Theory Notes

MODULE I — Fundamentals of C

Program Structure & OS Communication

How C talks to the OS

The **Operating System (OS)** sits between your software and hardware.

```
Your C Program
      ↓  (via system calls)
Operating System
      ↓  (via drivers)
Hardware (CPU, RAM, Disk)
```

- OS talks to **hardware** through → **Drivers**
- OS talks to **software** through → **System Calls**

C Library Functions

To use built-in functions, you `#include` the right header file.

Header File	What It Does	Key Functions
`<stdio.h>`	Standard Input/Output	`printf()`, `scanf()`, `fopen()`
`<stdlib.h>`	Standard Library	`malloc()`, `free()`, `exit()`
`<string.h>`	String handling	`strlen()`, `strcpy()`, `strcmp()`
`<math.h>`	Math operations	`sqrt()`, `pow()`, `abs()`
`<errno.h>`	Error handling	`errno` variable
`<setjmp.h>`	Jump functions	`setjmp()`, `longjmp()`

Preprocessor Directives

> **Preprocessor** runs BEFORE the compiler. All directives start with `#`.

Directive	Purpose	Example
`#include`	Include a file	`#include <stdio.h>`
`#define`	Define a constant/macro	`#define PI 3.14`
`#ifdef` / `#ifndef`	Conditional compilation	`#ifdef DEBUG`
`#undef`	Undefine a macro	`#undef PI`
`#pragma`	Compiler-specific instructions	`#pragma once`

```c
#include<stdio.h>
#define MAX100// MAX will be replaced by 100 everywhere
```

```c
int main() {
    printf("Max value:%d", MAX);  // prints: Max value: 100
    return 0;
}
```

Debugging and Efficiency

Debugging = Finding and fixing bugs in your code.

Steps:
1. **Reproduce** the error
2. **Identify** the cause (logical, syntax, or runtime error)
3. **Correct** the mistake

Efficiency = Analyse how fast/slow your code is using **Asymptotic Notations**:
- **O(1)** → Constant time (best)
- **O(n)** → Linear time
- **O(n²)** → Quadratic time (slowest, avoid)

Memory Models in C

```
+------------------+
|   Stack          | ← Local variables, function calls (auto-managed)
+------------------+
|   Heap           | ← Dynamic memory (malloc, free)
+------------------+
|   Data Segment   | ← Global and static variables
+------------------+
|   Text Segment   | ← Your compiled code (read-only)
+------------------+
```

> **Key point:** Stack = automatic. Heap = manual (you must free it!).

Data Types in C

Every variable must have a **data type** that tells C how much memory to use.

Simple Data Types Table

Data Type	Size	Range	Example
`int`	4 bytes	-2,147,483,648 to 2,147,483,647	`int age = 20;`
`short int`	2 bytes	-32,768 to 32,767	`short s = 100;`
`long int`	8 bytes	Very large	`long l = 100000L;`
`unsigned int`	4 bytes	0 to 4,294,967,295	`unsigned int x = 5;`
`float`	4 bytes	6 decimal places	`float f = 3.14f;`
`double`	8 bytes	15 decimal places	`double d = 3.14159;`
`char`	1 byte	-128 to 127	`char c = 'A';`

Integer Representation (How negatives work)

```
Example with 8 bits:
+75  → 01001011
-75  → 10110100  (one's complement)
-75  → 10110101  (two's complement ← C uses this!)
```

ASCII Character Set

Every character has a number (ASCII code) stored in 7 bits.

Character	ASCII Value
`'A'`	65
`'a'`	97
`'0'`	48
`' '` (space)	32
`'\n'` (newline)	10

```c
char c = 'A';
printf("%d", c);  // Output: 65
```

Compilation and Linking

```
Source Code (.c)
      ↓  [Preprocessor] → expands #include, #define
Modified Source
      ↓  [Compiler]     → converts to machine code
Object Code (.o)
      ↓  [Linker]       → links with libraries
Executable (.exe / a.out)
```

Types of Compilers

Type	What It Does
Cross Compiler	Compiles for a DIFFERENT CPU/OS than the one it runs on
Bootstrap Compiler	Written in the language it compiles
Decompiler	Converts low-level → high-level code
Transcompiler	Converts one high-level language to another

