Course Overview and Motivation

1.1 Systems Programming

- **Definition**: Systems Programming involves writing software that provides services to the computer hardware, the operating system, or other lower-level parts of a system. Examples include device drivers, operating system components, and system utilities.
- **Focus in this course**:
  - **C Programming**: A low-level but powerful language.
  - **Process Management**: Understanding how processes are created, scheduled, and managed by the operating system.
  - **Concurrency**: Dealing with multi-threaded programs and synchronization.
  - **Memory Management**: Learning how to allocate, free, and manage memory.

1.2 The "Money Making Computer Problem"

- **Scenario**: A Python program takes 1 hour to run, rewarding you 1 cent upon completion. You want it to run faster to earn more money.
- **Key Strategies**:
  1. **Switch to a faster language** (e.g., C) for speed benefits.
  2. **Use faster hardware** (upgraded processors, more CPU cores).
  3. **Parallelize** the workload to run multiple pieces of computation simultaneously.

1.3 Parallelism Example

- Suppose a loop has 10 iterations, each taking 1 unit of time sequentially → 10 total units of time.
- If you split these iterations across 10 separate CPUs, all 10 can run simultaneously → **1 total unit of time**.
- **Takeaway**: C can be fast on its own, and parallel programming in C allows even greater speedups, limited by hardware constraints.

---

2. Course Logistics and Success Tips

- **Grading**: Likely based on labs, assignments, quizzes/exams, and participation.
- **Course Difficulty**:
  - Easier in the first few weeks while learning C basics.
  - Significantly harder after about 5–6 weeks, when more advanced topics (memory management, concurrency) appear.
- **Best Practices**:

  - Start coding assignments **early**.
  - Seek help when stuck:
    - **TA Office Hours**
    - **Piazza** (or online forums)
    - **Professor’s Office Hours**

---

3. Introduction to the C Language

3.1 History and Relevance

- **Origins**: Developed in 1972 by Dennis Ritchie at Bell Labs.
- **Early Use**: Primary language for implementing the UNIX operating system.
- **Modern Relevance**: Although old, C remains popular and widely used for system-level and performance-critical tasks. It frequently ranks near the top in programming language popularity.

3.2 Key Design Principles

1. **Simplicity**: Focus on procedures/functions rather than complex features like object-orientation.
2. **Low-Level Access**: Direct memory manipulation with pointers, no automatic garbage collection.
3. **Efficiency**: Produces fast, compact binaries. Offers fine-grained control over hardware resources.
4. **Typed Language**: Variables have data types (e.g., `int`, `char`, `float`), allowing for safer operations.

3.3 Procedural Paradigm

- **Procedural (Imperative) Approach**:
  - Centered on writing sequences of commands (procedures/functions).
  - The main unit of abstraction is the *function*.
- **Contrast to OOP**: No built-in classes, objects, or inheritance in C.

---

4. Basic C Program Structure

Below is a minimal “Hello World” in C. We will dissect each part.

c

CopyEdit
```c
#include <stdio.h>  /* Including the standard I/O library */

/* Multi-line comment:
   This program prints "Hello, world!" to the console */

int main(void) {
    // This line prints to the console
    printf("Hello, world!\n");
    return 0;
}
```

4.1 Comments in C

- **Multi-line**: Everything between `/*` and `*/` is ignored by the compiler.
- **Single-line**: Prefix with `//`, and everything on that line is ignored.

4.2 `#include <stdio.h>`

- **Preprocessor Directive**: Tells the compiler to include the standard I/O header file.
- **Header Files**: Contain function declarations, macros, and shared definitions.

4.3 `int main(void)`

- **Entry Point**: `main()` is where the OS starts executing your program.
- **Return Type**: `int`, commonly used to report success/failure to the OS.
- **Parameters**: `void` means "no arguments." Variants like `int main(int argc, char *argv[])` are also common.

4.4 `printf(...)`

- **Function**: Part of the C standard library (declared in `stdio.h`).
- **Purpose**: Prints formatted output to the standard output (usually the screen).
- **Escape Sequences**: `\n` moves to the next line.

4.5 `return 0;`

- **Exit Status**: Returns an integer to the operating system:
  - `0` typically indicates success.
  - Non-zero values indicate different kinds of failures or status codes.

5. Compiling and Executing C Programs

5.1 The Compilation Process

When you run something like `cc hello.c` or `gcc hello.c`, several steps occur under the hood:

1. **Preprocessing**: Handles `#include`, `#define`, and other directives.
2. **Compilation**: Translates the preprocessed C code into **assembly code**, which is then turned into **machine code**placed in `.o` (object) files.
3. **Linking**: Combines your `.o` files with external libraries (like the C standard library) into a single **executable**.

In a terminal:

bash
CopyEdit
```
# Compile and link in one step (common approach)
cc hello.c

# This generates an executable, typically named "a.out"
./a.out

# Alternatively, specify a custom name:
cc hello.c -o hello
./hello
```

5.2 Separate Steps (Optional but Useful)

- **Preprocessing Only**: `cc -E hello.c -o hello.i`
- **Compilation Only**: `cc -S hello.i -o hello.s` (assembly)
- **Assembling**: `cc -c hello.s -o hello.o` (object file)
- **Linking**: `cc hello.o -o hello` (final executable)

---

6. Additional C Examples and Concepts

6.1 Variables and Data Types

c
CopyEdit

```c
#include <stdio.h>

int main(void) {
    int x = 10;        // Integer variable
    float y = 3.14f;   // Floating-point variable
    char c = 'A';      // Character variable

    printf("x = %d\n", x);
    printf("y = %.2f\n", y);
    printf("c = %c\n", c);

    return 0;
}
```

- **`%d`**: Format specifier for integers.
- **`%f`**: Format specifier for floats/doubles. Here, `%.2f` prints 2 decimal places.
- **`%c`**: Format specifier for a single character.

6.2 Control Structures

6.2.1 If/Else

c
CopyEdit

```c
#include <stdio.h>

int main(void) {
    int number = 10;
    if (number > 0) {
        printf("Positive number\n");
    } else {
        printf("Non-positive number\n");
    }
    return 0;
}
```

6.2.2 Loops

c

CopyEdit

```c
#include <stdio.h>

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

6.3 Functions

c

CopyEdit

```c
#include <stdio.h>

// Function prototype
int addTwoNumbers(int a, int b);

int main(void) {
    int sum = addTwoNumbers(5, 7);
    printf("Sum is %d\n", sum);
    return 0;
}

