C PROGRAMMING & DATA STRUCTURES - COMPLETE NOTES

TABLE OF CONTENTS

---

1. C PROGRAMMING FUNDAMENTALS

1.1 Introduction to C

What is C? - General-purpose programming language - Developed by Dennis Ritchie at Bell Labs (1972) - Middle-level language (combines features of high-level and low-level languages) - Foundation for many modern languages (C++, Java, C#)

Why Learn C? - Fast execution - Direct memory access - System programming - Embedded systems - Understanding computer architecture

---

1.2 Structure of C Program

```c
// Preprocessor Directives
#include <stdio.h>

// Global Declarations
int globalVar = 10;

// Function Declarations
int add(int a, int b);

// Main Function
int main() {
    // Local Variables
    int result;

    // Function Call
    result = add(5, 3);
```

```c
    // Output
    printf("Result: %d\n", result);

    return 0;
}

// Function Definition
int add(int a, int b) {
    return a + b;
}
```

Components: 1. **Preprocessor Directives**: `#include`, `#define` 2. **Global Declarations**: Variables/functions accessible throughout 3. **main() Function**: Entry point of program 4. **Function Definitions**: Actual code implementation

---

1.3 Compilation Process

4 Stages:

```
Source Code (.c)
    ↓
[1. Preprocessor] → Expanded Source Code (.i)
    ↓
[2. Compiler] → Assembly Code (.s)
    ↓
[3. Assembler] → Object Code (.o)
    ↓
[4. Linker] → Executable (.exe)
```

1. Preprocessor: - Removes comments - Expands macros - Includes header files - Processes directives (#include, #define)

2. Compiler: - Converts to assembly language - Performs syntax checking - Generates intermediate code

3. Assembler: - Converts assembly to machine code - Creates object file (.o or .obj)

4. Linker: - Links multiple object files - Resolves external references - Creates executable

---

1.4 Memory Layout of C Program

```
High Address
```

```
Stack           Local variables, function calls
↓               (Grows downward)


(Free Space)


↑
Heap            Dynamic memory allocation

BSS Segment     Uninitialized global/static variables

Data Segment    Initialized global/static variables

Text/Code       Program instructions (Read-only)

Low Address
```

Segments Explained:

1. Text Segment: - Contains executable code - Read-only (prevents modification) - Sharable among processes

2. Data Segment: - Initialized global and static variables - Example: `int x = 10;` (global)

3. BSS Segment (Block Started by Symbol): - Uninitialized global and static variables - Initialized to 0 by default - Example: `int y;` (global)

4. Heap: - Dynamic memory allocation - `malloc()`, `calloc()`, `realloc()` - Grows upward - Manual management (must free)

5. Stack: - Local variables - Function parameters - Return addresses - Grows downward - Automatic management

---

2. OPERATORS, DATA TYPES & LOOPS

2.1 Data Types in C

Primitive Data Types

Type	Size (bytes)	Format Specifier	Range
`char`	1	%c	-128 to 127
`unsigned char`	1	%c	0 to 255
`short`	2	%hd	-32,768 to 32,767
`unsigned short`	2	%hu	0 to 65,535
`int`	4	%d	-2,147,483,648 to 2,147,483,647

Type	Size (bytes)	Format Specifier	Range
`unsigned int`	4	%u	0 to 4,294,967,295
`long`	4/8	%ld	Platform dependent
`float`	4	%f	±3.4E-38 to ±3.4E+38
`double`	8	%lf	±1.7E-308 to ±1.7E+308

Note: Size may vary based on compiler and architecture.

Type Modifiers

```c
signed int x;       // Can store negative values (default)
unsigned int y;     // Only positive values
short int z;        // Smaller range
long int w;         // Larger range
long long int v;    // Even larger range
```

2.2 Operators in C

1. Arithmetic Operators

```c
int a = 10, b = 3;

printf("%d\n", a + b);  // Addition: 13
printf("%d\n", a - b);  // Subtraction: 7
printf("%d\n", a * b);  // Multiplication: 30
printf("%d\n", a / b);  // Division: 3 (integer division)
printf("%d\n", a % b);  // Modulus: 1 (remainder)
```

Important: - `10 / 3 = 3` (both integers → integer result) - `10.0 / 3 = 3.333...` (float division)

---

2. Relational Operators

```c
int x = 5, y = 10;

x == y  // Equal to: false (0)
x != y  // Not equal: true (1)
x > y   // Greater than: false (0)
x < y   // Less than: true (1)
x >= y  // Greater than or equal: false (0)
x <= y  // Less than or equal: true (1)
```

---

3. Logical Operators

```
int a = 1, b = 0;

a && b  // Logical AND: 0 (both must be true)
a || b  // Logical OR: 1 (at least one true)
!a      // Logical NOT: 0 (inverts value)
```

Truth Table: | A | B | A && B | A || B | !A | |—|—|———|———-|—-| | 0 | 0 | 0 | 0 | 1 | | 0 | 1 | 0 | 1 | 1 | | 1 | 0 | 0 | 1 | 0 | | 1 | 1 | 1 | 1 | 0 |

---

4. Bitwise Operators

```
int a = 5;   // Binary: 0101
int b = 3;   // Binary: 0011

a & b   // AND: 1 (0001)
a | b   // OR: 7 (0111)
a ^ b   // XOR: 6 (0110)
~a      // NOT: -6 (1010 in 2's complement)
a << 1  // Left shift: 10 (1010)
a >> 1  // Right shift: 2 (0010)
```

Bitwise Operations: - **AND (&)**: 1 if both bits are 1 - **OR (|)**: 1 if at least one bit is 1 - **XOR (^)**: 1 if bits are different - **NOT (~)**: Inverts all bits - **Left Shift («)**: Multiply by 2 - **Right Shift (»)**: Divide by 2

---

5. Assignment Operators

```
int x = 10;

x += 5;   // x = x + 5;  → 15
x -= 3;   // x = x - 3;  → 12
x *= 2;   // x = x * 2;  → 24
x /= 4;   // x = x / 4;  → 6
x %= 4;   // x = x % 4;  → 2
```

---

6. Increment/Decrement Operators

```
int a = 5, b;

// Pre-increment
b = ++a;  // a = 6, b = 6 (increment first, then assign)
```

```
// Post-increment
b = a++;  // a = 7, b = 6 (assign first, then increment)

// Pre-decrement
b = --a;  // a = 6, b = 6

// Post-decrement
b = a--;  // a = 5, b = 6
```

Memory Tip: - **Pre (++a)**: Update **before** use - **Post (a++)**: Update **after** use

---

7. Conditional (Ternary) Operator

```
condition ? expression1 : expression2
```

```
// Example
int a = 10, b = 20;
int max = (a > b) ? a : b;  // max = 20

// Equivalent to:
if (a > b)
    max = a;
else
    max = b;
```

2.3 Operator Precedence & Associativity

Precedence (High to Low):

Precedence	Operator	Description	Associativity
1	`() [] -> .`	Parentheses, Arrays, Members	Left to Right
2	`! ~ ++ -- + - * &`	Unary	Right to Left
3	`* / %`	Multiplicative	Left to Right
4	`+ -`	Additive	Left to Right
5	`<< >>`	Shift	Left to Right
6	`< <= > >=`	Relational	Left to Right
7	`== !=`	Equality	Left to Right
8	`&`	Bitwise AND	Left to Right
9	`^`	Bitwise XOR	Left to Right
10	`|`	Bitwise OR	Left to Right

Precedence	Operator	Description	Associativity
11	`&&`	Logical AND	Left to Right
12	`||`	Logical OR	Left to Right
13	`?:`	Conditional	Right to Left
14	`= += -=` etc.	Assignment	Right to Left
15	`,`	Comma	Left to Right

Example:

```c
int result = 2 + 3 * 4;  // 14 (not 20)
// Explanation: * has higher precedence than +
// So: 2 + (3 * 4) = 2 + 12 = 14
```

