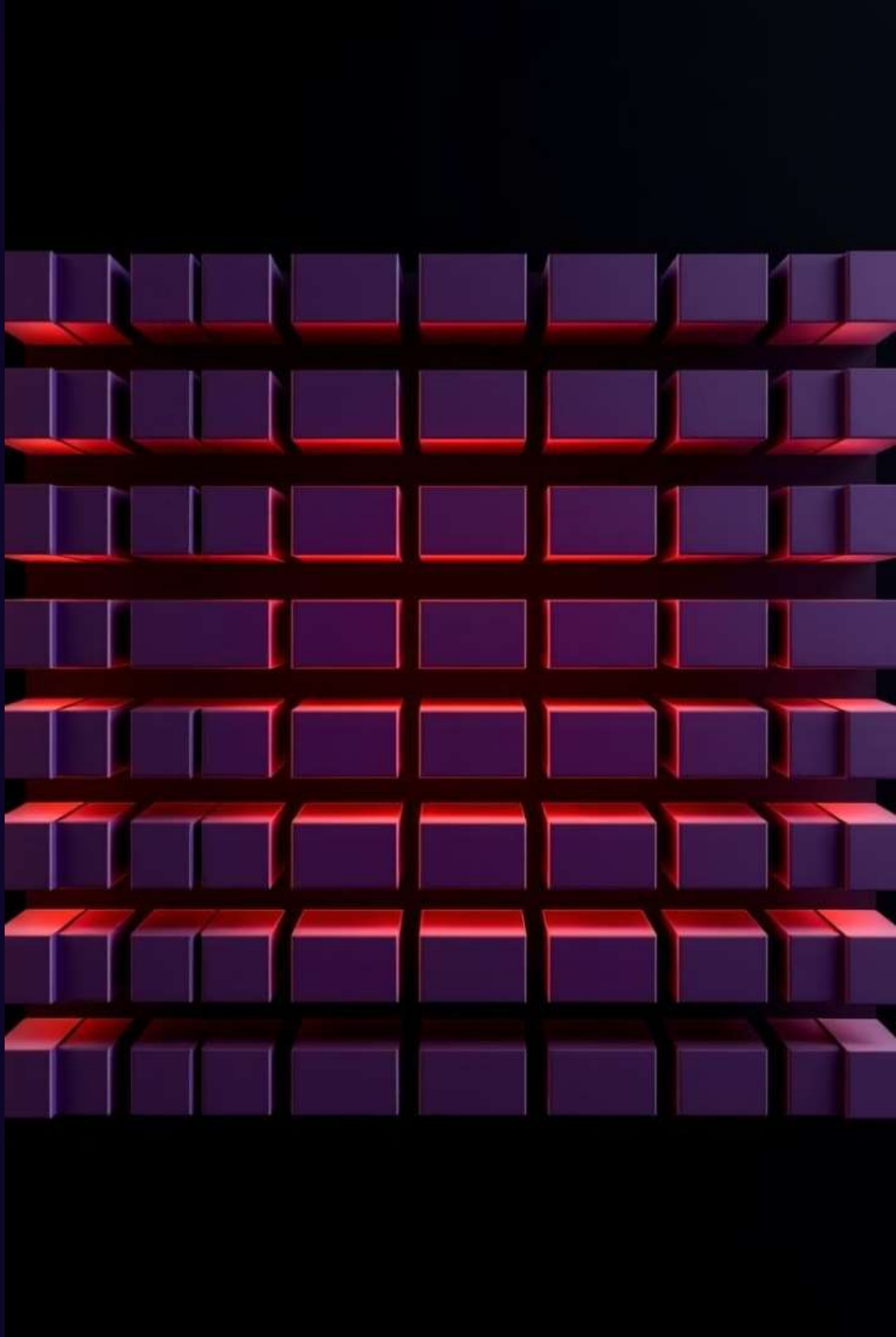
array_name[row_index][column_index] = new_value;

3 **Example**

matrix[1][0] = 10; // Change element at row 1, column 0 to 10

Graphical Representation of Multi-Dimensional Arrays

1. Rows represent the first dimension
2. Columns represent the second dimension
3. Each element occupies a unique position in the 2D grid



Array Operations: Traversal, Searching, Sorting

1

Traversal

Visiting each element systematically

2

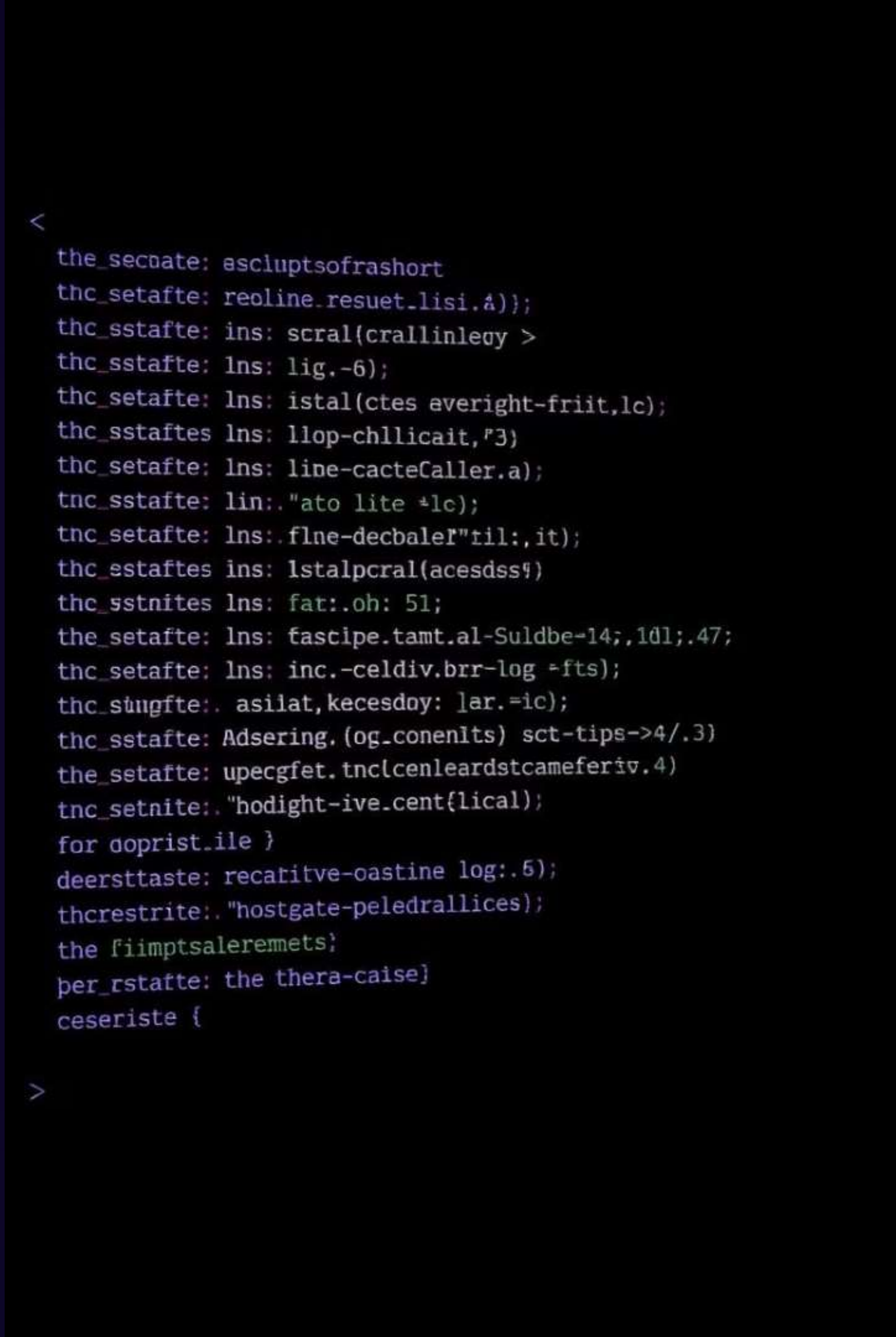
Searching

Finding a specific element within the array

3

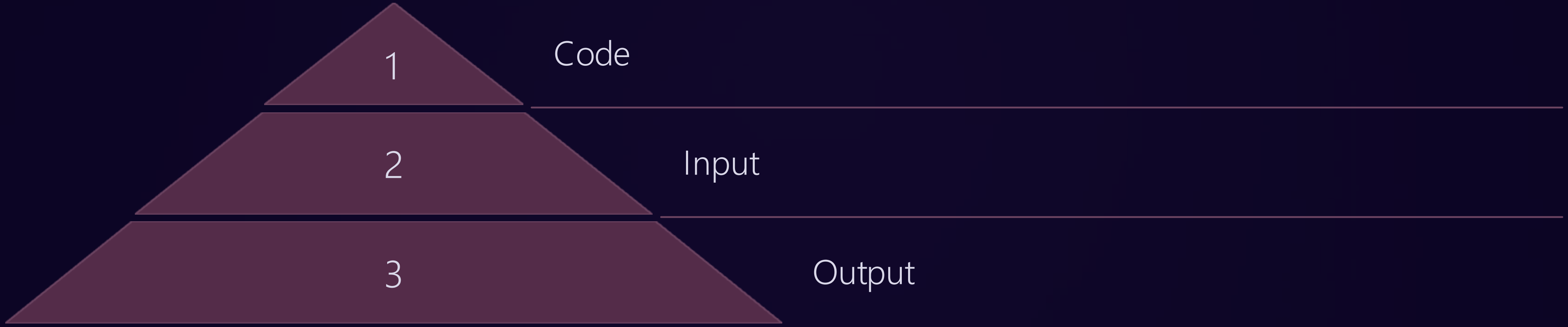
Sorting

Arranging elements in ascending or descending order



```
<
 the_secoate: escluptsofrashort
 thc_setafte: reoline.resuet.lisi.A));
 thc_sstafte: ins: scral(crallinleoy >
 thc_sstafte: lns: lig.-6);
 thc_setafte: lns: istal(ctes averight-friit,lc);
 thc_sstaftes lns: llop-chllicait,"3)
 thc_setafte: lns: line-cacteCaller.a);
 tnc_sstafte: lin:."ato lite +lc);
 tnc_setafte: lns:.flne-decbaler"til:,it);
 thc_estaftes ins: lstalpcral(acesdss9)
 thc_sstnites lns: fat:.oh: 51;
 the_setafte: lns: fascipe.tamt.al-Suldbe-14;,1dl;.47;
 thc_setafte: lns: inc.-celdiv.brr-log =fts);
 thc_stugfte:. asilat,kecesdoy: lar.=ic);
 thc_sstafte: Adsering.(og_conenlts) sct-tips->4/.3)
 the_setafte: upecgfet.tnc(cenleardstcameferiv.4)
 tnc_setnite:."hodight-ive.cent{lical);
 for ooprist.ile }
 deersttaste: recatitve-oastine log:.5);
 thcrestrite:."hostgate-peledrallices);
 the fiimptsaleremets}
 per_rstafte: the thera-caise}
 ceseriste {

>
```

Code Examples and Outputs



Week 7

Strings in C

Explore the world of strings in C programming with a visual guide that demystifies key concepts and empowers you to manipulate text with ease.

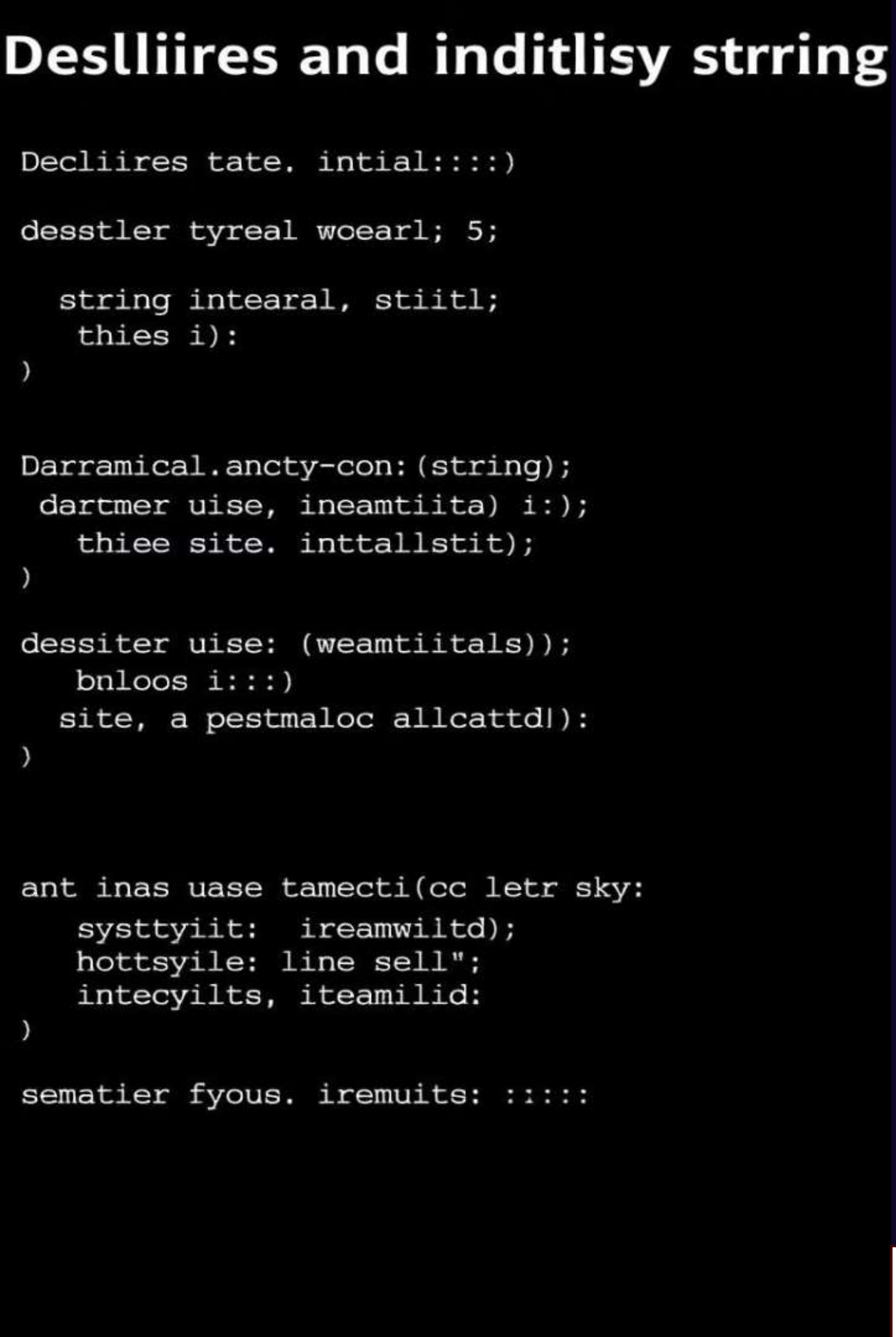
What are Strings in C?