// Function definition
int addTwoNumbers(int a, int b) {
    return a + b;
}
```

- **Prototype**: Declaration of the function signature before use.
- **Definition**: Actual implementation.

---

7. Why C is Key for Systems Programming

1. **Performance**: Close to the hardware, minimal runtime overhead.
2. **Control**: You handle memory allocation, pointers, direct hardware interfacing.
3. **Portability**: While close to hardware, standard C code can be compiled on many architectures.

---

8. Parallel Programming (Preview)

- **Motivation**: If sequential performance saturates, concurrency and parallelism can speed up tasks.
- **In C**: Often use libraries like **POSIX Threads (pthreads)**, or **OpenMP** for parallel loops, or **MPI** for distributed computing.
- **Example** (conceptual sketch with pseudocode; actual code would include pthread headers, etc.):

c
CopyEdit
```c
#include <stdio.h>
#include <pthread.h>

void* worker(void* arg) {
    // Perform some tasks
    printf("Hello from thread!\n");
    return NULL;
}

int main(void) {
    pthread_t thread;
    pthread_create(&thread, NULL, worker, NULL);
    // main thread can do something else here...
    pthread_join(thread, NULL);
    return 0;
}
```

- **Note**: This course will introduce the needed concurrency concepts in detail later.

---

9. Final Remarks and Course Tips

1. **Practice, Practice, Practice**: Write small programs every day to get comfortable with syntax and the compiler.
2. **Read Error Messages**: C compiler errors or warnings often guide you to the solution.
3. **Debugging**: Tools like `gdb` or `lldb` are invaluable for understanding crashes or incorrect behaviors.
4. **Memory Checks**: Utilities like `valgrind` help find memory leaks and invalid accesses, crucial in C programming.
5. **Ask Questions**: Utilize office hours, Piazza, and classmates to clarify misunderstandings quickly.

Putting It All Together (Mini Project Example)

Below is a small program that ties together basic input, arithmetic, conditionals, and printing. While simple, it showcases multiple features of C:

c

CopyEdit

```c
#include <stdio.h>

/*
   This program calculates the factorial of a user-provided integer.
   It demonstrates:
   - Input/Output
   - Loops
   - Basic error checking
*/

int main(void) {
    int n, i;
    long long factorial = 1;  // long long to hold large results

    printf("Enter a positive integer: ");
    if (scanf("%d", &n) != 1 || n < 0) {
        printf("Invalid input.\n");
        return 1;  // Non-zero return for error
    }

    // Calculate factorial
    for (i = 1; i <= n; i++) {
```

```
        factorial *= i;
    }

    printf("%d! = %lld\n", n, factorial);
    return 0;  // 0 indicates success
}
```

Explanation:

- **`scanf`**: Reads user input from standard input. Format specifier `%d` for integer.
- **Error Checking**: If `scanf` fails or if the user types a negative number, we print an error message and return `1`.
- **Factorial Computation**: Loop from 1 to `n`, multiplying an accumulator.
- **Printing**: `%lld` is the format specifier for `long long`.

Conclusion

- C remains a crucial, powerful language in systems programming, enabling direct interaction with hardware and high-performance code.
- This first lecture sets the stage: we've covered the course objectives, basic "Hello World," essential compilation steps, and hints of parallel programming.
- Future lectures will dive deeper into memory management, processes, threads, and more advanced C features.

Remember: **start early, practice often, and don't hesitate to seek help**. Mastering C and systems programming will pay off in speed, control, and a deeper understanding of how computers really work!

o1

You said:
Lecture 2 CSE 3100 Expressions and Basic Data Types.pptx
Presentation
do it again

ChatGPT said:

Here are detailed notes explaining each concept from **Lecture 2: Expressions and Basic Data Types in C**, with **clear explanations and C examples**.

📌 1. Static vs. Dynamic Typing

1.1 What is Typing in Programming?

Typing in programming languages refers to how variable types are handled.

1.2 Statically Typed vs. Dynamically Typed Languages

Type	Characteristics	Example Languages
Statically Typed	- Type checking is done at **compile-time**. - Errors are caught before execution. - Requires explicit type declarations.	C, C++, Java
Dynamically Typed	- Type checking is done at **run-time**. - More flexible but errors may only appear when the program runs. - No need to explicitly declare types.	Python, JavaScript

1.3 Advantages of Static Typing

1. **Faster Execution** – No need for type checking at run-time.
2. **Early Error Detection** – Many bugs are caught before execution.
3. **Memory Efficiency** – More control over how memory is allocated.

Example: Type Checking in Python (Dynamic)

python
CopyEdit

```python
x = "hello"  # String
x = 5        # Now an integer (allowed in Python)
```

Example: Type Checking in C (Static)

c
CopyEdit

```c
#include <stdio.h>

int main(void) {
    int x = "hello";  // ❌ Compilation error: Type mismatch
    return 0;
}
```

- C **does not allow** changing types dynamically.
- If the type is incorrect, the **compiler stops execution**.

📌 2. Expressions and Operators

An **expression** in C is a combination of variables, constants, and operators that evaluate to a value.

2.1 Operators in C

Category	Operators	Example
Arithmetic	`+`, `-`, `*`, `/`, `%`	`a + b`, `x * y`
Bitwise	`&`, `	`, ^, ~, <<, >>``
Logical	`&&`, `	
Assignment	`=`, `+=`, `-=`, `*=`, `/=`, `%=`	`x += 5` (same as `x = x + 5`)
Comparison	`==`, `!=`, `<`, `>`, `<=`, `>=`	`if (x == y)`
Increment/Decrement	`++`, `--`	`x++`, `--y`

2.2 Precedence and Associativity

- **Precedence** determines the order of execution.
- **Associativity** determines evaluation direction (`left-to-right` or `right-to-left`).

Example: Operator Precedence

c
CopyEdit

```
#include <stdio.h>

int main(void) {
    int a = 10, b = 2, c = 5;
    int result = a + b * c; // Multiplication (*) happens first
```

```
    printf("%d\n", result); // Output: 20
    return 0;
}
```

To enforce order, use **parentheses**:

c
CopyEdit
```
int result = (a + b) * c; // Addition happens first
```

📌 3. Memory for Data Types and Characters

3.1 Basic Data Types in C

Data Type	Description	Size (bytes)
`char`	Stores a single character	1 byte
`int`	Integer values	4 bytes
`float`	Floating-point numbers	4 bytes
`double`	Double-precision floating-point numbers	8 bytes

Example: Variable Declaration and Initialization
c
CopyEdit
```
#include <stdio.h>

int main(void) {
    char c = 'A';
    int num = 100;
    float pi = 3.14;
    double precise_pi = 3.1415926535;

    printf("char: %c, int: %d, float: %f, double: %lf\n", c, num, pi,
precise_pi);
    return 0;
```

```
}
```

Output:

cpp
CopyEdit
```
char: A, int: 100, float: 3.140000, double: 3.141593
```

📌 4. Constants in C

A **constant** is a fixed value that **cannot be modified** after declaration.

4.1 Declaring Constants

c
CopyEdit
```
#define PI 3.14159   // Macro constant (preprocessor)
const int max_users = 100;  // Constant variable
```

4.2 Types of Constants

Type	Example
Character	`'a'`, `'\n'`
Integer	`100`, `0xFF` (hex), `071` (octal)
Floating Point	`3.1415`, `1.3E20`

📌 5. Operators in Detail

5.1 Increment and Decrement

- **Post-Increment (`x++`)** → Use `x`, then increase.
- **Pre-Increment (`++x`)** → Increase `x`, then use.

Example: Post-Increment

c
CopyEdit
```
int i = 4;
int c = i++; // c = 4, then i becomes 5
```

Example: Pre-Increment

c
CopyEdit
```
int i = 4;
int c = ++i; // i becomes 5, then c = 5
```

📌 6. Assignment Operators

Operator	Equivalent
`a += b`	`a = a + b`
`a -= b`	`a = a - b`
`a *= b`	`a = a * b`
`a /= b`	`a = a / b`

Example: Using Assignment Operators

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

int main(void) {
    int a = 5;
    a += 10;  // Same as a = a + 10;
    printf("%d\n", a); // Output: 15
    return 0;
}
```

