Computer Science(UX Design) -Lecture 9

1. Origin & History of C

- **1972**: C was created by **Dennis Ritchie** at **Bell Labs**.
- It evolved from earlier languages: **ALGOL**, **BCPL**, and **B**.
- Designed primarily to build utilities for the **Unix** operating system.
- By the **mid-1970s**, C was widely used inside the Bell system.
- The **Unix kernel** was largely rewritten in C → this proved C's power and portability.
- **Johnson's Portable C Compiler (PCC)** enabled C to run on many platforms.
- **1978**: *The C Programming Language* by **Brian Kernighan & Dennis Ritchie** became the informal standard.
- **ANSI C (ANSI X3.159)** standardized the language.
- Around **1990**, C was adopted by **ISO**.
- Later standards include: **C99, C11, C17**.

2. Structure of a C Program

A typical C program consists of:

1. **Header Files (.h)**
   - Contain function declarations and macros.
   - Included via **preprocessor directives**.
   - Shared across multiple source files.
2. **main() Function**
   - Entry point of every C program.
   - Execution always starts here.
3. **Program Body**
   - Contains executable statements and logic.
   - May call other functions defined elsewhere.
4. **Return Statement**
   - Terminates execution.
   - Returns a value to the OS.
   - If return type is void, no value is required.
5. **Comments**
   - Used for explanation and documentation.
   - Supports:
     - Multi-line (/* */)
     - Single-line (//)

3. Core Features of the C Language

Performance & Efficiency

- Extremely **fast** and **resource-efficient**.
- Minimal abstraction → close to machine level.

Middle-Level Language

- Bridges **high-level logic** and **low-level hardware access**.

7. Optimization & Resource Management

- C exposes memory and CPU usage directly.
- Makes optimization concepts tangible.
- No automatic garbage collection.
- Programmer must manage memory manually:
  - Allocate when needed
  - Deallocate when done

This teaches discipline and precision.

8. Programming Paradigm of C

- **Imperative**
  - Describes *how* to do things step-by-step.
- **Procedural**
  - Code organized into functions.
  - Functions are modular and scope-bound.
- No built-in support for paradigms like:
  - Functional, Object-oriented (native)

Result:

- Easier to read & maintain
- Encourages good program design

9. C Standard Library & APIs

- Defined by ANSI/ISO standards.
- Provides:
  - Macros, Type definitions, Functions

Used for:

- String handling, Math, Input/output, Memory management, OS services

Libraries are exposed via **header files**.

API (Application Programming Interface)

- Defines:
  - What functions exist
  - How to call them
  - Data formats & conventions

10. Header Files in C

Types

1. **System Header Files**
   - Interface to OS services.
   - Enable system calls and standard libraries.
2. **User-Defined Header Files**
   - Interface between source files of your program.

Each #include copies declarations into the source during preprocessing.

11. Memory Management in C

- Functions exist for:
  - Allocating memory, Releasing memory, Resizing memory blocks
- Critical for:
  - Embedded systems, Low-resource platforms
- Memory leaks occur when allocated memory becomes unreachable.
- Poor memory handling can cause:
  - Crashes, Corruption, Undefined behavior
- Especially dangerous in systems like:
  - Embedded controllers, Safety-critical systems

12. Debugging & Analysis Tools

- **Lint / A-Lint**
  - Detect portability issues, Syntax errors, Poor coding practices
- Works during:
  - Compilation, Runtime

13. Industries & Real-World Usage of C

Embedded Systems

- Extremely small memory footprint.
- Example:
  - Arduino-like devices (very limited RAM & flash).
- Used in:
  - Cars, Washing machines, Routers, CCTV, Set-top boxes, BIOS firmware

Operating Systems

- Most OS kernels written in C.
- Example:
  - **Linux** kernel, Windows internals

Browsers & Infrastructure

- **Google** projects, **Google Chrome** (Chromium), Firefox (Mozilla)

Compilers

- C is ideal for compiler development due to:
  - Low-level control, High readability

Games & Graphics

- Early games written entirely in C.
- Engines/tools built with C/C++:
  - **Unity, Unreal Engine, Blender,** CryEngine, Buildbox, Cube

14. Final Takeaway

- C is one of the **oldest continuously used** programming languages.
- Its age is a strength:
  - Massive codebase, Endless learning resources, Proven reliability
- Learning C sharpens:
  - Logical thinking, Systems understanding, Performance awareness

Computer Science(UX Design) -Lecture 10

1. What is an IDE?
IDE = Integrated Development Environment
An IDE is a **software application** that bundles all tools needed for software development into one place.
You *can* write code using a plain text editor and compile manually, but IDEs exist to **dramatically increase productivity**.
Core Components of an IDE
An IDE typically includes:

- **Source Code Editor**
- **Compiler**
- **Debugger**
- **Build automation tools**
- **Plugins / extensions (language-specific)**

Purpose:

- Reduce setup time
- Catch errors early
- Speed up development
- Simplify repetitive tasks

