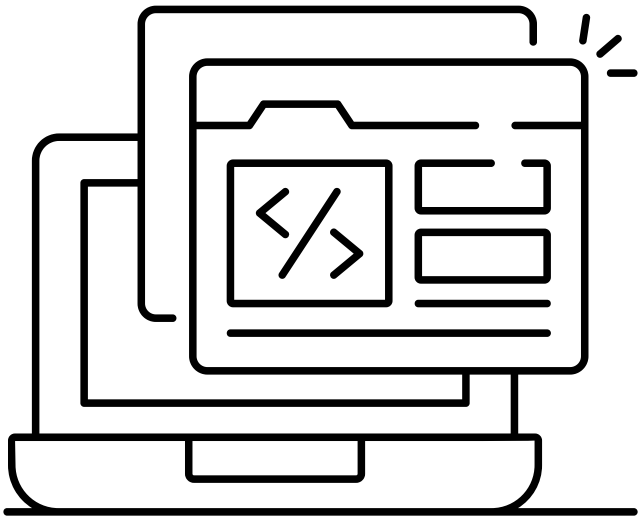
Computer Science (UX Design)



This document contains the notes I prepared during my 8th semester internship, which I joined with the intention of learning and gaining experience in UX Design.

However, throughout the internship, the content appeared to focus more on Computer Science concepts rather than UX Design. Even the stored file names reflected this, as they were labeled under "Computer Science," further indicating a stronger emphasis on CS topics than on UX Design.

Table of Contents

Lecture (Page No)

All related information and resources can be found at the following link:

GitHub → Runarok → Guides → Code Quest → Events → UX Design

Computer Science(UX Design) -Lecture 1

1. What Computer Science *Actually* Is

Computer Science (CS)

→ The study of **computational systems**

→ How problems are **modeled**, **solved**, and **automated** using computers

Core idea

- Computers don't "think" like humans
- They **follow logic and instructions**
- CS teaches us **how to design those instructions**

Reality vs theory

- **On paper**: Theory → Development → Deployment
- **In practice**: Continuous improvement, updates, optimization

Every update on your phone is computer science in motion.

2. Why Computer Science Matters

Key advantages of computers

1. **Automation**
   - Repetitive tasks handled instantly
   - Example: billing, payroll, checkout systems
2. **Speed**
   - Massive computations done in seconds
   - Searching the internet vs manual lookup
3. **Accuracy**
   - Eliminates human fatigue errors
   - Caveat: **Garbage In → Garbage Out**
4. **Scalability**
   - One system can serve millions simultaneously

3. Core Terminology (You'll See These Forever)

Programming

- Designing **instructions** a computer can follow

Coding

- Writing those instructions in **specific languages**

Programming Language

- A formal system for communicating with computers
- Example: Python, Java, C, etc.

4. Mathematics & Computing Relationship

- Nature itself is **mathematical**
- Computers model:
  - Motion, Patterns, Proportions, Geometry, Probability

Examples

- Flight simulation, Facial recognition, Physics engines, Weather forecasting, Biomechanics

This is why pilots can train **without flying**.

5. Major Fields of Computer Science

Foundational Areas

- Artificial Intelligence (AI), Computer Systems & Networks, Database Systems, Software Engineering, Human-Computer Interaction (HCI), Computer Vision & Graphics, Bioinformatics, Theory of Computation (algorithms & complexity)

Key insight

- You **don't learn everything**
- You **specialize deeply** in one or two

6. Historical Evolution of Computers

Early Computing

- **Abacus** – simple calculations (2000+ years ago)

Mechanical Era

- **Charles Babbage**
  - Built the **Difference Engine**
  - Introduced:
    - Storage
    - Mechanical computation
    - Output mechanisms

First General-Purpose Computer

- **ENIAC**
  - 30 tons, 18,000 vacuum tubes
  - 160 kW power, Programmable
  - Used for:
    - Weather
    - Physics, Military calculations

Architecture Breakthrough

- **John von Neumann**
  - Introduced **stored-program architecture**
  - Foundation of all modern computers

Transistor & IC Era

- Transistors replaced vacuum tubes
- Integrated Circuits (ICs):
  - Smaller
  - Faster
  - More reliable

Programming Evolution

- Assembly → High-level languages
- COBOL for business
- Operating systems introduced

7. Generations of Computers (High Level)

1. Vacuum tubes
2. Transistors
3. Integrated circuits
4. Microprocessors
5. Modern computing (parallelism, graphics, AI)

8. Artificial Intelligence

- Definition: Programming machines to **mimic human intelligence**

- Foundational Figure: **Alan Turing**
- Turing Test: If a machine fools humans ≥30% of the time in conversation → considered intelligent
- Notable Event: 2014: Eugene Goostman chatbot claimed to pass Turing Test

9. Machine Learning (ML)

What it is

- Systems that **learn from data**
- Not explicitly programmed for every rule

Core components

- Data, Model, Training, Prediction

Applications

- Spam filtering, Search engines, OCR, Fraud detection, Computer vision

10. Machine Learning Models

Artificial Neural Networks

- Inspired by the human brain
- Nodes = neurons
- Connections = synapses

Deep Learning

- Neural networks with **many layers**
- Used in: Vision, Speech recognition

Genetic Algorithms

- Inspired by natural selection
- Uses: Mutation, Crossover, Fitness evaluation

Decision Trees

- Tree-based logic
- Branches = conditions
- Leaves = outcomes

Federated Learning

- Learning from **distributed data**
- Example: keyboard prediction without uploading personal data

11. Internet of Things (IoT)

- Definition: Interconnection of physical devices with intelligence

Examples

- Smart speakers, Smart watches, Health monitors, Home automation systems

12. Quantum Computing

- Based on **quantum mechanics**
- Uses qubits instead of bits
- Potential to outperform classical computers massively
- Currently: Mostly theoretical, Some working prototypes exist

13. Real-World Applications of Computing

Business & Finance

- Banking systems, Stock markets, Payroll, Budgeting, Financial analytics

Communication

- Digital calls, Messaging platforms, Collaboration tools

Manufacturing

- Robotics, Precision automation

Consumer Electronics

- Smartphones, Smart TVs, Wearables, AI chips in devices

Healthcare & Accessibility

- Assistive technologies, Brain-controlled prosthetics, Medical simulations

Science & Research

- Weather prediction, Space exploration, Molecular modeling, Physics simulations

14. Career & Opportunity Outlook

- Web development, Software engineering, Data analysis, Mobile development, Systems design

Demand for computing skills is **accelerating**, not slowing.

15. Core Takeaway

- Computer science = problem-solving at scale
- Data is the fuel
- Algorithms are the logic
- Computers amplify human capability
- The field has no fixed ceiling

Computer Science(UX Design) -Lecture 2

1. Information Processing (Core Concept)

Definition

- **Data** → raw facts
- **Information** → processed data presented meaningfully

Conceptual analogy

- **Cake baking**
  - Ingredients = data
  - Recipe = program
  - Baker = CPU
  - Cake = information

2. Information Processing Cycle

2.1 Data Collection

- First step of processing
- Data sources:
  - Environment (via sensors)
  - Input devices (keyboard, mouse, touch, camera)
  - Files & databases
  - Previously processed data
  - Computer-generated data

Key idea

- Data must be in a **usable format**
- Raw data is **converted to machine-readable form**

2.2 Data Processing

- Done by **software programs**
- Executed by **CPU**
- Programs = instructions that define:
  - What data to use
  - Order of operations
  - Duration & conditions

Performance depends on:

- Processor speed
- Bus size
- Cache size
- Architecture

2.3 Storage

Temporary Storage (Memory)

- **RAM (Random Access Memory)**
- Volatile → cleared when power is off
- Stores currently used data & programs

Permanent Storage

- Drives (HDD, SSD)
- Flash storage
- Phone storage
- Retains data after shutdown

Storage formats

- Files, Databases

2.4 Output (Information Presentation)

- Output forms: Visual (screens, print), Audio (sound, music), Physical actions (robots, machines)

3. Operating System (OS)

3.1 Definition

An **Operating System** is a collection of procedures that:

- Allows **multiple users** to share computing resources
- Manages **hardware & software coordination**

3.2 Core Role of OS

- Acts as **middle layer** between:
  - Hardware, Applications, Users

4. Boot Process (System Startup)

Key terms

- **Boot** = starting the system
- Origin: “pulling oneself up by bootstraps”

Boot sequence

1. Power ON
2. **BIOS** (Basic Input Output System)
   - Initializes hardware
   - Performs checks
3. **Boot Loader**
   - Locates OS
   - Loads kernel into RAM
4. **Kernel takes control**
5. Drivers, services, applications loaded
6. Runtime state reached (system usable)

Rebooting

- **Hard reboot**: power cut
- **Soft reboot**: RAM cleared, power maintained

5. Kernel (Heart of the OS)

Definition:

- Core component of OS
- Always resident in memory
- Full control over system resources

Responsibilities

- Memory control
- Process scheduling
- Device communication
- Security enforcement

6. OS Functional Responsibilities

1. Memory Management
2. Processor Scheduling
3. Device Management (via drivers)
4. File Management
5. Security & Permissions
6. Resource Allocation
7. Error Handling
8. Application Platform

7. OS Architecture Layers

- Hardware, Kernel, Shell (interface), Utilities & Applications

8. Kernel Types

8.1 Monolithic Kernel

- Kernel & services in same space
- Fast execution
- Larger OS
- Example: **Linux**

8.2 Microkernel

- Minimal kernel
- Services run in user space
- Modular, secure
- Message-based communication

8.3 Hybrid Kernel

- Combines monolithic speed + microkernel modularity
- Example: **Microsoft Windows**

8.4 Nanokernel

- Extremely small
- Hardware abstraction only

8.5 Exokernel

- Minimal abstractions
- Application-specific customization

9. Kernel Space vs User Space

- **Kernel Space**
  - Privileged
  - Direct hardware access
- **User Space**
  - Applications
  - Isolated for safety

Purpose: prevent total system failure if an app crashes.

10. Device Drivers

What they are

- Software layer between hardware & OS
- Abstract hardware as files

Why they exist

- Prevent each app from managing hardware itself
- Improve efficiency & compatibility

Example

- Brightness change:
  - OS writes value → driver
  - Driver translates to hardware signal

11. Operating System Categories

- Three major families: Unix-like systems, Linux, Windows

12. Unix-Like Systems

Types:

1. **Genetic Unix** – original codebase
2. **Branded Unix** – commercial variants
3. **Functional Unix** – Unix-like behavior

13. macOS

- Unix-derived (BSD roots)
- Developed by **Apple Inc.**
- Written in:
  - C, C++, Objective-C, Swift
- Known for:
  - Stability, Security, Performance

14. Free Software Movement

Key Figure

- **Richard Stallman**

Contributions

- GNU Project (1983)
- Free Software Foundation (1985)
- GNU General Public License (GPL)

15. Linux

Characteristics

- Open source
- Kernel + distributions
- Highly customizable

Distributions

- Desktop: Linux Mint, Ubuntu
- Lightweight: Puppy Linux
- Enterprise: Red Hat, SUSE

Desktop environments

- GNOME, KDE, XFCE, MATE

Usage

- ~96% of servers
- Supercomputers
- Embedded systems
- IoT devices

16. Android

- Based on Linux kernel
- Developed by **Google**
- Uses:
  - Linux Kernel
  - Android Runtime (ART)
- Runs apps inside a **virtual machine**
- Requires more RAM due to abstraction layers

Market impact

- ~2 billion users
- ~85% smartphone market share
- Used in phones, cars, TVs, consoles, cameras

17. Virtual Machines

- Definition: OS running on top of another OS

Terms

- Host OS → hardware access
- Guest OS → virtualized environment

Trade-off

- Compatibility ↑
- Performance ↓

18. Windows

- Proprietary OS by **Microsoft**
- Written in: C (kernel), C++, C#
- Market share: ~77% desktop OS
- Closed source
- Apps typically released here first

19. APIs (Application Programming Interfaces)

Purpose

- Allow developers to access OS functionality
- Avoid rewriting low-level code

Importance

- Critical in proprietary OS
- Central to Android development

20. Why This Matters to Programmers

- OS = code written by programmers
- Understanding OS internals helps:
  - Choose correct platform
  - Avoid unsupported designs
  - Write efficient code
  - Prevent wasted time & cost

21. Core Takeaway

- Information processing = **input → process → store → output**
- OS is the **most important software**
- Kernel is the **brain**
- Drivers are the **translators**
- APIs are the **bridges**
- Knowing OS internals = **better engineering decisions**

Computer Science(UX Design) -Lecture 3

1. Computing Systems — Overview
A **computer system** is built using a predefined architecture, just like a house follows a blueprint.
Core components (present in all computers)

- **Input devices** – feed raw data
- **CPU** – processes data
- **Storage devices** – store data & programs
- **Output devices** – present results

2. Central Processing Unit (CPU)
The CPU is a **small chip**, not the computer cabinet.
Internal components of the CPU

2.1 Arithmetic Logic Unit (ALU)

- Performs:
  - Arithmetic operations (add, subtract, etc.)
  - Logical operations (AND, OR, comparisons)
- Handles **decision making**

2.2 Registers

- Very small, ultra-fast memory inside CPU
- Types:
  - **General-purpose registers**
  - **Special-purpose registers**
- Faster than cache and RAM

2.3 Cache

- High-speed memory inside the processor
- Stores:
  - Frequently used data
  - Recently used instructions
- Purpose: reduce access time to RAM
- Measured in MB (e.g., 3 MB cache)

2.4 Buses
High-speed communication pathways

- **Address bus** → carries memory addresses
- **Data bus** → carries actual data
- **Control bus** → carries control signals

2.5 Clock

- Synchronizes all CPU operations
- Measured in **Hertz (Hz)**
- Example:
  - 2.5 GHz = 2.5 billion cycles/second
- One cycle ≈ one instruction step

3. Von Neumann Architecture

Proposed in 1945 by **John von Neumann**

Core idea

- **Programs and data are stored together in memory**
- Enables **general-purpose computing**

Why it mattered

- Eliminated hard-wiring for each task
- Made computers programmable via software

3.1 Special Registers in Von Neumann Architecture

Register	Purpose
PC (Program Counter)	Holds next instruction address
CIR (Current Instruction Register)	Holds instruction being executed
MAR (Memory Address Register)	Holds memory address to access
MDR (Memory Data Register)	Holds data being transferred
ACC (Accumulator)	Stores intermediate & final results

4. Instruction Sets

Instruction Set

- Complete set of commands a CPU understands
- Controls transistor switching

4.1 CISC vs RISC

CISC – Complex Instruction Set Computer

- Goal: fewer instructions per program
- Uses **microcode**
- Instructions span multiple cycles
- Hardware complexity: high
- Pros:
  - Uses less RAM
  - Flexible instruction expansion

RISC – Reduced Instruction Set Computer

- Simple instructions

- One instruction per clock cycle
- Enables **pipelining**
- More registers, fewer transistors
- Pros:
  - Predictable performance
  - Easier compiler optimization
- Cons:
  - Requires more RAM

Analogy

- **CISC** → adult given one complex task
- **RISC** → child guided step-by-step

5. Digital Computing Fundamentals

Binary System

- Uses only:**0** (off), **1** (on)
- Based on transistor states

Bit:

- Single binary digit (0 or 1)

5.1 Byte & Data Units

- **Nibble** = 4 bits
- **Byte** = 8 bits (standard)
- **Word** = 16 bits

6. Memory Addressing & Powers of Two

- Memory addresses are binary
- Adding one address line → doubles capacity

Example

- 4 address lines → 2^4 = 16 locations
- 5 address lines → 32 locations

Why powers of two?