2.4 Control Structures

2.4.1 Conditional Statements if Statement:

```c
int age = 18;

if (age >= 18) {
    printf("Eligible to vote\n");
}
```

if-else Statement:

```c
int num = -5;

if (num >= 0) {
    printf("Positive\n");
} else {
    printf("Negative\n");
}
```

if-else-if Ladder:

```c
int marks = 75;

if (marks >= 90) {
    printf("Grade: A\n");
} else if (marks >= 80) {
    printf("Grade: B\n");
} else if (marks >= 70) {
    printf("Grade: C\n");
} else {
    printf("Grade: D\n");
}
```

Nested if:

```c
int age = 25;
int hasLicense = 1;

if (age >= 18) {
    if (hasLicense) {
        printf("Can drive\n");
    } else {
        printf("Get license first\n");
    }
}
```

---

switch Statement:

```c
int day = 3;

switch (day) {
    case 1:
        printf("Monday\n");
        break;
    case 2:
        printf("Tuesday\n");
        break;
    case 3:
        printf("Wednesday\n");
        break;
    default:
        printf("Invalid day\n");
}
```

Important: - `break` is crucial (prevents fall-through) - `default` is optional - Can only use integer/char types

---

2.4.2 Loops for Loop:

```c
// Syntax: for(initialization; condition; increment/decrement)

// Print 1 to 10
for (int i = 1; i <= 10; i++) {
    printf("%d ", i);
}
// Output: 1 2 3 4 5 6 7 8 9 10

// Infinite loop
```

```c
for (;;) {
    // Code
}
```

---

while Loop:

```c
// Entry-controlled loop

int i = 1;
while (i <= 5) {
    printf("%d ", i);
    i++;
}
// Output: 1 2 3 4 5
```

Important: Condition checked **before** execution

---

do-while Loop:

```c
// Exit-controlled loop

int i = 1;
do {
    printf("%d ", i);
    i++;
} while (i <= 5);
// Output: 1 2 3 4 5
```

Important: - Executes **at least once** (condition checked after) - Difference from while:

```c
int x = 10;

while (x < 5) {
    printf("while\n");  // Never executes
}

do {
    printf("do-while\n");  // Executes once
} while (x < 5);
```

Nested Loops:

```c
// Print multiplication table
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 3; j++) {
```

```
        printf("%d x %d = %d\n", i, j, i*j);
    }
}
```

2.4.3 Jump Statements break:

```
// Exit from loop immediately
for (int i = 1; i <= 10; i++) {
    if (i == 5) {
        break;  // Loop terminates at i=5
    }
    printf("%d ", i);
}
// Output: 1 2 3 4
```

---

continue:

```
// Skip current iteration
for (int i = 1; i <= 5; i++) {
    if (i == 3) {
        continue;  // Skip when i=3
    }
    printf("%d ", i);
}
// Output: 1 2 4 5
```

---

goto:

```
// Jump to labeled statement (not recommended)
int i = 0;

start:
    printf("%d ", i);
    i++;
    if (i < 5) {
        goto start;
    }
// Output: 0 1 2 3 4
```

Note: `goto` is generally avoided as it makes code hard to understand.

2.5 Practice Problems

Problem 1: Sum of n natural numbers

```c
#include <stdio.h>

int main() {
    int n, sum = 0;

    printf("Enter n: ");
    scanf("%d", &n);

    for (int i = 1; i <= n; i++) {
        sum += i;
    }

    printf("Sum = %d\n", sum);
    // Alternative: sum = n * (n + 1) / 2

    return 0;
}
```

Problem 2: Factorial

```c
#include <stdio.h>

int main() {
    int n, fact = 1;

    printf("Enter n: ");
    scanf("%d", &n);

    for (int i = 1; i <= n; i++) {
        fact *= i;
    }

    printf("%d! = %d\n", n, fact);

    return 0;
}
```

Problem 3: Check Prime Number

```c
#include <stdio.h>

int main() {
    int n, isPrime = 1;
```

```c
    printf("Enter number: ");
    scanf("%d", &n);

    if (n <= 1) {
        isPrime = 0;
    } else {
        for (int i = 2; i * i <= n; i++) {
            if (n % i == 0) {
                isPrime = 0;
                break;
            }
        }
    }

    if (isPrime) {
        printf("%d is prime\n", n);
    } else {
        printf("%d is not prime\n", n);
    }

    return 0;
}
```

---

3. RECURSION, FUNCTIONS & STORAGE

3.1 Functions in C

Definition: Block of code that performs a specific task.

Advantages: - Code reusability - Modularity - Easy debugging - Reduces complexity

---

3.1.1 Function Components

```c
// Function Declaration (Prototype)
int add(int a, int b);

// Function Definition
int add(int a, int b) {
    return a + b;
}

// Function Call
```

```c
int main() {
    int result = add(5, 3);
    return 0;
}
```

Parts: 1. **Return Type**: Data type of value returned 2. **Function Name**: Identifier 3. **Parameters**: Input values 4. **Function Body**: Code to execute 5. **return Statement**: Value to return

---

3.1.2 Types of Functions 1. No Arguments, No Return Value:

```c
void greet() {
    printf("Hello!\n");
}

int main() {
    greet();  // Output: Hello!
    return 0;
}
```

2. No Arguments, With Return Value:

```c
int getNumber() {
    return 42;
}

int main() {
    int num = getNumber();
    printf("%d\n", num);  // Output: 42
    return 0;
}
```

3. With Arguments, No Return Value:

```c
void printSum(int a, int b) {
    printf("Sum = %d\n", a + b);
}

int main() {
    printSum(5, 3);  // Output: Sum = 8
    return 0;
}
```

4. With Arguments, With Return Value:

```c
int multiply(int a, int b) {
    return a * b;
}
```

```c
int main() {
    int result = multiply(4, 5);
    printf("%d\n", result);  // Output: 20
    return 0;
}
```

3.2 Parameter Passing

3.2.1 Call by Value (Pass by Value) **Definition**: Copy of actual parameter is passed.

```c
#include <stdio.h>

void swap(int x, int y) {
    int temp = x;
    x = y;
    y = temp;
    printf("Inside function: x=%d, y=%d\n", x, y);
}

int main() {
    int a = 10, b = 20;

    swap(a, b);
    printf("In main: a=%d, b=%d\n", a, b);

    return 0;
}
```

Output:

```
Inside function: x=20, y=10
In main: a=10, b=20
```

Explanation: Changes inside function don't affect original variables.

3.2.2 Call by Reference (Pass by Pointer) **Definition**: Address of actual parameter is passed.

```c
#include <stdio.h>

void swap(int *x, int *y) {
    int temp = *x;
    *x = *y;
```

```c
    *y = temp;
}

int main() {
    int a = 10, b = 20;

    printf("Before: a=%d, b=%d\n", a, b);
    swap(&a, &b);
    printf("After: a=%d, b=%d\n", a, b);

    return 0;
}
```

Output:

```
Before: a=10, b=20
After: a=20, b=10
```

Explanation: Changes inside function **do** affect original variables.

3.3 Recursion

Definition: Function calling itself.

Components: 1. **Base Case**: Stopping condition 2. **Recursive Case**: Function calls itself

---

3.3.1 How Recursion Works **Example**: Factorial

```c
int factorial(int n) {
    // Base case
    if (n == 0 || n == 1) {
        return 1;
    }

    // Recursive case
    return n * factorial(n - 1);
}
```

Call Stack for factorial(4):

```
factorial(4)
    → 4 * factorial(3)
        → 3 * factorial(2)
            → 2 * factorial(1)
                → 1 (base case)
```

```
            ← returns 1
        ← returns 2 * 1 = 2
    ← returns 3 * 2 = 6
← returns 4 * 6 = 24
```

3.3.2 Types of Recursion 1. Direct Recursion:

```c
void fun() {
    // Some code
    fun();  // Calls itself
}
```

2. Indirect Recursion:

```c
void funA() {
    funB();
}

void funB() {
    funA();
}
```

3. Tail Recursion:

```c
// Recursive call is last statement
void print(int n) {
    if (n == 0) return;
    printf("%d ", n);
    print(n - 1);  // Last statement
}
```

4. Non-tail Recursion:

```c
// Recursive call is not last statement
int factorial(int n) {
    if (n == 1) return 1;
    return n * factorial(n - 1);  // Operation after recursive call
}
```

3.3.3 Recursion vs Iteration

Aspect	Recursion	Iteration
Memory	Uses stack (higher)	Uses fixed memory
Speed	Slower (function calls)	Faster
Code	Shorter, elegant	Longer

Aspect	Recursion	Iteration
Termination	Base case	Loop condition
Use Case	Tree traversal, backtracking	Simple loops

3.3.4 Classic Recursion Examples **Fibonacci Series**:

```c
int fibonacci(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fibonacci(n - 1) + fibonacci(n - 2);
}

// fibonacci(5) → 0, 1, 1, 2, 3, 5
```

Time Complexity: O(2) - Very inefficient!