2. Source Code Editor (Inside an IDE)
A source code editor is more than a text editor.
Key Features

- **Syntax highlighting** (visual cues for code structure)
- **Auto-completion**
- **Language-aware suggestions**
- **Real-time syntax checking**
- **Bug hints while typing**

Linting

- Automated code checking
- Warns about:
  - Syntax errors
  - Style guide violations
- Suggestions are **not always correct**
- IDE suggestions should not replace **understanding**

3. Debugger
A debugger is a tool used to **analyze program execution**.
What a Debugger Can Do

- Pause program execution
- Step through code **line by line**
- Inspect variable values

- Track:
  - Function calls
  - Control flow
  - Variable creation & modification
- Identify **semantic errors** (logic issues)

Key Debugging Concepts

Breakpoints

- Stop execution at a specific line

Stepping

- Execute one line at a time

Debuggers are essential for:

- Large programs
- Complex logic
- Understanding runtime behavior

4. Build Automation in IDEs

IDEs automate repetitive tasks such as:

- Compiling source code
- Linking binaries
- Running tests
- Packaging executables

This removes the need for manual command-line workflows for most use cases.

5. Intelligent Code Completion

Modern IDEs provide:

- **Context-aware suggestions**
- Reduced typing
- Fewer typos
- Faster development

Suggestions are based on:

- Language rules
- Code context
- Project structure

6. Source-to-Source Compilation (Transpilers)

Some IDEs support **transpilers**.

What is a Transpiler?

- Converts **source code → source code**
- Input and output languages are at the **same abstraction level**

Uses

- Backward compatibility
- Migrating codebases between languages
- Updating old code to newer language versions

⚠ Requires manual fixes in some cases.

7. Popular IDEs for C Development

Dev-C++

- Free & open-source
- Windows only
- Features:
  - Syntax highlighting
  - Code completion
  - Built-in GCC compiler
  - Debugger
  - Profiler
- Lightweight
- Used for demonstrations

Visual Studio

- Developed by Microsoft
- Windows-based
- Advanced features:
  - IntelliSense & Git integration
  - API support
  - Debugging at source & machine level
  - Deployment & database tools
- Very powerful
- Large disk and hardware requirements
- Paid license required for commercial use

Eclipse

- Free & open-source
- Windows, Mac, Linux
- Features:
  - Debugging
  - Compilation
  - Auto-completion
  - Refactoring
  - Static code analysis
- Beginner-friendly despite being powerful

Code::Blocks

- Free & open-source
- Runs on Windows, Linux, Mac
- Includes:
  - Compiler
  - Debugger
  - Profiling
  - Plugin support

Mobile IDEs

  - Android: IDEs with Git support available
  - iOS: C compiler apps available
- Limitations:
  - Small screens & Difficult for large projects

Other Noteworthy IDEs

- Sublime Text & NetBeans & Qt Creator & Brackets

8. Writing Code Without an IDE

- Possible using plain text editors
- Lose: Debugging & Auto-completion & Error highlighting
- Can be useful in **resource-constrained environments**

9. Version Control System: Git

What is Git?

Git is a **distributed version control system** used to track changes in source code.

- Created in **2005** by **Linus Torvalds**
- Developed for the **Linux** kernel
- Free and open-source

Why Git Matters

- Tracks history of code changes

- Enables collaboration
- Prevents conflicts
- Allows rollback to previous versions
- Supports parallel development

Distributed Model

- **Remote repository** (server)
- **Local repository** (each developer's machine)
- Every developer has a **full copy** of the codebase

This prevents:

- Overwriting others' work
- Conflicting edits
- Lost code

Industry Standard

- Not required to write code
- Essential for:
  - Team projects
  - Large codebases
  - Professional development
- Learning Git early reduces friction later

10. Choosing the Right IDE — Decision Factors

Cost

- Many IDEs are:
  - Free
  - Open-source
- Paid IDEs offer more features
- Choose based on **need**, not hype

Speed & Performance

- Bloated IDEs slow:
  - Startup
  - Compilation
  - Workflow
- Performance issues cost real time

System Requirements

- CPU & RAM & Storage
- Heavy IDEs can overwhelm weaker systems

Ease of Use

- Navigation should feel natural
- Complex UI reduces productivity

Compiler

- IDE must support a **stable, compatible compiler**
- Bundled compilers simplify workflow
- Unstable compilers cause:
  - Compilation failures
  - Unexpected behavior
- Switching compilers can solve unexplained bugs

11. Dev-C++ Installation (Windows Overview)

- Download from SourceForge > Run installer > Click through setup > Launch IDE > Ready to code

Android & iOS installs follow standard app-store installation steps.

12. Using Dev-C++ (Essentials Only)
Key Menus

- **File** → Create / Save source files
- **Execute** → Compile & Run (F11)
- **Edit** → Code editing
- **Search** → Find / Replace
- **View** → UI layout
- **Tools** → IDE configuration

13. Writing & Running First C Program
Workflow:

1. Create new source file
2. Write code
3. Save as **.c**
4. Compile
5. Run

Shortcut:

- **F11** → Compile + Run

Successful compilation + execution = program works.