- Prevent unused addresses
- Reduce controller complexity
- Avoid data loss

Powers of two became a **de facto standard**

7. Limits of Computing

7.1 Where Humans Are Better

- Emotional understanding
- Intuition
- Contextual decisions
- Creativity

7.2 Affective Computing

- AI branch focused on:
  - Emotion recognition
  - Facial expressions
  - Human emotional context

7.3 Creativity in AI

- **Inceptionism**
  - AI generates images from noise
- Example: Google “dreaming” AI, Pig-snails, camel-birds

7.4 Self-Improvement Limitation

- Computers don’t evolve autonomously
- Improvements require:
  - Engineers, Programmers
- Exception: learning systems

8. Learning Systems

Example

- **AlphaGo (DeepMind)**
  - Learned games autonomously
  - Improved through iteration

Still dependent on initial human-designed frameworks

9. Decision Making in Computers

- All decisions reduce to:
  - **True / False**
- Based on provided data & logic
- No genuine independent thought

9.1 Natural Language Challenges

- Accents
- Pronunciation variation
- Context
- Figurative language

9.2 Fuzzy Logic

- Allows partial truth values
- Avoids strict true/false boundaries
- Helps with:
  - Speech recognition
  - Pattern ambiguity

10. Fetch–Decode–Execute Cycle

CPU operation loop

- F**etch** instruction from RAM > **Decode** instruction > **Execute** instruction
- Synchronized by clock
- One complete loop = one instruction cycle

11. Factors Affecting Computer Speed

1. Clock Speed
   - More cycles per second → faster CPU
2. Cache Size
   - Larger cache → faster access
3. Number of Cores
   - Multiple processing units
   - Common in powers of two
   - Quad-core ≫ dual-core

12. Core Takeaway

- Computers are **deterministic machines**
- Everything reduces to:
  - Binary states, Timed instruction cycles
- Power comes from:
  - Architecture, Instruction design, Parallelism
- Limits remain in:
  - Emotion, Creativity, Contextual reasoning

Computer Science(UX Design) -Lecture 4

1. Natural Language vs Computer Language

Natural Language

- Refers to **human language**
- Used to express:
  - Thoughts
  - Identity
  - Emotions
  - Imagination
- Logical **and** emotional
- Highly **ambiguous**

Ambiguity

- Same sentence → different meanings based on:
  - Emphasis, Tone, Context
- Useful for humans, **problematic for computers**

Computer / Programming Language

- A **special-purpose language**
- Used to give **precise instructions**
- Must be:Unambiguous, Deterministic ,Strictly structured

If code is confusing to read, it is badly written code.

2. Language Structure (Borrowed into Programming)

Syntax

- Order of words / symbols, Rules of arrangement

Semantics

- Meaning of words / symbols

In simple terms:

Syntax = structure

Semantics = meaning

Programming languages borrow **syntax heavily** but minimize semantics to avoid ambiguity.

3. Why Programming Languages Exist

- Early computers used **pure binary (1s and 0s)**
- Extremely error-prone
- Impossible to scale

Purpose of programming languages

- Make computers usable
- Eliminate rewiring
- Express logic clearly
- Abstract hardware complexity

4. Binary Representation of Information

Binary System

- Based on: **0** (off), **1** (on)
- Fundamental to all computing

Bit

- Binary digit
- Smallest unit of information

Bytes & Units

- **Nibble** = 4 bits
- **Byte** = 8 bits (standard)
- **Word** = CPU-dependent (32-bit / 64-bit)

5. Character Encoding

Unicode

Designed to represent **all human languages**

UTF (Unicode Transformation Format)

- **UTF-7** – legacy email compatibility
- **UTF-8** – most popular
  - Variable width (8–48 bits)
  - Backward compatible with ASCII
- **UTF-16** – 16–32 bits
- **UTF-32** – fixed 32 bits

Capacity

- Characters = 2^n (n = number of bits)
- UTF-32 → ~4.2 billion characters

ASCII

- 7-bit encoding
- 2^7 = **128 characters**
- English-centric
- Counting starts at **0**

Limit:

Cannot represent global languages.

6. Parity Bits (Error Detection)

Used to check **data integrity**

- Even Parity: Total number of 1s must be even
- Odd Parity: Total number of 1s must be odd

Parity bit is adjusted accordingly.

7. Machine Language & Instruction Sets

Machine Language

- Actual 1s and 0s executed by hardware
- All programming languages compile/translate into this

Instruction Set Architecture (ISA)

- Programmer-visible CPU design
- Defines:
  - Supported operations
  - Instruction formats
  - Word size
- Example: **x86**, **ARM**

Word Size

- Amount of data CPU processes at once
- Common sizes:
  - 32-bit, 64-bit

Compatibility

- 32-bit programs → run on 64-bit systems
- 64-bit programs → □ on 32-bit systems

8. Computer Instructions

Each instruction has **three parts**:

1. **Opcode:** Operation to perform
2. **Address Field:** Memory/register location
3. **Mode Field:** How operand/address is interpreted

Instruction Types: Memory reference, Register reference, Input/Output

9. Transducers (Input Conversion)

- Purpose: Convert environmental data → digital signals

Examples

- Microphone, Accelerometer, Pressure sensor, Hall sensor, Display, Motors

Smartphones are packed with transducers.

10. Programming Languages (Conceptual View)

- Describe tasks **step-by-step**
- Follow strict rules
- Require understanding of:
  - Hardware, OS, Architecture

Before building a computer:

- An **abstract machine** is designed

11. Abstract Computational Models

These are **theoretical models**, not real machines.

11.1 Finite State Machines (FSM)

- Limited memory
- Generates **regular languages**
- Used for:
  - Simple logic
  - Pattern matching
  - Regular expressions

11.2 Pushdown Automata (PDA)

- FSM + **stack**
- Stack = LIFO (Last In, First Out)
- Accepts **context-free languages**
- Used in:
  - Parsers, Compilers, Matching parentheses

11.3 Linear Bounded Automata (LBA)

- PDA + finite tape
- Accepts **context-sensitive languages**

11.4 Turing Machine

Invented by **Alan Turing** (1936)

Characteristics

- Infinite tape (memory), Read/write head, Transition table
- Can:
  - Halt, Run forever

Importance

- Most powerful computational model, Defines limits of computability

Any real computer can be simulated by a Turing Machine (given enough memory).

12. Decidability

- **Decidable language**: TM halts for all inputs
- **Recognizable language**: TM may not halt for invalid inputs

All decidable languages are recognizable, but not vice versa.

13. Chomsky Hierarchy

Proposed by **Noam Chomsky** (1956)

Language hierarchy (from weakest → strongest)

	Type 3	**Type 2**	**Type 1**	**Type 0**
Grammar	Regular	Context-Free	Context-Sensitive	Unrestricted
Machine	Finite Automata	Pushdown Automata	Linear Bounded Automata	Turing Machine

Each level is a **subset of the next**.

14. Core Takeaway

- Human language → expressive but ambiguous
- Computer language → strict, deterministic
- Binary underpins everything
- Encoding enables global communication
- Instruction sets bridge software & hardware
- Abstract machines define **what is computable**
- Chomsky hierarchy classifies **language power**

Computer Science(UX Design) -Lecture 5

1. Brief History of Programming Languages

Early Mechanical Roots

- **Jacquard Loom (early 1800s)**
  - Used punch cards to control fabric patterns
  - First example of **machine instruction via symbolic input**

First Programmer

- **Ada Lovelace**
  - Wrote an algorithm for **Charles Babbage**'s Analytical Engine
  - Target problem: Bernoulli numbers
  - October 15 celebrated as **Ada Lovelace Day**

Stored-Program Era (1940s)

- Emergence of programmable electronic computers
- **Alec Glennie**
  - Developed **Autocode** for the Mark I
  - First high-level programming language concept

Assembly Language

- Invented by **Kathleen Booth**
- Replaced raw binary with mnemonics (ADD, SUB, MUL)
- Still used today for:
  - Device drivers
  - Embedded systems
  - Real-time systems

1960s–1970s

- Explosion of programming languages
- Evolution of:
  - C → C++
  - Modular programming
  - Object-oriented concepts

Internet Era

- Shift toward:
  - Programmer productivity
  - Rapid development
- Rise of:
  - Scripting languages
  - Interpreted languages

2. What Is a Programming Language?

Definition: A **systematic method** of describing a computational process using:

- Arithmetic, Logic, Control flow, Input / Output

3. Programming Paradigms (Models)

There are **two primary paradigms**:

3.1 Imperative Programming

Core idea

- Focuses on **how** a task is done, Program state changes step by step

Characteristics

- Assignment statements, Global state mutation, Close to machine architecture

Limitations

- Poor parallelism, Complex state management

Imperative Subtypes

Procedural Programming

- Sequential execution
- Heavy use of loops & variables
- Examples:
  - C, C++, Java, Pascal

Object-Oriented Programming (OOP)

- System modeled as interacting objects
- Key concepts: Encapsulation, Inheritance, Polymorphism
- Emphasizes reusability

Parallel Processing

- Divide-and-conquer strategy, Tasks distributed across processors

3.2 Declarative Programming

Core idea

- Focuses on **what** needs to be done, Not how it is achieved

Characteristics

- No explicit loops, No variable reassignment, High-level abstraction

Declarative Subtypes

Logic Programming

- Program = facts + rules
- Computer reasons about consequences
- Example: Prolog

Functional Programming

- Pure mathematical functions
- Immutable data
- Recursion instead of loops
- Functions passed as values
- Example: **ReactJS**

Database Programming

- Data-driven logic
- Queries describe desired result
- Uses **SQL**
- Managed by **DBMS**
- Strong data consistency guarantees

4. Low-Level vs High-Level Languages

Low-Level Languages

- Hardware-specific
- Assembly, machine code
- Fast but non-portable

High-Level Languages

- Hardware-independent, Portable source code
- Compiled per target architecture
- Large libraries & abstractions

Emulator

- Software that mimics another architecture, Inefficient and resource-heavy
- Avoided when high-level languages are available

5. Translation of Programs

All programs must become **machine language** to run.

5.1 Translation Tools

Assembler

- Assembly → machine code

Compiler

- High-level code → machine code
- Produces executable file
- Target-specific

Interpreter

- Translates line by line
- Executes immediately
- No standalone executable

6. Compilation Process (High-Level)

Phase 1: Preprocessing

- Handles macros, includes

Phase 2: Analysis

Lexical Analysis (Scanner)

- Converts code into tokens
- Removes whitespace/comments
- Detects invalid symbols

Syntax Analysis (Parser)

- Checks grammar rules
- Builds parse tree
- Reports syntax errors

Semantic Analysis

- Checks logical meaning
- Undeclared variables
- Type mismatches

Symbol Table

- Stores:
  - Variables, Functions, Classes
- Used across compiler phases

Phase 3: Intermediate Code Generation

- Architecture-independent
- Assembly-like representation

Phase 4: Code Optimization

- Improves speed
- Reduces size

Phase 5: Code Generation

- Translates to target machine code
- Output: **target program**

7. Assembly, Linking, and Loading

Assembler (Two Passes)

Pass 1

- Determines memory requirements
- Builds assembler symbol table

Pass 2

- Produces **relocatable machine code**

Linker

- Combines multiple object files
- Resolves references
- Produces single executable

Loader

- Loads executable into memory
- Starts execution
- Fails if references unresolved

8. Machine Code Types

- **Relocatable Machine Code:** Can load at different memory locations

- **Absolute Machine Code**
  - Final executable
  - Hardware-specific
  - Non-portable

9. Portability Implications

- Source code → portable
- Compiled executable → NOT portable
- Must compile separately for:
  - x86, ARM, Different operating systems

Example:

- Same Microsoft Office source
- Different binaries for Windows, macOS, Android

10. Core Takeaways

- Programming languages evolved with hardware
- Paradigms shape how problems are solved
- High-level languages trade control for productivity
- Compilation is a **multi-stage pipeline**
- Executables are architecture-specific
- Understanding this pipeline = better debugging & design

Computer Science(UX Design) -Lecture 6

1. Problem Formulation in Computer Science

Why problem formulation matters

- Before writing code, you must know **why** the program exists
- Prevents: Missing requirements, Incomplete solutions, Poor design decisions

Problem formulation

- Translating a **real-world need** into a **computationally solvable form**
- The programmer's responsibility is to:
  - Clarify ambiguity, Identify constraints, Define goals precisely

2. What Is a Computing Problem?

- In computer science, a **problem** is:
  - A task or a set of related tasks, That may be solvable by a computer

Key requirement

- Problems must be **explicitly stated**
- Computers cannot handle:
  - Vague questions, Emotional concepts
- Ambiguous goals: (e.g., "What is joy?")

3. Thinking Computationally

A computer scientist must:

- Break problems into smaller parts
- Identify what **can** and **cannot** be computed
- Decide if a problem is:
  - Solvable, Partially solvable, Unsolvable

4. Five Main Types of Computational Problems

4.1 Decision Problems

- Output is **YES or NO**
- Tests whether a property holds

Example: Is X + Y = Z?

Properties

- Closely related to function problems
- Problems can be:
  - **Decidable, Partially decidable, Undecidable**

Undecidable problems may **run forever** without producing an answer.

4.2 Search Problems

- Goal: **find a solution**, not just confirm existence
- Always associated with a decision problem

Defined by:

- Set of states, Start state, Goal state, Successor function, Goal test function

Example: Searching for a number in a list

- Search problems: Answer **where / how**, Not just **whether**

4.3 Counting Problems

- Goal: **count the number of valid solutions**
- Example: Number of perfect matchings in a graph
- Challenges: Brute-force search becomes impractical
- Requires clever algorithms (e.g., Edmonds' algorithm)

4.4 Optimization Problems

- Goal: find the **best solution**
- "Best" = **minimum cost / weight / time / distance**
- Weight: Represents resource usage

Examples: Route planning (maps), Airline scheduling, **Dijkstra's shortest path**
Types: Continuous optimization, Discrete optimization

4.5 Function Problems

- Every input has a corresponding output
- Output is **not just YES/NO**

Example: Mathematical functions (x / y)
Relationship

- Every function problem → decision problem
- Decision problem = graph of the function

5. General Characteristics of Computational Problems

Defined by: A **task**, A set of **input instances**

- Same task, different inputs
- Example: Addition is always the same process, Only numbers change

6. Tractable vs Intractable Problems

Tractable Problems

- Solvable: Algorithmically, In **reasonable (polynomial) time**

Intractable Problems

- Solvable only with:
  - Exponential or worse time complexity
- Impractical for large inputs
- Examples: Traveling Salesman Problem (TSP), School timetabling

Traveling Salesman Problem (TSP)

- Find shortest route visiting all cities once
- Cost = distance + time + resources

Reality: Exact solutions possible only for limited sizes
Record (2006): ~85,900 cities
Real-world use: PCB drilling paths, Manufacturing optimization

7. Approximate & Heuristic Solutions

Suboptimal Solutions

- Not perfect, Good enough, Solvable in reasonable time

Heuristic Algorithms

- Use: Experience, Rules of thumb, Judgment
- Avoid exhaustive search

Example: Always pick cheapest next route (greedy approach)

8. Polynomial Time vs Exponential Time

- **Polynomial time (P)**: Considered "fast", Scales reasonably with input size
- **Exponential time**: Grows too quickly, Becomes unusable

9. Unsolvable / Undecidable Problems

- Definition: No algorithm can solve the problem for all inputs
- Famous example: **Halting Problem**
- Question: Given a program and input, can we know if it will halt or run forever without running it?
- Result: Proven undecidable by **Alan Turing**

10. Mathematics and Computer Science

Why math is essential

- Computer science is built on:
  - Set theory, Graph theory, Probability, Number theory

Mathematics as a language

- Precise, Unambiguous, Universal

Computer science is often considered a **subset of mathematical sciences**.