Decision Making in C

```c
// if-else
if (marks >= 60) {
    printf("Pass");
} else {
```

```
    printf("Fail");
}

// Conditional (Ternary) Operator → shortcut for if-else
int result = (marks >= 60) ? 1 : 0;
// Syntax: condition ? value_if_true : value_if_false
```

Loops in C

```
// for loop
for (int i = 0; i < 5; i++) {
    printf("%d ", i);  // Output: 0 1 2 3 4
}

// while loop
int i = 0;
while (i < 5) {
    printf("%d ", i);
    i++;
}

// do-while loop (runs at least once)
do {
    printf("%d ", i);
    i++;
} while (i < 5);
```

Jump Statements

break — exits the loop immediately

```
for (int i = 0; i < 10; i++) {
    if (i == 5) break;
    printf("%d ", i);  // Output: 0 1 2 3 4
}
```

continue — skips current iteration

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

goto — jumps to a label (use rarely!)

```c
int i = 1;
start:
    printf("%d ", i);
    i++;
    if (i <= 10) goto start;
// Output: 1 2 3 4 5 6 7 8 9 10
```

return — exits the function

```c
int add(int a, int b) {
    return a + b;  // exits function and sends value back
}
```

Null Statement, Comma Operator

Null Statement — does nothing

```c
for (int i = 0; i < 5; i++)
    ;   // This semicolon IS the body (null statement)
```

Comma Operator — evaluates left first, returns right

```c
int x = (5 + 10, 6 + 9);
printf("%d", x);  // Output: 15  (returns the last value)
```

---

setjmp and longjmp

These allow **non-local jumps** (like exception handling in C).

```c
#include<setjmp.h>
#include<stdio.h>
jmp_buf buf;
void checkerror(){
    printf("Checking for Errors....\n");
    longjmp(buf,1);
}
int main(){
    if(setjmp(buf)){
        printf("Error Recovered!\n");
    }
    else{
        checkerror();
    }
```

```
    return 0;
}
```

> `setjmp(buf)` → saves current state, returns 0
> `longjmp(buf, i)` → goes back to setjmp, now it returns i

Storage Classes

Storage classes define **scope**, **lifetime**, and **visibility** of variables.

Storage Class	Where Declared	Lifetime	Default Value	Special Feature
`auto`	Inside function	Function duration	Garbage	Default for local vars
`extern`	Outside all functions	Entire program	0	Share across files
`static`	Anywhere	Entire program	0	Value preserved between calls
`register`	Inside function	Function duration	Garbage	Stored in CPU register (fast!)

```c
// Syntax: storage_class data_type variable_name;
auto int a = 32;
register char b = 'G';
extern int x;
static int count = 0;

// Static example — value is remembered!
void counter() {
    static int c = 0;
    c++;
    printf("%d\n", c);
}
// Call 3 times → prints 1, 2, 3 (not 1, 1, 1)
```

> You CANNOT use `&` (address-of) with `register` variables.

Arrays in C

An array = **fixed-size collection** of same-type elements in **contiguous memory**.

```c
// Declaration
int arr[5];              // 5 integers

// Declaration + Initialization
int arr[5] = {10, 20, 30, 40, 50};

// Without size (compiler counts)
int arr[] = {10, 20, 30};  // size = 3
```

```c
// Initialization using loop
for (int i = 0; i < 5; i++) {
    arr[i] = i * 10;
}

// Access (indexing starts at 0!)
printf("%d", arr[0]);  // 10 (first element)
printf("%d", arr[4]);  // 50 (last element, index = size-1)
```

Array of Pointers & Character Arrays

Array of Pointers

```c
int a = 10, b = 20, c = 30;
int *ptr[3] = {&a, &b, &c};  // array of 3 integer pointers

printf("%d", *ptr[0]);  // Output: 10
```

Array of Pointers to Characters (storing multiple strings)

```c
char *names[] = {"Alice", "Bob", "Charlie"};
printf("%s", names[0]);  // Output: Alice
printf("%s", names[1]);  // Output: Bob
```

> The array name itself is a pointer to its first element!
> `arr` is same as `&arr[0]`

MODULE II Advanced Data Types & Pointers

#define in C

```c
// Defining a constant
#define PI3.14159

// Defining an expression
#define SQUARE(x)((x)*(x))

int main() {
    printf("Area =%.2f\n", PI * SQUARE(5));  // Area = 78.54
}
```

> `#define` has NO semicolon at the end. It's text replacement, not a variable.