14. Core Takeaways

- IDEs exist to **increase productivity**
- Debuggers are essential for understanding runtime behavior
- Auto-completion reduces errors but does not replace understanding
- Git is the industry standard for version control
- IDE choice depends on:
  - Project needs
  - System capability
  - Personal workflow
- Compiler stability matters more than features
- IDEs are tools — mastery comes from understanding the code

Computer Science(UX Design) -Lecture 11

1. Documentation Section (Top of the Program)
Purpose: The documentation section appears at the very top of a C program and exists purely for **human understanding**.
What it typically contains

- Program name & Author name(s) & Date of creation & Version number & Update history (if modified later)
- Brief description of what the program does
- References:
  - User guides & Reports & Websites & External code sources

Why it matters

- Helps others (and future-you) understand the program quickly
- Provides legal and ethical clarity
- Required in professional and commercial projects

2. Copyright, Attribution & Plagiarism

- Using someone else's code **without permission or acknowledgment** is plagiarism.
- For educational use, this is usually acceptable.
- For **commercial software**, copyright rules **must be followed strictly**.
- If code is reused:
  - Credit the original author
  - Follow stated usage conditions
  - Clearly document what was reused
- If unclear, contact the original developer (contact info is often in documentation).
- Copyright violations can lead to **legal action and heavy fines**.

Best practice

- Always reference copied code, even if not required.
- Never present others' work as your own.

3. Historical Note: B Language

- C had a predecessor called **B**. It was Developed by **Dennis Ritchie**
- B is essentially **typeless C**
  - No data types in function definitions
  - Local variables declared using auto

4. Preprocessor Directives
Execution timing

- Preprocessor directives are handled **before compilation**
- They are part of the compilation pipeline

Two Main Sections: **Link Section** & **Definition Section**

5. Types of Preprocessor Directives

There are **four main categories**:

- File Inclusion
- Macros
- Conditional Compilation
- Miscellaneous Directives

6. File Inclusion (#include)

Purpose

- Includes external header files containing function declarations and macros

Syntax:

```
#include <stdio.h>      // Searches system directories
#include "myfile.h"     // Searches project/current directory
```

Why headers are used

- Avoid forward-declaring functions repeatedly
- Organize large programs
- Support multi-file projects
- Enable team collaboration

Important note

- Including the same header file twice causes compilation errors
- Header files usually pair with source (.c) files
- Headers provide forward declarations

7. Macros (#define)

What is a macro

- A named block of code or value
- Replaced **before syntax checking**
- Text substitution, not function execution

Important Rules

- No termination symbol (;)
- Lasts until undefined using #undef
- Not affected by block scope

Macro Characteristics

- Can represent: Constants & Expressions & Parameterized calculations
- Example use: Calculating area & Reusable logic

Risks: Can cause unexpected behavior & Hard to read if overused & Debugging becomes difficult

Predefined Macros

- Begin with __ , Used for diagnostics and documentation

#error

- Forces compilation to stop , Displays custom error message

8. Declaration Section (Global Scope)

Purpose: Declares elements visible across the entire program

- What belongs here: Global variables, Function declarations (prototypes) & User-defined functions (before main)

Global Variables

- Exist for the entire runtime, Accessible from all functions
- Local variables override globals with the same name
- Reusing names across scopes is discouraged (hurts readability)

9. Functions in C

- **Definition:** A function is a self-contained block of code with a name

User-Defined Functions

- Declared above main
- **Improves:** Readability & Reusability & Organization
- Called by name with arguments
- **Execution:** Control jumps to function, Executes function body, Returns to calling point

10. main() Function

- Mandatory entry point of every C program
- Execution always starts here

Structure of main

  - **Declaration Section:** Local variables & Constants
  - **Execution Section:** Statements executed line by line

Return Value

- return 0 → successful execution
- Non-zero → error or abnormal termination

11. Program Execution Flow

- Code above main is integrated during compilation
- Only **one entry point** exists: main
- Program exits when main returns

12. Comments in C

Purpose: Explain code for humans & Improve readability and maintainability

Important Notes:

- Comments are removed **before compilation**
- Do not affect execution
- Part of coding standards and best practices

13. Types of Comments

Single-Line Comments

- Syntax: //
- Apply only to that line
- Used for brief explanations

Multi-Line Comments

- Syntax: /* ... */
- Can span multiple lines
- Compiler ignores everything inside.