Optimized with Memoization:

```c
int fib[100] = {0};  // Cache

int fibonacci(int n) {
    if (n <= 1) return n;

    if (fib[n] != 0) return fib[n];  // Return cached value

    fib[n] = fibonacci(n - 1) + fibonacci(n - 2);
    return fib[n];
}
```

Time Complexity: O(n)

Tower of Hanoi:

```c
void towerOfHanoi(int n, char from, char to, char aux) {
    if (n == 1) {
        printf("Move disk 1 from %c to %c\n", from, to);
        return;
    }

    towerOfHanoi(n - 1, from, aux, to);
    printf("Move disk %d from %c to %c\n", n, from, to);
    towerOfHanoi(n - 1, aux, to, from);
}

// Call: towerOfHanoi(3, 'A', 'C', 'B');
```

Time Complexity: O(2)

GCD using Euclid's Algorithm:

```
int gcd(int a, int b) {
    if (b == 0) return a;
    return gcd(b, a % b);
}

// gcd(48, 18) → gcd(18, 12) → gcd(12, 6) → gcd(6, 0) → 6
```

Time Complexity: O(log min(a,b))

3.4 Storage Classes

Definition: Determines scope, lifetime, and visibility of variables.

Four Types: 1. auto 2. extern 3. static 4. register

3.4.1 auto (Automatic) **Characteristics**: - **Default** storage class for local variables - Scope: Within block - Lifetime: Until block executes - Initial value: Garbage

```
void func() {
    auto int x = 10;  // Same as: int x = 10;
    // x exists only within func()
}
```

3.4.2 extern (External) **Characteristics**: - Declares global variable defined elsewhere - Scope: Entire program (multiple files) - Lifetime: Entire program execution - Initial value: 0

File 1 (file1.c):

```
int globalVar = 100;  // Definition

void display() {
    printf("%d\n", globalVar);
}
```

File 2 (file2.c):

```
extern int globalVar;  // Declaration (defined in file1.c)
```

```c
int main() {
    printf("%d\n", globalVar);  // Accesses file1.c variable
    return 0;
}
```

3.4.3 static Characteristics: - Preserves value between function calls - Scope: Within function/file - Lifetime: Entire program execution - Initial value: 0

Static Local Variable:

```c
#include <stdio.h>

void counter() {
    static int count = 0;  // Initialized only once
    count++;
    printf("Count = %d\n", count);
}

int main() {
    counter();  // Output: Count = 1
    counter();  // Output: Count = 2
    counter();  // Output: Count = 3
    return 0;
}
```

Explanation: `count` retains value across function calls.

Static Global Variable:

```c
static int x = 10;  // Accessible only within this file

void func() {
    printf("%d\n", x);
}
```

Use Case: Restrict scope to single file (file-level privacy).

3.4.4 register Characteristics: - Suggests storing in CPU register (for faster access) - Scope: Within block - Lifetime: Until block executes - **Cannot** use `&` (address operator)

```c
void func() {
    register int i;

    for (i = 0; i < 1000000; i++) {
```

```
        // Faster loop counter
    }
}
```

Note: Modern compilers automatically optimize, so `register` is rarely used.

Storage Class Comparison

Storage Class	Scope	Lifetime	Default Value	Memory
auto	Block	Until block ends	Garbage	Stack
extern	Global	Entire program	0	Data segment
static	Block/File	Entire program	0	Data segment
register	Block	Until block ends	Garbage	CPU Register

3.5 Scope Rules

Scope: Region where variable is accessible.

Types of Scope: **1. Block Scope**:

```
{
    int x = 10;  // Visible only in this block
    printf("%d\n", x);
}
// x is not accessible here
```

2. Function Scope:

```
void func() {
    int x = 10;  // Visible throughout function
}
```

3. File Scope (Global):

```
int x = 10;  // Visible in entire file

void func1() {
    printf("%d\n", x);  // Accessible
}

void func2() {
    printf("%d\n", x);  // Accessible
}
```

4. Program Scope (extern):

```c
// Visible across multiple files
extern int x;
```

3.6 Variable Lifetime

Lifetime: Duration for which variable exists in memory.

```c
int global = 10;  // Lifetime: Entire program

void func() {
    int local = 20;        // Lifetime: Until function returns
    static int stat = 30;  // Lifetime: Entire program
}
```

---

4. POINTERS

4.1 Introduction to Pointers

Definition: Variable that stores memory address of another variable.

Why Pointers? - Dynamic memory allocation - Efficient array/string handling - Call by reference - Data structures (linked lists, trees) - Function pointers

---

4.2 Pointer Basics

Declaration and Initialization

```c
int x = 10;
int *ptr;  // Pointer declaration

ptr = &x;  // Pointer initialization (stores address of x)
```

Visualization:

```
Memory:
Address    Variable    Value
1000       x           10
2000       ptr         1000 (address of x)
```

Operators: - `&`: Address-of operator (gets address) - `*`: Dereference operator (gets value at address)

```c
int x = 10;
int *ptr = &x;

printf("Value of x: %d\n", x);        // 10
```

```c
printf("Address of x: %p\n", &x);      // 1000 (example)
printf("Value of ptr: %p\n", ptr);     // 1000
printf("Value at ptr: %d\n", *ptr);    // 10 (dereferencing)
```

4.3 Pointer Arithmetic

Allowed Operations: 1. Increment/Decrement 2. Addition/Subtraction of integer 3. Subtraction of two pointers 4. Comparison

```c
int arr[] = {10, 20, 30, 40};
int *ptr = arr;  // Points to arr[0]

printf("%d\n", *ptr);       // 10
printf("%d\n", *(ptr+1));  // 20 (moves by sizeof(int))
printf("%d\n", *(ptr+2));  // 30
```

Important: - `ptr + 1` moves by `sizeof(datatype)` bytes - For `int*`, `ptr + 1` adds 4 bytes (assuming 4-byte int)

Example:

```c
int arr[] = {10, 20, 30};
int *ptr = arr;

ptr++;  // Now points to arr[1]
printf("%d\n", *ptr);  // 20

ptr += 2;  // Now points to arr[3]
// ptr--;  // Moves back
```

Subtraction of Pointers:

```c
int arr[] = {10, 20, 30, 40, 50};
int *p1 = &arr[1];  // Points to 20
int *p2 = &arr[4];  // Points to 50

int diff = p2 - p1;  // 3 (number of elements between)
```

4.4 Types of Pointers

4.4.1 Null Pointer

```c
int *ptr = NULL;  // Points to nothing

if (ptr == NULL) {
    printf("Pointer is null\n");
}
```

Use: Initialize pointers to avoid garbage values.

4.4.2 Void Pointer (Generic Pointer)

```c
void *ptr;  // Can point to any data type

int x = 10;
float y = 3.14;

ptr = &x;  // Points to int
printf("%d\n", *(int*)ptr);  // Typecast required

ptr = &y;  // Points to float
printf("%.2f\n", *(float*)ptr);
```

Use: Generic data structures, `malloc()` returns `void*`.