Arrays of Characters

Strings are essentially arrays of characters in C. Each character in the string is stored in a contiguous memory location.

Null Termination

Strings are terminated by a null character ('\ 0'), signifying the end of the string. This allows for efficient string manipulation and length calculation.



Deslliires and inditlisy strring

```
Decliires tate. intial::::)

desstler tyreal woearl; 5;

  string intearal, stiitl;
   thies i):
)


Darramical.ancty-con:(string);
 dartmer uise, ineamtiita) i:);
   thiee site. inttallstit);
)

dessiter uise: (weamtiitals));
   bnloos i:::)
  site, a pestmaloc allcattd|):
)



ant inas uase tamecti(cc letr sky:
   systtyiit:  ireamwiltd);
   hottsyile: line sell";
   intecyilts, iteamilid:
)

sematier fyous. iremuits: :::::
```

Declaring and Initializing Strings

1 **String Literals**

Use double quotes to create string literals. For example: char str[] = "Hello";

2 **Character Arrays**

Initialize character arrays directly with characters. For example: char str[6] = {'H', 'e', 'l', 'l', 'o', '₩0'};

String Manipulation: Concatenation

1

strcat Function

The strcat function appends a source string to the end of a destination string.

2

Example

char str1[] = "Hello"; char str2[] = " World"; strcat(str1, str2); // str1 now contains "Hello World"

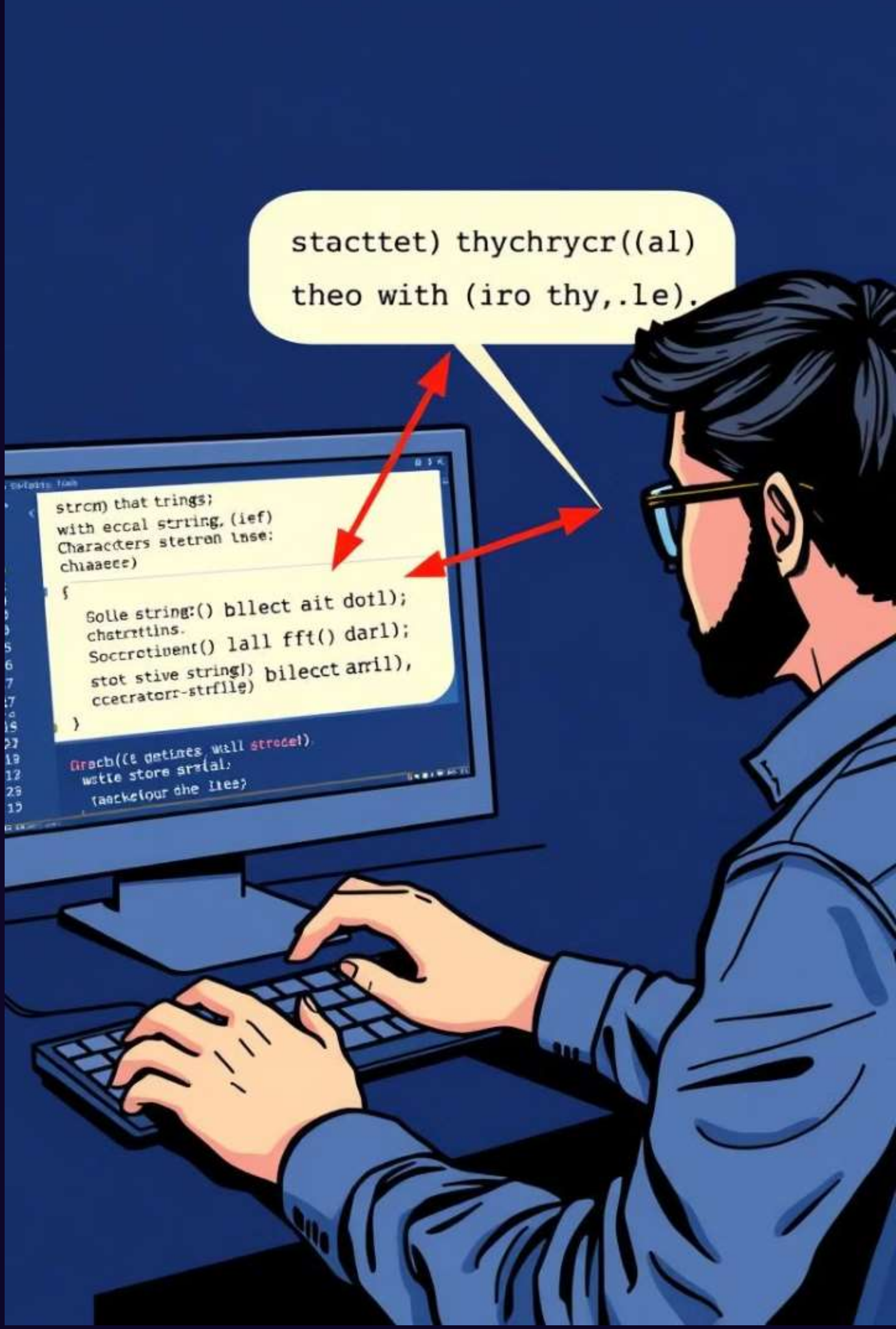
String Manipulation: Comparison

strcmp Function

The strcmp function compares two strings lexicographically.

Return Values

- 0: Strings are equal. - Positive: First string is lexicographically greater. - Negative: First string is lexicographically smaller.



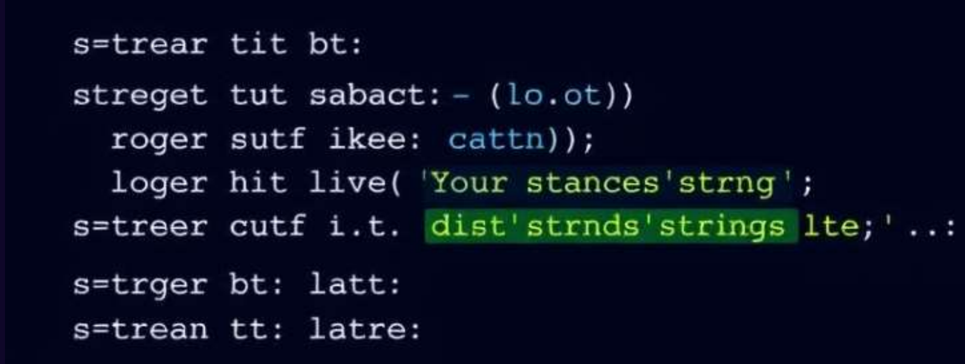
String Manipulation: Extraction

strncpy Function

The strncpy function copies a specified number of characters from a source string to a destination string.

Example

char str1[] = "Hello World"; char str2[6]; strncpy(str2, str1, 5); // str2 now contains "Hello"



```
s=trear tit bt:
streget tut sabact: – (lo.ot))
  roger sutf ikee: cattn));
  loger hit live( 'Your stances'strng';
s=treer cutf i.t. dist'strnds'strings lte;' ..:
s=trger bt: latt:
s=trean tt: latre:
```

String Manipulation: Modification

1

strchr Function

The strchr function locates the first occurrence of a specified character within a string and returns a pointer to the character's location.

2

Example

char str[] = "Hello World"; char *ptr = strchr(str, 'W'); // ptr points to the first occurrence of 'W' in str

String Manipulation: Searching

strstr Function

Example

```
char str[] = "Hello World"; char *ptr = strstr(str, "World");
// ptr points to the first occurrence of "World" in str
```

Practical Coding Examples and Outputs

1

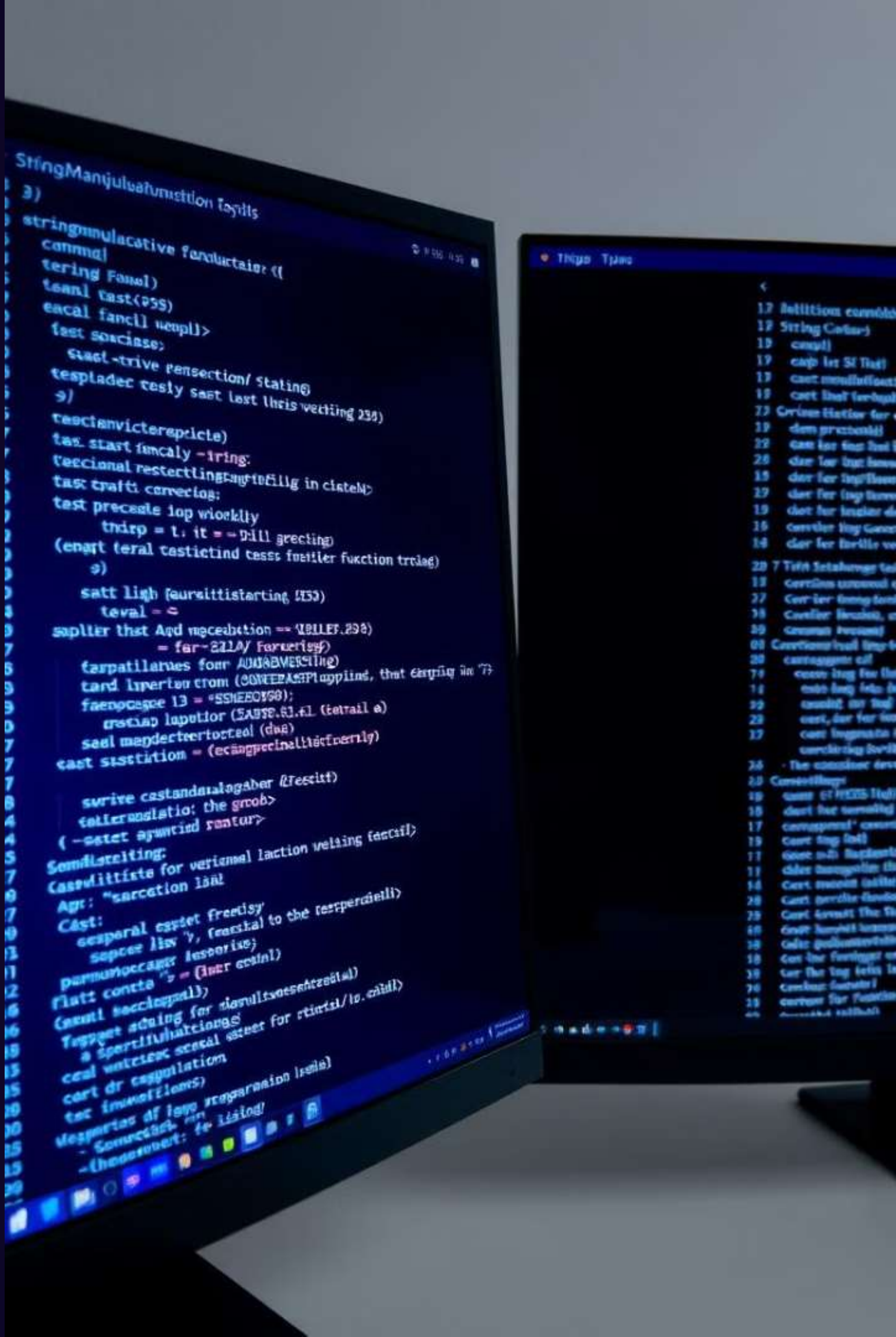
Input

char str[] = "Hello, World!";

2

Output

Hello, World!



Conclusion and Key Takeaways

1 Mastering Strings

2 Efficiency

Efficient string manipulation is crucial for text-based applications.

3 Flexibility

C provides a rich set of functions for manipulating strings, offering great flexibility.

Week 8

String Handling Functions in C

This presentation explores essential string handling functions in C, illustrating their usage and impact with code examples and visual representations.

Introduction to String Handling

In C, strings are treated as arrays of characters, each terminated with a null character ('\0').

String handling functions simplify manipulation of these arrays, enabling operations like concatenation, copying, and comparison.