Variable Length Arrays (VLA)

Arrays whose **size is decided at runtime** (not at compile time).

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

int arr[n];  // VLA — size decided when program runs!
for (int i = 0; i < n; i++) arr[i] = i;
```

> Available from C99 standard onwards.

Flexible Array Members

A structure can have an array of **unknown size at the end**:

```c
struct Student {
    int id;
    int marks[];  // flexible array — no size given!
};
// Size of structure = size of 'id' only (4 bytes)
// The flexible member doesn't count in sizeof()
```

Rules:
- Must be the **last member** of the structure
- Structure must have at least one other named member
- Used with dynamic memory allocation

Type Qualifiers

const — value cannot change

```c
const int MAX = 100;  // cannot do MAX = 200 later
// MAX = 200;  ← ERROR!
```

Types of const with pointers:

```c
int x = 10;

// 1. Pointer to constant (can't change value via pointer)
const int *ptr = &x;

// 2. Constant pointer (can't change where pointer points)
int *const ptr = &x;
```

```c
// 3. Constant pointer to constant (both fixed)
const int *const ptr = &x;
```

volatile — value can change unexpectedly (by hardware/OS)

```c
volatile int sensor_value;  // tells compiler: don't optimize this
```

restrict — pointer is the ONLY way to access that memory

```c
void func(int *restrict a, int *restrict b) {
    // compiler can optimize knowing a and b don't overlap
}
```

Functions in C

```c
// 1. Declaration (prototype) — tells compiler the function exists
int add(int a, int b);

// 2. Definition — actual code
int add(int a, int b) {
    return a + b;
}

// 3. Function Call
int main() {
    int result = add(5, 3);  // result = 8
}
```

Parameter Passing Techniques

Pass by Value (Call by Value)

The function gets a **COPY** — original variable is NOT changed.

```c
void square(int x) {
    x = x * x;  // changes only the local copy
}

int main() {
    int n = 5;
    square(n);
    printf("%d", n);  // still 5! original unchanged
}
```

Pass by Reference (Call by Reference)

The function gets the **ADDRESS** — original variable IS changed.

```
void square(int *x) {
    *x = (*x) * (*x);  // changes actual value
}

int main() {
    int n = 5;
    square(&n);  // pass address
    printf("%d", n);  // now 25!
}
```

> 🔑 **Key difference:** Value = copy (safe). Reference = actual address (powerful but risky).

7️⃣ Command Line Arguments

```
int main(int argc, char *argv[]) {
    // argc = number of arguments (including program name)
    // argv = array of argument strings
    printf("Program name:%s\n", argv[0]);
    printf("First argument:%s\n", argv[1]);
}
```

Example: Running `./program hello 42`

- `argc = 3`
- `argv[0] = "./program"`
- `argv[1] = "hello"`
- `argv[2] = "42"`

8️⃣ Structures in C

A structure groups **different data types** under one name.

```
// Define the structure
struct Student {
    int id;
    char name[50];
    float marks;
};

// Create a variable
struct Student s1;
s1.id = 101;
strcpy(s1.name, "Alice");
s1.marks = 95.5;
```

```c
// Or initialize directly
struct Student s2 = {102, "Bob", 88.0};

// Access members with dot operator
printf("%s scored%.1f\n", s2.name, s2.marks);
```

typedef with Structures (shorter syntax)

```c
typedef struct {
    int id;
    char name[50];
    float marks;
} Student;  // now use 'Student' instead of 'struct Student'

Student s1 = {101, "Alice", 95.5};
```

Nested Structures

```c
struct Date {
    int day, month, year;
};

struct Employee {
    int id;
    struct Date dob;  // nested!
};

struct Employee e;
e.dob.day = 15;  // access nested member
```

Bit Fields in Structures

```c
struct Flags {
    unsigned int isAdmin : 1;   // only 1 bit
    unsigned int isActive : 1;  // only 1 bit
    unsigned int level : 3;     // 3 bits (values 0-7)
};
```

offsetof() macro

```c
#include<stddef.h>
printf("%zu", offsetof(struct Student, marks));
// Returns how many bytes from start of struct to 'marks'
```

9️⃣ Unions in C

Like a structure, but **all members share the SAME memory location**.