Conventional Style

- Add * at the beginning of each line inside multi-line comments
- Improves readability & Common in real-world C codebases

14. Commenting Best Practices

- Use clear, narrative language
- Avoid shorthand
- Use sentence case
- Check spelling and grammar
- Place comments:
  - After the code block they explain (preferred)
- Avoid unnecessary blank lines
- Do not comment out large code blocks unless necessary
- Remove commented-out code in final versions unless documented
- Commented-out code during debugging is acceptable temporarily

15. Readability as a Core Principle

- Readability is as important as correctness
- Code is read more often than it is written
- Comments help:
  - Other developers
  - Reviewers
  - Your future self

16. Demonstrated Concepts (Practical)

- Adding a user-defined function above main
- Function does nothing until it is **called**
- Calling a function inserts its execution into program flow
- Inline comments do not change output
- Program output remains unchanged unless logic changes

17. Core Takeaways

- Documentation → humans
- Preprocessor → before compilation
- Headers → declarations, not execution
- Macros → text substitution
- Globals → lifetime = entire program
- Functions → modular execution
- main() → single entry point
- Comments → ignored by compiler, vital for clarity

Computer Science(UX Design) -Lecture 12

1. Tokens in C

A **token** is the **smallest meaningful unit** in a C program.

Programs are built by combining tokens, just like sentences are built from words.

Six Categories of Tokens

1. **Keywords**
2. **Identifiers**
3. **Constants**
4. **Strings**
5. **Special symbols (punctuators)**
6. **Operators**

During compilation:

- The **lexical analyzer** converts source code into a **stream of tokens**
- The **parser** uses token types to understand program structure

2. C Character Set

The **C character set** includes:

- Letters (uppercase and lowercase)
- Digits (0–9)
- Special characters (punctuation and symbols)

Role of Special Characters

- Used to define structure and meaning in code
- Examples:
  - { } → code blocks
  - [ ] → arrays
  - ; → statement termination
  - < > → system header inclusion
  - % → remainder in division

These characters help the compiler identify **where code sections start and end**.

3. Keywords

Definition

- Reserved words with **fixed meaning**
- Cannot be used as identifiers
- Always written in **lowercase**
- C has **32 keywords**

Common Keyword Groups

Control & Decision

- if, else, switch, case, default
- for, while, do
- break, continue, goto

Data Types

- int, float, char, double, long, void

Storage & Modifiers

- auto, extern, register
- signed, unsigned, const, volatile

Others

- return → exits function
- sizeof → size of variable/type
- struct, union, typedef
- enum

Case Sensitivity

- C is **case-sensitive**
- Using keywords with different casing technically works, but is **bad practice** and hurts readability

4. Identifiers

Definition: Identifiers are **user-defined names** for:

- Variables & Functions & Arrays & Structures & Other program elements

Rules for Identifiers

- Must start with: A letter (A–Z / a–z) or _
- Cannot start with:
  - A digit & Special characters
- Cannot contain:
  - Spaces or tabs & Punctuation or symbols
- Cannot be a keyword
- Case-sensitive (count ≠ Count)

Good naming improves readability; similar-looking names should be avoided.

5. Variables and Constants

Variables

- Represent **memory locations**
- Hold values that can change during execution
- Memory is allocated when declared
- Can be destroyed when no longer needed to save memory

Constants

- Values that **never change**
- Occupy memory like variables
- Used when modification is not allowed
- Example: π (22/7)

6. Punctuators (Special Symbols)

Definition

Punctuators give **syntactic meaning** but do not perform operations themselves.

Key Characteristics

- Often appear in pairs
- Missing one usually causes compilation errors
- Used to:
  - Group code & Mark boundaries & Define structure

Most Important Punctuator

- ; (semicolon)
  - Marks end of a statement
  - Missing semicolons are a common source of bugs
  - Program may compile but behave incorrectly

7. Operators in C

Operators instruct the computer to **perform operations**.

Main Categories

- **Arithmetic**: + - * / %
- **Relational**: < > <= >= == !=
- **Logical**: && || !
- **Bitwise**: & | ^ ~ << >>
- **Assignment**: = += -= *= /=

(Operators are covered in detail later in the course.)

8. Functions

Definition: A **function** is a named block of code that:

- Takes input
- Performs computation
- Produces output

Purpose

- Avoid code repetition
- Improve readability
- Improve maintainability

General Function Structure

- Return type
- Function name
- Parameters (arguments)
- Function body (code)

main() Function

- Mandatory entry point
- Program execution starts here
- Returns:
  - 0 → success
  - Non-zero → error

9. Control Structures

Control structures decide **when and how code executes**.

Conditional Execution

- if–else
- switch–case–default

Looping

- while → repeats while condition is true
- for → loop with initialization, condition, update
- do–while → executes at least once

10. Data Types (Overview)

- char → smallest addressable unit (stores characters)
- int, float, double, long
- unsigned long long → largest built-in type (memory heavy, often unnecessary)

Advanced types (covered later):

- Arrays & Pointers & Structures & Unions

11. Comments in C

Purpose

- Improve readability & Explain intent & Help maintain code

Important

- Comments are **removed before compilation**
- Do not affect execution

Types of Comments

Multi-line Comments

- /* ... */
- Can span multiple lines & Must be closed properly
- Forgetting to close comments can disable large portions of code