4.4.3 Wild Pointer

```c
int *ptr;  // Uninitialized (points to garbage address)
// *ptr = 10;  // Dangerous! May crash
```

Solution: Always initialize:

```c
int *ptr = NULL;
```

4.4.4 Dangling Pointer

```c
int *ptr = (int*)malloc(sizeof(int));
*ptr = 10;

free(ptr);  // Memory freed
// *ptr = 20;  // Dangling pointer (points to freed memory)
```

Solution: Set to NULL after freeing:

```c
free(ptr);
ptr = NULL;
```

4.5 Pointers and Arrays

Key Concept: Array name is a constant pointer to first element.

```c
int arr[] = {10, 20, 30, 40};

// These are equivalent:
printf("%d\n", arr[0]);    // 10
printf("%d\n", *arr);      // 10
printf("%d\n", *(arr+0));  // 10

printf("%d\n", arr[2]);    // 30
printf("%d\n", *(arr+2));  // 30
```

Relationship:

```
arr[i]   *(arr + i)
&arr[i]   (arr + i)
```

Pointer to Array

```c
int arr[] = {10, 20, 30};
int *ptr = arr;  // Points to first element

for (int i = 0; i < 3; i++) {
    printf("%d ", *(ptr + i));  // 10 20 30
}
```

4.6 Pointer to Pointer (Double Pointer)

```c
int x = 10;
int *ptr = &x;      // Pointer to int
int **pptr = &ptr;  // Pointer to pointer

printf("%d\n", x);      // 10
printf("%d\n", *ptr);   // 10
printf("%d\n", **pptr); // 10

// Accessing addresses:
printf("%p\n", &x);     // Address of x
printf("%p\n", ptr);    // Address of x (same as &x)
printf("%p\n", *pptr);  // Address of x (same as ptr)
printf("%p\n", &ptr);   // Address of ptr
printf("%p\n", pptr);   // Address of ptr (same as &ptr)
```

Visualization:

```
x = 10
ptr → x
```

```
pptr → ptr → x
```

4.7 Pointers and Functions

Returning Pointer from Function

```c
int* getMax(int *a, int *b) {
    return (*a > *b) ? a : b;
}

int main() {
    int x = 10, y = 20;
    int *max = getMax(&x, &y);
    printf("Max: %d\n", *max);  // 20
    return 0;
}
```

Warning: Don't return pointer to local variable!

```c
int* func() {
    int x = 10;  // Local variable
    return &x;   // WRONG! x destroyed after function returns
}
```

Solution: Use static or dynamic allocation:

```c
int* func() {
    static int x = 10;  // Static variable
    return &x;  // OK
}
```

4.8 Function Pointers

Definition: Pointer that points to a function.

```c
#include <stdio.h>

int add(int a, int b) {
    return a + b;
}

int main() {
    int (*funcPtr)(int, int);  // Function pointer declaration

    funcPtr = add;  // or funcPtr = &add;
```

```
    int result = funcPtr(5, 3);  // Function call through pointer
    printf("Result: %d\n", result);  // 8

    return 0;
}
```

Use Cases: - Callback functions - Dynamic function calls - Menu-driven programs

---

4.9 Pointer to Constant vs Constant Pointer

1. Pointer to Constant

```
const int x = 10;
const int *ptr = &x;

// *ptr = 20;  // ERROR: Cannot modify value
ptr = &y;      // OK: Can change pointer
```

2. Constant Pointer

```
int x = 10;
int *const ptr = &x;

*ptr = 20;   // OK: Can modify value
// ptr = &y; // ERROR: Cannot change pointer
```

3. Constant Pointer to Constant

```
const int x = 10;
const int *const ptr = &x;

// *ptr = 20;  // ERROR
// ptr = &y;   // ERROR
```

Memory Trick: - Read from right to left - `const int *ptr`: ptr is a pointer to constant int - `int *const ptr`: ptr is a constant pointer to int

---

4.10 Common Pointer Mistakes

1. Uninitialized Pointer:

```
int *ptr;     // Wild pointer
*ptr = 10;    // CRASH!
```

2. Dangling Pointer:

```c
int *ptr = (int*)malloc(sizeof(int));
free(ptr);
*ptr = 10;     // Accessing freed memory
```

3. Memory Leak:

```c
int *ptr = (int*)malloc(sizeof(int));
ptr = NULL;   // Lost reference, memory not freed
```

4. Buffer Overflow:

```c
int arr[5];
int *ptr = arr;
ptr[10] = 100;  // Out of bounds
```

---

5. ARRAYS, STACK, QUEUE & LINKED LISTS

5.1 Arrays

Definition: Collection of similar data types stored in contiguous memory.

5.1.1 Array Declaration and Initialization

```c
// Declaration
int arr[5];  // Uninitialized (garbage values)

// Initialization
int arr1[5] = {1, 2, 3, 4, 5};
int arr2[] = {1, 2, 3};  // Size automatically determined (3)
int arr3[5] = {1, 2};    // Rest initialized to 0: {1, 2, 0, 0, 0}
```

Memory Layout:

```
arr[0] arr[1] arr[2] arr[3] arr[4]
  1      2      3      4      5
[1000][1004][1008][1012][1016]  (Addresses assuming 4-byte int)
```

5.1.2 Accessing Array Elements

```c
int arr[] = {10, 20, 30, 40, 50};

// Using index
printf("%d\n", arr[0]);  // 10
printf("%d\n", arr[2]);  // 30

// Using pointers
```

```
printf("%d\n", *arr);       // 10
printf("%d\n", *(arr+2));   // 30
```

5.1.3 Multi-dimensional Arrays 2D Array:

```
int matrix[3][4] = {
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12}
};

// Accessing elements
printf("%d\n", matrix[0][0]);  // 1
printf("%d\n", matrix[1][2]);  // 7
```

Memory Layout (Row-major order in C):

```
1  2  3  4  5  6  7  8  9  10  11  12
```

Traversal:

```
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 4; j++) {
        printf("%d ", matrix[i][j]);
    }
    printf("\n");
}
```

5.1.4 Array Operations 1. Linear Search:

```
int linearSearch(int arr[], int n, int key) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == key) {
            return i;  // Index found
        }
    }
    return -1;  // Not found
}
```

Time Complexity: O(n)

---

2. Binary Search (Array must be sorted):

```
int binarySearch(int arr[], int n, int key) {
    int left = 0, right = n - 1;
```

```
    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (arr[mid] == key) {
            return mid;
        } else if (arr[mid] < key) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1;
}
```

Time Complexity: O(log n)

---

3. Bubble Sort:

```
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                // Swap
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}
```

Time Complexity: $O(n^2)$

5.2 Stack

Definition: Linear data structure following **LIFO** (Last In First Out).

Real-life Example: Stack of plates

Operations: 1. **Push**: Add element to top 2. **Pop**: Remove element from top 3. **Peek/Top**: View top element 4. **isEmpty**: Check if stack is empty 5. **isFull**: Check if stack is full

---

5.2.1 Stack Implementation (Array)

```c
#include <stdio.h>
#include <stdlib.h>

#define MAX 100

typedef struct {
    int data[MAX];
    int top;
} Stack;

// Initialize stack
void init(Stack *s) {
    s->top = -1;
}

// Check if stack is empty
int isEmpty(Stack *s) {
    return s->top == -1;
}

// Check if stack is full
int isFull(Stack *s) {
    return s->top == MAX - 1;
}

// Push element
void push(Stack *s, int value) {
    if (isFull(s)) {
        printf("Stack Overflow\n");
        return;
    }
    s->data[++s->top] = value;
}

// Pop element
int pop(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack Underflow\n");
        return -1;
    }
    return s->data[s->top--];
}

// Peek top element
```

```c
int peek(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack is empty\n");
        return -1;
    }
    return s->data[s->top];
}

// Main function
int main() {
    Stack s;
    init(&s);

    push(&s, 10);
    push(&s, 20);
    push(&s, 30);

    printf("Top: %d\n", peek(&s));  // 30
    printf("Popped: %d\n", pop(&s));  // 30
    printf("Top: %d\n", peek(&s));  // 20

    return 0;
}
```

Time Complexity: - Push: O(1) - Pop: O(1) - Peek: O(1)

5.2.2 Applications of Stack

1. **Function call management** (recursion)
2. **Expression evaluation** (infix, postfix, prefix)
3. **Parenthesis matching**
4. **Undo mechanism** (text editors)
5. **Browser history** (back button)
6. **Depth-First Search** (DFS)

Example: Balanced Parentheses

```c
#include <stdio.h>
#include <string.h>

#define MAX 100

char stack[MAX];
int top = -1;
```

```c
void push(char c) {
    stack[++top] = c;
}

char pop() {
    return stack[top--];
}

int isBalanced(char *expr) {
    for (int i = 0; i < strlen(expr); i++) {
        char ch = expr[i];

        if (ch == '(' || ch == '{' || ch == '[') {
            push(ch);
        } else if (ch == ')' || ch == '}' || ch == ']') {
            if (top == -1) return 0;

            char popped = pop();
            if ((ch == ')' && popped != '(') ||
                (ch == '}' && popped != '{') ||
                (ch == ']' && popped != '[')) {
                return 0;
            }
        }
    }
    return top == -1;
}

int main() {
    char expr[] = "{[()]}";

    if (isBalanced(expr)) {
        printf("Balanced\n");
    } else {
        printf("Not Balanced\n");
    }

    return 0;
}
```

5.3 Queue

Definition: Linear data structure following **FIFO** (First In First Out).