```c
union Data {
    int i;
    float f;
    char ch;
};

union Data d;
d.i = 65;
printf("%d\n", d.i);  // 65
printf("%c\n", d.ch); // 'A' (same memory!)

// Size of union = size of LARGEST member
printf("%zu", sizeof(d));  // = sizeof(float) = 4 bytes
```

> ⚠️ Only ONE member can hold a value at a time!

🔟 typedef in C

Give **new names** to existing data types.

```c
typedef unsigned long long int ull;
ull bigNumber = 9999999999ULL;

typedef int* IntPtr;
IntPtr p, q;  // both are int pointers!

typedef int Array5[5];
Array5 arr;  // int arr[5]
```

1️⃣1️⃣ Type Casting in C

Converting one data type to another.

Implicit (Automatic) Casting

```c
int i = 10;
float f = i;  // int automatically becomes float (10.0)
// Hierarchy: char < int < long < float < double
```

Explicit Casting

```c
float result = (float) 7 / 2;  // = 3.5 (without cast → 3)
int x = (int) 3.99;            // = 3 (truncates decimal)
```

1️⃣2️⃣ Pointers in C

A pointer **stores the memory address** of another variable.

```c
int x = 10;
int *ptr;       // declare pointer
ptr = &x;       // store address of x in ptr

printf("%d", x);    // 10 (value)
printf("%p", ptr);  // address (like 0x7ffd...)
printf("%d", *ptr); // 10 (dereference = get value at address)

*ptr = 20;  // change x through pointer
printf("%d", x);  // now 20!
```

> 🔑 `&` = "address of" | `*` = "value at" (dereference)

NULL Pointer

```c
int *ptr = NULL;  // points to nothing (safe initialization)
if (ptr == NULL) printf("Pointer not set!");
```

1️⃣3️⃣ Types of Pointers

1. Integer Pointer

```c
int x = 5;
int *ptr = &x;
```

2. Array Pointer

```c
int arr[5] = {1,2,3,4,5};
int (*ptr)[5] = &arr;  // pointer to entire array
// arr itself is pointer to arr[0]
```

3. Function Pointer

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

```c
int (*funcPtr)(int, int) = &add;  // pointer to function
int result = funcPtr(3, 4);       // call via pointer → 7
```

4. Structure Pointer

```c
struct Student s = {101, "Alice"};
struct Student *ptr = &s;

// Access members via pointer:
printf("%s", (*ptr).name);  // method 1
printf("%s", ptr->name);    // method 2 (arrow operator, cleaner!)
```

5. Array of Pointers

```c
int a=1, b=2, c=3;
int *arr[3] = {&a, &b, &c};
printf("%d", *arr[1]);  // 2
```

6. Passing Pointers to Functions

```c
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int x=10, y=20;
swap(&x, &y);
// x=20, y=10 now!
```

1️⃣4️⃣ Macros in C

Types of Macros

1. Object-Like Macros (constants)

```c
#define MAX_SIZE100
#define PI3.14159
```

2. Function-Like Macros

```c
#define ADD(a, b)((a)+(b))
#define SQUARE(x)((x)*(x))

printf("%d", SQUARE(5));  // Output: 25
```

3. Multi-Line Macros

```
#define PRINT_INFO(name, age)\
    printf("Name:%s\n", name);\
    printf("Age:%d\n", age)
```

4. Chain Macros (macros inside macros)

```
#define BASE10
#define DOUBLE_BASE(2* BASE)// uses BASE macro
```

> ⚠️ Always put parentheses around macro arguments to avoid bugs!
> `#define SQ(x) x*x` ← WRONG for `SQ(1+2)` = 1+2*1+2 = 5* ❌
> `#define SQ(x) ((x)*(x))` ← CORRECT → (1+2)*(1+2) = 9 ✅

🟡 MODULE III — Files, Recursion, Memory & Threads (10 hrs)

1️⃣ File Handling in C

Why use files?

- **Reusability** — data persists after program ends
- **Portability** — transfer data between computers
- **Efficiency** — handle large amounts of data
- **Storage** — don't need to store everything in RAM

Types of Files

Type	Description	Extension	Readable by
Text	ASCII characters, human-readable	`.txt`	Any text editor
Binary	Raw 0s and 1s	`.bin`	Only programs

File Operations with Functions