11. Mathematical Modeling

Definition

- Translating real-world problems into mathematical form
- Enables: Prediction, Simulation, Optimization
- Benefits: Reveals hidden relationships, Allows controlled experimentation

Types of Models

Mechanistic Models

- Based on physical laws
- Detailed, theory-heavy

Empirical Models

- Based on observed data
- Respond to changing conditions

Stochastic Models

- Probabilistic
- Predict distributions

Deterministic Models

- Same input → same output always

12. Dining Philosophers Problem

- Classic synchronization problem
- Models: Resource sharing, Deadlock prevention

Real-world analogy

- **Multithreading**, CPU time sharing, Device access coordination

13. Writing Problems Clearly

A well-formulated problem must be:

1. **Precise** (unambiguous)
2. Have a **clear initial state**
3. Have a **clear goal state**
4. Define **rules & constraints**
5. Include **all components**

Example: Water Jug Problem

- Jugs: 4L and 3L
- Goal: exactly 2L in 4L jug
- No measuring markers
- Clearly defined states & rules

14. Final Takeaways

- Not all problems are computable
- Not all solvable problems are practical
- Problem formulation determines:
  - Algorithm choice, Feasibility, Performance
- Mathematics provides the structure
- Heuristics make the impossible workable

Computer Science(UX Design) -Lecture 7

1. What Is Pseudocode (and why we use it)

Meaning: **Pseudo** = false (Greek), **Code** = instructions

- → *"Fake code" written for humans, not machines*

Purpose

- Explain **how an algorithm works**
- Capture **logic and flow** clearly
- Act as a **bridge** between:
  - Problem formulation
  - Flowcharts
  - Actual programming code

Key point: Pseudocode is written **to be understood**, not compiled.

2. Why Not Just Write Real Code?

Practical reasons

- Clients can't read real code
- Complex logic is easier to reason about in English
- Helps teams understand each other's work
- Acts as **documentation**
- Reduces bugs before coding begins
- Makes transcription to code faster

Industry reality

- Writing pseudocode first is **standard practice**
- Especially important for:
  - Large systems
  - Algorithms
  - Client-facing projects

3. Relationship Between Algorithm, Pseudocode, Flowchart

Concept	Algorithm	Pseudocode	Flowchart	Code
Purpose	List of logical steps	Human-readable version of algorithm	Visual representation of algorithm	Machine-executable version

→**Algorithm → Pseudocode → Flowchart → Code**

4. Characteristics of Good Pseudocode

Core goals

- **Clarity**, **Unambiguity**, **Readability**, **Transcribability** (Easy to turn into code)

What pseudocode is NOT

- Not tied to any programming language, Not syntactically strict, Not executable

5. Best Practices for Writing Pseudocode

1. Start with the goal

Example:

This program checks whether a number is a palindrome.

Immediately tells the reader:

- What the program does
- What to expect

2. Write in sequence

- Steps must follow logical execution order
- Especially important for complex problems

3. One statement = one action

Bad: Read number and check if it is even and display it

Good: Read number, Check if number is even & Display number

4. Use meaningful variable names

Instead of: x, y

Use: numberOne, numberTwo

- Helps both: Understanding & Transcription into code

5. Use indentation

- Shows structure
- Mirrors real programming logic
- Critical for:
  - IF blocks, LOOPS, PROCEDURES

6. Use common control keywords

Typical pseudocode keywords:

- START, END
- IF, THEN, ELSE
- WHILE, FOR, REPEAT UNTIL
- INPUT, OUTPUT
- PROCEDURE

Capitalize keywords to distinguish logic from text.

7. Avoid assumptions

- Specify:
  - Which sensor
  - Which value
  - Which condition
- No “magic happens here” steps

6. Example: Cake Baking Pseudocode (Why it works)

Why this example is strong:

- Clear procedures
- Logical sequence
- Visible decision point
- Proper indentation
- No ambiguity

It demonstrates:

- Sequential execution
- Conditional logic
- Iteration (re-bake if not done)

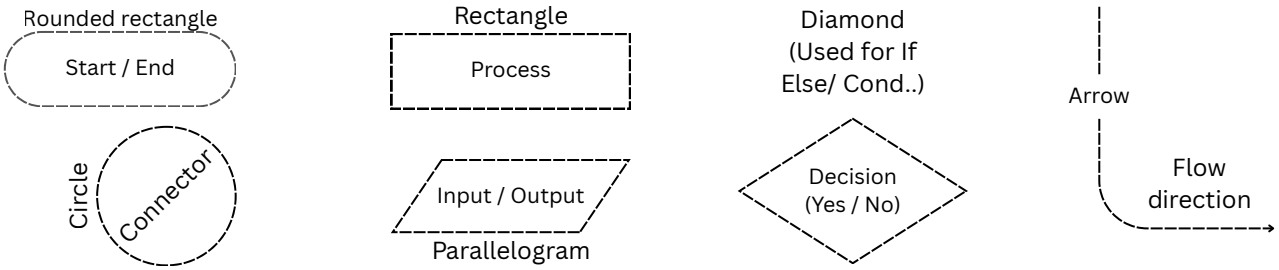
7. Flowcharts: Purpose and Role

What a flowchart does

- Visually shows:
  - Program start, Processing steps, Decisions, Loops, End state

Think of it as: A state machine diagram for a program

8. Flowchart Symbols (Must Know)



Flowcharts follow **stricter rules** than pseudocode.

9. Pseudocode vs Flowchart (Key Difference)

Aspect	Representation	Detail level	Flexibility	Rules	Best for
Pseudocode	Text	High	High	Loose	Logic
Flowchart	Visual	Medium	Lower	Strict	Process overview

They are **complementary**, not competitors.

10. Example: Even Numbers Program (What it shows)

Demonstrates:

- Input control (only 10 numbers), Looping, Decision making, Output filtering

Why it's good pseudocode

- Clear procedures, No hidden assumptions
- Easy to convert into:
  - Python, C, Java

11. Common Pseudocode Keywords (Expanded)

Keyword	INPUT	READ / GET	PRINT / DISPLAY	COMPUTE	SET / INIT	INCREMENT	DECREMENT
Meaning	Get data from user	Read from file	Output result	Perform calculation	Initialize variable	Increase value	Decrease value

12. Shape Calculation Task — How to Think

Correct approach

1. Identify requirements
2. Define valid inputs
3. Reject invalid shapes

1. Distinguish:
    - Flat shapes → area
    - Solid shapes → volume
2. Avoid repetition via procedures
3. Optimize for clarity

Key lesson: Good pseudocode improves **design quality**, not just code speed.

13. Final Takeaways

- Pseudocode is a **thinking tool**
- Flowcharts are a **visual validation tool**
- Writing code without either is:
    - Risky
    - Error-prone
    - Inefficient for complex problems
- Clear thinking → clear pseudocode → clean code

Computer Science(UX Design) -Lecture 8

1. What an Algorithm Really Is
Core idea: An **algorithm** is a **finite, step-by-step procedure** for solving a **well-specified problem**.
Key words that matter:

- **Finite** → must terminate
- **Step-by-step** → clarity and determinism
- **Well-specified** → no ambiguity

If the problem is vague, no algorithm—no matter how clever—can save you.

2. Why Studying Algorithms Matters (Even Today)
Even if:

- Computers were infinitely fast
- Memory was unlimited

We would **still** study algorithms because we must prove that a solution:

1. Exists, Is correct, Terminates, Is optimal

In the real world:

- Hardware **is limited**
- Users are **impatient**
- Efficiency **directly affects business**

88% of users abandon slow apps — optimization is not optional.

3. Algorithms vs Code (Critical Distinction)

Category	Logic	Language	Focus	Longevity
Algorithms	Focuses on logic	Language-independent	Focuses on what & how	Enduring
Code	Implementation details	Language-specific	Focuses on syntax	Changes with technology

Good developers write **code**. Great developers design **algorithms**.

4. Structure of an Algorithm
Every algorithm has **three parts**:

- 1. Input: Data the algorithm operates on
- 2. Process: Ordered steps applied to input
- 3. Output: Result produced by the process

If any of these are unclear, the algorithm is flawed.

5. Why Efficiency Is Everything
Two algorithms can: Solve the **same problem**, Produce the **same output**, But differ wildly in performance

Efficiency determines:

- Speed, Scalability, User experience, Cost

This is where **algorithm complexity** enters.

6. Algorithm Complexity (Plain English)

Algorithm complexity estimates:

How the number of steps grows as input size grows

We care about:

- **Growth rate**, not exact step count, Behavior as input becomes large

Why?

- Hardware differs, Platforms differ, Growth trends don't

7. Big-O Notation (The Language of Efficiency)

What Big-O does

- Describes **upper bound** of growth
- Ignores constants and low-order terms
- Allows fair comparison across machines

Why constants are dropped

- O(2n) and O(100n) grow the same way
- Growth rate matters more than exact counts

8. Major Time Complexities (Fast → Slow)

1. **O(1) — Constant Time**
   - Same number of steps, always

Example:

- Accessing an array element by index

Best possible performance.

2. **O(log n) — Logarithmic Time**
   - Input size is repeatedly halved

Example:

- Binary search

Extremely efficient and scalable.

3. **O(n) — Linear Time**
   - Steps grow directly with input size

Example:

- Simple loop over a list

Acceptable for moderate data sizes.

4. **O(n log n) — Linearithmic Time**
   - Common in efficient sorting algorithms

Example:

- Merge sort, quicksort (average case)

Often the **best practical balance**.

5. **O(n^2) — Quadratic Time**
   - Nested loops over input

Example:

- Bubble sort

Becomes slow very quickly as input grows.

6. **O(2^n) / O(n!) — Exponential / Factorial Time**
   - Growth explodes

Example:

- Brute-force Traveling Salesman Problem

Impractical beyond small inputs.

9. Why This Classification Matters

Imagine:

- 1,000 inputs

Complexity	O(1)	O(log n)	O(n)	O(n log n)	O(n^2)	O(2^n)
Approx. Steps	1	~10	1,000	~10,000	1,000,000	Impossible

Same problem. Vastly different realities.

10. Algorithm Design Is a Skill

Designing algorithms means:

- Choosing the right strategy
- Balancing time vs memory
- Thinking beyond “it works”

That’s why:

- Algorithms separate engineers from coders
- Optimization expertise is highly paid
- Interview processes obsess over it

11. Final Core Takeaways

Think of algorithms as:

- **Blueprints of logic**
- **Contracts of correctness**
- **Engines of performance**

Code is just the vehicle.

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

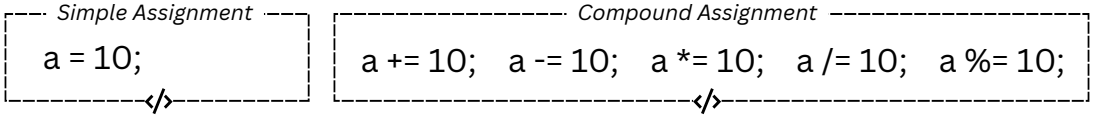
6. Division and Data Type Accuracy

- 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

Computer Science(UX Design) -Lecture 17

1. What "Decision Making" Means in Programming

- Programs don't always execute linearly.
- Decision making allows a program to **choose different execution paths** based on input or conditions.
- Internally, this is always driven by **boolean evaluations** (true / false).
- Real-world systems (GPS, camera modes, error handling) are just complex versions of the same idea.

2. Importance of Planning (Flowcharts & Logic)

- As programs grow, logic branches increase rapidly.
- Without planning, code becomes:
  - Hard to follow
  - Error-prone
  - Difficult to debug
- **Flowcharts** help:
  - Visualize control flow
  - Reduce ambiguity
  - Prevent tangled logic, especially with many conditions

3. Boolean Conditions (Foundation of Decisions)

- Every decision depends on a condition that evaluates to **true or false**.
- Comparison operators:
  - ==, !=, <, >, <=, >=
- These comparisons are **absolute**, not partial.
- Logical operators allow combining conditions:
  - && (AND)
  - || (OR)
  - ! (NOT)
- Operator precedence applies — grouping matters.

4. if Statement

- Executes code **only when a condition is true**.
- If the condition is false, execution continues normally.
- Curly braces {} are optional for single statements, but recommended for clarity.
- Common mistake:
  - Using = instead of == (assignment vs comparison)

5. if–else Statement

- Provides **two mutually exclusive execution paths**:
  - One for true
  - One for false
- Helps remove ambiguity by explicitly handling both outcomes.
- Improves readability and predictability.

6. else if Ladder

- Used when **multiple mutually exclusive conditions** exist.
- Conditions are evaluated **top to bottom**.
- Once a condition is satisfied:
  - Remaining else if and else blocks are skipped.
- Best for: Range-based checks (e.g., grading systems)
- If the ladder grows too long, consider switch.

7. Nested if Statements

- An if inside another if or else.
- Creates **hierarchical decision logic**.
- More powerful, but:
  - Easier to lose track of scope
  - Can lead to deeply nested, unreadable code
- Use carefully and consistently format your code.

8. switch Statement

- Used when selecting **one path from many discrete options**.
- Works like a multi-branch selection tree.
- Rules:
  - Expression must evaluate to **int or char**
  - case labels must be **constant values**
  - No duplicate cases
- Break is essential:
  - Without it, execution “falls through” to the next cases
- default handles unexpected values and ensures safe exit.

9. When to Use switch vs if–else

- Use switch when:
  - Comparing a single variable against fixed values
  - Logic is menu-like (e.g., USSD menus)
- Use if–else when:
  - Conditions involve ranges
  - Complex logical expressions are needed

10. Conditional (Ternary) Operator ?:

- Compact alternative to simple if–else.

Syntax

```
condition ? value_if_true : value_if_false
```

  - Useful for: Simple assignments & Short, readable decisions
- Limitations:
  - Becomes unreadable when nested
  - Not suitable for complex logic

- Both outcomes must be of the **same type**.

11. goto Statement

- Transfers control unconditionally to a labeled section.
- Strongly discouraged because it:
  - Breaks logical flow
  - Creates “spaghetti code”
  - Makes debugging difficult
- Not a true decision structure on its own.
- Should be avoided in structured programming.

12. The Core Takeaways

- Always prioritize **readability over cleverness**.
- Avoid deeply nested logic where simpler structures exist.
- Use:
  - if for conditional execution
  - if–else for binary decisions
  - else if for ordered conditions
  - switch for discrete selections
  - ternary operator for simple assignments only
- Plan before coding when logic becomes non-trivial.

Computer Science(UX Design) -Lecture 18

1. Repetition in Programming

- Repetition allows a program to execute the same block of code multiple times.
- Computers excel at repetition because they execute instructions consistently.
- Repetition is implemented using:
  - while / do–while loops & for loops & Recursion

2. While Loop

2.1 Concept

- Repeats a block of code as long as a condition remains true.
- Condition is checked **before** each iteration.
- May execute zero or more times.