Real-life Example: Queue at ticket counter

Operations: 1. **Enqueue**: Add element to rear 2. **Dequeue**: Remove element from front 3. **Front**: View front element 4. **Rear**: View rear element 5. **isEmpty**: Check if queue is empty 6. **isFull**: Check if queue is full

---

5.3.1 Queue Implementation (Array)

```c
#include <stdio.h>

#define MAX 100

typedef struct {
    int data[MAX];
    int front, rear;
} Queue;

// Initialize queue
void init(Queue *q) {
    q->front = -1;
    q->rear = -1;
}

// Check if queue is empty
int isEmpty(Queue *q) {
    return q->front == -1;
}

// Check if queue is full
int isFull(Queue *q) {
    return q->rear == MAX - 1;
}

// Enqueue element
void enqueue(Queue *q, int value) {
    if (isFull(q)) {
        printf("Queue Overflow\n");
        return;
    }

    if (q->front == -1) {
        q->front = 0;
    }
    q->data[++q->rear] = value;
}
```

```c
// Dequeue element
int dequeue(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue Underflow\n");
        return -1;
    }

    int value = q->data[q->front];

    if (q->front == q->rear) {
        q->front = q->rear = -1;  // Queue becomes empty
    } else {
        q->front++;
    }

    return value;
}

// Get front element
int front(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty\n");
        return -1;
    }
    return q->data[q->front];
}

int main() {
    Queue q;
    init(&q);

    enqueue(&q, 10);
    enqueue(&q, 20);
    enqueue(&q, 30);

    printf("Front: %d\n", front(&q));  // 10
    printf("Dequeued: %d\n", dequeue(&q));  // 10
    printf("Front: %d\n", front(&q));  // 20

    return 0;
}
```

Time Complexity: - Enqueue: O(1) - Dequeue: O(1) - Front: O(1)

5.3.2 Circular Queue Problem with Linear Queue: After multiple enqueue/dequeue, space at beginning is wasted.

Solution: Circular Queue (uses modulo arithmetic)

```c
#include <stdio.h>

#define MAX 5

typedef struct {
    int data[MAX];
    int front, rear;
} CircularQueue;

void init(CircularQueue *q) {
    q->front = -1;
    q->rear = -1;
}

int isEmpty(CircularQueue *q) {
    return q->front == -1;
}

int isFull(CircularQueue *q) {
    return (q->rear + 1) % MAX == q->front;
}

void enqueue(CircularQueue *q, int value) {
    if (isFull(q)) {
        printf("Queue Full\n");
        return;
    }

    if (q->front == -1) {
        q->front = 0;
    }

    q->rear = (q->rear + 1) % MAX;
    q->data[q->rear] = value;
}

int dequeue(CircularQueue *q) {
    if (isEmpty(q)) {
        printf("Queue Empty\n");
        return -1;
    }
```

```
    int value = q->data[q->front];

    if (q->front == q->rear) {
        q->front = q->rear = -1;
    } else {
        q->front = (q->front + 1) % MAX;
    }

    return value;
}
```

5.3.3 Applications of Queue

1. **CPU scheduling**
2. **Printer spooling**
3. **Breadth-First Search** (BFS)
4. **Buffering** (data streams)
5. **Handling requests** (web servers)

5.4 Linked List

Definition: Linear data structure where elements are stored in nodes, connected by pointers.

Advantages over Arrays: - Dynamic size - Efficient insertion/deletion - No memory wastage

Disadvantages: - No random access (O(n) to access element) - Extra memory for pointers

---

5.4.1 Singly Linked List **Node Structure**:

```
struct Node {
    int data;
    struct Node *next;
};
```

Visualization:

```
[10|•]→[20|•]→[30|NULL]
```

Implementation:

```c
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node *next;
};

// Create new node
struct Node* createNode(int value) {
    struct Node *newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}

// Insert at beginning
void insertAtBeginning(struct Node **head, int value) {
    struct Node *newNode = createNode(value);
    newNode->next = *head;
    *head = newNode;
}

// Insert at end
void insertAtEnd(struct Node **head, int value) {
    struct Node *newNode = createNode(value);

    if (*head == NULL) {
        *head = newNode;
        return;
    }

    struct Node *temp = *head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
}

// Delete node with given value
void deleteNode(struct Node **head, int value) {
    struct Node *temp = *head;
    struct Node *prev = NULL;

    // If head node holds the value
    if (temp != NULL && temp->data == value) {
```

```
        *head = temp->next;
        free(temp);
        return;
    }

    // Search for the node
    while (temp != NULL && temp->data != value) {
        prev = temp;
        temp = temp->next;
    }

    // If value not found
    if (temp == NULL) return;

    prev->next = temp->next;
    free(temp);
}

// Display list
void display(struct Node *head) {
    struct Node *temp = head;
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

// Search for element
int search(struct Node *head, int value) {
    struct Node *temp = head;
    int position = 0;

    while (temp != NULL) {
        if (temp->data == value) {
            return position;
        }
        temp = temp->next;
        position++;
    }
    return -1;  // Not found
}

int main() {
    struct Node *head = NULL;
```

```
    insertAtEnd(&head, 10);
    insertAtEnd(&head, 20);
    insertAtEnd(&head, 30);
    insertAtBeginning(&head, 5);

    display(head);  // 5 -> 10 -> 20 -> 30 -> NULL

    deleteNode(&head, 20);
    display(head);  // 5 -> 10 -> 30 -> NULL

    int pos = search(head, 30);
    printf("30 found at position: %d\n", pos);

    return 0;
}
```

Time Complexity: - Insert at beginning: O(1) - Insert at end: O(n) - Delete: O(n) - Search: O(n)

5.4.2 Doubly Linked List Node Structure:

```
struct Node {
    int data;
    struct Node *prev;
    struct Node *next;
};
```

Visualization:

```
NULL←[•|10|•] [•|20|•] [•|30|•]→NULL
```

Advantages: - Traverse in both directions - Delete node in O(1) if pointer to node is given

Implementation:

```
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node *prev;
    struct Node *next;
};

struct Node* createNode(int value) {
    struct Node *newNode = (struct Node*)malloc(sizeof(struct Node));
```

```
    newNode->data = value;
    newNode->prev = NULL;
    newNode->next = NULL;
    return newNode;
}

void insertAtBeginning(struct Node **head, int value) {
    struct Node *newNode = createNode(value);

    if (*head != NULL) {
        (*head)->prev = newNode;
    }

    newNode->next = *head;
    *head = newNode;
}

void display(struct Node *head) {
    struct Node *temp = head;
    printf("Forward: ");
    while (temp != NULL) {
        printf("%d <-> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

void displayReverse(struct Node *head) {
    if (head == NULL) return;

    struct Node *temp = head;
    while (temp->next != NULL) {
        temp = temp->next;
    }

    printf("Backward: ");
    while (temp != NULL) {
        printf("%d <-> ", temp->data);
        temp = temp->prev;
    }
    printf("NULL\n");
}
```

