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)
  - $n! = n \times (n-1)!$ 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

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