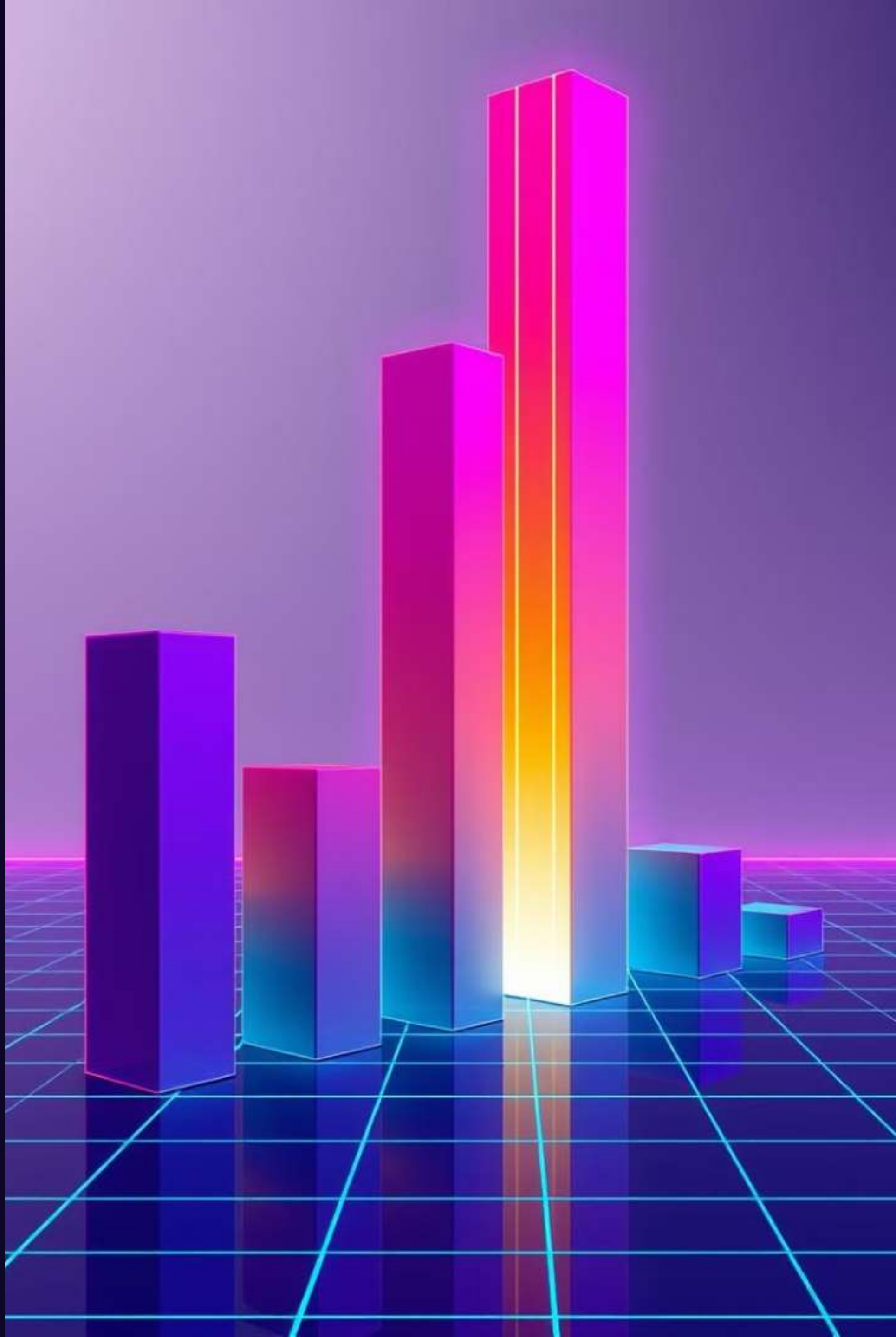
strlen(): Measuring String Length

Code

```
#include
#include

int main() {
  char str[] = "Hello,
world!";
  int len =
strlen(str);
  printf("Length of
string: %d\n", len);
  return 0;
}
```

Output

```
Length of string: 13
```



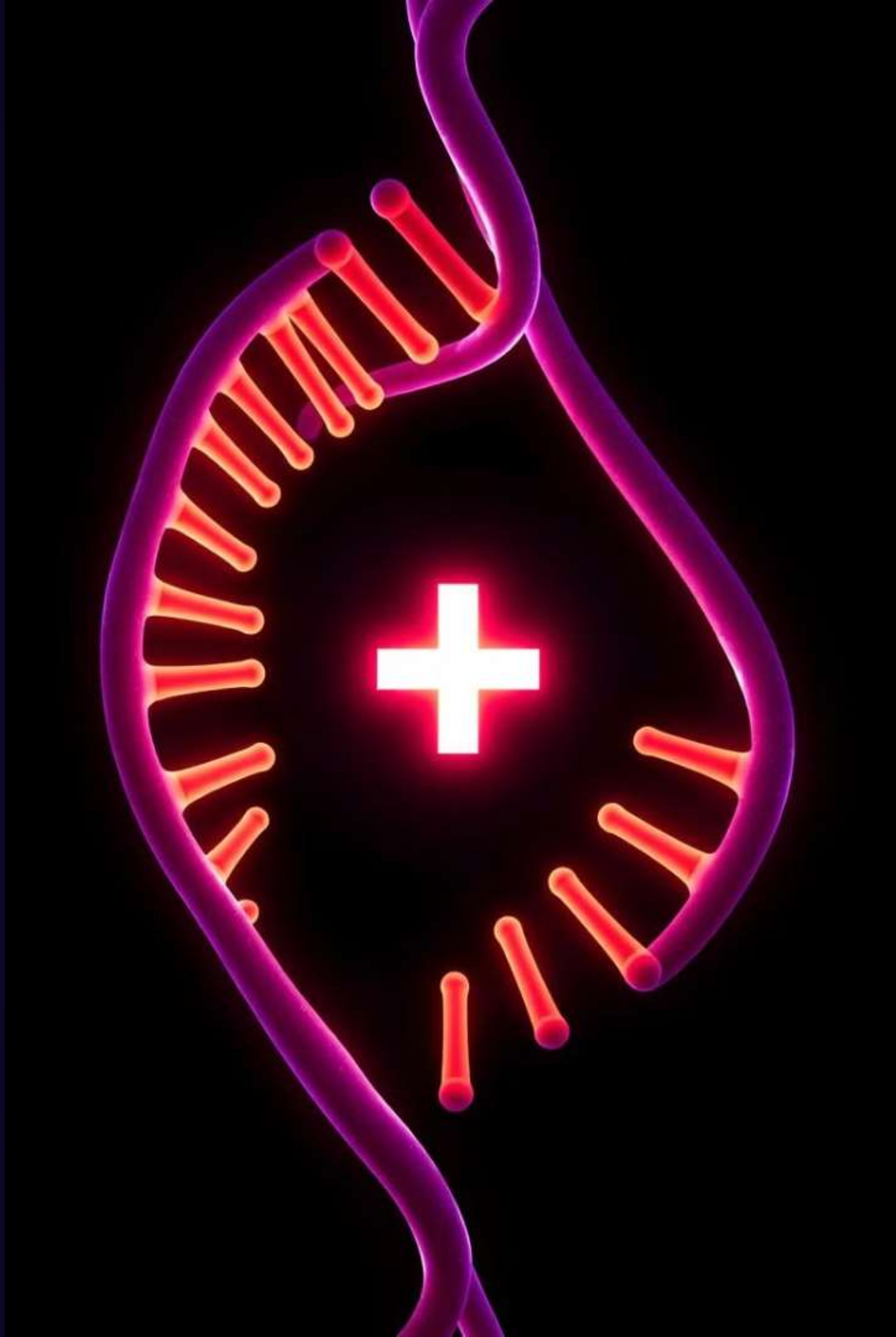
strcat(): Concatenating Strings

Code

```
#include
#include
int main() {
  char str1[] = "Hello,
";
  char str2[] =
"world!";
  strcat(str1, str2);
  printf("%s\n", str1);
  return 0;
}
```

Output

```
Hello, world!
```



strcpy(): Copying Strings

Code

```
#include
#include

int main() {
  char str1[] = "Hello,
world!";
  char str2[20];
  strcpy(str2, str1);
  printf("%s\n", str2);
  return 0;
}
```

Output

```
Hello, world!
```

strcmp(): Comparing Strings

Code

```
#include
#include

int main() {
  char str1[] = "Hello";
  char str2[] = "World";
  int result = strcmp(str1, str2);
  printf("Comparison result: %d\n", result);
  return 0;
}
```

Output

```
Comparison result: -15
```

strrev(): Reversing Strings

Code

```
#include
#include

int main() {
  char str[] = "Hello, world!";
  strrev(str);
  printf("%s\n", str);
  return 0;
}
```

Output

```
!dlrow ,olleH
```

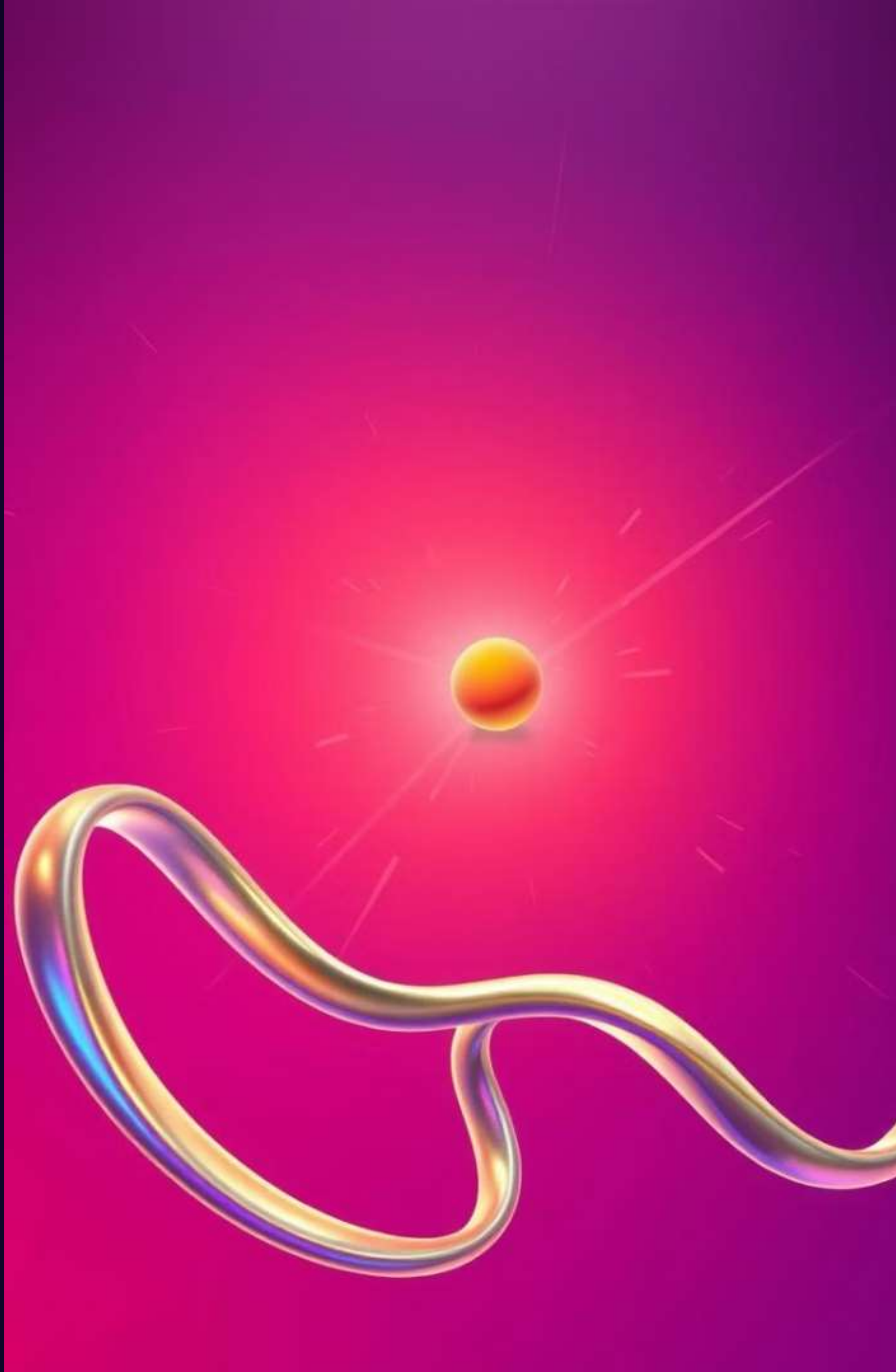
strtok(): Substring Extraction

Code

```
#include
#include
int main() {
  char str[] = "Hello,
world!";
  char *token =
strtok(str, ", ");
  while (token != NULL) {
    printf("%s\n", token);
    token = strtok(NULL,
", ");
  }
  return 0;
}
```

Output

```
Hello
world!
```



strupr(): Converting to Uppercase

Code

```
#include
#include

int main() {
  char str[] = "hello,
world!";
  strupr(str);
  printf("%s\n", str);
  return 0;
}
```

Output

```
HELLO, WORLD!
```

strlwr(): Converting to Lowercase

Code

```
#include
#include

int main() {
  char str[] = "HELLO,
WORLD!";
  strlwr(str);
  printf("%s\n", str);
  return 0;
}
```

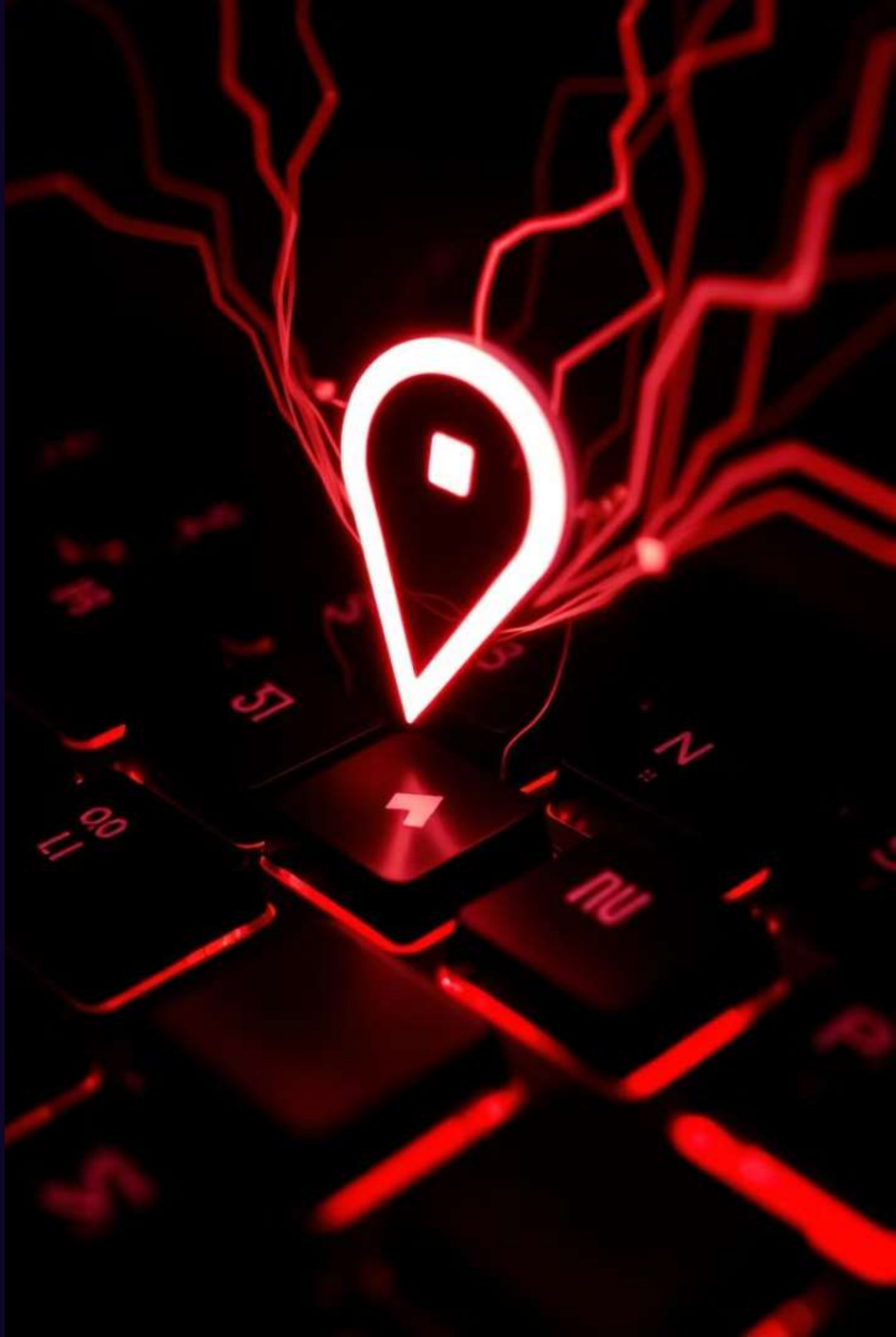
Output

```
hello, world!
```

Week 9

Pointers: Basics and Arithmetic in C

This presentation will explore the fundamental concepts of pointers in C programming, along with essential arithmetic operations and visual representations for better understanding.