📌 7. Memory and Type Conversion

7.1 Finding Memory Size of Data Types

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

int main(void) {
    printf("Size of int: %lu bytes\n", sizeof(int));
    printf("Size of char: %lu bytes\n", sizeof(char));
    printf("Size of float: %lu bytes\n", sizeof(float));
    return 0;
}
```

7.2 Automatic Type Conversion

C **automatically** converts types in mixed operations.

Example: Integer to Float Conversion

c
CopyEdit
```c
int x = 10;
float y = 2.5;
float result = x + y; // x converted to float before addition
```

7.3 Explicit Type Conversion (Casting)

c
CopyEdit
```c
int x = 5, y = 2;
double z = (double)x / y; // Converts x to double before division
printf("%lf\n", z); // Output: 2.500000
```

📌 8. Boolean Data Type

- Older C versions **did not have** a boolean type.
- C99 introduced `_Bool`.
- Before C99, programmers used:

c
CopyEdit
```
#define TRUE  1
#define FALSE 0
```

Example: Using `_Bool`

c
CopyEdit
```
#include <stdbool.h>

int main(void) {
    _Bool flag = 1;
    printf("Flag: %d\n", flag); // Output: 1 (TRUE)
    return 0;
}
```

📌 Final Notes

- **C is statically typed**, meaning **all variables must have a declared type**.
- **Operators and precedence matter** to avoid unintended behavior.
- **Understanding memory sizes** is critical for optimization.
- **Type conversion can happen automatically**, but **explicit casting** avoids surprises.

These notes cover **Lecture 2** in **detailed depth**, with **C examples** for **every concept**. Let me know if you need further explanations or more examples! 🚀

4o

You said:
Lecture 3 Control Flow.pptx
Presentation

Here are **detailed notes** explaining **Lecture 3: Control Flow in C**, with **clear explanations and C examples**.

📌 1. Introduction to Control Flow

1.1 What is Control Flow?

Control flow determines the **order** in which statements are executed in a C program.

- By default, C programs execute **sequentially**.
- To change the flow, we use **control flow statements**:
  - **Branching** (`if`, `if-else`, `switch`)
  - **Loops** (`while`, `do-while`, `for`)
  - **Jump statements** (`break`, `continue`)

📌 2. Blocks in C

2.1 What is a Block?

A **block** (also called a **compound statement**) is a group of statements enclosed by `{}`.

c
CopyEdit
```c
{
    int x = 10;
    printf("%d\n", x);
}
```

- Variables declared **inside a block** are **local** to that block.
- Blocks can be **nested** inside one another.

📌 3. If Statements

3.1 Basic If Statement

Used to execute code **only if a condition is true**.

c
CopyEdit
```c
if (condition) {
    // Code runs only if condition is true
}
```

Example: Checking for a Positive Number

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

int main(void) {
    int num = 10;
    if (num > 0) {
        printf("Number is positive.\n");
    }
    return 0;
}
```

3.2 If-Else Statement

Allows execution of one block if the condition is **true** and another if it is **false**.

c
CopyEdit
```c
if (condition) {
    // Runs if condition is true
} else {
    // Runs if condition is false
}
```

Example: Even or Odd Number

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

```c
int main(void) {
    int num = 7;
    if (num % 2 == 0) {
        printf("Even number\n");
    } else {
        printf("Odd number\n");
    }
    return 0;
}
```

3.3 Else-If Ladder

When there are **multiple conditions**, use `else if`.

c
CopyEdit

```c
if (condition1) {
    // Code for condition1
} else if (condition2) {
    // Code for condition2
} else {
    // Default case
}
```

Example: Grading System

c
CopyEdit

```c
#include <stdio.h>

int main(void) {
    int marks = 85;

    if (marks >= 90) {
        printf("Grade: A\n");
    } else if (marks >= 80) {
        printf("Grade: B\n");
    } else if (marks >= 70) {
```

```
        printf("Grade: C\n");
    } else {
        printf("Grade: Fail\n");
    }
    return 0;
}
```

📌 4. Ternary Operator (? :)

A **shortcut** for `if-else`. **Syntax**:

c
CopyEdit
```
condition ? expression_if_true : expression_if_false;
```

Example: Minimum of Two Numbers

c
CopyEdit
```
int min = (a < b) ? a : b;
```

Equivalent to:

c
CopyEdit
```
if (a < b) {
    min = a;
} else {
    min = b;
}
```

📌 5. Loops in C

5.1 While Loop

Repeats **while** a condition is `true`.

c
CopyEdit
```
while (condition) {
    // Loop body
}
```

Example: Sum of First 100 Numbers

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

int main(void) {
    int i = 0, sum = 0;
    while (i < 100) {
        sum += i;
        i++;
    }
    printf("Sum: %d\n", sum);
    return 0;
}
```

5.2 Do-While Loop

- **Executes at least once**, even if the condition is false.
- Condition is checked **after** the loop runs.

c
CopyEdit
```
do {
    // Loop body
} while (condition);
```

Example: Number Input Until Positive

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

```c
int main(void) {
    int num;
    do {
        printf("Enter a positive number: ");
        scanf("%d", &num);
    } while (num <= 0);

    printf("You entered: %d\n", num);
    return 0;
}
```

5.3 For Loop

Best for counting-based loops.
Syntax:

c
CopyEdit
```c
for (initialization; condition; update) {
    // Loop body
}
```

Example: Print Numbers 1 to 10
c
CopyEdit
```c
#include <stdio.h>

int main(void) {
    for (int i = 1; i <= 10; i++) {
        printf("%d ", i);
    }
    return 0;
}
```

5.4 Break and Continue

Statement	Use Case
`break`	Exits the loop **immediately**.
`continue`	Skips the **current** iteration and continues.

Example: Skip Even Numbers

c

CopyEdit

```c
#include <stdio.h>

int main(void) {
    for (int i = 1; i <= 10; i++) {
        if (i % 2 == 0) continue;  // Skip even numbers
        printf("%d ", i);
    }
    return 0;
}
```

Example: Stop at 5

c

CopyEdit

```c
#include <stdio.h>

int main(void) {
    for (int i = 1; i <= 10; i++) {
        if (i == 5) break; // Stops the loop at 5
        printf("%d ", i);
    }
    return 0;
}
```

📌 6. Switch Statement

Used for **multi-way branching** when dealing with **fixed values**.

6.1 Syntax

c
CopyEdit
```c
switch (expression) {
    case constant1:
        // Code
        break;
    case constant2:
        // Code
        break;
    default:
        // Default case
}
```

6.2 Example: Day of the Week

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

int main(void) {
    int day = 3;

    switch (day) {
        case 1:
            printf("Monday\n");
            break;
        case 2:
            printf("Tuesday\n");
            break;
        case 3:
            printf("Wednesday\n");
            break;
        default:
            printf("Invalid day\n");
    }
```

```
    return 0;
}
```

- **`break` is important** – prevents **fall-through** to the next case.

📌 7. Common C Mistakes

Mistake	Description
`=` vs. `==`	`if (x = 5) {}` assigns instead of comparing. Use `==`.
Forgetting `break` in `switch`	Leads to **fall-through** (unwanted execution of next cases).
Dangling `else`	Always use `{}` to avoid ambiguity in nested `if-else`.

