

- 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

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.

- Used when **multiple mutually exclusive conditions** exist.
- Conditions are evaluated **top to bottom**.
- Once a condition is satisfied:
- Range-based checks (e.g., grading systems)
- If the ladder grows too long, consider switch.
- More powerful, but:
 - Creates **hierarchical decision logic**.
 - An if inside another if or else.
- Nested if statements
- Used for **dynamic memory allocation**
- Memory allocated at runtime
- Characteristics:
 - Slower than stack, Much larger & Size can grow/shrink dynamically
 - Incorrect handling leads to:
 - Global/static variables with initial values, Read-write
 - Contains garbage values initially
 - Stack: Function calls, local variables
 - Heap: Dynamically allocated memory
- Rules:
 - Works like a multi-branch selection tree.
 - Used when selecting **one path from many discrete options**.
 - 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 case
 - default handles unexpected values and ensures safe exit.
- 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:
 - Complex logical expressions are needed
 - Conditions involve ranges
 - Syntactic alternative to simple if-else.
 - Useful for: Simple assignments & Short, readable decisions
- if-else if-else
 - Condition ? value_if_true : value_if_false
- Lessions:
 - Not suitable for complex logic
 - Becomes unreadable when nested

- 7. Program Memory Layout (Low → High Address)
 - 1. Text Segment
 - Executable code, Read-only, Shareable 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
 - 1. malloc
 - Allocates memory (uninitialized)
 - Returns pointer or NULL
 - Allocates memory and initializes to zero
 - 2. calloc
 - Returns pointer or NULL
 - Allocates memory and initializes to zero
 - 3. free
 - Does not return anything
 - Deallocates memory
 - Frees invalid or already freed memory → undefined behavior
 - 4. realloc
 - Returns NULL on failure (old block remains valid)
 - May move memory to a new location
 - Preserves old data up to smaller size
 - Resizes previously allocated memory
- 9. Special realloc Behaviors
 - realloc(ptr, 0) → frees memory, returns NULL
 - realloc(NULL, size) → same as malloc(size)

condition ? value_if_true : value_if_false

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

- 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.

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

4.2 Infinite Loop

- Condition is never true & Loop body never executes.