```
FILE *fptr;  // file pointer — used in ALL file operations
```

Opening a File

```
// Syntax: fopen("filename", "mode")
FILE *fptr = fopen("data.txt", "r");   // open for reading
FILE *fptr = fopen("data.txt", "w");   // open for writing (creates new/overwrites)
FILE *fptr = fopen("data.txt", "a");   // open for appending
```

```c
FILE *fptr = fopen("data.txt", "r+");  // read AND write

if (fptr == NULL) {
    printf("Error opening file!");
    return 1;
}
```

File Mode Table

Mode	Meaning
`"r"`	Read (file must exist)
`"w"`	Write (creates new or overwrites)
`"a"`	Append (adds to end)
`"r+"`	Read + Write
`"w+"`	Write + Read (overwrites)
`"rb"`	Read binary
`"wb"`	Write binary

Reading from a File

```c
char str[100];
fscanf(fptr, "%s", str);     // read word
fgets(str, 100, fptr);       // read line
char c = fgetc(fptr);        // read one character
```

Writing to a File

```c
fprintf(fptr, "Hello%s, age%d\n", "Alice", 20);
fputs("Hello World\n", fptr);
fputc('A', fptr);
```

Closing a File (ALWAYS do this!)

```c
fclose(fptr);
```

Binary File Read/Write

```c
// Write binary
struct Student s = {101, "Alice"};
fwrite(&s, sizeof(struct Student), 1, fptr);

// Read binary
fread(&s, sizeof(struct Student), 1, fptr);
```

2️⃣ Recursive Functions

A function that **calls itself** until a base condition is met.

```
// Basic structure
int recursiveFunc(int n) {
    if (n == 0) return 0;        // BASE CONDITION (stops recursion)
    return n + recursiveFunc(n-1); // RECURSIVE CASE (calls itself)
}

// Example: Sum of 1 to n
int nSum(int n) {
    if (n == 0) return 0;
    return n + nSum(n - 1);
}
// nSum(5) = 5 + 4 + 3 + 2 + 1 + 0 = 15
```

How Memory Works in Recursion

```
nSum(5) called
  → nSum(4) called
    → nSum(3) called
      → nSum(2) called
        → nSum(1) called
          → nSum(0) called → returns 0
        ← nSum(1) returns 1
      ← nSum(2) returns 3
    ← nSum(3) returns 6
  ← nSum(4) returns 10
← nSum(5) returns 15
```

Each call creates a **stack frame** in memory. Stack frames are destroyed when the function returns.

Types of Recursion

Type	Description	Example
Head Recursion	Recursive call at the START	Call first, process after
Tail Recursion	Recursive call at the END	Process first, call last
Tree Recursion	Multiple recursive calls	Fibonacci, Tree traversal
Indirect Recursion	A() calls B(), B() calls A()	Mutually recursive

---

3️⃣ Dynamic Memory Allocation

Allocate memory during **runtime** (not at compile time).

Function	Purpose	Initializes?
`malloc()`	Allocate memory	No (garbage values)
`calloc()`	Allocate + clear memory	Yes (zeros)
`realloc()`	Resize existing allocation	Keeps old values

Function	Purpose	Initializes?
`free()`	Release memory	—

```c
#include<stdlib.h>

// malloc — allocate memory
int *arr = (int*) malloc(5 * sizeof(int));
if (arr == NULL) { printf("Memory failed!"); }

// calloc — allocate and zero-initialize
int *arr = (int*) calloc(5, sizeof(int));  // 5 elements, each = 0

// realloc — resize
arr = (int*) realloc(arr, 10 * sizeof(int));  // now 10 elements

// free — ALWAYS free when done!
free(arr);
arr = NULL;  // good practice: set to NULL after free
```

4️⃣ Global vs Local Memory

	Global Memory	Local Memory
Declared	Outside all functions	Inside a function/block
Stored in	Data Segment	Stack
Scope	Entire program	Only inside that function
Lifetime	Whole program duration	Only during function execution
Default value	0	Garbage