5.4.3 Circular Linked List **Characteristics**: - Last node points to first node - No NULL pointer

Visualization:

```
[10|•]→[20|•]→[30|•]
  ↑_________________↓
```

Use Cases: - Round-robin scheduling - Circular buffers

5.5 Stack using Linked List

```c
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node *next;
};

struct Node *top = NULL;

void push(int value) {
    struct Node *newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = top;
    top = newNode;
}

int pop() {
    if (top == NULL) {
        printf("Stack Underflow\n");
        return -1;
    }

    struct Node *temp = top;
    int value = temp->data;
    top = top->next;
    free(temp);

    return value;
}

int peek() {
    if (top == NULL) {
        printf("Stack is empty\n");
```

```
        return -1;
    }
    return top->data;
}
```

5.6 Queue using Linked List

```
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node *next;
};

struct Node *front = NULL;
struct Node *rear = NULL;

void enqueue(int value) {
    struct Node *newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;

    if (rear == NULL) {
        front = rear = newNode;
        return;
    }

    rear->next = newNode;
    rear = newNode;
}

int dequeue() {
    if (front == NULL) {
        printf("Queue Underflow\n");
        return -1;
    }

    struct Node *temp = front;
    int value = temp->data;
    front = front->next;

    if (front == NULL) {
        rear = NULL;
```

```
    }

    free(temp);
    return value;
}
```

---

6. TREES & HASHING

6.1 Trees

Definition: Non-linear hierarchical data structure with nodes connected by edges.

Terminology: - **Root**: Top node - **Parent**: Node with children - **Child**: Node descended from parent - **Leaf**: Node with no children - **Height**: Longest path from root to leaf - **Depth**: Length of path from root to node - **Degree**: Number of children

---

6.2 Binary Tree

Definition: Tree where each node has at most 2 children (left and right).

Node Structure:

```
struct Node {
    int data;
    struct Node *left;
    struct Node *right;
};
```

Visualization:

```
      1
     / \
    2   3
   / \
  4   5
```

6.2.1 Binary Tree Types **1. Full Binary Tree**: - Every node has 0 or 2 children

2. Complete Binary Tree: - All levels filled except possibly last - Last level filled from left

3. Perfect Binary Tree: - All internal nodes have 2 children - All leaves at same level

4. Balanced Binary Tree: - Height difference between left and right subtrees 1 for all nodes

---

6.2.2 Binary Tree Traversals 1. Inorder (Left-Root-Right):

```c
void inorder(struct Node *root) {
    if (root == NULL) return;

    inorder(root->left);
    printf("%d ", root->data);
    inorder(root->right);
}
```

Output: 4 2 5 1 3

---

2. Preorder (Root-Left-Right):

```c
void preorder(struct Node *root) {
    if (root == NULL) return;

    printf("%d ", root->data);
    preorder(root->left);
    preorder(root->right);
}
```

Output: 1 2 4 5 3

---

3. Postorder (Left-Right-Root):

```c
void postorder(struct Node *root) {
    if (root == NULL) return;

    postorder(root->left);
    postorder(root->right);
    printf("%d ", root->data);
}
```

Output: 4 5 2 3 1

---

4. Level Order (BFS):

```c
void levelOrder(struct Node *root) {
    if (root == NULL) return;

    // Use queue for level order traversal
    struct Node *queue[100];
    int front = 0, rear = 0;

    queue[rear++] = root;

    while (front < rear) {
        struct Node *current = queue[front++];
        printf("%d ", current->data);

        if (current->left != NULL) {
            queue[rear++] = current->left;
        }
        if (current->right != NULL) {
            queue[rear++] = current->right;
        }
    }
}
```

Output: 1 2 3 4 5

6.3 Binary Search Tree (BST)

Properties: 1. Left subtree < Root 2. Right subtree > Root 3. Both subtrees are BSTs

Example:

```
       50
      /  \
    30    70
   / \    / \
  20 40  60 80
```

6.3.1 BST Operations 1. Insertion:

```c
struct Node* insert(struct Node *root, int value) {
    if (root == NULL) {
        struct Node *newNode = (struct Node*)malloc(sizeof(struct Node));
        newNode->data = value;
        newNode->left = newNode->right = NULL;
        return newNode;
```

```
    }

    if (value < root->data) {
        root->left = insert(root->left, value);
    } else if (value > root->data) {
        root->right = insert(root->right, value);
    }

    return root;
}
```

Time Complexity: O(h) where h = height

---

2. Search:

```
struct Node* search(struct Node *root, int value) {
    if (root == NULL || root->data == value) {
        return root;
    }

    if (value < root->data) {
        return search(root->left, value);
    }

    return search(root->right, value);
}
```

Time Complexity: O(h)

---

3. Deletion:

```
struct Node* minValueNode(struct Node *node) {
    struct Node *current = node;
    while (current && current->left != NULL) {
        current = current->left;
    }
    return current;
}

struct Node* deleteNode(struct Node *root, int value) {
    if (root == NULL) return root;

    if (value < root->data) {
        root->left = deleteNode(root->left, value);
    } else if (value > root->data) {
        root->right = deleteNode(root->right, value);
```

```
    } else {
        // Node with only one child or no child
        if (root->left == NULL) {
            struct Node *temp = root->right;
            free(root);
            return temp;
        } else if (root->right == NULL) {
            struct Node *temp = root->left;
            free(root);
            return temp;
        }

        // Node with two children
        struct Node *temp = minValueNode(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }

    return root;
}
```

6.4 Hashing

Definition: Technique to store and retrieve data in O(1) average time.

Components: 1. **Hash Function**: Maps key to index 2. **Hash Table**: Array to store data

Example:

```
Hash Function: h(key) = key % 10

Keys: 25, 36, 15, 42
Hash Table (size 10):

Index  Value
  0      -
  1      -
  2     42
  3      -
  4      -
  5     25, 15  (collision)
  6     36
  7      -
  8      -
  9      -
```

6.4.1 Collision Resolution 1. Chaining (Open Hashing):

```c
#include <stdio.h>
#include <stdlib.h>

#define SIZE 10

struct Node {
    int data;
    struct Node *next;
};

struct Node *hashTable[SIZE];

void init() {
    for (int i = 0; i < SIZE; i++) {
        hashTable[i] = NULL;
    }
}

int hashFunction(int key) {
    return key % SIZE;
}

void insert(int key) {
    int index = hashFunction(key);

    struct Node *newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = key;
    newNode->next = hashTable[index];
    hashTable[index] = newNode;
}

int search(int key) {
    int index = hashFunction(key);
    struct Node *temp = hashTable[index];

    while (temp != NULL) {
        if (temp->data == key) {
            return 1;  // Found
        }
        temp = temp->next;
    }
```

```
    return 0;  // Not found
}

void display() {
    for (int i = 0; i < SIZE; i++) {
        printf("%d: ", i);
        struct Node *temp = hashTable[i];
        while (temp != NULL) {
            printf("%d -> ", temp->data);
            temp = temp->next;
        }
        printf("NULL\n");
    }
}
```

---

2. Linear Probing (Open Addressing):

```
#define SIZE 10

int hashTable[SIZE];
int occupied[SIZE];  // 0 = empty, 1 = occupied

void init() {
    for (int i = 0; i < SIZE; i++) {
        occupied[i] = 0;
    }
}

int hashFunction(int key) {
    return key % SIZE;
}

void insert(int key) {
    int index = hashFunction(key);

    while (occupied[index] == 1) {
        index = (index + 1) % SIZE;  // Linear probing
    }

    hashTable[index] = key;
    occupied[index] = 1;
}

int search(int key) {
    int index = hashFunction(key);
```

```
    int startIndex = index;

    while (occupied[index] == 1) {
        if (hashTable[index] == key) {
            return 1;  // Found
        }
        index = (index + 1) % SIZE;

        if (index == startIndex) {
            break;  // Full cycle
        }
    }

    return 0;  // Not found
}
```

---

3. Quadratic Probing:

```
void insert(int key) {
    int index = hashFunction(key);
    int i = 0;

    while (occupied[(index + i * i) % SIZE] == 1) {
        i++;
    }

    index = (index + i * i) % SIZE;
    hashTable[index] = key;
    occupied[index] = 1;
}
```