Single-line (Inline) Comments

- //
- Apply only to the rest of the line & Do not require closing

Rules & Best Practices

- Comments cannot be nested
- Place inline comments:
  - On their own line & Or at end of a code line
- Avoid shorthand
- Use clear, readable language
- Remove commented-out code in final versions unless justified
- TODO comments indicate unfinished or future work

12. Whitespace in C

Definition

Whitespace includes:

- Spaces & Tabs & Newlines & Comments

Compiler Behavior

- Whitespace is ignored after tokenization
- Used only to separate tokens
- Blank lines have no effect on compilation

Example

```
int num1;
```

- Requires space between int and num1 so the compiler can distinguish tokens

13. Core Takeaways

- Tokens are the foundation of C programs
- Keywords are reserved and lowercase
- Identifiers follow strict naming rules
- Semicolons terminate statements
- Functions organize logic
- Control structures manage flow
- Comments are for humans, not machines
- Whitespace improves readability, not execution
- Readability is a core programming principle

Computer Science(UX Design) -Lecture 13

1. Why Data Types Matter in C

- C is **hardware-aware**: data types map closely to machine architecture.
- Sizes of data types **depend on hardware** (CPU, word size, OS).
- C guarantees **minimum sizes**, not exact sizes.
- Choosing the wrong data type can cause:
  - Incorrect results
  - Memory waste
  - Performance issues
  - Undefined behavior

2. Classes of Data Types in C

C data types fall into **four classes**:

1. **Basic (Arithmetic) Types**
2. **Enumerated Types**
3. **Void**
4. **Derived Types**

3. Basic (Arithmetic) Data Types

3.1 Integer Types

Used to store **whole numbers** (no fractions).

	char	**short int**	**int**	**long int**	**long long int**
Min Size	8 bits	≥16 bits	≥16 bits	≥32 bits	≥64 bits
Notes	Stored as numbers (character encoding)	Rarely used	Common default	Larger range	"Very large, often overkill"

Signed vs Unsigned

- **Signed**: can store negative and positive values
- **Unsigned**: only non-negative values, larger upper range

3.2 char Data Type

- Stored in **1 byte (8 bits)** minimum
- Internally stored as integers using character encoding
- Characters:
  - 'A'–'Z' are contiguous & 'a'–'z' are contiguous & '0'–'9' are contiguous
- Can be manipulated using arithmetic

Variants:

- char
- signed char → typically -128 to 127
- unsigned char → typically 0 to 255

3.3 int Data Type

- Stores integers without fractions
- Minimum range: -32767 to 32767
- Division discards fractional part
  - Example: 22 / 7 = 3

Use float or double when precision is required.

3.4 Floating-Point Types

Used for **real numbers with fractions**:

- float
- double
- long double

Used when precision matters (e.g., scientific calculations).

4. Boolean Type in C

C does **not** have a true boolean type internally.

Options:

_Bool

- Built-in type
- Can only hold 0 or 1
- Any non-zero assignment becomes 1

bool

- Available via <stdbool.h>
- Defined as:
  - true → 1
  - false → 0

Internally, booleans are still integers.

5. Void Data Type

void means **no value** or **no type**.

Used in **three cases**:

1. **Function return type**
   - Function returns nothing
2. **Function parameters**
   - Function accepts no arguments
3. **Void pointers (void*)**
   - Generic pointer to any data type
   - Common in memory allocation
   - Must be cast before use

6. Fixed-Width Integer Types

- Provide **guaranteed sizes**
- Useful for portability
- Defined in <stdint.h>

- Examples:
  - int8_t
  - int16_t
  - int32_t
  - int64_t

7. Choosing the Right Data Type — Key Factors

1. Requirements
- What values must be stored?
- Range?
- Precision?

2. Future-Proofing
- Will values grow over time?
- Avoid assumptions tied to current hardware

3. Convenience
- Readability matters
- Avoid overly complex representations

4. Performance
- Larger types cost more memory
- But **premature optimization is bad**

5. Memory Constraints
- Critical in:
  - Embedded systems
  - Battery-powered devices

8. Derived Data Types

Derived types build on basic types to provide structure and flexibility.

9. Arrays

Definition: Collection of elements of the **same data type**
- Stored in **contiguous memory**

Key Properties
- Indexed starting at **0**
- Fixed size at declaration
- Cannot exceed declared size

Declaration

```
char initials[15];
```

- Holds 15 characters
- Access via index:

```
initials[2] = 'E';
```

10. Pointers

Definition: A variable that stores the **address of another variable**

Key Facts
- All pointers store memory addresses
- Address is a large hexadecimal number

- Pointer type determines **what it points to**, not address size

Declaration

```
int *ptr;
```

Null Pointers

- Assigned value 0
- Indicates pointer points to nothing
- Prevents accidental memory access

Pointer Variations

- Pointer arithmetic (++, --, +, -)
- Arrays of pointers
- Pointer to pointer
- Passing pointers to functions (pass-by-reference)
- Returning pointers from functions

⚠ Powerful but dangerous if misused.