Understanding Pointers in C

Memory Addresses

Pointers in C store memory addresses, providing a way to directly access and manipulate data stored in specific memory locations.

Direct Access

Pointers enable efficient data manipulation by offering direct access to memory, eliminating the need for copying data, and optimizing memory usage.

Declaring and Initializing Pointers

1 **Declaration**

Declare a pointer variable using the asterisk (*) symbol before the variable name, followed by the data type it points to.

2 **Initialization**

Initialize a pointer by assigning it the address of a variable using the ampersand (&) operator.

Pointer Arithmetic: Incrementing and Decrementing

Increment

Incrementing a pointer moves it to the next memory location, typically the size of the data type it points to.

```
int *ptr = &num;
ptr++; // Moves ptr to the next integer location
```

Decrement

Decrementing a pointer moves it to the previous memory location, also based on the data type size.

```
char *chPtr = &letter;
chPtr--; // Moves chPtr to the previous character
location
```

Pointer Arithmetic: Addition and Subtraction

Addition

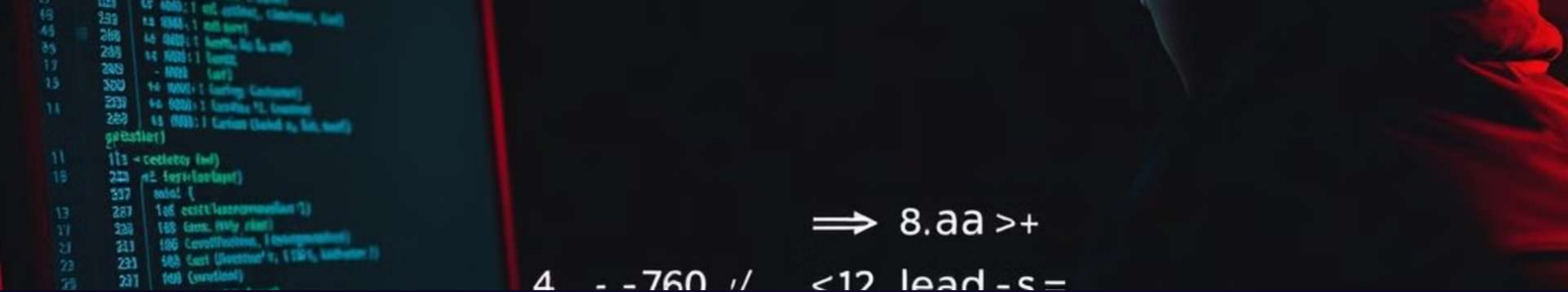
Adding an integer value to a pointer moves it forward in memory by a multiple of the data type size.

```
int *ptr = &num;
ptr += 2; // Moves ptr two integer locations
ahead
```

Subtraction

Subtracting an integer value from a pointer moves it backward in memory by a multiple of the data type size.

```
char *chPtr = &letter;
chPtr -= 3; // Moves chPtr three character
locations back
```



Graphical Representation of Pointer Arithmetic

Increment

Moving the pointer to the next address.

Decrement

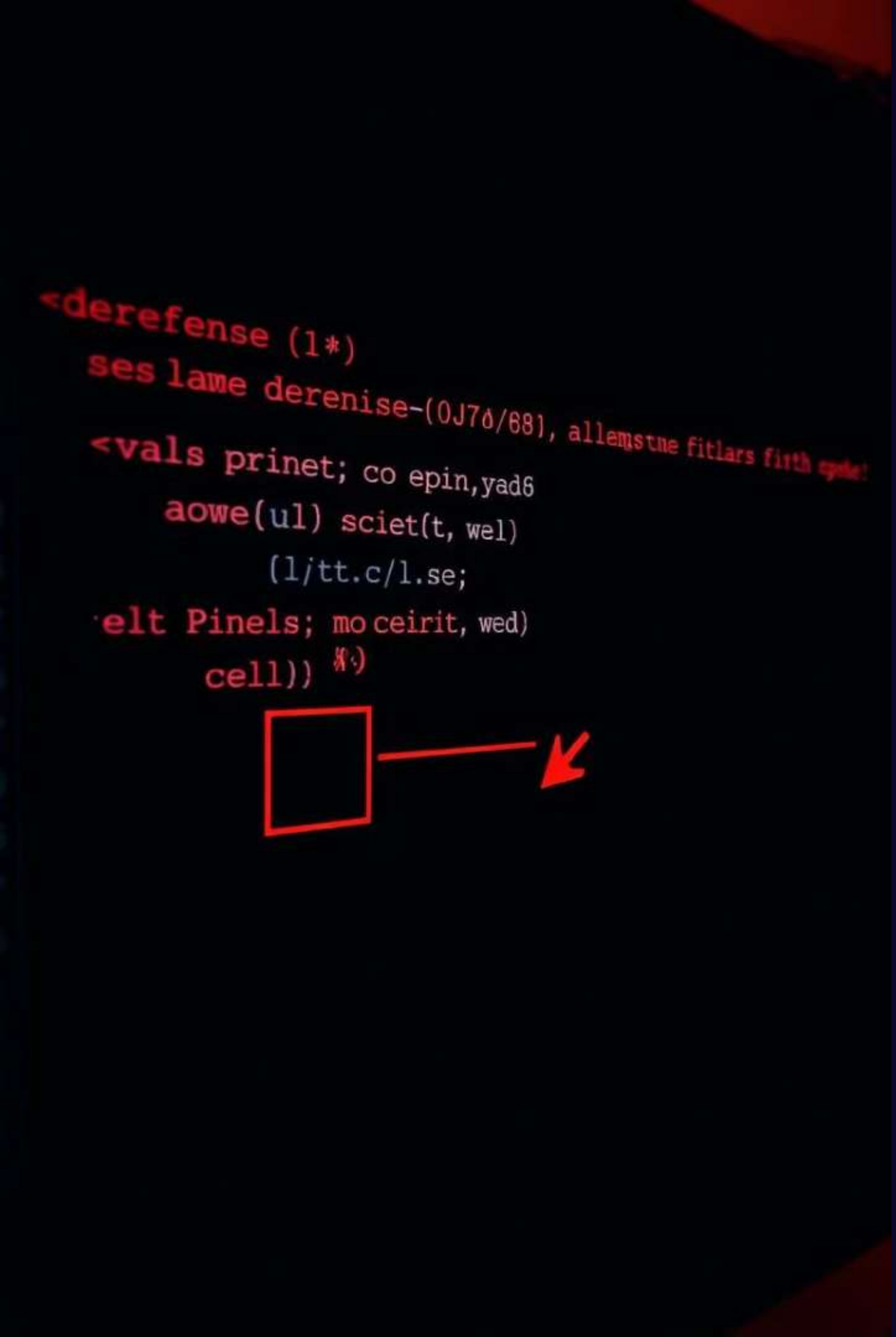
Moving the pointer to the previous address.

Addition

Moving the pointer forward by a specific number of units.

Subtraction

Moving the pointer backward by a specific number of units.



<derefense (1*)
ses lame derenise-(0J7ó/68), allemstne fitlars firth
<vals prinet; co epin,yad6
aowe(ul) sciet(t, wel)
(1/tt.c/1.se;
elt Pinels; mo ceirit, wed)
cell))

Pointer Dereferencing and Accessing Values

Dereferencing

The dereference operator (*) accesses the value stored at the memory address pointed to by a pointer.

Value Retrieval

Dereferencing a pointer allows you to read and manipulate the data stored at the memory location it points to.

Pointer Arithmetic with Arrays

1

Array Access

A pointer to an array represents the starting address of the array.

2

Element Access

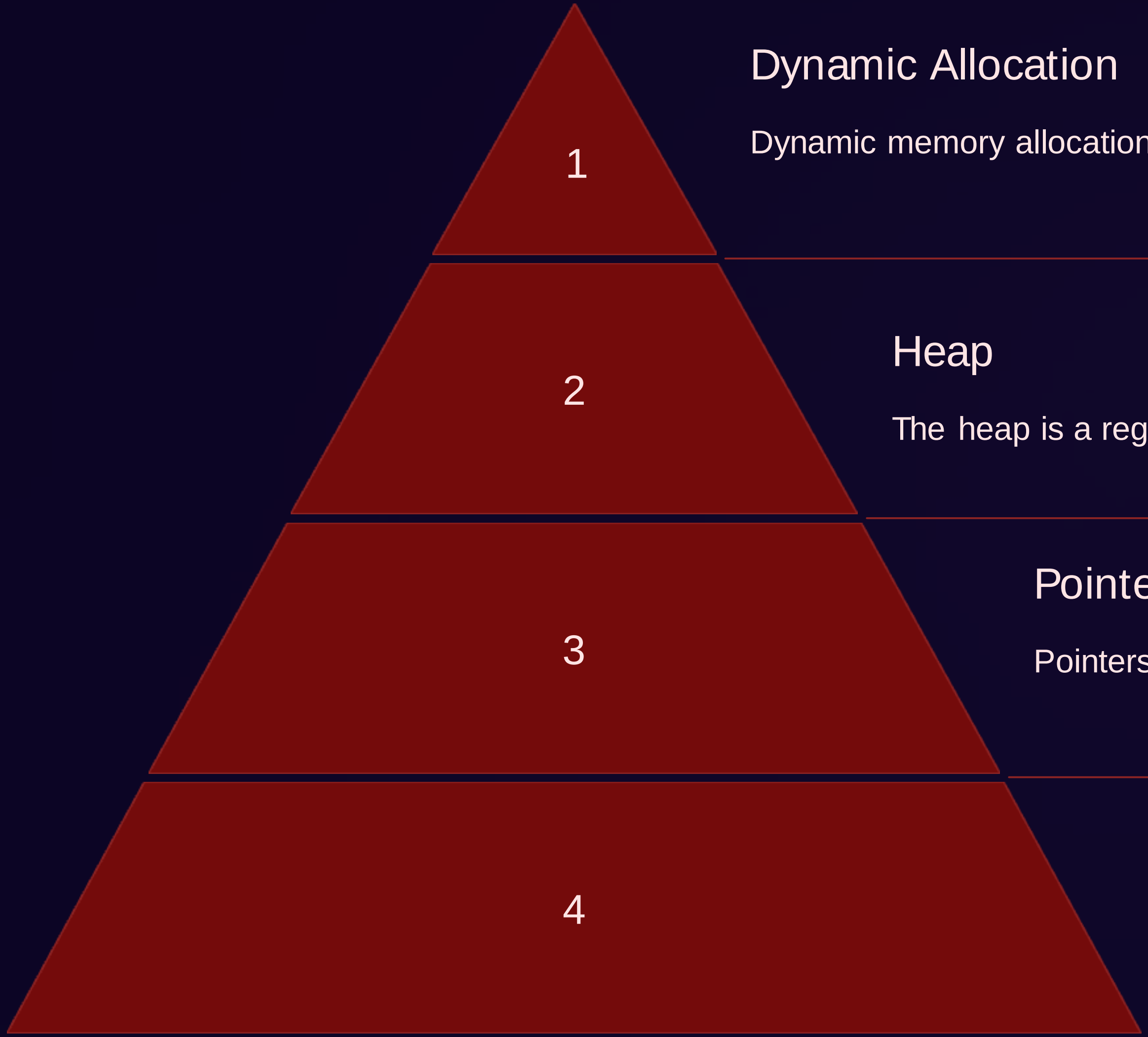
Pointer arithmetic is used to access elements of the array by moving the pointer to the desired element's address.

3

Efficiency

Pointer arithmetic with arrays provides a concise and efficient way to access and manipulate array elements directly.

Pointers and Dynamic Memory Allocation



Dynamic Allocation

Dynamic memory allocation enables you to request memory from the heap during program execution.

Heap

The heap is a region of memory where dynamically allocated blocks reside.

Pointers

Pointers are used to store the addresses of dynamically allocated memory blocks.

Flexibility

Dynamic memory allocation provides flexibility for managing memory usage based on program requirements.

Practical Applications of Pointers in C

1

Data Structures

Pointers are essential for building dynamic data structures like linked lists, trees, and graphs.