---

4. Double Hashing:

```
int hashFunction2(int key) {
    return 7 - (key % 7);  // Second hash function
}

void insert(int key) {
    int index = hashFunction(key);
    int stepSize = hashFunction2(key);
    int i = 0;

    while (occupied[(index + i * stepSize) % SIZE] == 1) {
        i++;
    }
```

```
    index = (index + i * stepSize) % SIZE;
    hashTable[index] = key;
    occupied[index] = 1;
}
```

6.4.2 Hash Function Properties **Good Hash Function**: 1. **Uniform Distribution**: Spreads keys evenly 2. **Fast Computation**: O(1) time 3. **Minimal Collisions**

Common Hash Functions:

1. Division Method:

```
int hash(int key, int tableSize) {
    return key % tableSize;
}
```

2. Multiplication Method:

```
int hash(int key, int tableSize) {
    double A = 0.618033;  // (√5 - 1) / 2
    return floor(tableSize * fmod(key * A, 1.0));
}
```

3. Mid-Square Method:

```
int hash(int key, int tableSize) {
    int squared = key * key;
    // Extract middle digits
    // Implementation varies
}
```

6.4.3 Load Factor **Definition**:

```
Load Factor ( ) = Number of keys / Table size
```

Impact: - $<$ **0.5**: Good performance - **0.7**: Acceptable - $>$ **0.8**: Poor performance, consider resizing

---

7. GATE QUESTIONS WITH SOLUTIONS

7.1 C Programming - Operators & Data Types

GATE 2024 Question 1 **Question**: What is the output of the following C code?

```c
#include <stdio.h>

int main() {
    int a = 5, b = 10;
    int c = a++ + ++b;
    printf("%d %d %d\n", a, b, c);
    return 0;
}
```

(A) 5 11 16

(B) 6 11 16

(C) 6 11 17

(D) 5 10 15

Solution:

Step-by-step: 1. `a++`: Post-increment → use current value (5), then increment 2. `++b`: Pre-increment → increment first, then use new value (11) 3. `c = 5 + 11 = 16` 4. After expression: `a = 6`, `b = 11`, `c = 16`

Output: 6 11 16

Answer: (B)

GATE 2023 Question 2 **Question**: Consider the following C code:

```c
#include <stdio.h>

int main() {
    char c = 255;
    c = c + 10;
    printf("%d\n", c);
    return 0;
}
```

What will be the output?

(A) 265

(B) 9

(C) -247

(D) Undefined

Solution:

Analysis: - `char` is 1 byte (8 bits) - Range: -128 to 127 (signed) or 0 to 255 (unsigned) - Compiler-dependent, but typically `char` is **signed**

Calculation: 1. `c = 255` → In signed char, this is -1 (two's complement) 2. `c = -1 + 10 = 9`

Answer: (B) 9

Note: If `unsigned char`, answer would be (255 + 10) % 256 = 9 (same result)

GATE 2022 Question 3 **Question**: What is the output?

```c
#include <stdio.h>

int main() {
    int x = 5;
    int y = (x << 2) | (x >> 2);
    printf("%d\n", y);
    return 0;
}
```

(A) 5

(B) 20

(C) 21

(D) 25

Solution:

Binary Analysis:

```
x = 5 (decimal) = 0101 (binary)

x << 2: Left shift by 2
0101 → 10100 = 20 (decimal)

x >> 2: Right shift by 2
0101 → 0001 = 1 (decimal)

y = 20 | 1 (bitwise OR)
  10100
```

```
| 00001
-------
  10101 = 21 (decimal)
```

Answer: (C) 21

7.2 Recursion & Functions

GATE 2024 Question 4 **Question**: What is the output of the following recursive function?

```c
#include <stdio.h>

int mystery(int n) {
    if (n <= 1) return 1;
    return n + mystery(n - 1);
}

int main() {
    printf("%d\n", mystery(5));
    return 0;
}
```

(A) 5

(B) 10

(C) 15

(D) 20

Solution:

Trace:

```
mystery(5) = 5 + mystery(4)
           = 5 + (4 + mystery(3))
           = 5 + 4 + (3 + mystery(2))
           = 5 + 4 + 3 + (2 + mystery(1))
           = 5 + 4 + 3 + 2 + 1
           = 15
```

Formula: Sum of n natural numbers = $n(n+1)/2 = 5 \times 6/2 = 15$

Answer: (C) 15

GATE 2023 Question 5 **Question**: Consider the following function:

```
int func(int n) {
    static int count = 0;
    count++;
    if (n > 1) {
        func(n / 2);
        func(n / 2);
    }
    return count;
}
```

What is the value of `func(8)`?

(A) 7

(B) 11

(C) 15

(D) 31

Solution:

Key: `static int count` - persists across function calls

Call Tree:

```
                func(8)
               /       \
          func(4)   func(4)
          /    \     /    \
     func(2) func(2) func(2) func(2)
     /  \    /  \    /  \     /  \
  func(1) func(1) ... (8 calls of func(1))
```

Counting: - Level 0: 1 call (func(8)) - Level 1: 2 calls (func(4)) - Level 2: 4 calls (func(2)) - Level 3: 8 calls (func(1))

Total = $1 + 2 + 4 + 8 = \mathbf{15}$

Answer: (C) 15

7.3 Pointers

GATE 2024 Question 6 **Question**: What is the output?

```c
#include <stdio.h>

int main() {
    int arr[] = {10, 20, 30, 40};
    int *ptr = arr + 2;
    printf("%d %d\n", *ptr, *(ptr - 1));
    return 0;
}
```

(A) 30 20

(B) 20 10

(C) 30 10

(D) 40 30

Solution:

Analysis:

```
arr[] = {10, 20, 30, 40}
         ↑   ↑   ↑   ↑
       arr arr+1 arr+2 arr+3

ptr = arr + 2 → points to 30

*ptr = 30
*(ptr - 1) = *(arr + 1) = 20
```

Answer: (A) 30 20

GATE 2023 Question 7 **Question**: Consider:

```c
int main() {
    int x = 10;
    int *p = &x;
    int **q = &p;

    **q = 20;

    printf("%d\n", x);
    return 0;
}
```

(A) 10

(B) 20

(C) Address of x

(D) Compilation error

Solution:

Pointer Chain:

```
x = 10
p → x
q → p → x

**q: Dereference q twice
**q = *(*q) = *p = x

**q = 20 means x = 20
```

Answer: (B) 20

7.4 Arrays & Searching

GATE 2024 Question 8 **Question**: Binary search is performed on a sorted array of size n. What is the maximum number of comparisons required?

(A) n

(B) n/2

(C) log (n)

(D) log (n+1)

Solution:

Binary Search Analysis: - Each comparison halves search space - Continues until 1 element remains

Recurrence: T(n) = T(n/2) + 1

Example: n = 16

```
16 → 8 → 4 → 2 → 1
```

Comparisons = 4 = log (16)

For n = 15:

`15 → 7 → 3 → 1`

Comparisons = 4 = log (16) = log (15+1)

Answer: (D) log (n+1)

7.5 Stack & Queue

GATE 2024 Question 9 Question: A queue is implemented using two stacks S1 and S2. Enqueue operation uses S1 and dequeue uses S2. What is the worst-case time complexity of dequeue operation?

(A) O(1)

(B) O(log n)

(C) O(n)

(D) $O(n^2)$

Solution:

Implementation:

Enqueue: Push to S1 → O(1)

Dequeue: - If S2 empty: Pop all from S1 and push to S2 → O(n) - Pop from S2 → O(1)

Worst Case: When S2 is empty and S1 has n elements

Answer: (C) O(n)

Note: **Amortized** complexity is O(1)

GATE 2023 Question 10 Question: Postfix expression: `5 3 + 8 2 - *`

What is the result?

(A) 16

(B) 24

(C) 32

(D) 48

Solution:

Evaluation using Stack:

```
Symbol  Stack       Operation
5       [5]
3       [5, 3]
+       [8]         5 + 3 = 8
8       [8, 8]
2       [8, 8, 2]
-       [8, 6]      8 - 2 = 6
*       [48]        8 * 6 = 48
```

Answer: (D) 48

7.6 Linked Lists

GATE 2024 Question 11 **Question**: What is the time complexity of inserting a node at the end of a singly linked list of length n?