Example: Dangling Else Problem

c

CopyEdit

```c
if (x > 0)
    if (y > 0)
        printf("Both positive\n");
    else
        printf("Only x is positive\n"); // Misleading!
```

- Fix with `{}`:

c

CopyEdit

```c
if (x > 0) {
    if (y > 0)
        printf("Both positive\n");
    else
        printf("Only x is positive\n");
}
```

✅ Final Notes

- **If-Else**: For decision-making.
- **Loops**: `while`, `do-while`, `for` for repetition.
- **Switch**: For multiple choices.
- **Break/Continue**: For flow control inside loops.
- **Readability is crucial** – use **indentation** and **braces** `{}` for clarity.

🚀 **Mastering these control structures is essential for efficient C programming!**

4o

You said:
Lecture 4 Functions and Global Variables.pptx
Presentation

Here are **detailed notes** explaining **Lecture 4: Functions and Global Variables in C**, with **clear explanations and C examples**.

📌 1. Functions in C

1.1 Why Use Functions?

- **Improves readability**: Breaks code into manageable sections.
- **Encourages reusability**: A function can be used multiple times.
- **Enhances maintainability**: Easier to debug and update.
- **Organizes logic**: Reduces redundancy and simplifies debugging.

1.2 Function Basics

General Syntax:

c
CopyEdit
```c
return_type function_name(parameter_list) {
    // Function body
    return value;  // Optional
}
```

- **`return_type`**: Type of value returned (use `void` for no return).
- **`function_name`**: The function's identifier.
- **`parameter_list`**: Input values (can be empty).

Example: Function to Add Two Numbers

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

int add(int a, int b) {  // Function declaration & definition
    return a + b;
}

int main(void) {
    int sum = add(5, 7);
    printf("Sum: %d\n", sum);
    return 0;
}
```

Output:

makefile
CopyEdit
```
Sum: 12
```

📌 2. Return Statement in Functions

- **Terminates function execution**.
- **Returns a value** to the calling function.

Example: Using Return

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

int getNumber() {
    return 42;  // Returns 42
```

```
}

int main(void) {
    int num = getNumber();
    printf("Number: %d\n", num);
    return 0;
}
```

Output:

javascript
CopyEdit

```
Number: 42
```

📌 3. Function Prototypes

3.1 What is a Function Prototype?

A **function prototype** declares a function **before its definition** to inform the compiler about its existence.

c
CopyEdit

```
return_type function_name(parameter_list);
```

Example: Function Prototype

c
CopyEdit

```
#include <stdio.h>

// Function prototype
int square(int x);

int main(void) {
    printf("%d\n", square(5));  // Calls function
    return 0;
```

```
}

// Function definition
int square(int x) {
    return x * x;
}
```

- **Why use prototypes?**
  - Ensures the compiler knows about the function before it's used.
  - Helps catch **type mismatches** early.

📌 4. Function Parameters and Scope

4.1 Local vs. Global Variables

Type	Scope	Lifetime
Local Variables	Inside a function/block	Created when function starts, destroyed when function ends
Global Variables	Declared outside functions	Exists for the program's entire runtime

Example: Local Variables

c

CopyEdit

```
#include <stdio.h>

void myFunction() {
    int localVar = 10;  // Local variable
    printf("%d\n", localVar);
}

int main(void) {
    myFunction();
    // printf("%d", localVar);  // ❌ Error! localVar is not
accessible here
    return 0;
```

```
}
```

Local variables are accessible only inside the function where they are declared.

4.2 Global Variables

- **Declared outside all functions**.
- **Accessible from any function** in the file.
- **Retained for the entire program's duration**.

Example: Global Variable

c

CopyEdit

```
#include <stdio.h>

int globalVar = 100;  // Global variable

void printGlobal() {
    printf("%d\n", globalVar);
}

int main(void) {
    printGlobal();
    return 0;
}
```

Output:

CopyEdit

```
100
```

4.3 Static Variables

Type	Scope	Lifetime
Static Local Variables	Inside a function	Exists for the entire program's runtime, retains value between function calls

Static Global Variables	Declared outside functions	Only accessible within the file

Example: Static Local Variable

c

CopyEdit

```
#include <stdio.h>

void counter() {
    static int count = 0;  // Retains value between calls
    count++;
    printf("Count: %d\n", count);
}

int main(void) {
    counter();
    counter();
    counter();
    return 0;
}
```

Output:

makefile

CopyEdit

```
Count: 1
Count: 2
Count: 3
```

Example: Static Global Variable

c

CopyEdit

```
#include <stdio.h>

static int hiddenVar = 5;  // Cannot be accessed outside this file

void printHidden() {
    printf("%d\n", hiddenVar);
```

```
}

int main(void) {
    printHidden();
    return 0;
}
```

Static global variables are private to the file and prevent external access.

📌 5. Recursion in C

A function **calls itself** to solve problems **recursively**.

Example: Recursive Factorial Function

c

CopyEdit

```c
#include <stdio.h>

int factorial(int n) {
    if (n == 0) return 1;  // Base case
    return n * factorial(n - 1);  // Recursive call
}

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

Output:

yaml

CopyEdit

```yaml
Factorial of 5: 120
```

- **Recursion uses the execution stack** to store intermediate states.
- **Base case is required** to stop infinite recursion.

📌 6. Memory Organization

6.1 Process Memory Layout

Every process in C has the following **memory sections**:

Memory Region	Description
Code (Text) Segment	Stores compiled machine code.
Data Segment	Stores **global** and **static** variables.
Heap	Stores **dynamically allocated** memory (`malloc()`, `free()`).
Stack	Stores **function call frames**, **local variables**, and **return addresses**.

6.2 Memory Diagram

sql
CopyEdit

```
----------------------------
|    Stack (grows down)    |
|--------------------------|
|       Heap (grows up)    |
|--------------------------|
|  Global/Static Variables |
|--------------------------|
|      Code (Text)         |
----------------------------
```

📌 7. Function Call Stack

- Each function call **creates a new stack frame**.
- A function's **local variables** exist **only within its stack frame**.
- **When a function returns, its stack frame is destroyed**.

Example: Understanding Stack Frames

c

CopyEdit

```c
#include <stdio.h>

void func1() {
    int x = 10;
    printf("x in func1: %d\n", x);
}

void func2() {
    int y = 20;
    printf("y in func2: %d\n", y);
}

int main(void) {
    func1();
    func2();
    return 0;
}
```

Each function call has its own separate stack frame.

📌 8. Key Takeaways

✅ **Use functions** to organize and simplify code.
✅ **Use function prototypes** to avoid implicit declarations.
✅ **Scope matters**: Use **local** variables when possible.
✅ **Global variables should be avoided** unless necessary.
✅ **Static variables retain values** between function calls.
✅ **Memory layout includes Stack, Heap, Data, and Code sections.**
✅ **Recursion is powerful but uses stack memory.**

🚀 **Mastering functions and memory management is essential for efficient C programming!**

4o

You said:
Lecture 5 Arrays and Pointer Basics.pptx
Presentation

Here are **detailed notes** explaining **Lecture 5: Arrays and Pointer Basics in C**, with **clear explanations and C examples**.

📌 1. Arrays in C

1.1 What is an Array?

An **array** is a **linear, contiguous collection** of elements of the **same type**.

1.2 Declaring Arrays

c
CopyEdit
```
type array_name[size];
```

Example: Declaring an Integer Array

c
CopyEdit
```
int numbers[5];  // Declares an array of 5 integers
```

1.3 Initializing Arrays

c
CopyEdit
```
int x[5] = {1, 2, 3, 4, 5};  // Explicit initialization
int y[] = {10, 20, 30};      // Size inferred from values
int z[5] = {1, 2};           // Remaining elements are 0
int a[5] = {1, 2, [4] = 5};  // C99 feature: Initialize specific
indexes
```

📌 2. Array Properties

2.1 Arrays in Memory

- Arrays are stored **contiguously in memory**.
- **Array elements can be accessed via indexing (`array[index]`).**
- **Indexing starts at `0` and ends at `size - 1`.**

Example: Accessing Array Elements

c

CopyEdit