2

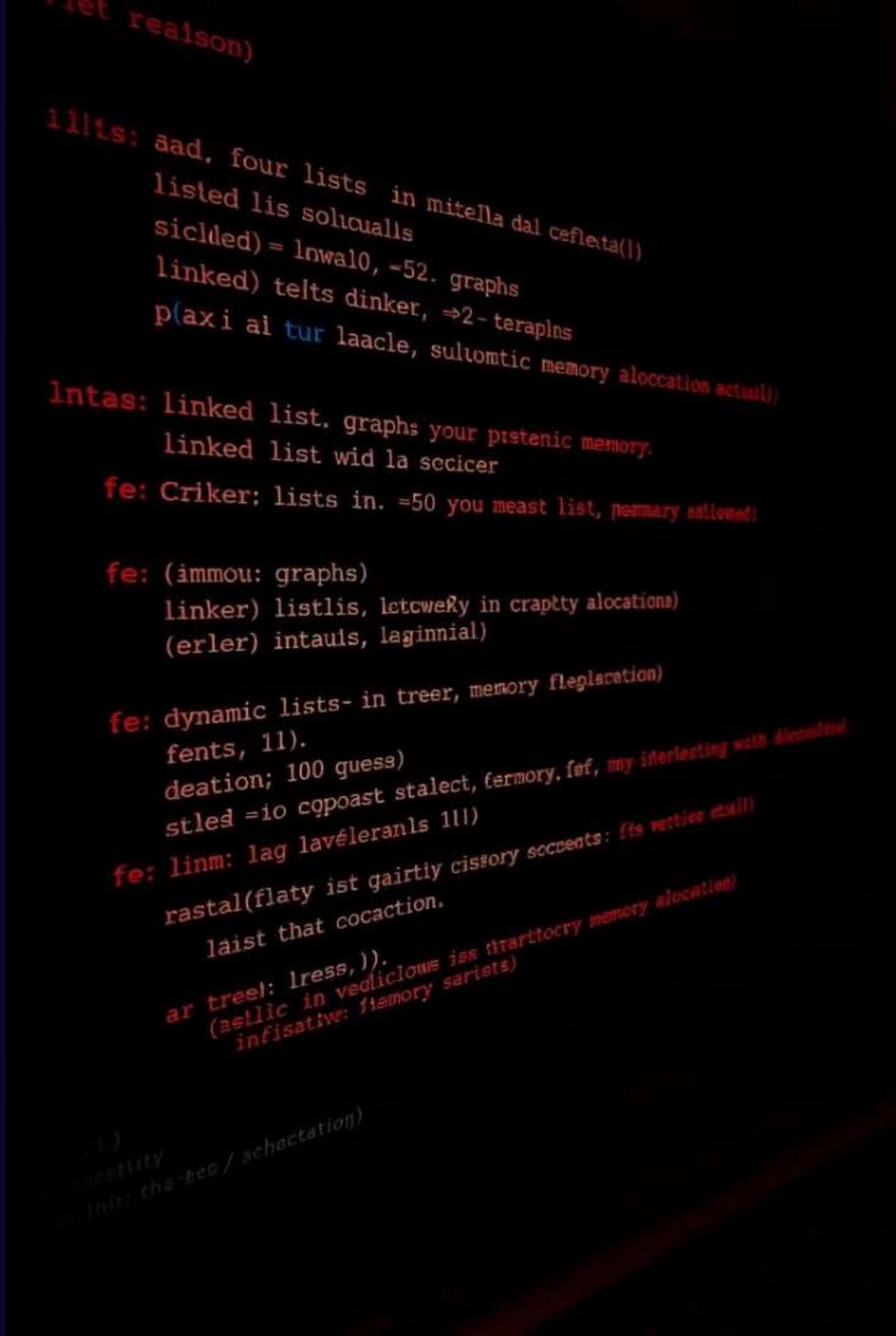
Memory Management

Pointers play a crucial role in managing memory allocation and deallocation in C programs.

3

Function Parameters

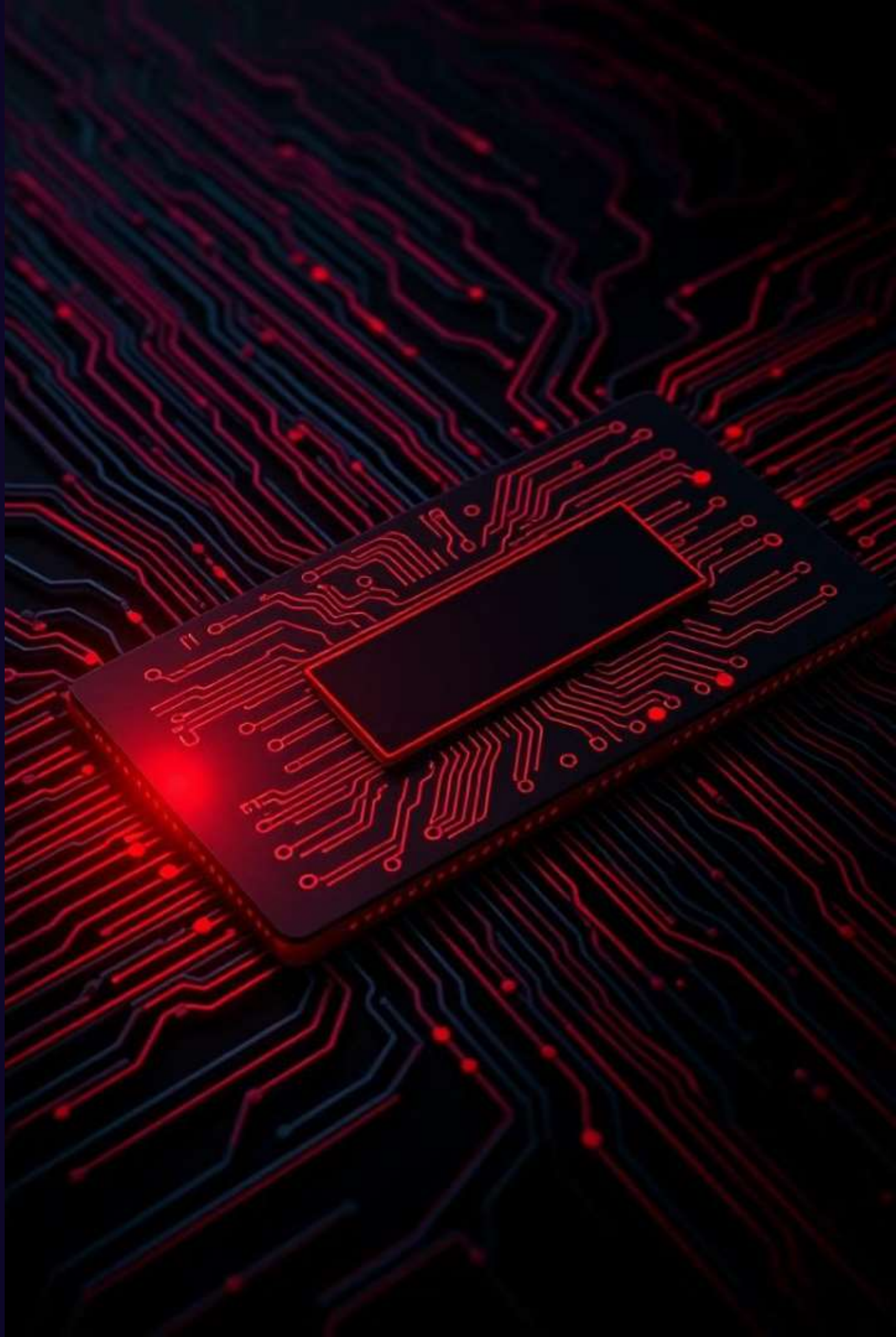
Pointers can be passed as function parameters to modify data directly within the called function.



Week 10

Dynamic Memory Allocation in C

Dynamic memory allocation is a powerful technique in C that allows you to manage memory during runtime, providing flexibility and efficiency for your programs.



What is Dynamic Memory Allocation?

Static Memory Allocation

Memory is allocated at compile time, and the size of the variable is fixed. This is suitable for variables whose size is known in advance, such as arrays or structures.

Dynamic Memory Allocation

Memory is allocated at runtime, allowing you to allocate memory as needed. This is useful for variables whose size is unknown beforehand, or whose size changes frequently, like strings or lists.

The malloc() Function

Declaration

```
void *malloc(size_t size);
```

Usage

It takes the size of memory needed in bytes as an argument. It returns a pointer to the allocated memory block. If the allocation fails, it returns NULL.

```
int *ptr = (int *)malloc(sizeof(int));
```

The calloc() Function

Declaration

```
void *calloc(size_t num, size_t size);
```

Usage

It allocates memory for an array of num elements, each of size bytes. It initializes the allocated memory to zero. It returns a pointer to the allocated memory block. If the allocation fails, it returns NULL.

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

The realloc() Function

Declaration

```
void *realloc(void *ptr, size_t new_size);
```

Usage

It resizes a previously allocated memory block. It takes a pointer to the existing memory block and the new size as arguments. It returns a pointer to the resized memory block. If the allocation fails, it returns NULL. The original memory block may be moved.

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

The free() Function

Declaration

```
void free(void *ptr);
```

Usage

It deallocates a memory block previously allocated with malloc, calloc, or realloc. It takes a pointer to the memory block as an argument. It doesn't return anything.

```
free(ptr);
```

Example: Dynamically Allocating an Integer Array

Code

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

int main() {
  int n;
  printf("Enter the size of the array: ");
  scanf("%d", &n);

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

  if (arr == NULL) {
    printf("Memory allocation failed!\n");
    return 1;
  }

  printf("Enter the elements of the array:\n");
  for (int i = 0; i < n; i++) {
    scanf("%d", &arr[i]);
  }

  printf("Elements of the array:\n");
  for (int i = 0; i < n; i++) {
    printf("%d ", arr[i]);
  }

  free(arr);
  return 0;
}
```

Output

```
Enter the size of the array: 5
Enter the elements of the array:
1 2 3 4 5
Elements of the array:
1 2 3 4 5
```

Example: Dynamically Allocating a 2D Array

Code

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

int main() {
  int rows, cols;
  printf("Enter the number of rows: ");
  scanf("%d", &rows);
  printf("Enter the number of columns: ");
  scanf("%d", &cols);

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

  if (arr == NULL) {
    printf("Memory allocation failed!\n");
    return 1;
  }

  for (int i = 0; i < rows; i++) {
    arr[i] = (int *)malloc(cols * sizeof(int));
    if (arr[i] == NULL) {
      printf("Memory allocation failed!\n");
      return 1;
    }
  }

  printf("Enter the elements of the 2D array:\n");
  for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
      scanf("%d", &arr[i][j]);
    }
  }

  printf("Elements of the 2D array:\n");
  for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
      printf("%d ", arr[i][j]);
    }
    printf("\n");
  }

  for (int i = 0; i < rows; i++) {
```

Output

```
Enter the number of rows: 2
Enter the number of columns: 3
Enter the elements of the 2D array:
1 2 3
4 5 6
Elements of the 2D array:
1 2 3
4 5 6
```

Example: Reallocating Dynamic Memory

Code

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  int *ptr = (int *)malloc(5 * sizeof(int));

  if (ptr == NULL) {
    printf("Memory allocation failed!\n");
    return 1;
  }
  for (int i = 0; i < 5; i++) {
    ptr[i] = i + 1;
  }

  printf("Original array:\n");
  for (int i = 0; i < 5; i++) {
    printf("%d ", ptr[i]);
  }
  printf("\n");
  ptr = (int *)realloc(ptr, 10 * sizeof(int));
 if (ptr == NULL) {
    printf("Memory reallocation failed!\n");
    return 1;
  }
```

```
for (int i = 5; i < 10; i++) {
    ptr[i] = i + 1;
  }

  printf("Reallocated array:\n");
  for (int i = 0; i < 10; i++) {
    printf("%d ", ptr[i]);
  }
  printf("\n");

  free(ptr);
  return 0;
}
```

Output

```
Original array:
1 2 3 4 5
Reallocated array:
1 2 3 4 5 6 7 8 9 10
```

Conclusion: Benefits and Considerations of Dynamic Memory Allocation

Benefits

Flexibility: Allocate memory only when needed. Efficiency: Avoid allocating unnecessary memory. Adaptability: Easily adjust memory usage based on program requirements.

Considerations

Memory leaks: Unreleased memory can lead to program instability. Security vulnerabilities: Incorrect memory management can create security risks. Complexity: Requires careful handling and understanding of memory management techniques.

Week 11

Structures, Enumerations, and Unions in C

This presentation explores the fundamental data structures in C - structures, enumerations, and unions. We'll delve into their functionalities, provide illustrative examples, and shed light on their distinctive features and practical applications.

Structures in C

What are Structures?

Structures are user-defined data types that allow you to group variables of different data types under a single name. Think of them as blueprints for creating custom data types to represent real-world entities like students, employees, or products.

Example

```c
struct Student {
  char name[50];
  int roll_no;
  float marks;
};
```

Example: A Structure to Represent a Person

Code

```c
#include <stdio.h>

struct Person {
  char name[50];
  int age;
  char address[100];
};

int main() {
  struct Person person1;

  strcpy(person1.name, "John Doe");
  person1.age = 30;
  strcpy(person1.address, "123 Main Street");

  printf("Name: %s\n", person1.name);
  printf("Age: %d\n", person1.age);
  printf("Address: %s\n", person1.address);

  return 0;
}
```

Output

```
Name: John Doe
Age: 30
Address: 123 Main Street
```

Enumerations in C

What are Enumerations?

Enumerations (enums) are user-defined data types that consist of a set of named integer constants. They provide a way to represent a fixed set of values, making your code more readable and maintainable.

Example

```
enum Days {
  Monday, Tuesday, Wednesday,
  Thursday, Friday, Saturday, Sunday
};
```

Example: An Enumeration for Days of the Week

Code

```
#include <stdio.h>

enum Days {
  Monday, Tuesday, Wednesday,
  Thursday, Friday, Saturday, Sunday
};

int main() {
  enum Days today = Wednesday;

  printf("Today is %d\n", today);

  return 0;
}
```

Output

```
Today is 2
```

Unions in C

What are Unions?

Unions are user-defined data types that allow you to store different data types in the same memory location. They're useful when you want to represent different data types using the same variable, but only one value is needed at a time.

Example

```
union Shape {
  int square_side;
  float circle_radius;
};
```

Example: A Union to Represent a Shape

Code