```c
int globalVar = 20;  // global

void myFunc() {
    int localVar = 10;  // local
    printf("%d%d", localVar, globalVar);  // both accessible here
}
// localVar is GONE after myFunc() ends
// globalVar exists until program ends
```

5️⃣ Error Handling in C

errno — global error variable

```c
#include<errno.h>
// After a function fails, check errno for error code
```

Error Handling Functions

```c
// 1. perror() — prints error message with your description
FILE *f = fopen("missing.txt", "r");
if (f == NULL) {
    perror("Error");  // prints: Error: No such file or directory
}

// 2. strerror() — returns error string
printf("%s", strerror(errno));

// 3. ferror() — check if file operation had error
if (ferror(fptr)) printf("File error occurred!");

// 4. feof() — check if end of file reached
if (feof(fptr)) printf("End of file!");

// 5. clearerr() — clear error/EOF flags
clearerr(fptr);
```

Exit Status

```c
#include<stdlib.h>
exit(EXIT_SUCCESS);  // 0 — program completed successfully
exit(EXIT_FAILURE);  // 1 — program had an error
```

Divide by Zero

> ⚠️ Dividing by zero → **undefined behavior** in C!

```c
// Always check before dividing!
if (denominator != 0) {
    result = numerator / denominator;
}
```

6️⃣ Interfacing C with Python

Calling C from Python (using ctypes)

Step 1: Write C code

```c
// example.c
int add_integers(int a, int b) {
    return a + b;
}
```

Step 2: Compile as shared library

```
gcc -shared -fPIC -o libexample.so example.c
```

Step 3: Call from Python

```
import ctypes
lib = ctypes.CDLL('./libexample.so')
lib.add_integers.argtypes = (ctypes.c_int, ctypes.c_int)
lib.add_integers.restype = ctypes.c_int
result = lib.add_integers(3, 5)  # → 8
```

Other methods:
- **CFFI** — more Pythonic interface
- **SWIG** — auto-generates wrapper code
- **Python C API** — write Python extension in C

Calling Python from C

```
#include<Python.h>

int main() {
    Py_Initialize();  // Start Python interpreter

    PyObject* pModule = PyImport_ImportModule("myscript");
    PyObject* pFunc = PyObject_GetAttrString(pModule, "my_function");
    PyObject* pValue = PyObject_CallObject(pFunc, NULL);

    Py_Finalize();  // Stop Python interpreter
    return 0;
}
```

Steps:
1. `Py_Initialize()` — start Python
2. `PyImport_ImportModule()` — import .py file
3. `PyObject_GetAttrString()` — get function
4. `PyObject_CallObject()` — call function
5. `Py_Finalize()` — stop Python

7️⃣ Threads in C (pthreads)

A **thread** = smallest unit of execution within a process.

Why use threads?

- **Concurrency** — multiple tasks “at the same time”
- **Performance** — use multiple CPU cores
- **Responsiveness** — app stays responsive

Thread vs Process

	Thread	Process
Memory	Shares with other threads	Own memory space
Speed	Faster to create	Slower to create
Communication	Easy (shared memory)	Hard (needs IPC)

Creating a Thread

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

void* myThreadFunc(void* arg) {
    printf("Hello from thread!\n");
    return NULL;
}

int main() {
    pthread_t thread_id;

    pthread_create(&thread_id, NULL, myThreadFunc, NULL);
    // Arguments: &thread_id, attributes, function, arguments

    pthread_join(thread_id, NULL);  // wait for thread to finish
    return 0;
}
// Compile: gcc program.c -lpthread
```

Passing Arguments to Thread

```c
void* thread_function(void* arg) {
    int* num = (int*) arg;
    printf("Got:%d\n", *num);
    return NULL;
}

int main() {
    pthread_t thread;
    int num = 10;
    pthread_create(&thread, NULL, thread_function, &num);
    pthread_join(thread, NULL);
}
```

8️⃣ Thread Synchronization

Problem: Multiple threads accessing same data = **race conditions** (unpredictable results)!

Solution: Synchronization techniques.