2.2 Key Characteristics

- Controlled by a boolean condition.
- Uses a control variable.
- Control variable must be modified inside the loop.

2.3 Execution Flow

1. Evaluate condition
2. If true → execute loop body
3. Modify control variable
4. Re-evaluate condition
5. Exit when condition becomes false

2.4 Common Uses

- Unknown number of iterations, Input validation & Event-driven repetition

3. Do–While Loop

3.1 Concept

- Executes the loop body first, then evaluates the condition.
- Guaranteed to run at least once.

3.2 Difference from While Loop

- while: entry-controlled loop
- do–while: exit-controlled loop

3.3 When to Use

- When the condition depends on user input
- When the condition must be established after execution

4. Common Loop Errors

4.1 Unreachable Loop

- Condition is never true & Loop body never executes.

4.2 Infinite Loop

- Condition never becomes false.
- Caused by: Incorrect update of control variable & Missing update

4.3 Uninitialized Control Variables

- Causes unpredictable behavior.
- Code may compile but behave inconsistently.

4.4 Rules to Remember

- Condition must be satisfiable.
- Condition must eventually become false.
- Control variables must be initialized.
- Condition must clearly define continuation and termination.

5. For Loop

5.1 Concept

- Used when the number of iterations is known in advance.
- Combines initialization, condition, and update in one statement.

5.2 Execution Order

1. Initialization (once)
2. Condition check
3. Execute loop body
4. Increment/decrement
5. Repeat condition check

5.3 Advantages

- Cleaner and more readable
- Lower risk of infinite loops
- Ideal for counting and traversal

5.4 Typical Use Cases

- Fixed iteration counts
- Array traversal
- Multiplication tables

6. While vs For Loop

Aspect	Iteration count	Control clarity	Error risk	Best use
While Loop	Unknown	Distributed	Higher	Input-driven logic
For Loop	Known	Centralized	Lower	Counting / traversal

7. Recursion

7.1 Concept

- A function calls itself to solve a problem.
- Each call works on a smaller version of the problem.

7.2 Essential Components

1. Base Case
   - Stops recursion
   - Solved without further recursive calls
2. Recursive Case
   - Reduces problem size
   - Moves toward base case

7.3 Execution Behavior

- Function calls stack until base case is reached.
- Results are resolved in reverse order.

8. Example: Factorial

- Recursive definition:

    - n! = 1 if n = 0 (base case)
    - n! = n × (n−1)! if n > 0 (recursive case)
- Demonstrates problem reduction and backtracking.

9. Applications of Recursion
- Mathematical computations
- Divide-and-conquer algorithms (e.g., merge sort)
- Data structures (linked lists, trees)
- Artificial intelligence problems
- Computer graphics (fractals)
- Classical puzzles (Towers of Hanoi)

10. Limitations of Recursion
- High memory usage due to call stack
- Slower than iteration in many cases
- Repeated computation of the same subproblems

11. Fibonacci and Algorithmic Complexity
- Naive recursive Fibonacci recalculates values repeatedly.
- Time complexity grows exponentially.
- Performance degrades rapidly for large inputs.
- Highlights why recursion must be used carefully.

12. Design Rules for Recursion
1. Always define a base case.
2. Each recursive call must move closer to the base case.
3. Assume recursive calls work correctly (inductive reasoning).
4. Avoid duplicate computation.

13. The Core Takeaways
- Use **while loops** for unpredictable repetition.
- Use **do–while loops** when execution must occur at least once.
- Use **for loops** when iteration count is known.
- Always initialize and update control variables.
- Infinite loops are valid only when intentional.
- Recursion is powerful but costly — use it deliberately.
- Prefer iteration when performance is critical.
- Clear termination logic matters more than syntax.

Computer Science(UX Design) -Lecture 19

1. What Scope Means

- **Scope** defines where a variable or function is **visible and accessible** in a program.
- Two primary types:
  - **Global scope** → visible everywhere in the program
  - **Local (block) scope** → visible only inside the block { } where declared
- Scope boundaries are defined by **curly braces**.

2. Global vs Local Variables

- **Global variables**
  - Declared outside all functions
  - Accessible across functions and source files
- **Local variables**
  - Declared inside functions or blocks
  - Exist only within that scope
- **Rule**: Local variables take precedence over global variables if names collide.

3. Function Scope

- Variables declared inside a function:
  - Are created when the function is entered
  - Are destroyed when the function exits
- Function parameters also have **local scope**.
- In C, only **goto labels** have function scope (unique rule).

4. Why Scope Is Necessary

- Prevents variable name collisions
- Enforces **information hiding**
- Ensures predictable program behavior
- Improves **memory efficiency**
- Reduces unintended side effects
- Enables safer and more maintainable code

5. Scope and Memory Management

- Local variables:
  - Created only if their scope is executed
  - Destroyed when scope ends
- Unused scopes do not allocate memory
- Helps reduce memory footprint and waste

6. Storage Classes Overview

Storage classes define:

- **Lifetime & Memory location & Visibility & Linkage**

Four storage classes:

- Auto, register, static & extern

7. Linkage (Visibility Across Scopes & Files)

Linkage controls whether identifiers refer to the same object across scopes.

Types of linkage:

1. **No linkage**
   - Block scope variables
   - Function parameters
2. **Internal linkage**
   - static variables at file scope
   - Visible within one source file
3. **External linkage**
   - Global non-static variables and functions
   - Visible across translation units

8. Translation Unit

- A **translation unit** = source file + included headers
- Determines visibility and linkage across files

9. Auto Storage Class

- Default for local variables
- Characteristics:
  - Block scope
  - Automatic lifetime
  - Uninitialized by default (garbage values)
- Memory allocated: On block entry
- Memory deallocated: On block exit
- Rarely written explicitly

10. Register Storage Class

- Suggests storing variable in a **CPU register**
- Used for:
  - Frequently accessed variables
  - Loop counters
- Characteristics:
  - Faster access
  - Not guaranteed (compiler decides)
- Restrictions:
  - No address (&) allowed
  - No pointers
  - Block scope only
- Lifetime same as auto
- No linkage

11. Static Storage Class

- Enforces **information hiding**
- Characteristics:
  - Static storage duration (entire program lifetime)

    - Retains value between function calls
- Can be used: At file scope or At block scope
- Effects:
    - Internal linkage at file scope
    - Local scope but persistent lifetime inside functions
- Cannot be used in function parameter lists

12. Static Arrays in Function Parameters
- Indicates array pointer is:
    - Non-null
    - At least a certain size
- Helps compiler optimization and safety

13. Extern Storage Class
- Used for sharing variables across source files
- Characteristics:
    - External linkage (unless declared static)
    - Static storage duration
- Memory:
    - Allocated before main()
    - Deallocated when program ends
- extern keyword optional if:
    - Declaration is outside a function
    - Variable already has file scope

14. Scope Rules for extern
- Inside a block:
    - Refers to existing global variable
- Outside functions:
    - Creates external linkage unless static
- If variable was previously declared static, extern will not override it

15. Choosing the Right Storage Class
Guidelines:
- Use **static** → when value must persist across calls
- Use **register** → for heavily reused variables
- Use **extern** → for shared global data
- Use **auto** → default for most local variables

16. Scope and Code Security
- Scope limits access → reduces attack surface
- Variables should only be accessible where needed
- Improper scope increases vulnerability risk
- Security begins with **controlled visibility**

17. Environment Awareness

- Program behavior depends on:
  - OS
  - Hardware
  - Execution environment
- Portable code may expose new vulnerabilities
- Environment-specific optimization improves:
  - Security
  - Performance

18. Scope as First Security Boundary

- Defines what parts of the program can be accessed
- Helps identify:
  - External interaction points
  - Attack surface
- Secure scope design reduces misuse potential

19. Demo Key Takeaways

- Function prototypes enable visibility before definition
- Variables declared below main() are not visible unless prototyped
- Local variables override globals with same name
- Static variables persist across function calls
- Scope position in code determines accessibility
- Formatting specifiers control output precision

20. Core Things to Remember

- Scope controls visibility
- Storage class controls lifetime and linkage
- Local > global in name conflicts
- Static ≠ global (lifetime ≠ scope)
- Extern links across files
- Smaller scope = safer code
- Secure code starts with proper scoping

Computer Science(UX Design) -Lecture 20

1. What an Array Is

- An **array** is a data structure that stores a **collection of elements of the same data type**.
- Elements are stored in **contiguous memory locations**.
- All elements are accessed using:
  - one variable name
  - an **index** to identify each element
- Arrays are **homogeneous** (same data type).

2. Why Arrays Are Needed

- Managing large collections of data (e.g., 1000 values) using individual variables is:
  - inefficient
  - unreadable
  - wasteful
- Arrays allow:
  - compact code
  - loop-based processing
  - scalable data handling
- Arrays are foundational to:
  - searching algorithms
  - sorting algorithms
  - stacks, queues, hash tables, linked lists
  - strings
  - databases
  - graphics and mathematical models

3. Historical Context (Conceptual)

- Early arrays were implemented via:
  - self-modifying code
  - memory segmentation
- Modern high-level languages provide native array support
- CPUs are often optimized for array operations

4. Array Declaration Methods

Method 1: Declare and initialize

```
int a[] = {1, 2, 3, 4};
```

- Compiler infers array size.

Method 2: Declare with size, initialize later

```
int a[10];
```

- Elements assigned at runtime (input, computation, file I/O).

5. Array Size Rules

- Size must be specified at compile time (except VLAs).
- Size **cannot be changed after compilation**.
- From C99 onward:
  - **Variable Length Arrays (VLA)** are supported.
- Over-allocating wastes memory.

6. Indexing Rules

- Array indices start at **0**.
- Valid indices:
  - 0 to size - 1
- Accessing out-of-bounds indices:
  - causes undefined behavior
  - may not be caught at compile time
  - often leads to runtime bugs

7. Arrays and Memory

- Arrays occupy **contiguous memory**.
- Memory usage = size × sizeof(data_type)
- Example:
  - int ages[14];
  - If sizeof(int) = 4 bytes, total = 56 bytes
- Efficient access due to predictable memory layout.

8. Using Loops with Arrays

- **for-loops** are the primary tool for array traversal.
- Common pattern:
  - iterate from 0 to size - 1
- Allows:
  - sequential access
  - bulk operations
  - scalable logic

9. Assigning and Accessing Elements

Assign using index:

```
ages[5] = 21;
```

- Random access is allowed (not just sequential).
- Arithmetic and logical operations work the same as with normal variables.

10. Common Array Errors

- Incorrect index calculations
- Mixing loop bounds
- Off-by-one errors
- Using uninitialized arrays
- Runtime errors instead of compile-time errors
- Most array bugs come from **index misuse**

11. Arithmetic Operations on Arrays

- Element-wise operations using loops:
  - addition, subtraction, multiplication, division, modulus

- Data type choice matters:
  - int division loses fractional part
  - Use float arrays for division results

12. Example Pattern (Two Arrays)

- Read values into arrays A and B
- Perform operations element-wise
- Store results in separate arrays
- Display results in tabular format

13. Logical Operations on Arrays

- Logical checks can be applied per element
- Index correctness is critical
- Useful for:
  - filtering data
  - condition-based processing

14. Strings as Arrays

- In C, a **string is a character array**.
- Ends with a **null terminator** '\0'.
- Each element occupies **1 byte**.

Syntax

```
char name[20];
```

15. String Initialization Methods

Using string literal:

```
char s[] = "Hello";
```

Specify size:

```
char s[10] = "Hello";
```

Character-by-character:

```
char s[] = {'H','e','l','l','o','\0'};
```

Character-by-character with size specified

```
char s[6] = {'H','e','l','l','o','\0'};
```

- '\0' must be included when manually initializing.

16. Reading Strings with scanf

- %s stops at whitespace.
- Cannot read full names with spaces.
- Solution: **edit set conversion**

```
scanf("%[^\n]", name);
```

- Reads until newline.

17. string.h Header File

Provides utilities for string manipulation:

- Reduces manual errors
- Must be included explicitly
- Header files should be explored before use

18. Common String Functions (Core)

- strcat() → concatenate strings
- strncat() → concatenate first n characters
- strcmp() → compare strings
- strncmp() → compare first n characters
- strcpy() → copy string
- strncpy() → copy first n characters
- strlen() → length excluding '\0'
- strchr() → first occurrence of character
- strrchr() → last occurrence of character

19. String Safety Concerns

- Many string functions:
  - lack bounds checking
  - fail silently
- Buffer overflows can:
  - crash programs
  - be exploited for attacks
- Prefer:
  - correct sizing
  - bounded versions (strncpy, strncat)
- Extra unused space is safer than overflow

20. Multi-Dimensional Arrays

- Arrays can have multiple dimensions.
- A **2D array** represents:
  - rows × columns (table)

int a[rows][cols];

- First index → row
- Second index → column

21. Higher-Dimensional Arrays

- 3D and beyond supported:

int a[x][y][z];

- Used in:
  - graphics
  - simulations
  - mathematical modeling
  - 3D data representation

22. Two-Dimensional Array Access

- Requires **nested loops**:
  - outer loop → rows
  - inner loop → columns
- Enables structured data processing

23. Demo: Multiplication Table

- Uses:
  - 2D array
  - nested loops
- Stores multiplication results
- Displays formatted table
- Demonstrates:
  - real-world array usage
  - separation of computation and display

24. The Core Takeaways

- **A**rrays store homogeneous data
- Indexing starts at zero
- Out-of-bounds access is dangerous
- Arrays occupy contiguous memory
- Size must match actual usage
- Loops are essential for arrays
- Strings are character arrays
- '\0' terminator is mandatory
- String functions require caution
- Most array bugs are logic bugs, not syntax errors
- Multi-dimensional arrays model structured data naturally

Computer Science(UX Design) -Lecture 21

1. What a Pointer Is (Core Idea)

- A **pointer is a variable that stores a memory address**, not a value.
- That address usually points to another variable.
- Pointers exist because programs ultimately operate on **memory locations**, not names.
- Especially critical in **C**, **low-level programming**, **OS kernels**, **embedded systems**, and **drivers**.

Key Point: Pointer = reference / direction sign, not the destination itself.

2. Memory, Addresses, and Base Address

- **Byte** is the smallest addressable unit of memory.
- Variables may occupy **multiple bytes** (e.g., long long int → 8 bytes).
- The **base address** = address of the first byte of a variable.
- Only the base address is needed because:
  - The compiler already knows the variable's size.
  - It computes the full memory span automatically.

3. Data Primitives vs Data Aggregates

- **Data primitive**: single readable/writable unit in memory.
- **Data aggregate**: group of primitives treated as one object.
  - Examples: arrays, structures.
- Arrays are:
  - Contiguous in memory
  - Each element has its own address
  - The array name refers to the **base address**

4. Why Pointers Exist (Practical Reasons)

Pointers are essential for:

1. **Dynamic data structures**
   - Linked lists, trees, graphs
2. **Dynamic memory allocation**
   - malloc, calloc, free
3. **Efficiency**
   - Passing addresses instead of copying data
4. **Pass-by-reference**
   - Modify variables across functions
5. **Returning multiple values**
   - Via output parameters
6. **Interfacing with low-level APIs**
7. **Reducing memory footprint**
   - Especially for arrays and large structures

5. Pointer Declaration and Dereferencing

```
------- Declaration -------
  int *p;
----------</>-------------
```

- * tells the compiler this variable is a pointer.
- Pointer must store an **address**, not a value.

```
------- Assignment -------
  p = &x;
----------</>-------------
```

- & gives the base address of x.

```
------ Dereferencing ------
  *p
----------</>-------------
```

- Accesses the **value stored at the address**.
- p → address
- *p → data at that address

6. Initialization Rules (Very Important)

- **Uninitialized pointers = undefined behavior**
- Always :
  - With a valid address
  - Or with NULL

```
------- initialization -------
  int *p = NULL;
----------</>-------------
```

- Use NULL, not 0, for clarity and safety.