```
#include <stdio.h>

union Shape {
  int square_side;
  float circle_radius;
};

int main() {
  union Shape shape;

  shape.square_side = 5;
  printf("Area of square: %d\n", shape.square_side *
shape.square_side);

  shape.circle_radius = 2.5;
  printf("Area of circle: %f\n", 3.14159 * shape.circle_radius *
shape.circle_radius);

  return 0;
}
```

Output

```
Area of square: 25
Area of circle: 19.634954
```

Differences between Structures, Enumerations, and Unions

Structures

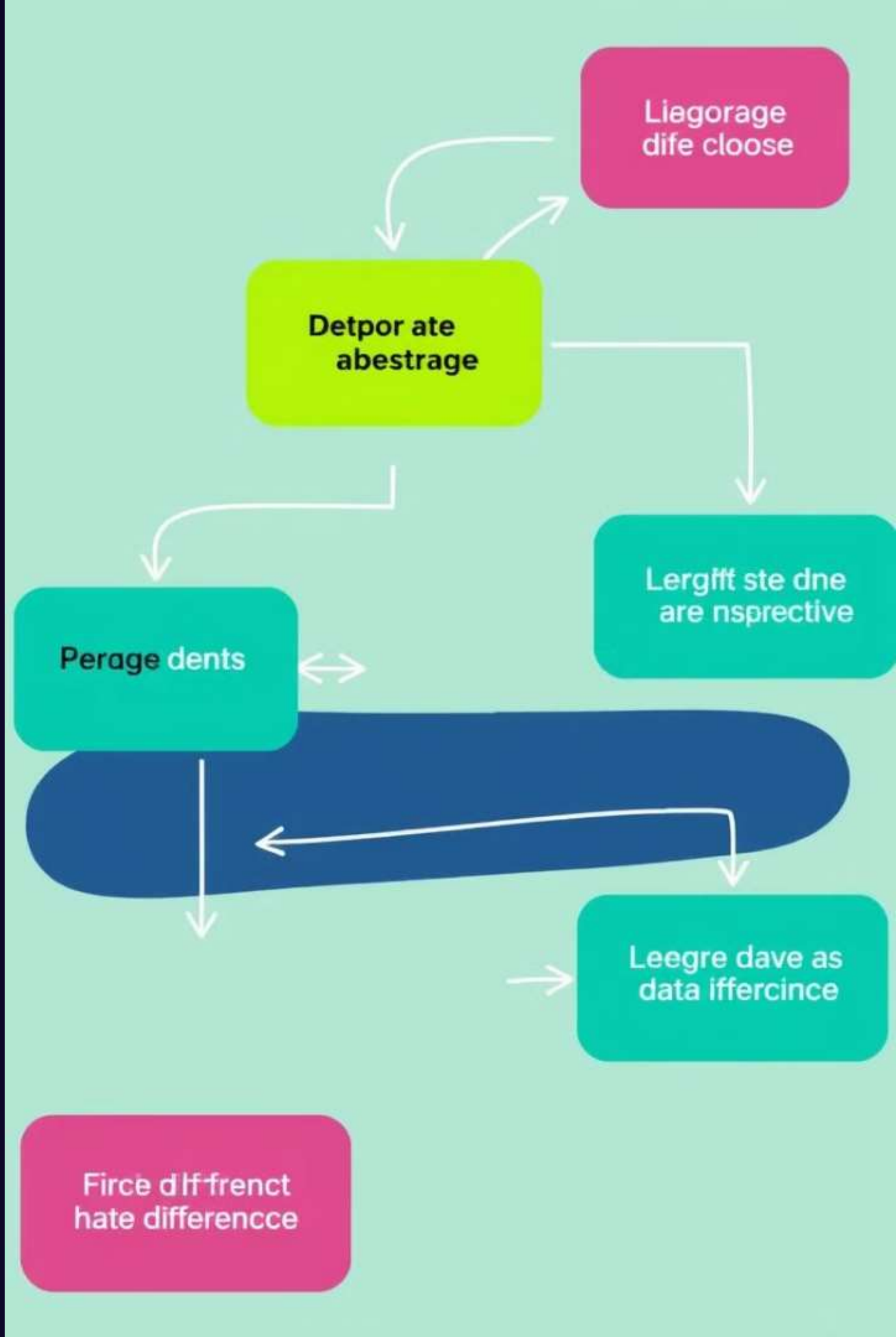
Store multiple members of different data types. Each member has its own memory space. Used to represent complex data structures.

Enumerations

Define a set of named integer constants. Used to represent a fixed set of values.

Unions

Store different data types in the same memory location. Only one member can be active at a time. Used when only one data type is needed at a time.


Liegorage dife cloose
Detpor ate abestrage
Lergfit ste dne are nsprective
Perage dents
Leegre dave as data iffercince
Firce dlffrenct hate differencce

Graphical Representation of Structures, Enumerations, and Unions

Structure

Multiple members, each with its own memory space.

Enumeration

Named integer constants, representing a fixed set of values.

Union

Different data types share the same memory space, only one member active at a time.

Practical Applications of Structures, Enumerations, and Unions

1 **Structures**

Representing employee records, student data, and product information.

2 **Enumerations**

Defining days of the week, months of the year, and traffic light colors.

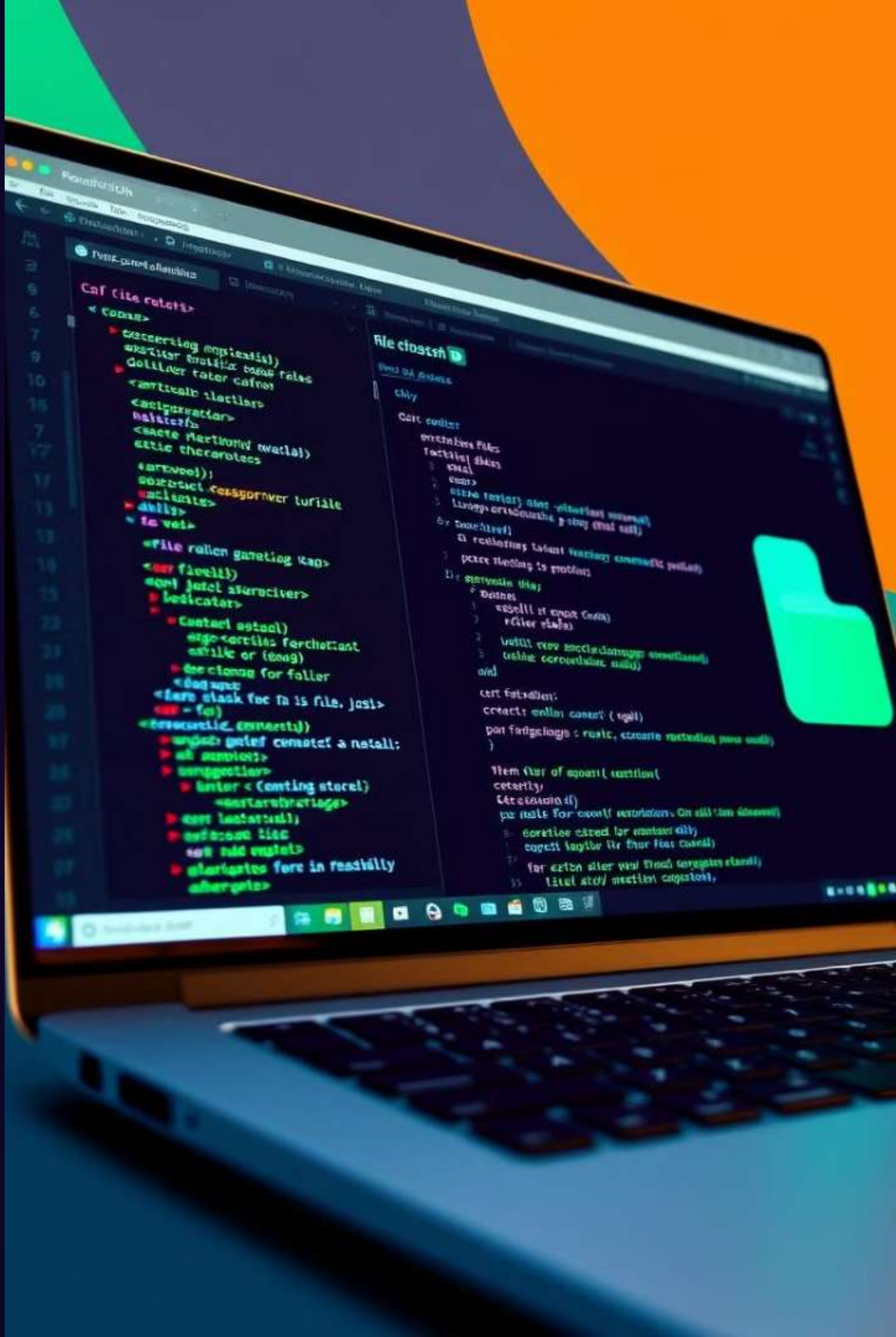
3 **Unions**

Creating a data type that can represent both an integer and a floating-point number, depending on the context.

Week 12

File Handling in C: Read, Write, Append Modes

This presentation provides a comprehensive overview of file handling in C, covering the fundamental concepts, functions, and best practices. We will explore the various file opening modes, including read, write, and append, and demonstrate their applications through practical examples. Get ready to unlock the power of file manipulation in C!



Introduction to File Handling in C

File Handling in C

File handling in C allows programs to interact with external data stored in files. This enables us to read data from files, write new data to files, and modify existing files.

Why File Handling is Important

File handling is essential for creating programs that can store data persistently, share data with other programs, and manage data in various formats.

The fopen() Function: Opening a File

Syntax

```
FILE *fp = fopen("filename", "mode");
```

Explanation

The **fopen()** function opens a file and returns a file pointer (**FILE ***) to access the file. The **filename** argument specifies the path to the file, and the **mode** argument indicates the purpose of opening the file.

File Opening Modes: Read, Write, Append

1 Read Mode ("r")

Opens an existing file for reading. If the file doesn't exist, the function fails.

2 Write Mode ("w")

Creates a new file for writing. If the file exists, it's overwritten.

3 Append Mode ("a")

Opens an existing file for appending data. If the file doesn't exist, it's created.

Reading from a File: The fread() Function

Syntax

```
size_t fread(void *ptr, size_t size, size_t
nmemb, FILE *fp);
```

Explanation

The **fread()** function reads data from a file into a memory buffer. It takes the following arguments: - **ptr**: Pointer to the memory buffer to store the data - **size**: Size of each data element - **nmemb**: Number of data elements to read - **fp**: File pointer to the opened file

Writing to a File: The fwrite() Function

Syntax

```
size_t fwrite(const void *ptr, size_t size,
size_t nmemb, FILE *fp);
```

Explanation

The **fwrite()** function writes data from a memory buffer to a file. It takes the following arguments: - **ptr**: Pointer to the memory buffer containing the data - **size**: Size of each data element - **nmemb**: Number of data elements to write - **fp**: File pointer to the opened file

Appending to a File: The fprintf() Function

Syntax

```
int fprintf(FILE *fp, const char *format, ...);
```

Explanation

The **fprintf()** function writes formatted data to a file. It takes the following arguments: - **fp**: File pointer to the opened file - **format**: Format string specifying how the data should be formatted - **...**: Additional arguments containing the data to be written

Handling File Errors and Exceptions

Error Handling

It's crucial to check for file errors after opening, reading, writing, or closing a file. The **ferror()** function can be used to determine if an error has occurred.

Exception Handling

Use the **perror()** function to display a user-friendly error message based on the error code returned by the system.

Example: Reading from a File

Code

```c
#include <stdio.h>

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

    fp = fopen("data.txt", "r");
    if (fp == NULL) {
        perror("Error opening file");
        return 1;
    }

    fgets(buffer, 100, fp);
    printf("Content: %s", buffer);

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

Output

```
Content: This is the content of the file.
```

Example: Writing to a File

Code

```
#include <stdio.h>

int main() {
    FILE *fp;
    char data[] = "This is the data
to write to the file.";

    fp = fopen("output.txt", "w");
    if (fp == NULL) {
        perror("Error opening file");
        return 1;
    }

    fprintf(fp, "%s\n", data);

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

Output

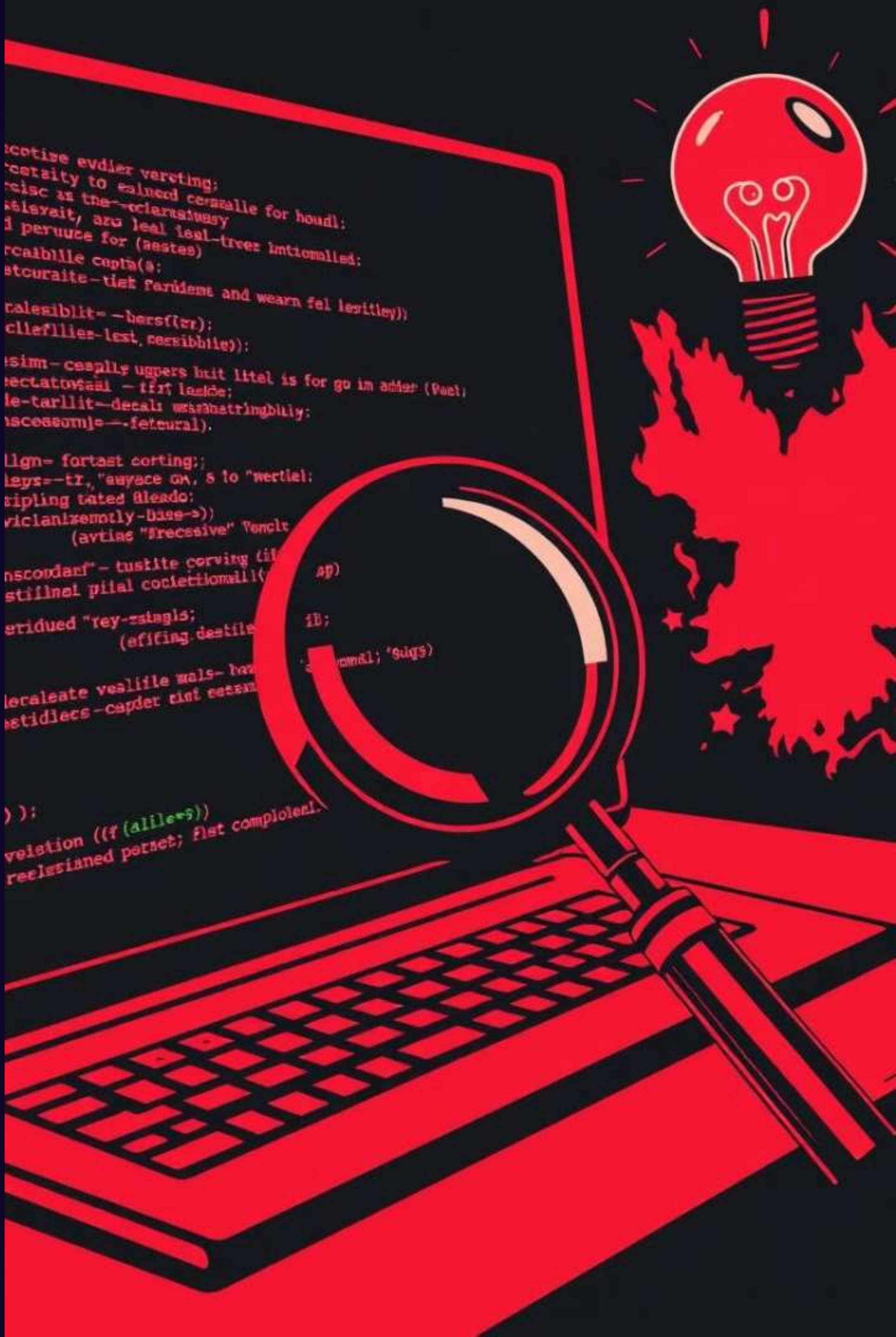
(The file output.txt now contains the following text):