```
#include <stdio.h>

int main(void) {
    int arr[3] = {10, 20, 30};

    printf("First element: %d\n", arr[0]);  // Output: 10
    printf("Second element: %d\n", arr[1]); // Output: 20
    printf("Third element: %d\n", arr[2]);  // Output: 30

    return 0;
}
```

📌 3. Strings in C

3.1 Strings as Character Arrays

A **string** is a **character array** ending with a **null (`\0`) character**.

Example: String Initialization

c

CopyEdit

```
char s1[6] = {'H', 'e', 'l', 'l', 'o', '\0'};  // Explicit null
terminator
char s2[] = "Hello";  // Implicit null terminator
char s3[10] = "Hello";  // Extra space is filled with '\0'
```

Example: Printing a String

c

CopyEdit

```
#include <stdio.h>

int main(void) {
    char greeting[] = "Hello, World!";
    printf("%s\n", greeting);
    return 0;
}
```

📌 4. Arrays and Functions

4.1 Passing Arrays to Functions

- Arrays are **always passed by reference**.
- The function **receives a pointer** to the first element.

Example: Passing an Array to a Function

c

CopyEdit

```
#include <stdio.h>

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

int main(void) {
    int numbers[] = {10, 20, 30, 40};
    printArray(numbers, 4);
    return 0;
}
```

📌 5. Passing by Value vs. Passing by Reference

5.1 Passing by Value

- **Primitives (`int`, `char`, `float`, etc.)** are **passed by value**.
- A **copy** is passed, so changes inside the function **do not affect** the original variable.

Example: Passing an Integer (Value)

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

void modify(int x) {
    x = 100;  // This change won't affect the original value
}

int main(void) {
    int num = 50;
    modify(num);
    printf("Value: %d\n", num);  // Output: 50
    return 0;
}
```

5.2 Passing by Reference

- **Arrays** are passed by **reference**.
- Modifying an array in a function **affects the original array**.

Example: Passing an Array (Reference)

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

void modifyArray(int arr[], int size) {
    arr[0] = 100;  // Modifies the original array
}
```

```c
int main(void) {
    int numbers[] = {1, 2, 3};
    modifyArray(numbers, 3);
    printf("First element: %d\n", numbers[0]);  // Output: 100
    return 0;
}
```

📌 6. Pointers in C

6.1 What is a Pointer?

A **pointer** is a variable that **stores the memory address** of another variable.

Concept	Explanation
Value	The actual data stored at an address.
Address	The memory location where the value is stored.
Pointer	Stores the address of a variable.

6.2 Declaring and Using Pointers

c
CopyEdit
```c
type *pointer_name;
```

Example: Pointer Declaration

c
CopyEdit
```c
int a = 10;
int *p = &a;  // `p` stores the address of `a`
```

6.3 Pointer Operators

Operator	Meaning
`&`	Address-of operator (gets the memory address)
`*`	Dereference operator (accesses value at the address)

Example: Using Pointers

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

int main(void) {
    int x = 42;
    int *p = &x;  // Pointer stores address of x

    printf("Value of x: %d\n", x);      // Output: 42
    printf("Address of x: %p\n", &x);   // Address of x
    printf("Value of p: %p\n", p);      // Same as &x
    printf("Value at p: %d\n", *p);     // Dereferencing pointer
(output: 42)

    return 0;
}
```

📌 7. Pointer Arithmetic

- Pointers **support arithmetic operations**.
- **Advances memory locations based on type size**.

Operation	Effect
`ptr + 1`	Moves to next element in memory
`ptr - 1`	Moves to previous element

Example: Pointer Arithmetic

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

int main(void) {
    int arr[] = {10, 20, 30};
    int *p = arr;

    printf("%d\n", *p);      // Output: 10
    printf("%d\n", *(p + 1));  // Output: 20
    printf("%d\n", *(p + 2));  // Output: 30

    return 0;
}
```

📌 8. Returning Arrays from Functions

C **does not allow returning entire arrays**. Instead, return a **pointer** to the first element.

8.1 Wrong Approach (Doesn't Work)

c
CopyEdit
```c
int myArray[3] = {1, 2, 3};
return myArray;  // ❌ Invalid
```

8.2 Correct Approach (Returning a Pointer)

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

int* getArray() {
    static int arr[3] = {4, 5, 6};  // Static to persist after
function returns
    return arr;
}
```

```
int main(void) {
    int *p = getArray();
    printf("%d\n", p[0]);  // Output: 4
    return 0;
}
```

📌 9. Multidimensional Arrays

9.1 Declaring and Initializing

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

9.2 Accessing Elements

c
CopyEdit
```
printf("%d\n", matrix[1][2]);  // Output: 6
```

✅ Key Takeaways

✔ Arrays store elements of the same type in contiguous memory.
✔ Strings are character arrays ending with `\0`.
✔ Passing primitives copies values, but arrays are passed by reference.
✔ Pointers store addresses and can be dereferenced using `*`.
✔ Pointer arithmetic allows movement in memory.
✔ Functions cannot return arrays but can return pointers.

🚀 **Mastering arrays and pointers is essential for low-level memory manipulation in C!**

4o

You said:

Lecture 6 Memory Allocation.pptx
Presentation

Here are **detailed notes** explaining **Lecture 6: Memory Allocation in C**, with **clear explanations and C examples**.

📌 1. Memory Organization in C

1.1 Memory Segments

A C program uses **three main memory pools**:

Memory Segment	Purpose	Lifetime
Static/Global	Stores global/static variables and string literals	Exists for the entire program execution
Stack	Stores local variables and function call frames	Allocated and deallocated **automatically**
Heap	Stores dynamically allocated memory (`malloc()`, `calloc()`, `realloc()`)	Allocated and **manually deallocated** by the programmer

1.2 Memory Layout

sql
CopyEdit

```
----------------------------
|       Stack (grows down) |
|--------------------------|
|       Heap (grows up)    |
|--------------------------|
|  Static/Global Variables |
|--------------------------|
|       Code (Text)        |
----------------------------
```

- The **stack** grows **downward** (toward lower addresses).
- The **heap** grows **upward** (toward higher addresses).

📌 2. Stack Memory (Automatic Allocation)

2.1 Stack Characteristics

- Stores **local variables** and **function call frames**.
- **Fast** allocation and deallocation.
- **Automatically** managed.
- **Limited size** (default in GCC: **2MB**).