7. Pointer Types

7.1 Data Pointers

- Point to variables of a specific type.

7.2 Function Pointers

- Store the address of a function.
- Used for:
  - Callbacks
  - Replacing switch statements
  - Avoiding code duplication

7.3 Void Pointers

- Generic pointer (void *)
- Can point to any data type
- **Cannot be dereferenced**
- Pointer arithmetic is **non-portable**

7.4 Double Pointers

- Pointer to another pointer (**)
- Used for:
  - Modifying pointers in functions
  - 2D arrays
  - API compatibility

8. Pass-by-Reference Using Pointers

- Allows a function to modify original data.
- Efficient for large data structures.
- Should only be used when modification is required.
- Improves performance and avoids unnecessary copying.

9. Pointer Arithmetic (Rules That Matter)

Allowed Operations:

- Increment (p++), Decrement (p--)
- Add/subtract integer
- Subtract two pointers (same object only)
- Comparison (same object only)
- Assignment

Forbidden / Dangerous

- Adding two pointers
- Multiplication/division
- Arithmetic on unrelated memory
- Crossing array bounds

10. How Pointer Arithmetic Works

- Pointer arithmetic is **scaled by data type size**
- Example:
  - int *p
  - p++ moves **4 bytes**, not 1

Formula

new_address =
old_address + (offset × sizeof(type))

11. Arrays and Pointers (Critical Relationship)

- Array name = base address
- Arrays decay into pointers when passed to functions

Access elements using:

```
*(p + i)
```

- Incrementing a pointer walks through array elements.
- Never exceed array bounds → undefined behavior / crash.

12. Operator Precedence with Pointers

Key distinctions:

	*p++	(*p)++	++*p	++p
Description	Increment pointer, then dereference	Increment value being pointed to	Increment value	Increment pointer

- * and + have same precedence
- **Associativity resolves ambiguity**
- Parentheses are strongly recommended

13. Linked Lists (Why Pointers Matter)

- Non-contiguous memory
- Dynamic size
- Each node contains:
  - Data
  - Pointer to next node
- Advantages:
  - Easy insertion/deletion
- Disadvantages:
  - No random access
  - Cache inefficiency
  - Extra memory per node

14. Dangling Pointers (Major Danger)

Occurs when:

1. Memory is freed but pointer still references it
2. Variable goes out of scope
3. Pointer references local variables after function exit

Result:

- Dereferencing yields garbage
- High risk of crashes

15. Best Practices to Remember

- Always initialize pointers
- Use NULL for safety
- Avoid complex expressions with ++ and --

- Never dereference invalid memory
- Don’t perform arithmetic on void pointers
- Only compare pointers within the same object
- Free memory and set pointer to NULL
- Prefer clarity over cleverness

16. The Core Takeaways

- Pointers are **fundamental**, not optional, in C
- They give **direct control over memory**
- With power comes responsibility:
  - Efficiency vs safety is a deliberate tradeoff
- Mastering pointers unlocks:
  - Dynamic memory
  - Data structures
  - Systems programming

Computer Science(UX Design) -Lecture 22

1. What a Structure Is

- A **structure (struct)** is a **composite data type** in C.
- It groups **different data types** under one name.
- Members are stored in **contiguous memory**.
- Represents a **record** (related fields grouped together).
- Access members using the **dot (.) operator**.
- Structure variables:
  - can be passed to functions
  - can be returned from functions
  - behave like ordinary variables

2. Structures vs Arrays

- **Arrays**
  - Store multiple values of the **same type**
- **Structures**
  - Store multiple values of **different types**
- Both use contiguous memory
- Arrays model collections
- Structures model **real-world records**

3. Structure Memory Behavior

- Memory allocated = sum of sizes of all members (plus padding)
- All members exist **at the same time**
- Each member has its own address
- Efficient for representing grouped data like:
  - student records
  - employee details
  - sensor data

4. Accessing Structure Members

- Use dot notation:
  - variable.member
- If accessed via pointer:
  - pointer->member
- Each member behaves like an independent variable

5. Passing Structures to Functions

  - Can be passed: by value (copy) or by reference (pointer)
- Passing by reference is preferred for:
  - performance & large structures
- Supports comparisons and assignments

6. What a Union Is

- A **union** is a composite data type similar to a structure.
- All members share the **same memory location**.
- Only **one member is valid at a time**.
- Size of a union = size of its **largest member**.

7. Structures vs Unions (Key Difference)

- **Structure:** All members exist simultaneously & More memory usage
- **Union:** Members overwrite each other & Memory-efficient
- Use unions when a value can take **only one form at a time**

8. Why Use Unions

- Save memory
- Ideal when:
  - data can be multiple types, but never simultaneously
- Common in:
  - embedded systems, low-level programming & protocol parsing

9. Type Casting (Concept)

- **Type casting** converts one data type into another.
- Ensures operations are performed correctly.
- Prevents unnecessary memory usage.
- Required when data types are **not naturally compatible**.

10. Type Promotion and Demotion

- **Promotion**
  - Smaller type → larger type
  - Safe (no data loss)
- **Demotion**
  - Larger type → smaller type
  - Risk of data loss
- Compiler follows a **data type hierarchy**

11. Explicit Type Casting

- Done **manually** by the programmer.
- Used when:
  - demotion is required, precision loss is acceptable
- Fractional parts are discarded when casting to int

Syntax

```
(type) expression
```

12. Correct Rounding Using Type Casting

- Add 0.5 **before casting** to int
- Parentheses are critical
- Prevents incorrect truncation
- Operator precedence matters

13. Implicit Type Conversion

- Done **automatically** by the compiler.
- Happens when:
  - data types are compatible
- Lower types are promoted to higher types
- No casting operator needed
- Occurs during compilation

14. Rules of Implicit Conversion

- char and short → int
- Float hierarchy applies:
  - long double > double > float
- Signed vs unsigned handled carefully
- Final result is converted to:
  - type of the left-hand side variable

15. When Implicit Conversion Is Blocked

- If conversion may cause data loss:
  - compiler issues a warning
- Programmer must use **explicit casting**
- Protects accuracy by default

16. Built-in Type Conversion Functions

- Provided via stdlib.h
- Mainly used for **string ⇔ numeric** conversion

17. String to Number Functions

- atof() → string to double
- atoi() → string to int
- atol() → string to long
- Return 0 if conversion fails
- No error reporting beyond return value

18. Number to String Functions

- itoa() → int to string
- ltoa() → long to string
- Require:
  - destination buffer
  - numerical base
- Buffer size must be sufficient
- Return pointer to null-terminated string

19. Typedef (Type Alias)

- typedef creates an **alias** for an existing type
- Does **not** create a new data type
- Used to simplify:
  - complex structures
  - unions
  - pointer-heavy code
- Improves readability and maintainability

20. Typedef vs #define

- typedef
  - handled by compiler
  - type-safe
  - scoped
- #define
  - handled by preprocessor
  - text substitution
  - no type checking
- Prefer typedef for data types

21. Structures + Typedef (Common Pattern)

- Used together to:
  - simplify syntax
  - hide implementation details
  - make APIs cleaner
- Very common in professional C codebases

22. Demo Program Key Concepts

- Structure definition using typedef
- Structure arrays for multiple records
- Function prototypes for scope visibility
- Constants used for array size safety
- Loop-based input and output
- Compound printf usage
- Clearing screen before output for clarity

23. The Core Takeaways

- Structures group **heterogeneous data**
- Unions share memory, only one member valid
- Structures consume more memory than unions
- Type casting controls precision and memory
- Explicit casting can cause data loss
- Implicit conversion is compiler-driven
- Parentheses affect casting correctness
- Built-in conversion functions lack robust error handling
- Typedef improves code clarity
- Use structures for records, unions for alternatives
- Memory safety matters when converting and copying data

Computer Science(UX Design) -Lecture 23

1. Purpose of Revisiting the Compilation Process

- To understand **where header files fit** in compilation.
- Explains **why header files behave differently** from normal source code.
- Critical for understanding **preprocessor rules and limitations**.

2. High-Level Compilation Pipeline

1. **Preprocessor**
2. **Compiler** → produces *relocatable code*
3. **Assembler** → produces *assembly*
4. **Linker & Loader** → produces *machine code*

Each stage transforms code into a different form.

3. The Preprocessor (Core Focus)

- Runs **before compilation**
- Is a **separate program** from the compiler
- Operates mainly on lines starting with #
- Output is still **valid C code**, just expanded

Main Responsibilities

- Header file inclusion
- Macro expansion
- Conditional compilation
- Line control

4. Preprocessing Stages (In Order)

1. **Trigraph Replacement**
   - Converts trigraph sequences into single characters
   - Rarely used today, mostly legacy
2. **Line Splicing**
   - Joins physical lines split using \
3. **Tokenization**
   - Breaks code into tokens and whitespace
   - **Comments are removed entirely**
4. **Directive & Macro Processing**
   - Executes #include, #define, #if, etc.
   - Inserts external file contents directly into code

5. Preprocessor Directives

5.1 #include

- Inserts file contents **textually**
- Treated as if you wrote that code yourself

Forms

- <file.h> → search standard include paths
- "file.h" → search project directory first

5.2 #pragma

- Requests **compiler-specific behavior**
- Syntax and supported tokens vary by compiler

- Ignored if unsupported
- From **C99**, can be generated via macros

Used mainly in **large or specialized builds**

6. Header Files: Key Rules

- Typically use .h, sometimes .hpp
- Meant to be **included once**
- Multiple inclusion causes errors unless guarded
- Contain:
  - Function declarations
  - Macros
  - Type definitions

Special Case

- assert.h **can be included multiple times**
  - Used to enable/disable assertions

7. Why Header Files Exist

- Avoid rewriting common functionality
- Improve portability
- Separate **interface** from **implementation**
- Enable reuse and standardization

8. The C Standard Library

- Collection of standard header files
- Defined by ANSI → ISO standards
- Intentionally **small and minimal**
- Designed for:
  - Portability
  - Low-level system access
  - Predictable behavior across platforms

9. Important Standard Header Files (Grouped)

Debugging
assert.h
→ runtime checks, aborts on failure

Error Handling
errno.h
→ global error reporting via errno

Variadic Functions
stdarg.h

- Rules:
  - At least one fixed argument
  - va_end must be called
  - Type promotion rules apply

Signals (Mostly Unix)
signal.h
→ asynchronous signal handling

- **Not portable**

Character Handling
ctype.h
→ character classification & mapping

Mathematics
math.h, float.h, limits.h
→ Advanced math, floating-point behavior, type limits

Control Flow
setjmp.h
→ non-local jumps (manual exception-like behavior)

Localization
locale.h
→ regional formats (date, time, currency)

Core Utilities

- stddef.h → common types and macros
- stdio.h → input/output (files, streams, devices)

- stdlib.h → memory allocation, conversions, utilities
- string.h → string manipulation
- time.h → date and time functions (second-level precision)

10. Hosted vs Freestanding Environments

Hosted

- Full standard library available
- Typical desktop/server systems

Freestanding

- Minimal headers only:
  - float.h
  - limits.h
  - stdarg.h
  - stddef.h
- Common in embedded systems

11. User-Defined Header Files

- Created to build **custom libraries**
- Included using quotes "file.h"
- Behave exactly like standard headers
- Excessive non-standard headers hurt portability

12. C Standards Overview

ANSI C

- First formal standard (1980s)
- C89 / C90

ISO C

- Adopted in 1990
- Revisions:
  - C99
  - C11
  - C18 (latest)

What the Standard Defines

- Program representation
- Syntax and constraints
- Semantics
- Input representation
- Output representation

13. POSIX Standard

- Superset of the C standard library
- Adds:
  - Multithreading
  - Networking
  - Regex
- Focused on **Unix-like systems**
- Improves OS-level compatibility, not portability across all platforms

14. The Core Takeaways

- Preprocessor runs **before compilation**
- #include performs **text substitution**
- Header files define interfaces, not logic
- Standard library is intentionally minimal
- Portability depends on sticking to standard headers
- POSIX ≠ Standard C
- Compiler behavior ≠ Preprocessor behavior
- Freestanding ≠ Hosted environments
- Comments never reach the compiler
- Multiple inclusion must be controlled

Computer Science(UX Design) -Lecture 24

1. What Memory Is

- Memory stores **data** (binary 0s and 1s) used by the computer.
- Data is unprocessed information; instructions operate on it to produce results.
- Physically implemented using **transistors and capacitors** representing charge.

2. Memory Hierarchy (Closest → Farthest from CPU)

1. **CPU Registers**
   - Smallest, fastest, most expensive
   - Directly accessed by CPU
2. **Cache**
   - Slightly slower than registers
   - Stores data likely needed soon
   - Usually only a few MB
3. **RAM (Main Memory)**
   - Largest working memory
   - Slower than cache
   - Programs can run directly from RAM
4. **Secondary Storage (Disk)**
   - Very slow compared to RAM
   - Used when RAM is insufficient

3. Latency and Performance

- **Latency** = time to complete a read/write
- Higher latency → slower memory
- Performance depends on:
  - Speed (latency)
  - Size
  - Cost
  - Distance from CPU

4. Memory Allocation at Program Start

- When a program starts:
  - OS assigns memory via the **memory manager**
  - Allocator decides how memory is distributed
- Memory movement between RAM and disk is handled by OS
- Programmer declares variables → compiler converts them to **memory addresses**
- Variable names mean nothing at runtime; only addresses matter

5. Stack Memory

- Stores **automatic (local) variables**
- Memory freed automatically when variables go out of scope
- Characteristics:
  - Fast access, Size must be known at compile time & Limited size
- Uses **LIFO (Last In, First Out)**
- Each function call:
  - Pushes return address and local variables
- Recursive calls rely heavily on stack behavior

6. Heap Memory

- Used for **dynamic memory allocation**
- Memory allocated at runtime
- Characteristics:
  - Slower than stack, Much larger & Size can grow/shrink dynamically
- Programmer must manually manage it
- Incorrect handling leads to:
  - **Memory leaks,** Crashes, Undefined behavior

7. Program Memory Layout (Low → High Address)

1. **Text Segment**
   - Executable code, Read-only, Sharable across processes
2. **Initialized Data Segment**
   - Global/static variables with initial values, Read-write
3. **Uninitialized Data Segment (BSS)**
   - Global/static variables without initialization
   - Contains garbage values initially
   - **Stack:** Function calls, local variables
   - **Heap:** Dynamically allocated memory

8. Dynamic Memory Allocation in C

- Used when data size is unknown at compile time
- Overcomes fixed-size array limitations
- Defined in standard library headers

Four Core Functions

1. **malloc**
   - Allocates memory (uninitialized)
   - Returns pointer or NULL
2. **calloc**
   - Allocates memory and initializes to zero
   - Returns pointer or NULL
3. **free**
   - Deallocates memory
   - Does not return anything
   - Freeing invalid or already freed memory → undefined behavior
4. **realloc**
   - Resizes previously allocated memory
   - Preserves old data up to smaller size
   - May move memory to a new location
   - Returns NULL on failure (old block remains valid)

9. Special realloc Behaviors

- realloc(NULL, size) → same as malloc(size)
- realloc(ptr, 0) → frees memory, returns NULL

- If expansion fails:
  - Original block is not freed

10. Fragmentation

- **External Fragmentation**
  - Free memory exists but in unusable chunks
  - Causes allocation failure despite available memory
- Handled by allocator, mostly out of programmer's control

11. Common Memory Errors

1. **Memory Leak**
   - Allocated memory never freed
   - Program memory usage grows uncontrollably
2. **Dangling Pointer**
   - Pointer refers to freed memory
   - Causes crashes and security vulnerabilities
3. **Wild Pointer**
   - Pointer never initialized
   - Points to random memory
4. **Invalid Free**
   - Freeing memory that was never allocated
   - Freeing memory twice

12. Best Practices

- Always initialize pointers (use NULL)
- Always free memory you allocate
- Set pointer to NULL after freeing
- Limit user-controlled memory sizes
- Validate allocation results (NULL checks)
- Be extra careful on constrained systems (embedded)

13. The Core Takeaways

- Stack is fast, automatic, limited
- Heap is flexible, manual, dangerous if mishandled
- C gives **full control** → also full responsibility
- Compiler will not protect you from memory mistakes
- Correct memory management = stable, efficient programs

Computer Science(UX Design) -Lecture 25

1. Why Function Arguments Matter

- Functions break large problems into smaller ones.
- For functions to cooperate, **data must be shared**.
- Arguments allow functions to **receive and operate on data**.