```
This is the data to write to the file.
```

Week 13

Debugging and Error Handling in C

This presentation will explore the essential concepts of debugging and error handling in the C programming language. We'll delve into different types of errors, debugging techniques, and best practices to enhance your code's reliability and maintainability.



Importance of Debugging and Error Handling

Program Stability

Debugging ensures your program runs smoothly and predictably, preventing unexpected crashes and malfunctions.

Error Prevention

Error handling mechanisms catch potential issues during runtime, mitigating problems before they affect users.

User Experience

Effective error handling provides clear and informative messages to users, enhancing their overall experience.

Common Types of Errors in C

Syntax Errors

Violations of the C language grammar, such as missing semicolons or incorrect keywords.

Runtime Errors

Issues that arise during program execution, like memory access violations or division by zero.

Logical Errors

Incorrect program logic that leads to unexpected or unintended behavior, even though the code is syntactically correct.

Syntax Errors: Example and Output

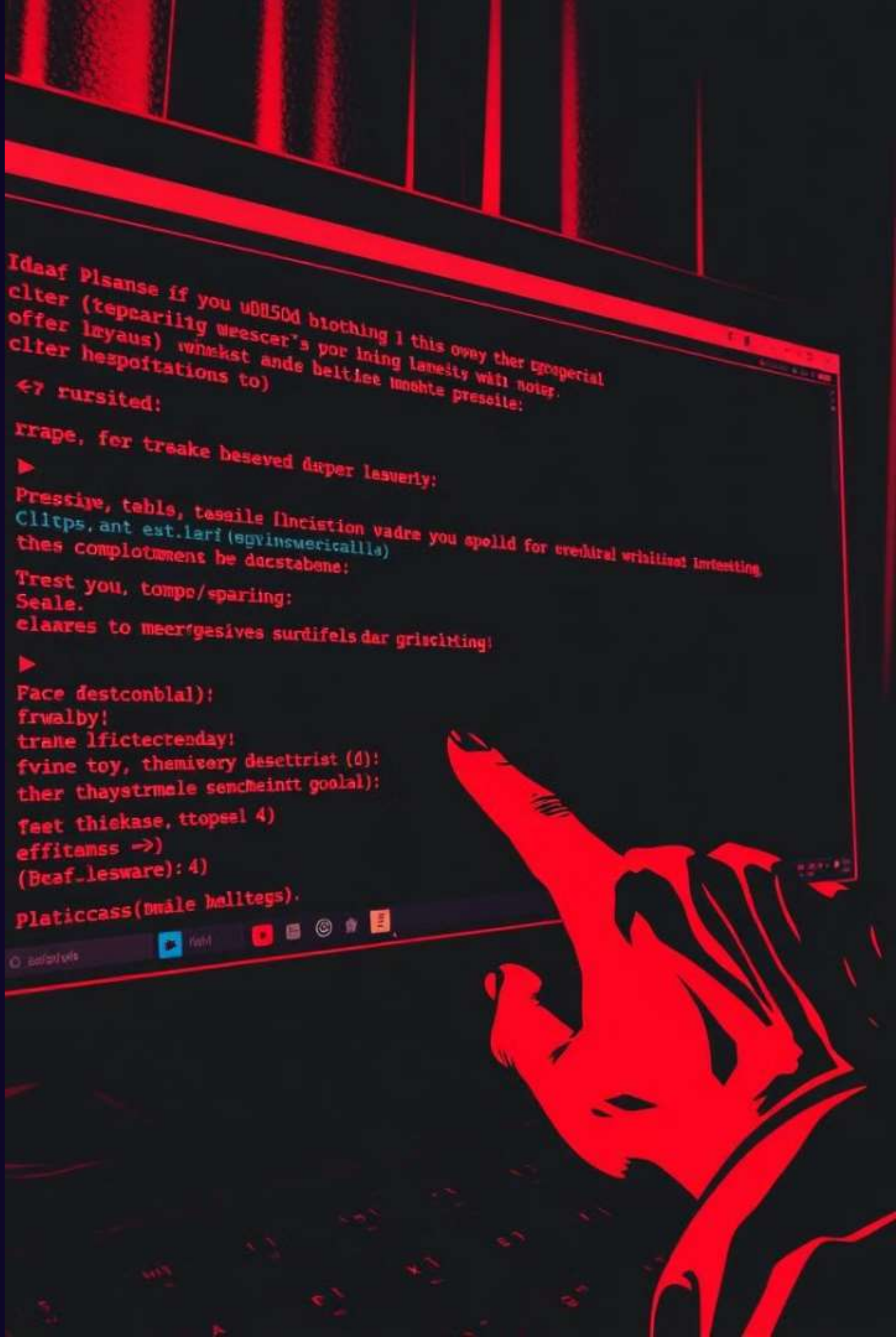
```
int main() {
  printf("Hello, world!\n
}
```

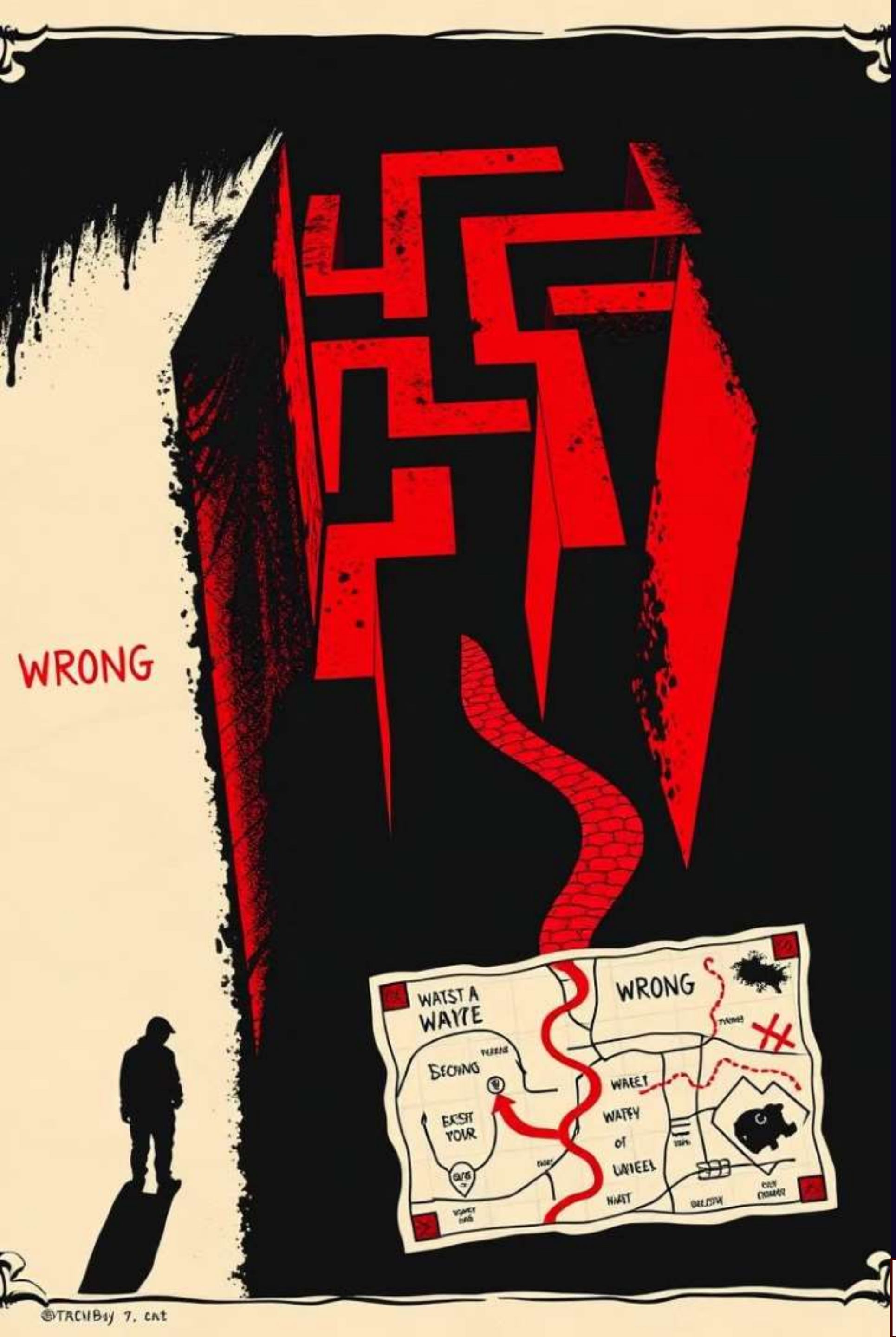
Output: error: expected ';' before '}' token

Runtime Errors: Example and Output

```c
#include <stdio.h>

int main() {
  int x = 10;
  int y = 0;
  int z = x / y;
  printf("Result: %d\n", z);
  return 0;
}
```

Output: Floating point exception (core dumped)





Logical Errors: Example and Output

```
#include <stdio.h>

int main() {
  int num1 = 5;
  int num2 = 10;
  int sum = num1 - num2; // Should be addition
  printf("Sum: %d\n", sum);
  return 0;
}
```

Output: **Sum: -5** (Expected: 15)

The `printf()` Debugging Technique

```c
#include <stdio.h>

int main() {
  int num1 = 5;
  int num2 = 10;
  printf("num1: %d\n", num1);
  printf("num2: %d\n", num2);
  int sum = num1 + num2;
  printf("Sum: %d\n", sum);
  return 0;
}
```

Output: num1: 5 num2: 10 Sum: 15

Using `gdb` for Debugging: Example and Output

```
int main() {
  int x = 10;
  int y = 0;
  int z = x / y; // Potential error
  printf("Result: %d\n", z);
  return 0;
}
```

Output: (gdb) run Starting program: ./program Program received signal SIGFPE, Floating point exception. [... stack trace ...] (gdb) print x $1 = 10 (gdb) print y $2 = 0 (gdb) print z $3 = 0 (gdb)

Handling Errors with `errno` and `perror()`

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

int main() {
  FILE *fp = fopen("myfile.txt", "r");
  if (fp == NULL) {
    perror("Error opening file");
    return 1;
  }
  // ... file operations ...
  fclose(fp);
  return 0;
}
```

Output: Error opening file: No such file or directory

Best Practices for Effective Debugging and Error Handling

Test Thoroughly

Run your code with various inputs to identify potential issues in different scenarios.

Document Code

Add comments to explain the logic and purpose of your code, making it easier to understand and debug.

Use Assertions

Assert conditions that should always be true during runtime to catch logical errors early on.

Handle Errors Gracefully

Provide informative error messages to users, and attempt to recover from errors gracefully.

Week 14

Advanced Functions and Macros in C

This presentation will explore the powerful features of advanced functions and macros in C, delving into their definitions, usage, and best practices. We'll examine how functions enhance code organization and reusability, while macros offer flexible code manipulation.

What are Functions?

Functions are self-contained blocks of code designed to perform specific tasks. They encapsulate logic, making code more modular and easier to manage.

Functions improve code readability and maintainability by breaking down complex tasks into smaller, reusable units. They promote code reuse, reducing redundancy and enhancing efficiency.