11. Unions

Definition

- Multiple variables share the **same memory location**
- Only one member holds a value at a time

Key Properties

- Memory size = size of largest member
- Used for memory efficiency

Access

```
u.member;
```

Use case:

- Low-memory systems
- Multiple interpretations of same data

12. Structures (struct)

Definition

- Group of **different data types**
- Stored in **contiguous memory**

Use Cases

- Records
- File metadata
- Complex data modeling

Initialization Methods

- Direct initialization
- Designated initializers
- Assignment from another struct

Key Notes

- Field alignment depends on machine word size
- sizeof gives total memory used
- Structures can be:
  - Assigned
  - Passed by pointer
  - Accessed using . or ->

13. Type Qualifiers

Type qualifiers modify how a variable behaves.

const

- Value cannot be modified after initialization

volatile

- Value may change outside program control
- Prevents compiler optimizations

restrict

- Used with pointers
- Informs compiler pointer is sole reference
- Enables optimization

⚠ restrict and volatile **cannot** be used together.

14. sizeof Operator

- Returns size (in bytes) of:
  - Variables
  - Data types
  - Structures
- Hardware-dependent
- Essential for portability checks

15. Core Takeaways

- Data types are **contracts with hardware**
- Size is minimum-guaranteed, not fixed
- Smaller ≠ better, larger ≠ safer
- Choose types based on:
  - Meaning
  - Range
  - Precision
  - Context
- Arrays → contiguous memory
- Pointers → memory addresses
- Structs → grouped records
- Unions → shared memory
- Qualifiers → behavioral rules

Computer Science(UX Design) -Lecture 14

1. Variables and Memory

- Memory content is **not fixed**
- Variable values change because memory content changes
- When a variable is read:
  - The **value is copied** from memory
  - The copy is used in calculations

Variables are simply **named memory locations** formatted to hold a specific data type.

2. Variable Declaration

General Syntax

```
data_type variable_name;
```

- Memory size is determined by the **data type**
- Memory location is labeled using the **variable name**
- Variable declaration:
  - Reserves memory
  - Defines how that memory will be interpreted

Variables can be declared:

- At the start of a block
- Immediately before use

⚠ Placement affects **visibility (scope)**.

3. Types of Variables in C

C supports several variable categories:

1. **Local variables**
2. **Global variables**
3. **Static variables**
4. **Automatic variables**
5. **External variables**

(Static, automatic, and external are covered deeper later.)

4. Variable Naming Rules & Conventions

Rules (Mandatory)

- Must start with:
  - Letter or _
- Cannot:
  - Be a keyword
  - Contain whitespace
  - Contain special characters
- Case-sensitive

Naming Best Practices (Strongly Recommended)
Variable names should:

- Reflect purpose
- Improve readability
- Reduce mental load

Bad:

```
int CFDdrs;
```

Good:

```
int age;   int user_age;
```

5. Naming Conventions
Common Styles

snake_case

```
user_age
```

camelCase

```
userAge
```

Hungarian Notation

```
iCount // integer
sName // string
arrData // array
```

PascalCase

```
UserAge
```

Using conventions helps:

- Reviewers & Future maintainers & Yourself

6. Variables vs Constants
Variables

- Value **can change**
- Represent changing state

Constants

- Value **never changes**
- Assigned once
- Protected against modification

7. Why Use Constants
Constants:

- Prevent accidental modification
- Improve readability
- Communicate intent
- Reduce bugs in large codebases
- Are thread-safe (read-only)
- Often reused across program

8. Declaring Constants
Method 1: #define (Preprocessor)

```
#define PI 3.14
```

- No data type
- Text replacement
- No memory safety

Method 2: const keyword (Preferred)

```
const float PI = 3.14;
```

- Typed
- Stored in memory
- Compiler-enforced protection

- Best Practices: Assign value immediately & Use **UPPERCASE names** & Treat constants as immutable

9. Scope of Variables
Scope = Where a variable is visible
Local Variables

- Declared inside a block or function
- Accessible **only within that block**
- Invisible outside

Global Variables

- Declared outside all functions
- Accessible **anywhere in program**

Scope mistakes cause:

- Compilation errors
- Logic bugs

10. Function Parameters & Arguments

Terminology (Important)

- **Parameter** → variable in function definition
- **Argument** → actual value passed during function call

Also known as:

- Parameters → *formal parameters*
- Arguments → *actual parameters*

11. Formal vs Actual Parameters

Formal Parameters

- Defined in function declaration
- Act as local variables inside function
- Receive values at function call

Actual Parameters

- Values or expressions passed during call
  - Can be: Variables or Constants or Literals

12. Passing Arguments

Pass by Value

- Copy of value is passed
- Caller and callee have **separate variables**
- Changes inside function **do not affect caller**

Used when:

- Safety matters
- Multi-threaded systems
- Distributed systems

Pass by Reference (Pass by Address)

- Memory address is passed
- Caller and callee share same data
- Changes **affect original variable**

Used when:

- Large data structures
- Performance matters
- Modification is intended

13. Argument & Parameter Rules

- Number of arguments must match parameters
- Data types must match
- Parameters are local to the function
- Parameters exist only during function execution
- Variable-length parameter lists are exceptions

14. Variable Initialization

Why Initialization Matters

- Memory is reused
- Uninitialized variables may contain **garbage values**
- Causes:
  - Random behavior , Crashes & Hard-to-debug bugs

Best Practice: Always initialize variables:

```
int count = 0;
```

If not used immediately:

- Set to 0

15. Program Demo Overview (Even Numbers Program)

Problem

- Ask user for **10 numbers**
- Display **only even numbers**

16. Design Process

1. Flowchart
2. Pseudocode
3. Implementation
4. Testing
5. Commenting

17. Core Program Elements Used

- Documentation section
- #include <stdio.h>
- Arrays
- Index variable
- Functions
- Loops (while, for)
- Conditionals (if)
- Modulus operator %
- Input (scanf)
- Output (printf)
- Return values

18. Key Logic Used

Even Number Check

number % 2

- Remainder 0 → even
- Remainder 1 → odd

19. Array Handling

- Array size fixed (10 elements)
- Index starts at 0
- Index must be reset after use
- Prevents out-of-bounds errors

20. Why Reset Variables

Resetting variables:

- Prevents stale values
- Simplifies debugging
- Improves readability
- Makes execution predictable

21. Return Values

- int main() must return an integer
- return 0; → success
- Non-zero → error

22. Comments in Practice

- Inline comments used for explanation
- Comments:
  - Do not affect execution
  - Improve understanding
- Too many comments can hurt readability
- Use comments to explain **why**, not obvious **what**

23. Core Takeaways

- Variables = named memory locations
- Initialize everything
- Use meaningful names
- Prefer const over magic numbers
- Scope controls visibility
- Parameters ≠ arguments
- Pass by value = safe
- Pass by reference = efficient
- Arrays require careful indexing
- Reset reused variables
- Comments improve maintainability
- Logic clarity > clever tricks

Computer Science(UX Design) -Lecture 15

1. Purpose of Input and Output in C

- Computers need to know **what type of data** they are receiving or displaying.
- This is done using **format specifiers**.
- The programmer is responsible for telling the computer how to interpret data.
- Input → process → output is the core flow.

2. Required Header Files

- Input/output is **not built into core C**.
- You must include:
  - stdio.h → standard, portable, required
  - conio.h → non-standard, mostly MS-DOS / old compilers
- conio.h is **not portable** and not part of ISO C.

3. Input and Output Functions

Character-based

- getchar() → reads a single character
- putchar() → outputs a single character

String-based

- puts() → outputs a string (adds newline automatically)

General-purpose (most important)

- scanf() → input & printf() → output
- These handle:
  - characters, integers, floats, strings

4. Syntax Rules for scanf and printf

scanf

- Requires:
  - format specifier
  - variable **address** (use & for non-strings)
- Strings do **not** use &
- Example logic:
  - %d → &integer
  - %f → &float
  - %s → string name only

printf

- Uses format specifiers to replace values inside text
- Does **not** use &
- Output formatting is controlled by escape sequences

5. Common Format Specifiers

- %c → character

- %d / %i → integer
- %f → float
- %e / %E → scientific notation
- %g / %G → shortest float representation
- %hi → short signed integer
- %hu → short unsigned integer
- Length modifiers matter for correct memory interpretation

6. Escape Sequences (Formatting Symbols)

Used to control layout and readability:

- \n → new line
- \t → horizontal tab
- \\ → backslash
- \" → double quote
- \a → alert sound
- \b → backspace

Used to:

- align output
- separate columns
- improve presentation

7. Default Output Behavior

- Text is **right-aligned**
- Field width equals content width unless specified
- No formatting → output appears in a straight horizontal line

8. Program Design Workflow

You are expected to follow this order:

1. Problem statement
2. Pseudocode
3. Flowchart
4. Actual C code

Reason:

- Prevents logic errors
- Makes large programs manageable
- Any change in code should reflect back in pseudocode

9. Sample Program Logic (Student Average)

Inputs Required

- Name (string)
- Surname (string)
- Maths mark (int)
- English mark (int)
- Science mark (int)

Processing

- Average = (maths + english + science) / 3

Output

- Student name
- Student surname
- Average mark
- Displayed vertically and aligned

10. Variable Declaration Rules

- Strings must have:
  - fixed size
  - proper null termination
- Integers should be initialized
- Variables *can* be declared inline but:
  - not recommended
  - reduces readability
  - increases error risk

11. Screen Control

- system("cls") clears the screen on Windows
- Platform-dependent
- Avoid in portable programs

12. Output Alignment Techniques

- Use \t to align columns
- Multiple tabs may be required
- Works similarly to word processors
- Improves readability of results

13. Core Takeaways

- Always include stdio.h
- Match format specifier to data type
- Use & in scanf for non-strings
- Strings do not use &
- Initialize variables
- Escape sequences affect layout, not data
- Avoid non-standard headers unless required
- Follow design steps for clarity and correctness

Computer Science(UX Design) -Lecture 16

1. Arithmetic in C (Basics)

- Arithmetic in C follows the **same rules as mathematics**.
- Operators:
  - + addition
  - - subtraction
  - * multiplication
  - / division
  - % modulus
- Key difference from maths:
  - **Result is always stored on the left-hand side**
  - Example: total = 2 + 3;

2. Arithmetic Using Variables

- Operations can be done using:
  - constants
  - variables
  - user input
- Using variables allows **runtime flexibility**.