4. Common Loop Errors

<p>1. Repetition in Programming</p> <ul style="list-style-type: none"> • Repetition allows a program to execute the same block of code multiple times. • Computers repeat a block of code as long as a condition remains true. • If true → execute loop body • Evaluate condition • Header files define interfaces, not standards • Standard library is intentionally minimal • Portability depends on sticking to standard headers • POSIX ≠ Standard C • ISO C • First formal standard (1980s) • C89 / C90 • Adopted in 1990 • Revisions: • C99 • C11 • C18 (latest) • What the Standard Defines • Comments never reach the compiler • Environments • Freestanding ≠ Hosted behavior • Compiler behavior ≠ Preprocessor behavior • Standard library is intentionally portable • Header files define interfaces, not standards • Standard library is intentionally portable • Portability depends on sticking to standard headers • POSIX ≠ Standard C • ISO C • First formal standard (1980s) • C89 / C90 • Adopted in 1990 • Revisions: • C99 • C11 • C18 (latest) • Output representation • Input representation • Semantics • Syntax and constraints • Program representation • Multiple inclusion must be controlled • Compiler never reaches the control flow 	<p>10. Hosted vs Freestanding Environments</p> <ul style="list-style-type: none"> • time.h → date and time functions (second-level precision) • string.h → string manipulation • stdlib.h → memory allocation, conversions, utilities <p>11. User-Defined Header Files</p> <ul style="list-style-type: none"> • Created to build custom libraries • Included using quotes "file.h" • Behave exactly like standard headers • Excessive non-standard headers hurt portability • If true → execute loop body • Evaluate condition • Header files define interfaces, not standards • Standard library is intentionally minimal • Portability depends on sticking to standard headers • POSIX ≠ Standard C • ISO C • First formal standard (1980s) • C89 / C90 • Adopted in 1990 • Revisions: • C99 • C11 • C18 (latest) • What the Standard Defines • Comments never reach the compiler • Environments • Freestanding ≠ Hosted behavior • Compiler behavior ≠ Preprocessor behavior • Standard library is intentionally portable • Header files define interfaces, not standards • Standard library is intentionally portable • Portability depends on sticking to standard headers • POSIX ≠ Standard C • ISO C • First formal standard (1980s) • C89 / C90 • Adopted in 1990 • Revisions: • C99 • C11 • C18 (latest) • Output representation • Input representation • Semantics • Syntax and constraints • Program representation • Multiple inclusion must be controlled • Compiler never reaches the control flow 	<p>12. C Standards Overview</p> <ul style="list-style-type: none"> • ANSI C • First formal standard (1980s) • C89 / C90 • Adopted in 1990 • Revisions: • C99 • C11 • C18 (latest) • What the Standard Defines • Comments never reach the compiler • Environments • Freestanding ≠ Hosted behavior • Compiler behavior ≠ Preprocessor behavior • Standard library is intentionally minimal • Portability depends on sticking to standard headers • POSIX ≠ Standard C • ISO C • First formal standard (1980s) • C89 / C90 • Adopted in 1990 • Revisions: • C99 • C11 • C18 (latest) • Output representation • Input representation • Semantics • Syntax and constraints • Program representation • Multiple inclusion must be controlled • Compiler never reaches the control flow 	<p>2.3 Execution Flow</p> <ul style="list-style-type: none"> • Control variable must be modified inside the loop. • Control variable must be modified inside the loop. • If true → execute loop body • Evaluate condition • Header files define interfaces, not standards • Standard library is intentionally minimal • Portability depends on sticking to standard headers • POSIX ≠ Standard C • ISO C • First formal standard (1980s) • C89 / C90 • Adopted in 1990 • Revisions: • C99 • C11 • C18 (latest) • What the Standard Defines • Comments never reach the compiler • Environments • Freestanding ≠ Hosted behavior • Compiler behavior ≠ Preprocessor behavior • Standard library is intentionally portable • Header files define interfaces, not standards • Standard library is intentionally portable • Portability depends on sticking to standard headers • POSIX ≠ Standard C • ISO C • First formal standard (1980s) • C89 / C90 • Adopted in 1990 • Revisions: • C99 • C11 • C18 (latest) • Output representation • Input representation • Semantics • Syntax and constraints • Program representation • Multiple inclusion must be controlled • Compiler never reaches the control flow
<p>2.4 Common Uses</p> <ul style="list-style-type: none"> • Exit when condition becomes false • Unknown number of iterations, input validation & Event-driven repetition • Guarantees the loop body first, then evaluates the condition. • Executes the loop body first, then evaluates the condition. • do-while: exit-controlled loop • while: entry-controlled loop • When the condition depends on user input • When the condition is never true & Loop body never executes. 	<p>3.1 Concept</p> <ul style="list-style-type: none"> • Guarantees the loop body first, then evaluates the condition. • Executes the loop body first, then evaluates the condition. • do-while: exit-controlled loop • while: entry-controlled loop • When the condition depends on user input • When the condition is never true & Loop body never executes. 	<p>3.2 Difference from While Loop</p> <ul style="list-style-type: none"> • Guarantees the loop body first, then evaluates the condition. • Executes the loop body first, then evaluates the condition. • do-while: exit-controlled loop • while: entry-controlled loop • When the condition depends on user input • When the condition is never true & Loop body never executes. 	<p>3.3 When to Use</p> <ul style="list-style-type: none"> • When the condition depends on user input • When the condition is never true & Loop body never executes.
<p>3.4 Common Loops</p> <ul style="list-style-type: none"> • do-while Loop • for-loop • while Loop • switch Statement • break Statement • continue Statement • goto Statement • label Statement • do-while Loop • for-loop • while Loop • switch Statement • break Statement • continue Statement • goto Statement • label Statement 	<p>3.5 When to Use</p> <ul style="list-style-type: none"> • When the condition depends on user input • When the condition is never true & Loop body never executes. 	<p>3.6 When to Use</p> <ul style="list-style-type: none"> • When the condition depends on user input • When the condition is never true & Loop body never executes. 	<p>3.7 When to Use</p> <ul style="list-style-type: none"> • When the condition depends on user input • When the condition is never true & Loop body never executes.