2. Types of Function Arguments

2.1 Formal Arguments

- Declared in the **function definition or prototype**
- Act as placeholders for incoming data
- Usually variables, but not strictly limited to variables

2.2 Actual Arguments

- Supplied **when the function is called**
- Can be: Variables, Constants, Expressions & Function calls

3. Ways Arguments Are Passed

3.1 Pass by Value

- A **copy** of the data is passed
- Changes inside the function **do not affect** the original variable

3.2 Pass by Address

- The **memory address** is passed
- Function directly modifies the original data
- Uses pointers

4. Why Variable Arguments Are Needed

- In real programs, **data size is often unknown at runtime**
- Fixed-argument functions force rigid design
- Examples where argument count varies:
  - Printing output (printf)
  - Dynamic data processing
  - Runtime-generated arrays

5. Variadic Functions (Variable Argument Functions)

- Functions that accept **a variable number of arguments**
- Syntax requirement:
  - At least **one named parameter**
  - Followed by ... (ellipsis)

Example pattern:

```
function(type fixed_arg, ...)
```

6. Real-World Example

- printf and scanf
- They accept different numbers of arguments per call
- Compiler does **not** enforce argument count or type correctness

7. Standard Argument Handling (stdarg.h)
This header enables variadic functions.
Core Components

1. **va_list**
   - Tracks the position in the argument list
2. **va_start**
   - Initializes va_list
   - Uses the last named parameter as reference
3. **va_arg**
   - Retrieves the next argument
   - Requires the **correct expected type**
4. **va_end**
   - Cleans up when argument processing is done

8. How Variadic Argument Processing Works

1. Function receives fixed arguments
2. va_start points to first unnamed argument
3. va_arg retrieves arguments one by one
4. Programmer controls **how many arguments are read**
5. Reading beyond available arguments → undefined behavior

9. Critical Limitations of Variadic Functions

- Function **cannot know**:
  - Number of arguments
  - Types of arguments
- Caller and callee must **agree on structure**
- No built-in runtime validation
- Compiler usually cannot detect mismatches

10. Undefined Behavior Risks
Occurs if:

- Wrong argument type is retrieved
- Too few arguments are supplied
- Type promotions are misunderstood
- Format strings do not match arguments
- NULL passed without correct type (int* vs void*)

Even a simple printf("Hello %d") without an argument is undefined behavior.

11. Passing Variadic Arguments Safely

- Many libraries provide **v-functions**
  - Accept va_list instead of raw arguments
  - Example: vprintf
- Passing va_list by reference avoids unsafe copying
- Providing variadic APIs **without** va_list versions is bad practice

12. Variadic Functions vs Variadic Macros

- Variadic macros can sometimes bypass limitations
- Variadic functions are runtime-based and riskier
- Portability suffers if relying on compiler extensions

13. Relation to Dynamic Memory

- Variadic functions pair well with dynamic allocation
- Enable:
  - Flexible array creation
  - Runtime-sized processing
- Must still control memory growth manually

14. Demo Program Concept (Marks Average)

- User enters number of subjects at runtime
- Function accepts variable number of marks
- Uses variadic arguments to:
  - Sum values
  - Compute average
- Demonstrates flexibility but highlights:
  - Need for dynamic memory for true efficiency

15. Program Design Process Recap

1. Understand the problem
2. Propose multiple solutions
3. Choose the best strategy
4. Write pseudocode
5. Draw flowchart
6. Implement actual code
7. Test and refine

Skipping planning leads to inefficient or broken programs.

16. The Core Takeaways

- Variadic functions trade **flexibility for safety**
- Compiler trusts the programmer completely
- Argument count and type must be manually controlled
- stdarg.h is mandatory
- Always provide a clear termination condition
- Misuse leads to silent, dangerous bugs

Computer Science(UX Design) -Lecture 26

1. What the Software Development Process Is

- A **structured approach** to building, deploying, and maintaining software.
- Applies to **software, hardware, systems engineering, and information systems**.
- Necessary for **large or complex projects** to avoid failure.
- Follows a **life cycle** from idea → development → maintenance → retirement.

2. Why Planning Is Non-Negotiable

- Skipping planning leads to:
  - Scope confusion
  - Budget overruns
  - Poor quality
  - Project failure
- Planning enables:
  - Strategic decision-making
  - Correct prioritization
  - Risk management
  - Controlled change handling

3. Core Reasons for Planning a Software Project

1. **Define goals**
2. **Determine requirements**
3. **Estimate costs**
4. **Create timelines**
5. **Improve software quality**
6. **Understand project value**

4. Goal Definition

- Transforms vague ideas into **clear, realistic objectives**
- Unrealistic goals significantly increase failure risk
- Goals must be **measurable and achievable**

5. Requirements Determination

- Identifies everything needed to build the software:
  - Team, Skills, Hardware/software, User input, Budget
- Requirements often **change during development**
- Without planning, changes cause chaos and overspending

6. Cost Planning

- Determines: Development cost, Operational cost, Return on investment
- Prevents waste and financial failure
- Closely tied to **scope definition**

7. Timeline Creation

- Establishes: Milestones, Deadlines, Deliverables & Its Essential for multi-person projects
- Software is never “perfect”; timely release matters

8. Quality and Risk Control

- Planning provides:
  - Benchmarks for quality
  - Metrics for evaluation
- Deviations are allowed but must be **controlled**
- Enables handling unexpected issues without panic

9. Understanding Project Value
- Value is evaluated continuously using:
  - Milestones, Deliverables, Cost vs benefit
- Helps decide whether to continue, pivot, or terminate a project

10. Software Development Life Cycle (SDLC)
Standard stages followed by most methodologies:
- Analysis, Feasibility Study, Design, Coding, Testing, Deployment, Maintenance

11. Analysis Stage
Purpose: **Decide whether the project is worth doing**
Key activities:
- Gather user requirements
- Identify users, inputs, outputs, environment
- Define objectives
- Identify available resources
- Analyze competitors (if applicable)

Sub-steps:
1. Requirements gathering
2. Scope definition
3. Risk, cost, and quality planning

12. Scope Definition
- Defines **upper and lower limits** of the solution
- Prevents infinite expansion of features
- Balances: User needs, Time, Budget & Resources

13. Feasibility Study
Determines whether the project is **practical and viable**
Produces: **Software Requirements Specification (SRS)**
Five Feasibility Aspects
1. **Technical** – Can existing systems handle it?
2. **Economic** – Is it financially viable?
3. **Legal** – Is it legally compliant?
4. **Operational** – Can it be operated and maintained?
5. **Schedule** – Can it be completed on time?

Failure here usually guarantees project failure later.

14. Design Phase

Produces two documents:

14.1 High-Level Design (HLD)

- Module overview
- Module interactions
- System architecture
- Technology stack
- Database identification

14.2 Low-Level Design (LLD)

- Detailed logic
- Data structures
- Database schemas
- UI details
- Error handling
- Input/output mapping

These documents guide coding and testing.

15. Coding Phase

- Actual implementation of the system
- Developers follow HLD and LLD
- Modules assigned to teams
- Longest phase of the life cycle
- Language and tools already chosen

16. Testing Phase

Ensures correctness and quality.

Testing Types

- **Black-box testing**
  - Tests external behavior
  - Focuses on user experience
- **White-box testing**
  - Tests internal logic
  - Focuses on correctness and structure

Goal: eliminate as many bugs as possible before release.

17. Deployment Phase

- Software installed in real environment
- Pilot testing performed
- Final commissioning after user approval
- Some bugs may appear only after deployment

18. Maintenance Phase

Continues for the rest of the software's life.

Types of Maintenance

- **Preventive:** Bug fixes, OS compatibility updates, Performance optimization
- **Perfective:** New features, Enhancements based on feedback
- **Adaptive:** Internal improvements, UI updates, Technology upgrades

19. Major SDLC Models

19.1 Waterfall Model

- Linear, sequential stages
- Each phase must finish before next begins

Best suited for:

- Small projects
- Stable requirements
- Fixed scope

Advantages

- Simple and structured
- Well-documented
- Easy task distribution

Disadvantages

- Late visibility of working software
- Poor at handling changes
- High risk for complex projects

19.2 V-Model

- Extension of waterfall
- Each development phase has a corresponding testing phase

Testing stages:

1. Unit testing
2. Integration testing
3. System testing
4. Acceptance testing

Same strengths and weaknesses as waterfall, with better testing alignment.

19.3 Iterative Model

- Software built in **small increments**
- Each iteration delivers working functionality

Advantages

- Early usable product
- Supports parallel work
- Lower cost of changes

Disadvantages

- Requires strong management
- Less detailed initial requirements
- Not ideal for small projects

19.4 Spiral Model

- Combines iterative development with risk analysis
- Heavy focus on customer feedback

Best suited for:

- High-risk projects
- Changing requirements
- Tight budgets

Disadvantages

- Complex management
- High documentation overhead
- Not suitable for low-risk projects

20. Agile Development

- Based on **iterative and incremental delivery**
- Emphasizes:
  - Customer collaboration, Rapid delivery, Continuous improvement

Used in frameworks like:

- Scrum, Kanban, Extreme Programming (XP), Feature-Driven Development

21. Agile vs Waterfall (Key Differences)

- Agile is flexible; waterfall is rigid
- Agile delivers early and often; waterfall delivers late
- Agile embraces change; waterfall resists it
- Agile prioritizes working software; waterfall prioritizes documentation
- Agile integrates testing continuously; waterfall tests late

22. Seven Root Causes of Project Failure

1. **Obscurity** – Unclear goals
2. **Insentience** – Incompetent leadership
3. **Tardiness** – Missed deadlines
4. **Lethargy** – Lack of urgency
5. **Inevitability** – Ignoring risk
6. **Obsequiousness** – Poor authority and scope creep
7. **Evolution** – Uncontrolled change

23. The Core Takeaways

- SDLC stages are universal across methodologies
- Planning saves time, money, and effort
- Scope control is critical
- Feasibility determines success early
- Design documents guide everything downstream
- Testing is continuous, not optional
- Maintenance dominates software lifespan
- Agile fits modern, fast-changing environments
- Poor planning is the primary cause of failure

Computer Science(UX Design) -Lecture 27

1. Why Agile Was Needed (Problem with Waterfall)

1. **Rigidity**
   - Changes after project start were extremely costly.
   - Requirements fixed too early.
2. **Long Development Cycles**
   - Software projects could last years.
   - By completion, technology and business needs had already changed.
3. **Late Feedback**
   - Customers saw the product only near the end.
   - High risk of building the *wrong* product.
4. **High Failure Rate**
   - Many projects were cancelled mid-way as they no longer met business needs.

2. Core Idea Behind Agile

1. **Expect Change, Don't Fight It**
2. **Deliver Early and Frequently**
3. **Customer Involvement Throughout**
4. **Working Software Over Heavy Documentation**
5. **Iterative and Incremental Development**

Agile focuses on **fast feedback**, **continuous improvement**, and **flexibility**.

3. Key Differences: Agile vs Linear (Waterfall)

Aspect	Planning	Change Handling	Customer Feedback	Delivery	Risk
Waterfall	Heavy upfront	Difficult	Late	End of project	High
Agile	Minimal upfront	Built-in	Continuous	Every iteration	Reduced early

4. Benefits of Agile Development

1. **Easy Adaptation to Change**
2. **Lower Technical Debt:** Maintenance tasks handled in each sprint.
3. **Risk Reduction:** Early testing and feedback.
4. **Higher Quality:** Small features perfected before release.
5. **Predictable Delivery:** Short iterations with visible progress.
6. **Better Communication:** Daily interaction between team and client.

5. Agile Is Iterative by Nature

1. Work starts **immediately**
2. Deliver **imperfect but usable** software early

3. Improve through repeated iterations
4. Final product evolves based on real feedback

6. Scrum Framework (Agile Implementation)

6.1 Scrum Roles

1. **Product Owner**
   - Defines features, Prioritizes backlog, Decides release dates, Accepts or rejects work
2. **Scrum Master**
   - Facilitates Scrum process, Removes obstacles, Shields team from external interference, Ensures rules are followed
3. **Development Team**
   - 5–9 members, Self-organizing, Cross-functional, Responsible for development and delivery

6.2 Scrum Artifacts

- **User Stories:** Describe *what* the system does (not how).
- **Product Backlog:** All approved user stories.
- **Release Backlog:** Stories planned for a release.
- **Sprint Backlog:** Stories selected for the sprint.
- **Burndown Chart:** Visual progress tracking.
- **Block List:** Obstacles affecting progress.

6.3 Scrum Ceremonies

1. **Sprint Planning**
2. **Daily Scrum:** 15 minutes
3. **Sprint Review:** 2–4 hours
4. **Sprint Retrospective:** Process improvement

7. Scrum Process Flow

1. Product Owner creates backlog
2. Sprint planning → sprint backlog
3. Sprint (2–4 weeks)
4. Daily scrums
5. Sprint review
6. Repeat until:
   - Budget ends
   - Deadline reached
   - Product owner satisfied

8. Extreme Programming (XP)

8.1 Purpose

- Designed for **small teams**
- Handles **unclear and changing requirements**
- Assumes cost of change can remain constant

8.2 Core XP Practices

1. Unit tests before coding
2. Continuous testing and integration
3. Pair programming
4. Simple design
5. Frequent releases
6. Customer always involved

8.3 Advantages of XP

1. Faster delivery
2. Very low defect rate
3. Strong customer alignment
4. Reduced project cost
5. High team cohesion

9. Kanban Methodology

9.1 Origin

- Inspired by **Toyota's manufacturing system**
- Focuses on **flow efficiency**

9.2 Core Values

1. Transparency
2. Balance
3. Collaboration
4. Customer focus
5. Flow
6. Leadership
7. Agreement
8. Respect

9.3 Key Kanban Practices

1. Visualize work (Kanban board)
2. Limit Work-In-Progress (WIP limits)
3. Manage flow
4. Explicit policies
5. Feedback loops
6. Continuous improvement

9.4 Kanban Roles (Optional)

1. **Service Request Manager**
2. **Service Delivery Manager**

(No mandatory roles — existing roles adapt.)