3. Increment and Decrement Operators

Standard form

- a = a + 1;
- a = a - 1;

Shorthand form

- a++ → increment
- a-- → decrement

4. Pre-Increment vs Post-Increment

Pre-increment (++a)

- Variable is incremented **before** use.
- Used value is the incremented value.

Post-increment (a++)

- Variable is used **first**, then incremented.

Risk

- Incorrect usage can cause:
  - infinite loops
  - unreachable code
  - compilation errors

5. Multiplication and Parentheses

- Multiplication uses *

- Parentheses must explicitly include *
- Brackets group expressions just like in maths
- C distinguishes:
  - () parentheses
  - {} braces
  - [] brackets

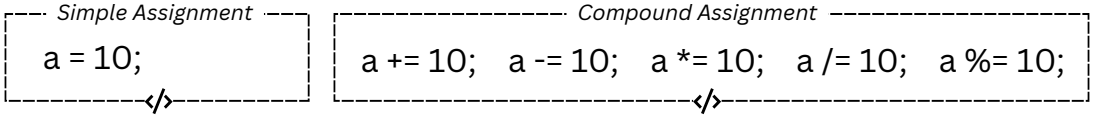
6. Division and Data Type Accuracy

- Division often produces **fractional results**
- Storing division results in int:
  - fractional part is lost
- Correct approach:
  - store division results in float
- Integers can be divided and stored in float

7. Logical Operators

- Used for conditional execution
- Operators:
  - && logical AND
  - || logical OR
  - ! logical NOT
- Work exactly like logic gates
- Useful when **multiple conditions** control execution

8. Assignment Operators



Benefits:

- Shorter code
- More readable
- Fewer CPU operations
- More efficient in loops and large programs

9. Performance Implications

- Compound assignments reduce:
  - instruction count
  - assembly code size
- Small changes scale massively in:
  - large codebases
  - tight loops
- Critical in system-level languages like C

10. Address, Pointer, and Size Operators

Address-of (&)

- Returns memory address of a variable
- Used in scanf
- Necessary because arguments are **passed by value**

Pointer (*)

- Used to access value at an address

Size-of (sizeof)

- Returns size (in bytes) of a data type or variable
- Hardware-dependent
- Essential for:
  - dynamic memory allocation
  - portable programs
  - array size calculation

11. Why & Is Required in scanf

- Function arguments are passed **by value**
- Without &, input is stored in a copy
- Using & ensures:
  - actual variable is modified
  - data persists after function returns

12. Expressions and Operators

- Every statement contains operators (directly or indirectly)
- Expressions are the **core of computation**
- Incorrect operator use leads to:
  - wrong results
  - hard-to-find bugs

13. Operator Precedence (BODMAS)

Evaluation order:

1. Parentheses
2. Multiplication / Division / Modulus
3. Addition / Subtraction
4. Assignment

- Same rules as mathematics
- Parentheses override precedence

14. Associativity

- Determines evaluation order **within the same precedence level**
- Most operators: left to right
- Assignment and ternary: right to left
- Compiler always follows rules — intent is programmer's responsibility

15. Quadratic Equation Demo (Concept)

- Uses:
  - arithmetic operators
  - precedence
  - math.h for sqrt
- Formula:
  - (-b ± sqrt(b² - 4ac)) / (2a)
- Error occurs when:
  - discriminant is negative
- Fix requires:
  - condition checking (covered later)

16. Operator Classification by Operands

- **Unary** → one operand (a++)
- **Binary** → two operands (a + b)
- **Ternary** → three operands (condition ? x : y)
- Helps:
  - debugging
  - correctness checking

17. Bitwise Operators

- Operate at **bit level**
- Faster than arithmetic
- Use less power
- Important for:
  - embedded systems
  - battery-powered devices
  - performance-critical code

18. The Core Takeaways

- Parentheses
- Prefix increment/decrement
- Postfix increment/decrement
- Unary operators
- Cast / address / sizeof
- Arithmetic (BODMAS)
- Relational operators
- Bitwise operators
- Logical operators
- Ternary operator
- Assignment operators
- Comma operator

- Core Things to Remember
  - Results are stored on the left
  - Data type choice affects accuracy
  - Increment operators are powerful but dangerous
  - Always respect precedence and associativity
  - Use float for division results
  - Use & correctly in input
  - Efficient code matters at scale
  - Parentheses remove ambiguity — use them