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

Used mainly in **large or specialized builds**

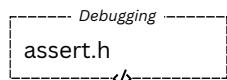
6. Header Files: Key Rules

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

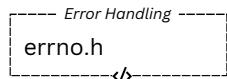
Special Case

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

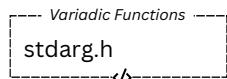
9. Important Standard Header Files (Grouped)



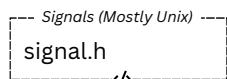
→ runtime checks, aborts on failure



→ global error reporting via errno



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



→ asynchronous signal handling

- **Not portable**

Core Utilities

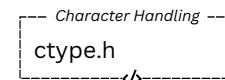
- stddef.h → common types and macros
- stdio.h → input/output (files, streams, devices)

7. Why Header Files Exist

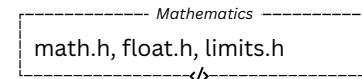
- Avoid rewriting common functionality
- Improve portability
- Separate **interface** from **implementation**
- Enable reuse and standardization

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



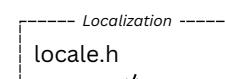
→ character classification & mapping



→ Advanced math, floating-point behavior, type limits



→ non-local jumps (manual exception-like behavior)



→ regional formats (date, time, currency)

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 | 2. Recursive Case |
| <ul style="list-style-type: none"> ◦ Stops recursion ◦ Solved without further recursive calls | <ul style="list-style-type: none"> ◦ 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:

Computer Science UX Design - Lecture 23

- **Purpose of Reviewing the Compilation Process**
- To understand where header files fit in compilation.
- Explains why header files behave differently from normal source code.
- Mathematical compilations
- Critical for understanding preprocessor rules and limitations.

2. High-Level Compilation Pipeline

- 2. 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)

- Main Responsibilities
- Header file inclusion
- Macro expansion
- 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
- Inserts external file contents **textually**
- Forms

4. Preprocessing Stages (In Order)

- Output is still **valid C code**, just expanded
- Operates mainly on lines starting with #
- Conditional compilation
- Line control

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

2. Line Splicing

- Joins physical lines split using \

3. Tokenization

- Breaks code into tokens and whitespace

4. Directives & Macro Processing

- Inserts file contents **textually**

5. Preprocessor Directives

- **#include**

5.1. #include

- **#include** → search standard include paths
- “file.h” → search project directory first
- Treated as if you wrote that code yourself
- Inserts file contents **textually**
- Forms

5.2. #pragma

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

- Demonsrates problem reduction and backtracking.
- $n! = 0$ if $n = 0$ (base case)
- $n! = n \times (n-1)!$ if $n > 0$ (recursive case)

9. Applications of Recursion

- 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
- Time complexity grows exponentially.
- Naive recursive Fibonacci calculates values repeatedly.
- 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

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

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 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

- Auto, register, static & extern
- Four storage classes:

14. Rules of Implicit Type Conversion

- Done automatically by the compiler.
- Happens when:
 - 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

- Block scope variables
- Function parameters
- Global non-static variables and functions
- Visible within one source file
- Static variables at file scope
- A translation unit = source file + included headers
- Determines visibility and linkage across files

3. External linkage

- Visible across translation units
- Global variables and functions
- A translation unit
- Determined by default (garbage values)
- Memory allocated: On block entry
- Rarely written explicitly
- Used for:
 - Register storage class
 - Loop counters
 - Characteristics:
 - No linkage
 - Life-time same as auto
 - No pointers
 - Block scope only
 - No guarantees (compiler decides)

9. Auto Storage Class

- Default for local variables
- Characteristics:
 - Automatic lifetime
 - Block scope
 - Used for storing variable in a CPU register

10. Register Storage Class

- Registers storing variable in a CPU register
- Characteristics:
 - Not guaranteed (compiler decides)
 - Faster access
 - Loop counters
 - Characteristics:
 - No linkage
 - Simple syntax
 - Used together to:
 - hide implementation details
 - make APIs clearer
 - very common in professional C codebases

11. Static Storage Class

- Characteristics:
 - Static storage class
 - Structure arrays for multiple records
 - Structure definition using typeid
 - Constants used for array size safety
 - Loop-based input and output
 - Compound print usage
 - Clearing screen before output for clarity
- Not guaranteed (compiler decides)
- Life-time same as auto
- No pointers
- Block scope only
- No guarantees (compiler decides)

12. Enforcees Information hiding

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

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)

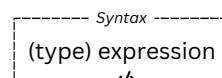
- **Type casting** converts one data type into another.
- Ensures operations are performed correctly.
- Prevents unnecessary memory usage.
- Required when data types are **not naturally compatible**.