9.5 Kanban Feedback Cadences

1. Strategy review – quarterly
2. Operations review – monthly
3. Risk review – monthly
4. Service delivery review – bi-weekly
5. Replenishment meeting – weekly
6. Daily Kanban meeting
7. Delivery planning – per release

10. DevOps

10.1 What DevOps Is

- Collaboration between **development and operations**
- Enables **continuous integration and delivery**

10.2 Core DevOps Principles

1. Customer-centric action
2. End-to-end responsibility
3. Continuous improvement
4. Automation everywhere
5. Monitoring and testing
6. One unified team

10.3 DevOps Life Cycle

1. Development
2. Testing
3. Integration
4. Deployment
5. Monitoring

10.4 DevOps Benefits

1. Faster time-to-market
2. Lower failure rates
3. Easy rollback
4. Higher system stability
5. Reduced risk
6. Cost efficiency

11. Feature-Driven Development (FDD)

11.1 Characteristics

1. Short iterations (≈2 weeks)
2. Feature-centric
3. Strong focus on quality
4. Minimal overhead
5. Accurate progress tracking

11.2 Key FDD Roles

1. Project Manager
2. Chief Architect
3. Development Manager
4. Chief Programmers
5. Class Owners
6. Domain Experts
7. Supporting Roles (testers, build engineers, etc.)

11.3 FDD Best Practices

1. Domain object modeling
2. Develop by feature
3. Individual class ownership
4. Regular inspections
5. Frequent builds
6. Visible reporting

12.The Core Takeaways

1. Agile exists because **change is inevitable**
2. Agile prioritizes **working software and customer feedback**
3. Scrum = roles + artifacts + ceremonies
4. XP = Agile pushed to the extreme
5. Kanban = flow-based, no fixed roles
6. DevOps = Agile + Operations automation
7. FDD = feature-centric agile model
8. Iteration + feedback = reduced risk and higher quality

Computer Science(UX Design) -Lecture 28

1. What a File Is

1. A **file** is the basic unit of data storage in a computer.
2. Everything in a computer system (programs, OS, user data) is stored as files.
3. A file is a **container of binary data (0s and 1s)** formatted so the computer knows:
    - where data starts
    - where data ends
4. The **operating system manages files**.

2. Why Files Are Necessary

1. Binary data by itself is meaningless without structure.
2. Files separate and organize binary sequences.
3. Files allow:
    - identification of data, reuse, storage beyond RAM

3. Storage and File Paths

1. Files are stored on:
    - Internal storage (hard drive / SSD)
    - Removable storage (USB, external drives)
2. Files are accessed using a **file path**.
3. A file path is a string that locates a file in a **directory structure**.

4. File Directories and Structure

1. All files are organized in a **file directory system**.
2. Directories are structured as a **tree**:
    - Root at the top, Subdirectories and files below
3. A file path represents traversal through this tree.

5. History of Files

1. Term “file” used in computing since the **1940s**.
2. Initially referred to **punched cards**.
3. Later evolved to represent **binary data stored on disks**.
4. Modern meaning: arbitrary binary data stored on storage media.

6. Operating Systems and Files

1. An operating system itself is a **collection of files**.
2. Types of OS files:
    - Executables (kernel, drivers, applications)
    - Non-executables (configs, documents, images)
3. Files are organized by a **file system**.

7. File Systems

1. A file system defines:
    - How files are stored/retrieved & Permissions on files
2. Without a file system, data access would be chaotic.
3. Permissions restrict access to files (read/write/execute).

8. Key Benefits of Using Files

1. **Reusability** – data can be reused later.
2. **Large Storage Capacity** – storage > RAM.
3. **Time Saving** – avoids repeated manual input.
4. **Portability** – files can be transferred across systems.

9. Why Programs Need Files

1. Data in memory is lost when the program exits.
2. Files provide **persistent storage**.
3. Real-world programs handle large datasets.
4. Files reduce: user effort, human error

10. File Usage in C Programming

1. C allows: Reading from files, Writing to files
2. Enables: Configuration storage, Data processing and persistence
3. Most software bundles: Executables, Configuration files

11. Configuration Files

1. Store settings such as: display resolution, refresh rate, hardware properties
2. Usually stored as **text files**.
3. Loaded automatically when programs start.

12. File Systems Across Operating Systems

12.1 Windows File System

1. Uses **drive letters** (C:, D:)
2. Uses **backslashes (\)** in paths.
3. File path format:
4. C:\folder\subfolder\file.ext

12.2 Linux File System

1. No drive letters.
2. All storage unified under **single root (/)**.
3. Devices listed under /dev.
4. Uses **forward slashes (/)**.
5. File path format:
6. /folder/subfolder/file.ext

13. File Extensions and Encoding

1. File extension indicates format.
2. Text files commonly use:
    - ASCII
    - ANSI
    - Unicode
3. Encoding defines how text maps to binary.

14. Types of Files in C

14.1 Text Files

1. Human-readable.
2. Can be opened by text editors.
3. Advantages:
   - Easy to edit
   - Highly portable
4. Disadvantages:
   - Low security
   - Larger size
   - No compression

14.2 Binary Files

1. Not human-readable.
2. Stored as raw binary.
3. Advantages:
   - Faster read/write
   - Random access
   - Compact storage
   - Higher security
4. Used for large structured data.

15. Binary Files and File Pointers

1. Binary files behave like arrays on disk.
2. Support **random access**.
3. C uses a **file pointer**:
   - Points to byte location in file
4. Operations move the pointer automatically.
5. fseek() moves pointer manually.

16. File Operations in C

Five fundamental operations:

1. Create a file
2. Open a file
3. Read from a file
4. Write to a file
5. Close a file

16. File Operations in C

Five fundamental operations:

1. Create a file
2. Open a file
3. Read from a file
4. Write to a file
5. Close a file

17. File Pointer in C

1. Files are treated as data structures.
2. A pointer of type FILE * is required.
3. Used to communicate between program and file.

18. Common File Handling Functions in C

18.1 File Control

1. fopen() – open/create file
2. fclose() – close file

18.2 Writing Functions

1. fputc() – write character
2. fputs() – write string
3. fprintf() – formatted write

18.3 Reading Functions

1. fgetc() – read character
2. fgets() – read string
3. fscanf() – formatted read
4. fgetws() – wide string input

18.4 File Positioning

1. fseek() – move pointer
2. ftell() – current position
3. rewind() – move to start

19. File Opening Modes

1. Mode determines file behavior.
2. Example:
   - Read mode cannot create a file.
3. If file does not exist:
   - Some modes create it automatically.

20. Importance of Closing Files

1. Frees system resources.
2. Prevents accidental data corruption.
3. Systems limit number of open files.

21. Command Line Arguments in C

21.1 Purpose

1. Allow programs to behave differently at runtime.
2. Reduce hardcoded logic.

21.2 Components

1. argc – argument count
2. argv – argument vector (array of strings)

21.3 Usage

1. Arguments passed after program name.
2. Typically processed using loops.
3. Used to configure program behavior dynamically.

22. Demo Program Overview (File Encryption)

1. Program:
   - Creates a file
   - Writes user input
   - Encrypts file
   - Allows decryption
2. Uses:
   - File I/O
   - Menu-driven logic
   - Switch-case
   - Character arithmetic

23. Encryption Logic Used

1. Encryption:
   - Adds 100 to each character.
2. Decryption:
   - Subtracts 100.
3. Demonstrates:
   - File reading
   - File writing
   - Pointer movement

24. Limitations of the Demo Program

1. Cannot detect encrypted vs decrypted state.
2. Double decryption corrupts data.
3. No state persistence.
4. Limited navigation in menu.

25. Suggested Improvements

1. Use a **flag stored in file** to track encryption state.
2. Improve menu navigation.
3. Better error handling.
4. Make program more portable.

26. The Core Takeaways

1. Files store persistent binary data.
2. OS manages files via file systems.
3. Directories form a tree structure.
4. C supports text and binary files.
5. Binary files offer speed and random access.
6. File pointers control file navigation.
7. Always close files.
8. Command line arguments enable flexible programs.

Computer Science(UX Design) -Lecture 29

1. What a Bug Is

1. A **bug** is an error, flaw, or fault that causes a program to:
   - behave unexpectedly, or
   - produce incorrect results
2. Bugs can exist in **software and hardware**.
3. Bugs are **unintentional** mistakes made by developers.
4. A bug is **not** malicious behavior.

2. Bug vs Virus (Important Distinction)

1. **Bug:** Unintentional error & Caused by developer mistakes
2. **Virus / Malware:** Intentional behavior & Designed to steal data, damage systems, or misuse resources

- Therefore: Malware ≠ Bug

3. Origin of the Term "Bug"

1. Term used in engineering since the **1800s**.
2. First recorded computer bug:
   - **September 9, 1947**
   - A moth found in the Harvard Mark II relay computer
3. The incident popularized the term "computer bug".
4. The moth and logbook are preserved in a museum.

4. Real-World Cost of Bugs (Why They Matter)

4.1 Ariane 5 Rocket Failure (1996)

1. Cause: **Integer overflow**
2. Error: 64-bit number forced into 16-bit space
3. Result:
   - Rocket exploded 40 seconds after launch
   - Loss ≈ **$370 million**

4.2 Y2K Bug (Millennium Bug)

1. Cause: Years stored using two digits (e.g., 60 instead of 1960)
2. Problem: Year 2000 interpreted as 1900
3. Impact:
   - Banking systems
   - Power plants
   - Transportation systems
4. Fixing cost: **Billions worldwide**

4.3 Mars Climate Orbiter (1998)

1. Cause: **Unit mismatch**
   - Imperial vs Metric units
2. Result:
   - Incorrect trajectory
   - Orbiter destroyed
3. Loss ≈ **$125 million**

4.4 Integer Overflow Example (YouTube)

1. View count exceeded 32-bit integer limit
2. Limit: 2,147,483,647
3. Gangnam Style exceeded this value
4. Caused system overflow

5. Bug Classification by Nature

5.1 Functional Defects

1. Software does not meet functional requirements
2. Example:
    - Accepts floats instead of integers
3. Found during **functional testing**

5.2 Performance Defects

1. Poor speed, stability, or resource usage
2. Example: 5-second delay after mouse click
3. Found during **performance testing**

5.3 Usability Defects

1. Software is hard to use or overly complex
2. Example:
    - Multi-step signup when fewer steps are required
3. Found during **usability testing**

5.4 Compatibility Defects

1. Software behaves differently across:
    - Devices, OS versions, Platforms
2. Example:
    - Works on desktop but breaks on mobile
    - Works on Android 9 but not Android 11

5.5 Security Defects

1. Vulnerabilities in code
2. Examples:
    - Weak authentication
    - Buffer overflows
    - Encryption errors
3. Increases attack surface

6. Bug Classification by Severity

6.1 Critical

1. Testing cannot proceed
2. Core functionality broken
3. Must be fixed immediately

6.2 High

1. Major functionality affected
2. Software partially usable
3. Must be fixed before release

6.3 Medium

1. Minor functionality affected
2. Does not break core logic
3. Can be deferred

6.4 Low

1. Cosmetic issues
2. Example:Text alignment
3. Can be released with defect

7. Bug Classification by Priority

7.1 Urgent

1. Needs immediate fix
2. Can include low-severity bugs if user impact is high

7.2 High Priority

1. Fixed in upcoming release
2. Users can temporarily work around it

7.3 Medium Priority

1. Fixed in later release
2. Not blocking users

7.4 Low Priority

1. Minimal user impact
2. Fixed when time permits

8. Importance of Correct Severity & Priority

1. Speeds up defect resolution
2. Improves testing efficiency
3. Helps meet deadlines and milestones

9. How to Avoid Bugs (Prevention)

9.1 Code Simplification

1. Functions should do **one thing only**
2. Avoid overly complex logic

9.2 Modularity

1. Code divided into:Functions, Modules, Macros
2. Easier to isolate and test bugs

9.3 Clear Dependencies

1. Functions interact via: Well-defined parameters, Shared variables
2. Avoid hidden coupling

9.4 Use Existing Libraries

1. Prefer well-tested libraries
2. Avoid reinventing solutions
3. Reduces debugging effort

10. Debugging Basics

1. Debugging applies to **hardware and software**
2. Bugs appear:
    - During testing
    - After deployment
3. Debugging is part of the **software development life cycle**

11. Debugging Process (Steps)

1. Identify the error
2. Locate the error
3. Analyze the code
4. Prove the analysis
5. Build the fix
6. Test and validate

12. Explanation of Debugging Steps

12.1 Error Identification

1. Correct problem understanding is critical
2. Poor identification wastes time
3. User reports may be vague

12.2 Error Location

1. Find where the bug originates
2. No fixing without location

12.3 Code Analysis

1. Understand:
   - What code does
   - What it should do
2. Check ripple effects
3. Assess fix risks

12.4 Prove the Analysis

1. Document bug behavior
2. Write automated tests
3. Prevent regression

12.5 Build the Fix

1. Modify code
2. Append or replace logic

12.6 Test and Validate

1. Retest entire program
2. Ensure fix didn't add new bugs

13. Fundamental Debugging Rule

1. Know what the program is supposed to do
2. Detect when it doesn't
3. Fix it
4. Repeat

14. Common Debugging Mistakes

1. Randomly changing code
2. Debugging by output only
3. Not understanding program flow
4. Ignoring input data
5. Debugging wrong source version

15. Use of Assertions

1. assert() checks assumptions
2. Stops program if condition fails
3. Helps pinpoint errors early
4. Useful during development

16. Debugging Tools

16.1 GDB

1. Step-by-step execution, Inspect variables and flow
2. Available on Linux and some Windows compilers

16.2 Valgrind

1. Detects: Memory leaks & Invalid memory access
2. Useful for pointer-heavy programs, Helps improve security

17. Breakpoints

1. Intentional stop points in execution
2. Used to inspect program state
3. Triggered by conditions or tool rules

18. Debugging Strategies

18.1 Automated Testing

1. GUI testing
2. API testing
3. Key part of agile development

18.2 Incremental Development

1. Build small units
2. Test after each addition
3. Reduces bug accumulation

18.3 Logging

1. Write execution steps to log file
2. Helps debug production issues
3. Useful for users and developers

18.4 Error Clustering

1. Group similar bugs
2. Fix root cause
3. Can resolve multiple issues at once

18.5 Debugging Output

1. Useful only if code is understood
2. Patterns in output can guide fixes
3. Not a substitute for code analysis

18.6 Change Your Perspective

1. Bug may not be where expected
2. Question assumptions
3. Inspect test cases and input data
4. Explain problem aloud

18.7 Ensure Correct Build

1. Debug correct source version
2. Verify libraries and build configuration
3. Use build tools properly

18.8 Take Breaks

1. Fatigue reduces effectiveness
2. Fresh perspective helps
3. Often reveals overlooked bugs

19. The Core Takeaways

1. Bugs are unavoidable but manageable
2. Bugs can be extremely expensive
3. Classification helps manage fixes
4. Severity ≠ Priority
5. Prevention is better than debugging
6. Debugging is systematic, not random
7. Tools help, understanding matters more

Computer Science(UX Design) -Lecture 30

1. Bugs vs Errors

- **Bug**:
  - Caused by incorrect program logic or implementation
  - Program behaves differently from what the programmer intended
  - Examples: wrong calculations, missing menu, incorrect output
- **Error**:
  - Caused by **external or unexpected runtime events**
  - Not a fault in program logic
  - Examples: removed flash drive, missing file, invalid user input
- Not all bugs produce errors
- Not all errors are bugs