(A) O(1)

(B) O(log n)

(C) O(n)

(D) O(n log n)

Solution:

Analysis: - Need to traverse entire list to reach end - Traverse n nodes - Insert $\rightarrow$ O(1) - Total $\rightarrow$ O(n)

Optimization: Maintain tail pointer $\rightarrow$ O(1)

Answer: (C) O(n) [without tail pointer]

GATE 2023 Question 12 **Question**: A circular linked list is used to represent a queue. A single variable `p` is used to access the queue. To which node should `p` point such that both enqueue and dequeue operations can be performed in constant time?

(A) Front node

(B) Rear node

(C) Node next to front

(D) Any node

Solution:

Analysis:

If `p` points to **rear**: - `p->next` = front (O(1) access) - `p` = rear (O(1) access)

Enqueue: Insert after `p`, update `p` → O(1)
Dequeue: Delete `p->next` → O(1)

If `p` points to **front**: - Front access: O(1) - Rear access: O(n) (need to traverse)

Answer: (B) Rear node

7.7 Trees

GATE 2024 Question 13 Question: A binary tree has 20 nodes. The inorder and preorder traversals are given:

Inorder: D B E A F C
Preorder: A B D E C F

Which node is the root?

(A) A

(B) B

(C) C

(D) D

Solution:

Property: First node in **Preorder** is always **root**

Preorder: A B D E C F
↑
root

Answer: (A) A

GATE 2023 Question 14 **Question**: What is the maximum number of nodes in a binary tree of height h?

(A) 2^h

(B) 2^h - 1

(C) 2^(h+1) - 1

(D) 2^(h-1)

Solution:

Analysis: - Height h → (h+1) levels - Level i has maximum 2^i nodes

Sum:

```
Level 0: 2^0 = 1
Level 1: 2^1 = 2
Level 2: 2^2 = 4
...
Level h: 2^h

Total = 2^0 + 2^1 + ... + 2^h
      = 2^(h+1) - 1  (GP sum)
```

Example: h = 2

```
       O        Level 0: 1 node
      / \
     O   O      Level 1: 2 nodes
    / \ / \
   O  O O  O    Level 2: 4 nodes
```

Total = 1 + 2 + 4 = 7 = 2^3 - 1 = 2^(2+1) - 1

Answer: (C) 2^(h+1) - 1

GATE 2022 Question 15 **Question**: In a Binary Search Tree, the inorder traversal gives:

Inorder: 5 10 15 20 25 30 35

Which of the following could be the preorder traversal?

(A) 20 10 5 15 30 25 35

(B) 20 10 15 5 30 25 35

(C) 20 15 10 5 25 30 35

(D) 20 5 10 15 25 30 35

Solution:

Property: Inorder of BST is **sorted**

Analysis: - Root must split inorder into left and right subtrees - Root = 20 (middle element for balanced tree)

Preorder: Root → Left → Right

```
Inorder:  5 10 15 | 20 | 25 30 35
          ↑ left     ↑    ↑ right
                   root
```

```
Preorder must start with: 20
Then left subtree: 5, 10, 15
Then right subtree: 25, 30, 35
```

Check options: - (A) 20 10 5 15 … (10 before 5,15)

Answer: (A) 20 10 5 15 30 25 35

7.8 Hashing

GATE 2024 Question 16 Question: A hash table of size 13 uses open addressing with linear probing. Keys are inserted in order: 18, 41, 22, 44, 59, 32, 31, 73.

Hash function: h(k) = k mod 13

Which slot does 73 occupy?

(A) 8

(B) 9

(C) 10

(D) 11

Solution:

Insertions:

```
18 % 13 = 5 → Slot 5
41 % 13 = 2 → Slot 2
```

```
22 % 13 = 9 → Slot 9
44 % 13 = 5 → Collision! → Slot 6 (next empty)
59 % 13 = 7 → Slot 7
32 % 13 = 6 → Collision! → Slot 8 (next empty)
31 % 13 = 5 → Collision! → Probe: 5,6,7,8,9 → Slot 10
73 % 13 = 8 → Collision! → Probe: 8,9,10,11 → Slot 11
```

Table:

```
Slot:  0  1  2  3  4  5  6  7  8  9  10 11 12
       -  - 41  -  - 18 44 59 32 22 31 73  -
```

Answer: (D) 11

GATE 2023 Question 17 **Question**: In a hash table with chaining, n keys are inserted into a table of size m. What is the expected length of the longest chain?

(A) $\Theta(1)$

(B) $\Theta(\log n)$

(C) $\Theta(n/m)$

(D) $\Theta(\log(n/m))$

Solution:

Load Factor: $= n/m$

Expected chain length $=$ $= n/m$

With uniform distribution, average case is $\Theta(n/m)$

Worst case (all keys in one chain): $\Theta(n)$

Answer: (C) $\Theta(n/m)$

7.9 Storage Classes

GATE 2022 Question 18 **Question**: What is the output?

```c
#include <stdio.h>

void func() {
    static int x = 0;
```

```c
    x++;
    printf("%d ", x);
}

int main() {
    func();
    func();
    func();
    return 0;
}
```

(A) 0 0 0

(B) 1 1 1

(C) 1 2 3

(D) 0 1 2

Solution:

Static Variable: Initialized once, retains value

Execution:

```
func() call 1: x = 0 → x++ → print 1
func() call 2: x = 1 → x++ → print 2
func() call 3: x = 2 → x++ → print 3
```

Answer: (C) 1 2 3

7.10 Complexity Questions

GATE 2024 Question 19 **Question**: What is the time complexity of the following code?

```c
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= i; j++) {
        printf("*");
    }
}
```

(A) O(n)

(B) O(n log n)

(C) $O(n^2)$

(D) O(2^n)

Solution:

Analysis:

```
i = 1: inner loop runs 1 time
i = 2: inner loop runs 2 times
i = 3: inner loop runs 3 times
...
i = n: inner loop runs n times

Total = 1 + 2 + 3 + ... + n
      = n(n+1)/2
      = O(n²)
```

Answer: (C) $O(n^2)$

GATE 2023 Question 20 **Question**: Recurrence: $T(n) = 2T(n/2) + n$

What is the time complexity?

(A) O(n)

(B) O(n log n)

(C) $O(n^2)$

(D) O(log n)

Solution:

Using Master Theorem:

$T(n) = aT(n/b) + f(n)$

Here: $a = 2$, $b = 2$, $f(n) = n$

Compare n^(log_b a) with f(n): - n^(log 2) = n^1 = n - f(n) = n

Case 2: $f(n) = \Theta(n^{\log_b a})$

Result: $T(n) = \Theta(n \log n)$

Answer: (B) O(n log n)

SUMMARY & QUICK REVISION

Important Formulas

1. Array/Pointer:

```
arr[i] = *(arr + i)
&arr[i] = arr + i
```

2. Binary Tree:

```
Max nodes at level i = 2^i
Max nodes with height h = 2^(h+1) - 1
Min height for n nodes =  log (n+1)
```

3. Time Complexities:

```
Linear Search: O(n)
Binary Search: O(log n)
Bubble Sort: O(n²)
Merge Sort: O(n log n)
Quick Sort: O(n log n) average, O(n²) worst
```

4. Hashing:

```
Load Factor   = n/m
Expected chain length =
```

PRACTICE STRATEGY

Week 1-2: C Basics - Operators, data types, loops - Practice 50+ programs - Focus on pointer arithmetic

Week 3: Functions & Recursion - Write 20+ recursive programs - Understand call stack - Practice tree problems

Week 4: Data Structures - Implement all from scratch - Arrays, stack, queue, linked list - Practice 30+ problems

Week 5: Trees & Hashing - Tree traversals - BST operations - Hashing collision resolution

Week 6: GATE Previous Years - Solve 100+ GATE questions - Time yourself - Identify weak areas

---

END OF NOTES

All the best for GATE preparation!