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



12. Correct Rounding Using Type Casting

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

- 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**

- 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 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.

- **Valid indices:**
- **Array indices start at 0.**
- **Pointer arithmetic is scaled by data type size**
- **Example:**
- $$\text{new_address} = \text{old_address} + (\text{offset} \times \text{sizeof}(type))$$

- Access elements using: $\boxed{\text{*(p + i)}}$
- Increases occupying contiguous memory.
- Increases memory usage = $\text{size} \times \text{sizeof(data-type)}$

- Arrays decay into pointers when passed to functions

Access elements using:
* (p + i)

2. Operator Precedence With Polarity

- key distinctions:

- Increasing a pointer walks through array elements.
- Never exceed array bounds \leftrightarrow undefined behavior / crash.

8. Using Loops with Arrays					
	Description	Increment Value	then dereference	being pointed to	increment value
	for-loops are the primary tool for array traversal.				

- | Description | Incrementeal pointer,
incrementeal value | being pointed to
by reference | then dereference | incrementeal |
|-------------|---|----------------------------------|------------------|--------------|
|-------------|---|----------------------------------|------------------|--------------|

Description	Increment point value	then dereference	increment pointer to	pointer
Increment point value	increment value	being pointed to	incremented to	then dereference

- Allows:
 - iterate from 0 to size - 1
 - * and + have same precedence
 - * and + have same precedence
 - Parentheses are strongly recommended
 - Associativity resolves ambiguity
- sequential access
- bulk operations
- scalable logic

- **Associativity** resolves ambiguity
 - Parentheses are strongly recommended

- **Associativity** resolves ambiguity
- Parentheses are strongly recommended

- Dynamic size
 - Easy insertion/deletion
- Each node contains:
 - Disadvantages:

- Each node contains:
 - Easy insertion
 - Dynamic size
 - Disadvantages:
 - Each node contains:

- Each node contains:
 - Easy insertion/deletion
 - Disadvantages:

- Random access is allowed (not just sequential).
- Extra memory per node
- Arithmetical and logical operations work the same as with normal variables.
- Arithmetic Pointers (Major Danger)

- #### 4. Dangling Pointers (Major Danger)

o Extra memory per node

10. Common Array Errors

- 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
- Run-time errors instead of compile-time errors
- Most array bugs come from **index misuse**

- High risk of crashes
 - Different yields grabage
 - Best Practices to Remember

1. Memory is freed but pointer still references it

2. Variable goes out of scope

3. Pointers references local variables after function exit

- High risk of crashes
- Derefencing yields garbage

esult:

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

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

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

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

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).
- Only the base address is needed because:
- The compiler already knows the variable's size.
- It compiles the full memory span automatically.
- Contiguous in memory
- Each element has its own address
- The array name refers to the **base address**

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

2. Dynamic memory allocation

3. Efficiency

4. Pass-by-reference

o Passing addresses instead of copying data

5. Returning multiple values

o Modify variables across functions

6. Interfacing with low-level APIs

o Via output parameters

7. Reducing memory footprint

o Especially for arrays and large structures

24. The Core Takeaways

- o separation of computation and display
- o real-world array usage

23. Demo: Multiplication Table

- Uses:
- o 2D array
- o nested loops
- o stores multiplication results
- o displays formatted table
- Demonstrates:

- Enables structured data processing
- o inner loop → columns
- o outer loop → rows

22. Two-Dimensional Array Access

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

21. Higher-Dimensional Arrays

- Extra unused space is safer than overflow
- o bounded versions (strncpy, strncat)
- o correct sizing

20. Multi-Dimensional Arrays

- Prefer:
- o be exploited for attacks
- o crash programs
- o buffer overflows can:
- o fail silently
- o lack bounds checking
- o many string functions:

19. String Safety Concerns

- Many string functions:
- o lack bounds checking
- o fail silently
- o buffer overflows can:
- o crash programs
- o be exploited for attacks

18. Performance Concerns

- Performance concerns:
- o nested loops can:
- o extra unused space is safer than overflow
- o correct sizing
- o bounded versions (strncpy, strncat)
- o extra unused space is safer than overflow