2. Errors vs Exceptions (Conceptual Mapping)

- In many languages, runtime errors are called **exceptions**
- In C, errors are typically handled manually
- Errors occur **during execution**, not compilation
- Programs should handle **foreseeable errors** gracefully

3. Good Error Handling Practices

- Programs should:
  - Detect errors
  - Inform users clearly
  - Avoid cryptic or technical messages
- Error messages should:
  - Explain what happened
  - Explain what the user can do next
- Poor error messages reduce usability and reliability

4. Error Handling in C (Big Picture)

- C provides **no built-in exception system**
- Error handling is mostly the **programmer's responsibility**
- Support exists via:
  - Header files
  - Return values
  - Global and local indicators
  - Signals and jumps

5. Two Phases of Error Handling

1. **Error Detection**
   - Identify that something went wrong

2. **Error Recovery**
    - Decide how to respond without crashing
    - Restore program to a valid state if possible

6. Error Handling Strategies in C

Common approaches include:

1. Prevention
2. Termination
3. Global error indicators
4. Local (non-global) error indicators
5. Return values
6. Assertions
7. Signals
8. Goto chains
9. Non-local jumps
10. Error callbacks

7. Error Prevention

- Write code that **prevents invalid states**
- Examples:
    - Restrict input to digits for numeric fields
    - Limit ranges (dates, sizes, lengths)
- Prevention is not always practical
    - Some mathematical operations naturally overflow
    - Over-restriction can reduce functionality

8. Termination (Fail-Hard Strategy)

- Used when execution **must not continue**
- Program exits immediately
- Useful when:
    - Continuing would corrupt data
    - State cannot be recovered
- Libraries should avoid forcing termination unless unavoidable
- Responsibility shifts to higher-level code to decide recovery

9. Global Error Indicators (Fail-Soft)

- Use shared variables to indicate error state
- Common examples: errno, Floating-point error indicators, Platform-specific equivalents

Key Rules for errno

- Set errno = 0 before calling a function
- Check it **only after** a function signals failure
- Library functions must not reset it to zero
- Meaningful only when the function explicitly documents its use
- Misuse can silently corrupt program state

10. Local (Non-Global) Error Indicators

- Used for **object-specific error states**
- Common in file handling

- Typical functions: ferror, clearerr
- Safer and more contained than global indicators

11. Jumps for Error Handling

- Uses goto or similar jump mechanisms
- Allows jumping to cleanup or recovery code
- Strongly discouraged because:
  - Breaks structured flow
  - Creates hard-to-debug logic
  - Easy to forget or misuse

Use structured control flow instead whenever possible.

12. Return Values as Error Indicators

- Many C functions signal errors via return values
- Example:
  - malloc() returns NULL on failure

Pros	Cons
• Simple • Explicit • Widely used	• Return value cannot carry other information • Easy to ignore • Ignoring return values can cause security vulnerabilities

13. Responsibility of the Caller

- The **caller must always check**:
  - Return values
  - Error indicators
- Ignoring errors can lead to:
  - Crashes
  - Data corruption
  - Security flaws

14. Error Handling Decision Flow

- Choose error handling based on:
  - Severity
  - Recoverability
  - Scope (local vs global)
- No single method fits all cases
- Mixing methods carefully is common in real systems

15. Project Context: Student Records System

Goal:

- Store, retrieve, delete, and view student records
- File-based persistent storage
- Login-protected system

16. Flowchart Design (High-Level Logic)

Main stages:

1. Start
2. Initialization
3. Login
4. Main Menu
5. Operations:
    - Add student
    - Search records
    - View records
    - Delete records
    - Change credentials
6. Return to menu after each operation
7. Exit program

17. Initialization Phase (Critical Setup)

Includes:

- Variable initialization
- File creation/checking
- Credential setup
- Defining limits:
    - Year ranges
    - String sizes
    - File names

18. Data Structures Defined

- Date structure
- Credential structure
- Student structure:
    - Name
    - Address
    - Guardian
    - Enrollment date

19. Functional Decomposition

Planned functions include:

- Initialization
- Login handling
- Menu control
- Add student
- Search student
- View records
- Delete records
- Update credentials

- Duplicate checking
- Input validation
- File validation

20. Pseudocode Development Strategy

- Start from flowchart blocks
- Translate each block into:
  - Functions
  - Variables
  - Structures
- Gradually refine into executable code
- Keep track of required components as development progresses

21. The Core Takeaways

- Bugs ≠ errors
- Errors can be external and unavoidable
- C relies on manual error handling
- Always check return values
- Use error indicators carefully
- Prefer clarity and safety over cleverness
- Design flow before writing code
- Initialization and validation prevent most runtime failures

Computer Science(UX Design) -Lecture 31

1. What Version Control Is (Core Idea)

- **Version Control Systems (VCS)** manage multiple versions of files over time.
- Allow multiple people to work on the same project without overwriting each other.
- Preserve history so changes can be reviewed, reverted, or merged.
- Essential for **modern software development**, especially with large codebases.

2. Why Version Control Became Necessary

- Early programs were small (hundreds–thousands of lines).
- Modern software (e.g., operating systems) can contain **millions of lines of code**.
- Agile development creates **frequent revisions**.
- Only one version is released, but many exist during development.
- Without VCS:
  - Changes are lost
  - Mistakes are irreversible
  - Collaboration becomes unsafe

3. Version Control ≠ Just Software Development

- Exists in:
  - Cloud storage (file history)
  - Word processors and spreadsheets
  - Operating systems
  - Content management systems
  - Collaborative platforms like Wikipedia
- Fundamentally about **tracking change over time**.

4. What Version Control Provides

- Complete change history
- Timestamps and author tracking
- Ability to:
  - Compare versions
  - Revert changes
  - Merge work
- Backup against accidental deletion
- Safe collaboration at scale

5. Historical Evolution of VCS

- Early systems:
  - SCCS (1970s), RCS, CVS
- Modern systems:
  - Subversion
  - **Git** (dominant today)

Terminology equivalence:

- Version Control
- Source Control
- Source Code Management
- Revision Control

All refer to the same concept.

6. Importance in Team Development

- Multiple developers work concurrently.
- Each developer can: Access full history, Restore removed code
- Enables:
  - Parallel work, Feature isolation, Safe experimentation
- Without VCS, development is fragile and risky.

7. Key Capabilities of an Effective VCS

1. **Complete change history**
2. **Concurrent development**
3. **Branching**
   - Isolate new features or bug fixes
4. **Merging**
   - Integrate approved changes
5. **Development tracking**
   - Know who changed what, when, and why

8. Scale of Modern Version Control Usage

- Platforms like **GitHub** host:
  - Tens of millions of users
  - Hundreds of millions of repositories
- Demonstrates how fundamental VCS is to software engineering.

9. Types of Version Control Systems

9.1 Local Version Control

- Files stored on a single machine
- Simple but risky
- Easy to overwrite or lose data
- No collaboration support

9.2 Centralized Version Control (CVCS)

- Single central server holds repository
- Developers check out files from server
- Advantages: Central visibility, Access control
- Disadvantages: **Single point of failure,** Server downtime halts work

9.3 Distributed Version Control (DVCS)

- Each developer has a **full copy of the repository**
- Local repositories contain:
  - All files, Full history
- Advantages:
  - Offline work, Built-in backups, Efficient branching and merging
- Central server used mainly for coordination, not dependency

10. File Access Models in VCS

10.1 File Locking

- Only one developer edits a file at a time, Others can only read
- Pros: Prevents merge conflicts
- Cons: Bottlenecks, Encourages bypassing the system, Forgotten locks block progress

10.2 Revision Merging

- Multiple developers edit simultaneously
- System merges changes on check-in
- Works best for: Text files
- Problematic for: Binary files
- Requires manual conflict resolution when overlaps occur

11. Reserved Edits

- Explicitly lock files even when merging is possible
- Useful for:
  - Large or sensitive changes
  - Complex refactors

12. Baselines, Labels, and Tags

- **Label / Tag**: Identifies a snapshot in history
- **Baseline**:
  - A snapshot of special importance, Often used for releases or milestones
- Meaning:
  - Baseline = significance
  - Tag/Label = mechanism

13. Core Version Control Terminology

- Baseline: Approved reference version for future changes
- Atomic Operation: Either fully completes or leaves system unchanged

Commit

- Records changes permanently
  - Includes: Author, Timestamp, Message
- **Branch:** Independent line of development
- **Change / Diff / Delta:** A specific modification to files
- **Change List:** Group of changes committed together

14. Repository Operations

- **Checkout:** Create a local working copy
- **Clone:** Copy entire repository including history
- **Pull:** receive changes, **Push:** send changes
- **Fetch:** Retrieve changes without updating working files

15. Conflicts and Resolution

- Conflict occurs when:
  - Multiple edits affect the same region
- Resolution requires:
  - Manual intervention
  - Choosing or combining changes

16. Merging Scenarios

- Sync local work with upstream changes
- Merge branches back into main line
- Apply fixes across branches (cherry-picking)

17. Pull Requests

- Formal request to merge changes
- Enables: Review, Discussion, Controlled integration

18. Role of External Knowledge Bases

- Platforms like **Stack Overflow**:
  - Help debug errors
  - Explain language behavior
  - Share best practices
- Acts as a practical extension of documentation.

19. Project Context: Student Records System

Goal:

- File-based student management system
- Operations: Add, Search, View, Delete, Update credentials
- Protected by login system

20. Pseudocode as a Design Tool

- Bridges flowchart and real code
- Acts as: Low-level design document, Coding roadmap
- Focuses on: Logic, Order, Responsibility separation

21. Functional Breakdown (Project)

Key functions include:

- Initialization
- File checking
- Login handling
- Menu routing
- Input validation
- Duplicate detection
- CRUD operations
- Credential updates

22. Design Philosophy Emphasized

- No single correct solution
- Focus on:
  - Correctness
  - Efficiency
  - Maintainability
- Structure first, implementation second

23. The Core Takeaways

- Version control is **non-optional** in real software
- Always track changes
- Always preserve history
- Prefer distributed systems for resilience
- Branch before experimenting
- Merge carefully
- Use pseudocode to reduce coding errors
- Tools support you, but **design discipline matters more**

Computer Science(UX Design) -Lecture 32

1. Purpose of the Final Demo

- Review the **complete working solution** for the project.
- Code shown is **one valid implementation**, not a universal solution.
- Focus is on:
  - Correctness
  - Structure
  - Practical application of concepts from the module
- Emphasis on **understanding design choices**, not memorising code.

2. Development Flow Recap

The solution followed a disciplined progression:

1. Problem description
2. Problem statement
3. Flowchart
4. Pseudocode
5. Full implementation
6. Testing and validation

Each stage reduced ambiguity and coding errors.

3. Program Overview

- Command-line student records system
- Stores data in a **binary file**
- Supports:
  - Login authentication
  - Add student
  - Search student
  - View all students
  - Delete student
  - Update credentials
- Program structure is modular and function-driven
- Main function is minimal and orchestration-focused

4. Header Files and Libraries Used

- stdio.h → input/output
- stdlib.h → memory management and utilities
- string.h → string manipulation
- time.h → date handling

Each header has a **clear functional purpose**.

5. Macros and Configuration Constants

Defined at the top for safety and maintainability:

- Year limits (min/max)
- Username length
- Password length
- File name
- Maximum field sizes (names, address, etc.)

This prevents:

- Magic numbers
- Input overflows
- Inconsistent validation logic

6. File Storage Design

- Uses a **binary file**
- Advantages:
  - Faster access
  - Structured storage
  - Less human-readable (basic security)
- Same file stores:
  - Credentials
  - Student records

7. Data Structures

Date Structure

- Year
- Month
- Day

Credentials Structure

- Username
- Password

Student Record Structure

- Student ID
- Guardian name
- Student name
- Address
- Enrollment date

All structures are defined **before function usage** to avoid compilation issues.

8. Header Display Function

- Prints centered titles for each screen
- Uses:
  - String length
  - Screen width calculation
- Improves readability and user experience
- Reused across multiple functions

9. Welcome Screen Function

- Clears screen
- Displays program introduction
- No logic beyond UI presentation

10. Name Validation Function

- Checks input for valid alphabetic characters
- Uses local variables (scope-safe)
- Avoids global variable side effects
- Returns validity status

11. Leap Year Validation Function

- Implements standard leap year rules:
  - Divisible by 4
  - Divisible by 100
  - Divisible by 400
- Returns boolean result
- Used to validate February dates correctly

12. Date Validation Function

Checks:

- Year range (configured limits)
- Month range (1–12)
- Day range (month-specific)
- Leap year rules for February
- Rejects invalid dates through repetition

Ensures **data integrity** before storage.

13. Add Student Function

Core steps:

1. Open file in append binary mode (ab+)
2. Validate file pointer
3. Flush input buffer
4. Read student ID
5. Check for duplicate ID
6. Capture guardian name
7. Capture student name
8. Capture address
9. Validate and capture date
10. Write record to file
11. Close file

Prevents: Duplicate records, Invalid input, Corrupt file writes

- Reads through file using student ID
- If ID exists:
  - Displays record
  - Aborts insertion
- If ID does not exist:
  - Allows insertion to continue

Maintains **database uniqueness**.

15. Search Student Function

- Opens file in **read-only mode**
- Reads records sequentially
- If found: Displays record
- If not found: Displays appropriate message, Always closes file after operation

Follows **least-privilege access** principle.

16. View All Students Function

- Opens file in read-only mode
- Iterates through all records
- Displays: Student details, Record count
- Handles empty database case
- Closes file after reading

17. Delete Student Function

- Opens original file (read)
- Opens temporary file (write)
- Copies all records **except the one to delete**
- If deletion succeeds:
  - Replaces original file with temporary file
- Ensures data safety even if deletion fails

Trade-off: Less efficient, More reliable

18. Update Credentials Function

- Reads new username and password
- Enforces size limits
- Writes updated credentials to file
- Forces logout
- Requires immediate re-login

Ensures credentials are **validated and tested instantly**.

19. Menu Function

- Displays menu options
- Uses switch statement
- Routes execution to correct function
- Handles invalid input gracefully
- Central control point of the program

20. Login Function

- Reads stored credentials from file
- Compares with user input
- Limits attempts to **three**
- Exits program on repeated failure
- Grants access on success

Implements basic authentication security.

21. File Existence Check Function

- Verifies if data file exists
  - Returns status: Exists / Does not exist

Used during initialization.

22. Initialization Function

- Runs on program start
- If file does not exist:
  - Creates binary file
  - Writes default credentials
- Ensures program is usable on first launch

23. Main Function Design

- Extremely minimal:
  1. Initialize system
  2. Display welcome message
  3. Handle login
  4. Exit

All logic delegated to helper functions.

24. Compilation and Runtime Testing

Tested scenarios include:

- Adding valid records
- Rejecting duplicate IDs
- Validating incorrect dates
- Searching existing and non-existing records
- Viewing all records
- Deleting records
- Updating credentials
- Exiting program safely

All core paths tested successfully.

25. The Core Takeaways

- Design before coding saves time
- Binary files improve structure and performance
- Input validation is non-negotiable
- Use read-only access when possible
- Temporary files improve safety during deletion
- Modular functions improve maintainability
- Main function should orchestrate, not compute
- Testing edge cases is as important as normal cases