2.2 Example: Stack Allocation

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

void func() {
    int x = 10;  // Stored on stack
    printf("x = %d\n", x);
}

int main(void) {
    func();
    return 0;
}
```

- `x` is **created** when `func()` is called and **destroyed** when it returns.

📌 3. Heap Memory (Dynamic Allocation)

3.1 Why Use Heap Memory?

- Allows **dynamic allocation** at runtime.
- Useful when **size is unknown** at compile time.
- **Manually managed** by the programmer.

3.2 `malloc()` - Allocating Memory

- Allocates a **block of memory** of **size bytes**.
- Returns a **`void*` pointer** (must be typecast).
- Returns **`NULL`** if allocation fails.

Syntax:

c

CopyEdit

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

Example: Allocating an Integer

c

CopyEdit

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

int main(void) {
    int *ptr = (int *)malloc(sizeof(int));  // Allocate space for 1
int

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

    *ptr = 42;  // Store value in allocated memory
    printf("Value: %d\n", *ptr);

    free(ptr);  // Free allocated memory
    return 0;
}
```

3.3 `calloc()` - Allocating and Initializing Memory

- Similar to `malloc()`, but initializes memory **to zero**.
- Allocates space for **multiple elements**.

Syntax:
c
CopyEdit
```
void* calloc(size_t num, size_t size);
```

Example: Allocating an Integer Array
c
CopyEdit
```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int *arr = (int *)calloc(5, sizeof(int));  // Allocate space for 5
integers

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

    for (int i = 0; i < 5; i++) {
        printf("%d ", arr[i]);  // Prints 0 (calloc initializes to
zero)
    }

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

3.4 `realloc()` - Resizing Memory

- Resizes previously allocated memory.
- If **expanding**, new memory is **uninitialized**.
- If **shrinking**, excess memory is freed.

Syntax:
c
CopyEdit

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

Example: Expanding an Array

c
CopyEdit

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

int main(void) {
    int *arr = (int *)malloc(3 * sizeof(int));

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

    arr = (int *)realloc(arr, 5 * sizeof(int));  // Resize array to 5
elements

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

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

3.5 `free()` - Deallocating Memory

- Frees **heap-allocated** memory.
- **Avoids memory leaks**.
- **Do not free a pointer twice!**
- Always **set pointers to `NULL`** after freeing.

Example: Freeing Memory

c
CopyEdit

```c
#include <stdlib.h>

int main(void) {
    int *ptr = (int *)malloc(sizeof(int));
    free(ptr);  // Free memory
    ptr = NULL; // Prevents accidental access
    return 0;
}
```

📌 4. Tracking Array Size

4.1 Problem: How Do We Track an Array's Size?

- **Fixed-size arrays**: Use `sizeof()` in the **same function**.
- **Dynamically allocated arrays**: Keep track **manually**.

Example: Getting Fixed Array Size

c
CopyEdit

```c
#include <stdio.h>

int main(void) {
    int arr[5] = {1, 2, 3, 4, 5};
    int size = sizeof(arr) / sizeof(arr[0]);  // 5
    printf("Size of array: %d\n", size);
    return 0;
}
```

4.2 Why Can't We Use `sizeof()` in Functions?

c
CopyEdit

```c
void printArray(int arr[]) {
    printf("Size of array: %lu\n", sizeof(arr));  // Wrong! Size of
pointer, not array
}
```

- Arrays **decay into pointers** when passed to functions.
- The function only sees the **pointer to the first element**.

Solution: Pass the Size Explicitly

c

CopyEdit

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

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

📌 5. Multi-Dimensional Dynamic Arrays

5.1 Array of Pointers

c

CopyEdit

```
int **arr = malloc(rows * sizeof(int *));
for (int i = 0; i < rows; i++) {
    arr[i] = malloc(cols * sizeof(int));
}
```

5.2 Freeing a 2D Array

c

CopyEdit

```
for (int i = 0; i < rows; i++) {
    free(arr[i]);
```

```
}
free(arr);
```

✅ Key Takeaways

✔ **Stack**: Stores **local variables**, automatically managed.
✔ **Heap**: Used for **dynamic memory allocation**, manually managed.
✔ `malloc()` **allocates memory**, but leaves it uninitialized.
✔ `calloc()` **allocates and initializes** memory to zero.
✔ `realloc()` **resizes** allocated memory.
✔ `free()` **deallocates** heap memory.
✔ **Keep track of array sizes manually when using `malloc()`.**
✔ **Multi-dimensional dynamic arrays** require **arrays of pointers**.

🚀 **Mastering memory management is crucial for efficient C programming!**

4o

You said:
Lecture 7 More on Memory, Pointers and Structures.pptx
Presentation

Here are **detailed notes** explaining **Lecture 7: More on Memory, Pointers, and Structures in C**, with **clear explanations and C examples**.

📌 1. Arrays in Memory

1.1 Stack vs. Heap for Arrays

Array Type	Memory Location	Lifetime
`int arr[5];`	**Stack**	Deallocated when function exits
`int *arr = malloc(5 * sizeof(int));`	**Heap**	Exists until `free(arr)` is called

Example: Stack vs. Heap Allocation

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

void stackArray() {
    int arr[3] = {1, 2, 3};  // Stack allocation
}

int* heapArray() {
    int *arr = malloc(3 * sizeof(int));  // Heap allocation
    return arr;  // Must free later
}

int main(void) {
    int *arr = heapArray();
    free(arr);  // Must free allocated memory
    return 0;
}
```

💡 **Stack memory is faster, but heap memory is more flexible**.

📌 2. Pointer Arithmetic

2.1 Understanding Pointer Arithmetic

- **Adding an integer to a pointer moves it by `sizeof(type)`**.
- **Subtracting two pointers gives the number of elements between them**.

Example: Pointer Addition

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

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

```
    int *p = arr;  // Points to arr[0]

    printf("p points to: %d\n", *p);      // Output: 10
    p++;  // Moves pointer to the next element
    printf("p now points to: %d\n", *p);  // Output: 20

    return 0;
}
```

💡 **Pointer arithmetic respects type size (increments by `sizeof(type)`)**.

2.2 Pointer Subtraction

- **Subtracting two pointers** of the same type returns **the number of elements between them**.

Example: Pointer Difference

c
CopyEdit

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

int main(void) {
    int *p = malloc(10 * sizeof(int));
    int *last = p + 9;

    printf("Distance: %ld\n", last - p);  // Output: 9

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

💡 **Subtracting pointers gives the number of elements, not bytes**.

2.3 Pointer Comparisons

Operator	Meaning
`p1 == p2`	Check if pointers are equal
`p1 != p2`	Check if pointers are different
`p1 < p2`	Check if `p1` is before `p2` in memory
`p1 > p2`	Check if `p1` is after `p2` in memory

Example: Pointer Comparison

c

CopyEdit

```c
#include <stdio.h>

int main(void) {
    int arr[5] = {1, 2, 3, 4, 5};
    int *p1 = &arr[0], *p2 = &arr[4];

    if (p1 < p2) {
        printf("p1 is before p2\n");
    }

    return 0;
}
```

📌 3. Arrays and Pointers

3.1 Arrays as Pointers

- **Arrays decay into pointers** when passed to functions.
- **They are NOT the same**, but can often be used interchangeably.

Example: Array and Pointer Equivalence

c

CopyEdit

```c
#include <stdio.h>

int main(void) {
    int arr[] = {10, 20, 30};
    int *p = arr;  // Equivalent to &arr[0]

    printf("%d %d\n", arr[0], *p);  // Both print 10
    return 0;
}
```

💡 **`arr[i]` is the same as `*(arr + i)`**.

📌 4. Structures in C

4.1 What is a Structure?

A **structure (`struct`)** groups related variables of different types.

