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