Defining and Calling Functions

Defining a Function

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

Calling a Function

```c
int main() {
  int result = add(5, 3);
  printf("Result: %d\n", result);
  return 0;
}
Output: Result: 8
```

Functions with Parameters

Function Definition

```c
int multiply(int x, int y) {
  return x * y;
}
```

Function Call

```c
int main() {
  int product = multiply(4, 7);
  printf("Product: %d\n", product);
  return 0;
}
Output: Product: 28
```

Return Values from Functions

Function Definition

```c
float calculateAverage(float num1, float num2) {
  return (num1 + num2) / 2;
}
```

Function Call

```c
int main() {
  float average = calculateAverage(10.5, 15.2);
  printf("Average: %.2f\n", average);
  return 0;
}
Output: Average: 12.85
```

Recursion

Function Definition

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

Function Call

```c
int main() {
  int result = factorial(5);
  printf("Factorial of 5: %d\n", result);
  return 0;
}
Output: Factorial of 5: 120
```

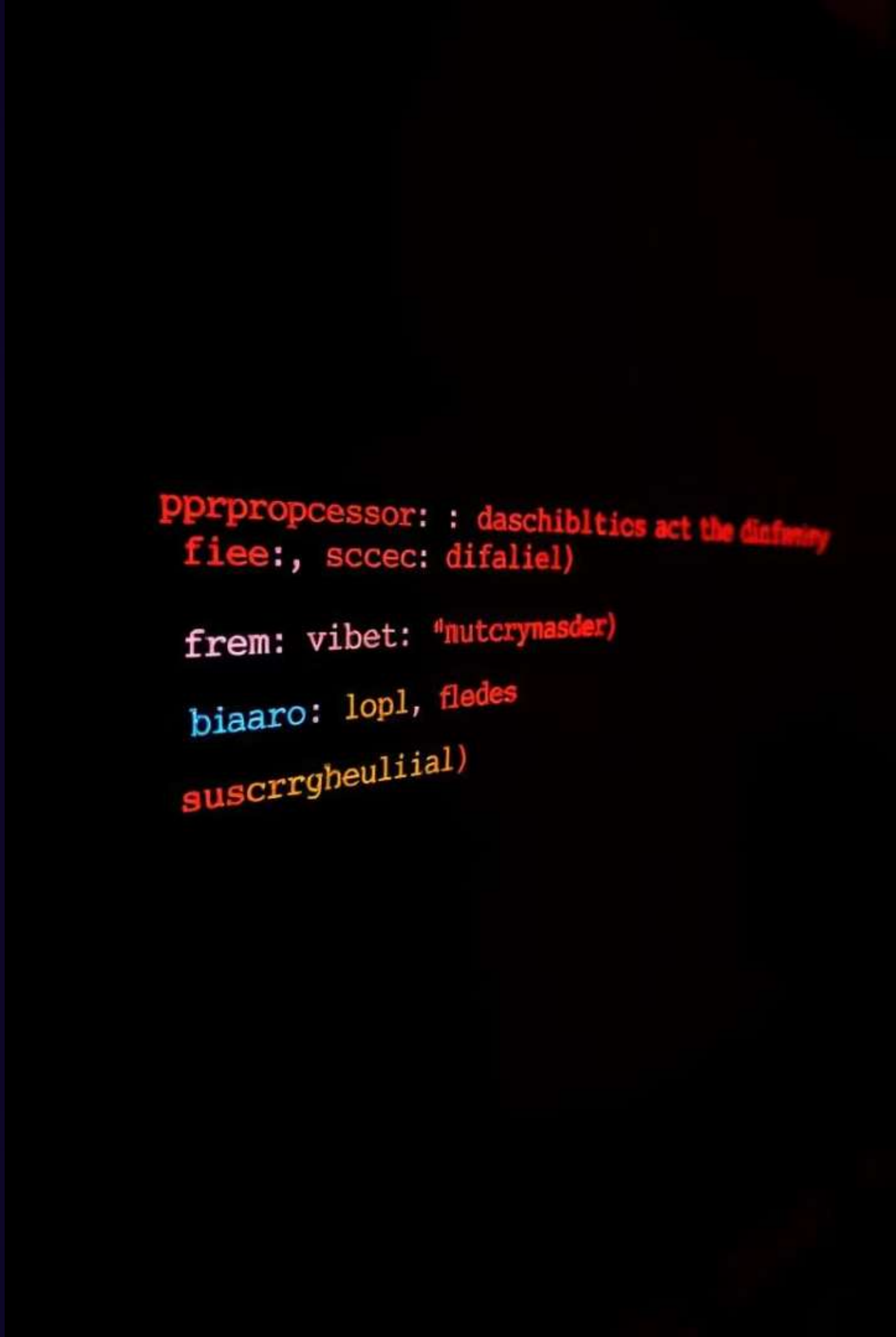
Preprocessor Directives and Macros

1 Preprocessor Directives

Instructions processed before compilation, influencing how the source code is treated.

2 Macros

Code snippets replaced with equivalent text during compilation, providing flexibility and optimization.


pprpropcessor: : daschibltics act the dinfamy
fiee:, sccec: difaliel)
frem: vibet: "nutcrynasder)
biaaro: lopl, fledes
suscrrgheuliial)

Macro Substitution

Macro Definition

```
#define PI 3.14159
```

Macro Usage

```
int main() {
  float circumference = 2 * PI * 5;
  printf("Circumference: %.2f\n", circumference);
  return 0;
}
Output: Circumference: 31.42
```

Macros with Arguments

Macro Definition

```
#define SQUARE(x) (x * x)
```

Macro Usage

```
int main() {
  int number = 7;
  int squared = SQUARE(number);
  printf("Squared value: %d\n", squared);
  return 0;
}
Output: Squared value: 49
```

Pitfalls of Macros

1 **Side Effects**

Macros can lead to unexpected results when applied to expressions with side effects (e.g., function calls), potentially causing unexpected behavior.

2 **Type Safety**

Macros lack type checking, making them prone to errors when used with different data types.

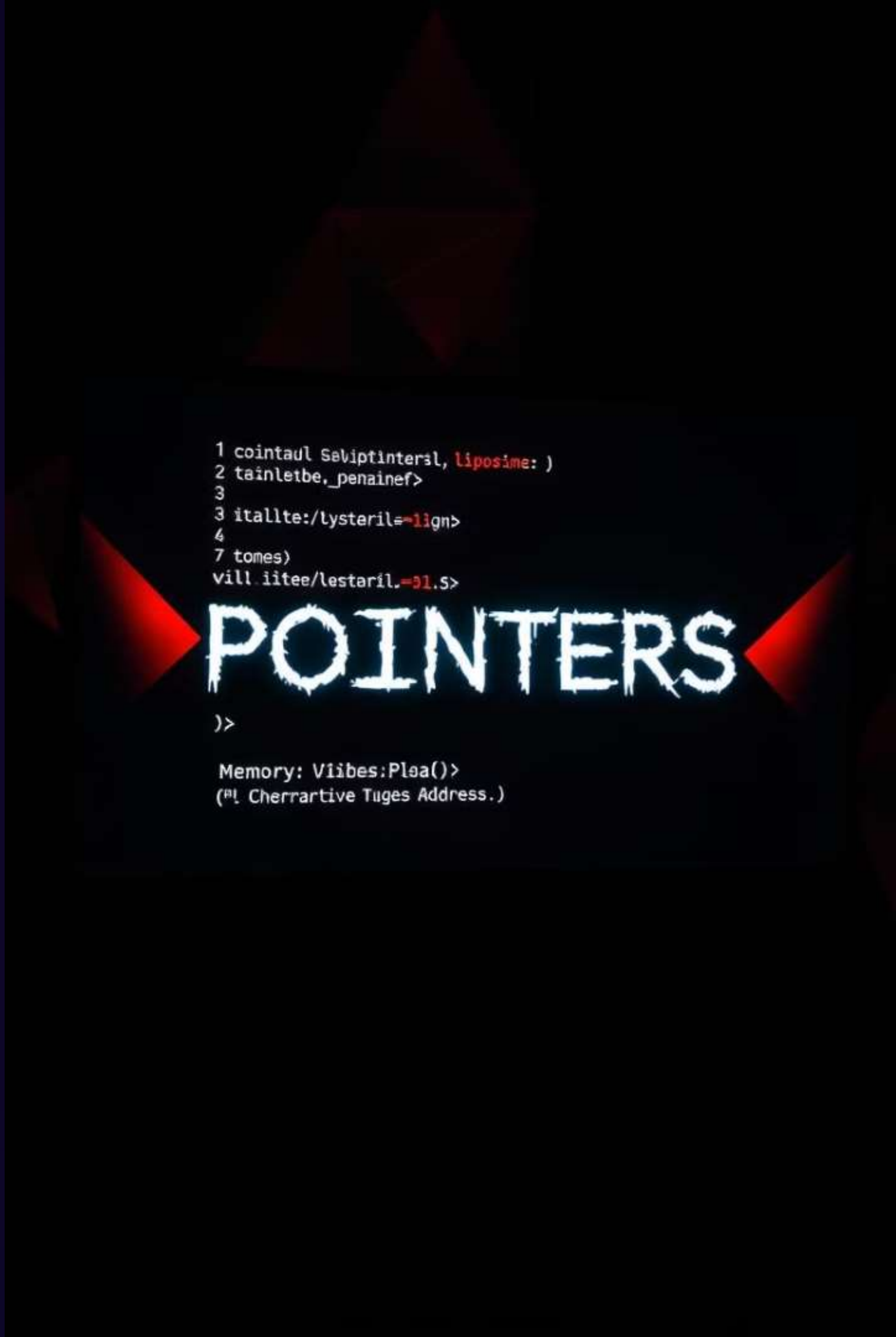
3 **Debugging Challenges**

Macros are expanded during pre-processing, making it challenging to debug issues directly in the source code.

Week 15

Advanced Pointer Techniques in C

This presentation will explore some advanced techniques for using pointers in C programming, including pointers to pointers, pointers to arrays, dynamic memory allocation, pointers and structures, and pointers as function arguments.



1 cointaul Seliptinteral, liposime:)
2 tainletbe,_penainef>
3 itallte:/lysteril=-1ign>
7 tomes)
vill iitee/lestaril.=91.5>
POINTERS
)>
Memory: Viibes:Plea()>
(ªl Cherrartive Tuges Address.)

Pointers to Pointers

Concept

A pointer to a pointer is a variable that stores the memory address of another pointer. It's like having a pointer that points to a pointer, allowing you to indirectly access the data pointed to by the original pointer.

Example

```
int x = 10;
int *ptr1 = &x;
int **ptr2 = &ptr1;

printf("%d\n", **ptr2); // Output: 10
```

Example: Swapping two integers using pointers to pointers

Code

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

int main() {
  int a = 10, b = 20;
  int *ptr1 = &a, *ptr2 = &b;
  swap(&ptr1, &ptr2);
  printf("a = %d, b = %d\n", a, b); // Output: a = 20,
b = 10
  return 0;
}
```

Explanation

The `swap` function takes two pointers to pointers as arguments. By dereferencing these pointers, it swaps the values pointed to by the original pointers.

Pointers to Arrays

Concept

A pointer to an array is a variable that stores the memory address of the first element of the array. It allows you to access and manipulate array elements directly using pointer arithmetic.

Example

```
int arr[] = {1, 2, 3, 4, 5};
int *ptr = arr;

printf("%d\n", *ptr); // Output: 1
printf("%d\n", *(ptr + 1)); // Output: 2
```

Example: Accessing array elements using pointers to arrays

Code

```
int main() {
  int arr[] = {10, 20, 30, 40, 50};
  int *ptr = arr;

  for (int i = 0; i < 5; i++) {
    printf("%d ", *(ptr + i)); // Output: 10 20
30 40 50
  }

  return 0;
}
```

Explanation

The code iterates through the array using a `for` loop and accesses each element using the pointer `ptr`. The pointer arithmetic `*(ptr + i)` accesses the element at the `i`-th index.

Dynamic Memory Allocation

Concept

Dynamic memory allocation allows you to allocate memory during program execution, unlike statically allocated memory, which is fixed at compile time. Functions like `malloc()`, `calloc()`, and `realloc()` provide this capability.

Example

```
int *ptr = (int *) malloc(sizeof(int));
if (ptr == NULL) {
  // Handle memory allocation failure
}

*ptr = 10;
printf("%d\n", *ptr); // Output: 10

free(ptr); // Release the allocated memory
```

Example: Dynamically allocating and initializing an array

Code

```c
int main() {
  int n;
  printf("Enter the size of the array: ");
  scanf("%d", &n);

  int *arr = (int *) malloc(n * sizeof(int));
  if (arr == NULL) {
    // Handle memory allocation failure
  }

  for (int i = 0; i < n; i++) {
    arr[i] = i + 1; // Initialize the array elements
  }

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

  free(arr); // Release the allocated memory
  return 0;
}
```

Explanation

The code first prompts the user for the array size, dynamically allocates memory using `malloc`, and then initializes the elements of the allocated array.

Pointers and Structures

Concept

You can use pointers to access and manipulate the members of a structure. This allows for efficient data manipulation and passing structures as arguments to functions.

Example

```
struct Student {
  char name[50];
  int rollno;
};

int main() {
  struct Student s1 = {"John Doe", 101};
  struct Student *ptr = &s1;

  printf("%s %d\n", ptr->name, ptr->rollno); //
Output: John Doe 101
  return 0;
}
```

Example: Accessing structure members using pointers

Code

```
struct Book {
  char title[100];
  char author[50];
  int pages;
};

void printBook(struct Book *book) {
  printf("Title: %s\n", book->title);
  printf("Author: %s\n", book->author);
  printf("Pages: %d\n", book->pages);
}

int main() {
  struct Book b1 = {"The Hitchhiker's Guide to the Galaxy",
"Douglas Adams", 42};
  printBook(&b1);
  return 0;
}
```

Explanation

The `printBook` function takes a pointer to a `Book` structure. It accesses the members of the structure using the arrow operator (`->`) to display the book details.

Pointers and Function Arguments

Concept

Passing pointers as arguments to functions allows you to modify the original data directly within the function. This is more efficient than passing copies of large data structures.

Example

```
void increment(int *num) {
  *num = *num + 1;
}

int main() {
  int num = 10;
  increment(&num);
  printf("%d\n", num); // Output: 11
  return 0;
}
```

Week 16

Chapter 16

Project Integration and Review

Beginner Projects

1. **Basic Calculator**
    1. Features: Perform basic arithmetic operations (+, -, *, /).
    2. Skills: Input handling, control structures, and functions.
2. **Unit Conversion Tool**
    1. Features: Convert between units (e.g., length, weight, temperature).
    2. Skills: Switch-case, modular programming.
3. **Student Record System**
    1. Features: Add, view, and delete student records.
    2. Skills: Arrays, file handling, basic menu systems.
4. **Number Guessing Game**
    1. Features: The program generates a random number, and the user guesses it with hints provided.
    2. Skills: Loops, random number generation, conditional statements.
5. **Simple Tic-Tac-Toe Game**
    1. Features: Two players can play a classic game of tic-tac-toe on a 3x3 grid.
    2. Skills: 2D arrays, nested loops, and conditional logic.

1. Library Management System
Features:
•Add, delete, and search for books.
•Issue and return books to/from students.
•View issued books and calculate late return fines.
•**Skills Used**:
•File handling (to store book and student data).
•Structures (to manage records).

2. Hospital Management System
Features:
•Add, search, and delete patient records.
•Manage doctor schedules and appointments.
•Generate bills for services provided.
• **Skills Used**:
•File handling (to save patient and doctor data).
•Structures and dynamic memory allocation.
•Input validation.

3. Inventory Management System
Features:
•Add, update, and remove products.
•Track inventory levels and notify when stock is low.
•Generate purchase and sales reports.
• **Skills Used**:
•Arrays or linked lists for inventory storage.
•File handling for saving and retrieving data.
•Functions for modularity.

4. Employee Management System
Features:
•Add, update, and delete employee details (e.g., name, position, salary).
•Calculate salary, including bonuses and deductions.
•Search for employees by name or ID.
Skills Used:
•Structures for employee records.
•File handling for persistence.