Defining a Structure

c

CopyEdit

```c
struct Student {
    char name[50];
    int id;
    float gpa;
};
```

4.2 Declaring Structure Variables

c

CopyEdit

```c
struct Student s1 = {"Alice", 101, 3.5};
struct Student s2 = {"Bob", 102, 3.8};
```

Example: Accessing Structure Members

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

struct Student {
    char name[50];
    int id;
    float gpa;
};

int main(void) {
    struct Student s = {"Alice", 101, 3.5};

    printf("Name: %s\n", s.name);
    printf("ID: %d\n", s.id);
    printf("GPA: %.2f\n", s.gpa);

    return 0;
}
```

4.3 Using `typedef` for Structures

- **Shortens structure names** for readability.

c
CopyEdit
```
typedef struct {
    char name[50];
    int id;
    float gpa;
} Student;

Student s1 = {"Charlie", 103, 3.7};
```

📌 5. Pointers to Structures

5.1 Declaring a Pointer to a Structure

c
CopyEdit
```
struct Student s = {"Alice", 101, 3.5};
struct Student *p = &s;
```

5.2 Accessing Structure Members via Pointer

- Use `->` instead of `.` for pointers.

Example: Pointer to Structure

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

typedef struct {
    char name[50];
    int id;
    float gpa;
} Student;

int main(void) {
    Student s = {"Alice", 101, 3.5};
    Student *p = &s;

    printf("Name: %s\n", p->name);
    printf("ID: %d\n", p->id);
    printf("GPA: %.2f\n", p->gpa);

    return 0;
}
```

💡 **`p->name` is equivalent to `(*p).name`.**

📌 6. Arrays of Structures

- **Useful for storing multiple records**.

Example: Array of Structures

c

CopyEdit

```
#include <stdio.h>

typedef struct {
    char name[50];
    int id;
    float gpa;
} Student;

int main(void) {
    Student students[2] = {{"Alice", 101, 3.5}, {"Bob", 102, 3.8}};

    for (int i = 0; i < 2; i++) {
        printf("Student %d: %s, ID: %d, GPA: %.2f\n",
               i + 1, students[i].name, students[i].id,
students[i].gpa);
    }

    return 0;
}
```

📌 7. Structure Memory Alignment

- **Structures are aligned for efficiency**.
- **Padding may be added** to match memory word boundaries.

Example: Structure Size with Padding

c

CopyEdit

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

```c
typedef struct {
    char c;
    int x;
} Test;

int main(void) {
    printf("Size of struct: %lu bytes\n", sizeof(Test));  // Usually 8 bytes
    return 0;
}
```

💡 **Padding can waste memory, but improves speed**.

✅ Key Takeaways

✔ **Stack memory is faster but limited. Use heap for large allocations.**
✔ **Pointer arithmetic works based on data type size.**
✔ **Arrays decay into pointers when passed to functions.**
✔ **Structures allow grouping of related variables.**
✔ **Use `->` for accessing structure members via pointers.**
✔ **Padding in structures affects memory layout and efficiency.**

🚀 **Mastering pointers, memory, and structures is crucial for writing efficient C programs!**

4o

You said:
Lecture 8 Linked Lists, Enums and Function Pointers(1).pptx
Presentation

Here are **detailed notes** explaining **Lecture 8: Linked Lists, Enums, and Function Pointers in C**, with **clear explanations and C examples**.

📌 1. Linked Lists in C

1.1 What is a Linked List?

A **linked list** is a **dynamic data structure** consisting of **nodes** that contain:

1. **Data** (value)
2. **Pointer to the next node** (for singly linked lists)

Feature	Arrays	Linked Lists
Memory Allocation	Fixed size	Dynamic size
Insertion/Deletion	Slow (shifting elements)	Fast (just pointer updates)
Random Access	O(1)	O(n)

1.2 Defining a Linked List Node

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

typedef struct Node {
    int data;
    struct Node* next;
} Node;
```

- The **`struct Node* next`** creates a **self-referential structure**.

1.3 Creating a New Node

c
CopyEdit
```c
Node* createNode(int value) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}
```

- **Dynamically allocates** memory for a new node.
- Initializes `data` and `next` pointer.

1.4 Adding a Node at the Beginning (Prepend)

c
CopyEdit
```c
Node* prepend(Node* head, int value) {
    Node* newNode = createNode(value);
    newNode->next = head;
    return newNode;  // New head of the list
}
```

💡 **Time Complexity:** O(1) (Constant time)

1.5 Adding a Node at the End (Append)

c
CopyEdit
```c
Node* append(Node* head, int value) {
    Node* newNode = createNode(value);

    if (head == NULL) return newNode;

    Node* temp = head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
    return head;
}
```

💡 **Time Complexity:** O(n) (Linear time, needs to traverse)

1.6 Finding the Last Node

c
CopyEdit

```c
Node* findLast(Node* head) {
    if (head == NULL) return NULL;
    while (head->next != NULL) {
        head = head->next;
    }
    return head;
}
```

1.7 Deleting a Node

c
CopyEdit

```c
Node* deleteNode(Node* head, int value) {
    if (head == NULL) return NULL;

    if (head->data == value) {  // If the head needs to be removed
        Node* temp = head;
        head = head->next;
        free(temp);
        return head;
    }

    Node* current = head;
    while (current->next != NULL && current->next->data != value) {
        current = current->next;
    }

    if (current->next != NULL) {
        Node* temp = current->next;
        current->next = temp->next;
        free(temp);
    }

    return head;
}
```

📌 2. Enumerations in C

2.1 What is an `enum`?

An `enum` (enumeration) is a **user-defined type** for **named integer constants**.

c
CopyEdit
```
typedef enum {
    RED,    // 0
    ORANGE, // 1
    YELLOW, // 2
    GREEN,  // 3
    BLUE    // 4
} Color;
```

2.2 Using `enum` in Code

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

typedef enum { RED, GREEN, BLUE } Color;

int main(void) {
    Color myColor = GREEN;

    if (myColor == GREEN) {
        printf("Green is selected.\n");
    }

    return 0;
}
```

💡 **Benefit:** Improves **code readability** compared to using integer constants.

2.3 Setting Custom Enum Values

c
CopyEdit
```
typedef enum {
    RED = 1,
    GREEN = 5,
    BLUE = 10
} CustomColor;
```

- **Values start from 1, 5, and 10 respectively**.

📌 3. Function Pointers in C

3.1 What is a Function Pointer?

A **function pointer** stores the **address of a function** and can be used to **call functions dynamically**.

c
CopyEdit
```
return_type (*pointer_name)(parameter_list);
```

3.2 Declaring a Function Pointer

c
CopyEdit
```
int (*funcPtr)(int, int);  // Pointer to function taking (int, int)
and returning int
```

3.3 Assigning a Function to a Pointer

c

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

int main(void) {
    int (*funcPtr)(int, int);  // Declare function pointer
    funcPtr = add;  // Assign function address
    printf("Sum: %d\n", funcPtr(3, 4));  // Call function via pointer
    return 0;
}
```

💡 **Function pointers allow dynamic function execution.**

3.4 Function Pointers as Arguments

c
CopyEdit
```
void execute(int a, int b, int (*operation)(int, int)) {
    printf("Result: %d\n", operation(a, b));
}

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

int main(void) {
    execute(5, 3, multiply);  // Passing function pointer
    return 0;
}
```

💡 **Useful for callback functions (like `qsort()`).**

3.5 `qsort()` and Function Pointers

`qsort()` is a **generic sorting function** in `<stdlib.h>` that uses **function pointers for comparisons**.

Syntax

c
CopyEdit
```c
void qsort(void *base, size_t nel, size_t width, int (*compare)(const
void *, const void *));
```

Example: Sorting Integers Using `qsort()`

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

int compareInts(const void *a, const void *b) {
    return (*(int*)a - *(int*)b);
}

int main(void) {
    int arr[] = {4, 2, 9, 1, 5};
    int size = sizeof(arr) / sizeof(arr[0]);

    qsort(arr, size, sizeof(int), compareInts);

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

    return 0;
}
```

💡 **`qsort()` works generically for any data type using function pointers!**

✅ Key Takeaways

✔ **Linked lists are dynamic and use pointers for efficient insertions/deletions.**
✔ **Enumerations (`enum`) improve code readability by replacing integer constants.**
✔ **Function pointers store function addresses and enable dynamic function execution.**
✔ **`qsort()` uses function pointers for generic sorting.**

🚀 **Mastering these concepts is crucial for efficient and flexible C programming!**

4o

You said:
Lecture 9 Input Output and Files.pptx
Presentation

Here are **detailed notes** explaining **Lecture 9: Input/Output and Files in C**, with **clear explanations and C examples**.

📌 1. Error Handling in C

1.1 What is `errno`?

- Many **C library functions can fail**.
- They return a **flag** (`-1` or `NULL`) to indicate failure.
- The **global variable `errno`** stores the **specific error code**.

1.2 Using `errno` and `perror()`

c
CopyEdit

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

int main(void) {
    FILE *fp = fopen("nonexistent.txt", "r");  // Try opening a
non-existent file

    if (fp == NULL) {
```

```
        perror("Error opening file");  // Print readable error message
        exit(EXIT_FAILURE);
    }

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

💡 **`perror()` prints a human-readable error message based on `errno`.**

📌 2. Files and Input/Output

2.1 What is a File?

- A **file** is a **sequential stream of bytes**.
- In Linux, **everything is a file** (including devices, pipes, etc.).

2.2 File Operations in C

Function	Description
`fopen()`	Opens a file
`fclose()`	Closes a file
`fread()` / `fwrite()`	Reads/Writes binary data
`fprintf()` / `fscanf()`	Reads/Writes formatted text
`fgetc()` / `fputc()`	Reads/Writes a single character
`fgets()` / `fputs()`	Reads/Writes a string
`ftell()` / `fseek()`	Moves the file pointer

2.3 Opening a File

c
CopyEdit
```
FILE *fopen(const char *filename, const char *mode);
```

Mode	Description
`"r"`	Read-only (file must exist)
`"w"`	Write (creates or overwrites file)
`"a"`	Append (creates file if not exists)
`"r+"`	Read and write (file must exist)
`"w+"`	Read and write (creates or overwrites)
`"a+"`	Read and append

Example: Opening a File

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

int main(void) {
    FILE *fp = fopen("data.txt", "r");  // Open file in read mode

    if (fp == NULL) {
        perror("Error opening file");
        return 1;
    }

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

📌 3. Reading from Files

3.1 Reading an Entire Line (`fgets()`)

c
CopyEdit
```c
char *fgets(char *str, int n, FILE *stream);
```

- Reads **at most `n-1` characters**.
- Stops at **newline (`\n`)** or **end-of-file (EOF)**.

Example: Reading Lines from a File

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

int main(void) {
    FILE *fp = fopen("data.txt", "r");
    char buffer[100];

    if (fp == NULL) {
        perror("Error opening file");
        return 1;
    }

    while (fgets(buffer, 100, fp) != NULL) {  // Read line by line
        printf("%s", buffer);
    }

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

💡 **Best for reading structured text files like logs or CSVs.**

3.2 Reading a Single Character (`fgetc()`)

c
CopyEdit
```c
int fgetc(FILE *stream);
```

- Reads **one character** from the file.

Example: Reading Characters from a File

c

CopyEdit

```
#include <stdio.h>

int main(void) {
    FILE *fp = fopen("data.txt", "r");
    int ch;

    if (fp == NULL) {
        perror("Error opening file");
        return 1;
    }

    while ((ch = fgetc(fp)) != EOF) {  // Read character by character
        putchar(ch);
    }

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

💡 **Useful for low-level character-based file processing.**

📌 4. Writing to Files

4.1 Writing a String (`fputs()`)

c

CopyEdit

```
int fputs(const char *str, FILE *stream);
```

- Writes a **null-terminated string** to the file.

Example: Writing a String

c

CopyEdit

```c
#include <stdio.h>

int main(void) {
    FILE *fp = fopen("output.txt", "w");

    if (fp == NULL) {
        perror("Error opening file");
        return 1;
    }

    fputs("Hello, World!\n", fp);

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

4.2 Writing a Single Character (`fputc()`)

c

CopyEdit

```c
int fputc(int c, FILE *stream);
```

- Writes **one character** to the file.

Example: Writing Characters

c

CopyEdit

```c
#include <stdio.h>

int main(void) {
    FILE *fp = fopen("output.txt", "w");

    if (fp == NULL) {
        perror("Error opening file");
        return 1;
```

```c
    }

    fputc('A', fp);
    fputc('\n', fp);

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

4.3 Writing Formatted Output (`fprintf()`)

c
CopyEdit
```c
int fprintf(FILE *stream, const char *format, ...);
```

- Works **just like `printf()`**, but writes to a file.

Example: Writing Formatted Data

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

int main(void) {
    FILE *fp = fopen("scores.txt", "w");

    if (fp == NULL) {
        perror("Error opening file");
        return 1;
    }

    fprintf(fp, "Name: %s, Score: %d\n", "Alice", 95);
    fprintf(fp, "Name: %s, Score: %d\n", "Bob", 88);

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

📌 5. Moving the File Pointer

5.1 `ftell()` - Get Current Position

c
CopyEdit
```c
long ftell(FILE *stream);
```

- Returns **current file position**.

5.2 `fseek()` - Move File Pointer

c
CopyEdit
```c
int fseek(FILE *stream, long offset, int origin);
```

`origin`	Meaning
`SEEK_SET`	Move to `offset` from **start**
`SEEK_CUR`	Move to `offset` from **current position**
`SEEK_END`	Move to `offset` from **end of file**

Example: Finding File Size

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

int main(void) {
    FILE *fp = fopen("data.txt", "r");

    if (fp == NULL) {
        perror("Error opening file");
        return 1;
    }
```

```
    fseek(fp, 0, SEEK_END);  // Move to end of file
    long size = ftell(fp);   // Get file size
    rewind(fp);              // Move back to beginning

    printf("File size: %ld bytes\n", size);

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

✅ Key Takeaways

✔ **Error handling (`errno`, `perror()`) helps diagnose file I/O issues.**
✔ **Use `fopen()` with correct modes (`r`, `w`, `a`, etc.).**
✔ **`fgets()`, `fgetc()` read files; `fputs()`, `fputc()`, `fprintf()` write files.**
✔ **Use `fseek()` and `ftell()` to move the file pointer.**
✔ **Always `fclose()` files to avoid memory leaks.**

🚀 **Mastering file I/O is essential for handling real-world data in C!**