Mutex (Mutual Exclusion) — most important!

```
pthread_mutex_t lock;

void* thread_function(void* arg) {
    pthread_mutex_lock(&lock);    // LOCK — only 1 thread enters
    counter++;                    // Critical section
    pthread_mutex_unlock(&lock);  // UNLOCK — others can enter now
    return NULL;
}

int main() {
    pthread_mutex_init(&lock, NULL);   // initialize
    // ... create threads ...
    pthread_mutex_destroy(&lock);      // cleanup
}
```

Condition Variables — for thread signaling

```
pthread_mutex_t lock;
pthread_cond_t cond;
int ready = 0;

// Waiting thread:
pthread_mutex_lock(&lock);
while (!ready) {
    pthread_cond_wait(&cond, &lock);  // wait for signal
}
pthread_mutex_unlock(&lock);

// Signaling thread:
pthread_mutex_lock(&lock);
ready = 1;
pthread_cond_signal(&cond);           // wake up waiting thread
pthread_mutex_unlock(&lock);
```

Key functions:

- `pthread_cond_wait()` — wait for condition
- `pthread_cond_signal()` — wake ONE thread
- `pthread_cond_broadcast()` — wake ALL threads

MODULE IV — Network Security

Generic Security Software

Type	What It Does	Examples
Antivirus/Anti-malware	Detects/removes malicious software	Norton, McAfee, Bitdefender
Firewalls	Filters network traffic	Windows Firewall, Cisco ASA
Encryption Tools	Makes data unreadable to unauthorized users	VeraCrypt, BitLocker
IDS	Detects intrusions, sends alerts	Snort, OSSEC
VPN	Encrypts internet traffic, hides IP	NordVPN, OpenVPN
SIEM	Monitors and analyzes security events	Splunk, IBM QRadar

Windows Firewall

What is it?

Built-in Windows security feature that **monitors and filters network traffic**.
- First introduced in **Windows XP**
- Free and built-in (no extra cost)

Three Profiles

Profile	When Used	Trust Level
Domain	Connected to corporate/domain network	High
Private	Home network	Medium
Public	Public Wi-Fi, coffee shops	Low

How It Works

1. **Packet Filtering** — examines each data packet
2. **Stateful Inspection** — tracks state of connections
3. **Inbound Traffic** — blocks unsolicited by default
4. **Outbound Traffic** — allows by default (can restrict)

Strengths vs Limitations

Strengths	Limitations
Free and built-in	No VPN support
Easy to use	Advanced config needs networking knowledge
Lightweight	Limited home user central management
Auto-updates with Windows	—

Linux Firewalls

Types of Linux Firewalls

Tool	Description	Used On
iptables	Traditional, command-line packet filter	All Linux
nftables	Modern replacement for iptables	All Linux
firewalld	Dynamic frontend (works with iptables/nftables)	RHEL, CentOS
UFW	Uncomplicated Firewall, beginner-friendly	Ubuntu, Debian

iptables — Key Commands

```
# Allow SSH (port 22)
iptables -A INPUT -p tcp --dport 22 -j ACCEPT

# Block HTTP (port 80)
iptables -A INPUT -p tcp --dport 80 -j DROP

# List all rules
iptables -L
```

Chains in iptables:
- `INPUT` — incoming traffic to this machine
- `OUTPUT` — outgoing traffic from this machine
- `FORWARD` — traffic passing through this machine

UFW — Simple Commands

```
ufw enable          # Turn on firewall
ufw allow ssh       # Allow SSH
ufw allow 80/tcp    # Allow HTTP
ufw status          # Check status
```

nftables Example

```
nft add rule inet filter input tcp dport 22 accept  # allow SSH
nft add rule inet filter input tcp dport 80 accept  # allow HTTP
```

firewalld Example

```
firewall-cmd --zone=public --add-port=80/tcp  # allow HTTP in public zone
firewall-cmd --list-all                        # list all rules
```

Access Control Lists (ACLs)

ACLs = **rules that allow/deny access** to network resources in firewalls/routers.

Types of ACLs

Type	Filters Based On	Use Case
Standard ACL	Source IP only	Simple access control
Extended ACL	Source IP + Dest IP + Protocol + Port	Complex filtering
Named ACL	User-defined names	Easier management
Dynamic ACL	Temporary rules	Condition-based changes

ACL Syntax

```
access-list [number] [permit|deny] [protocol] [source] [wildcard] [destination] [port]
```

Examples

```
# Standard ACL – allow a subnet
access-list 10 permit 192.168.1.0 0.0.0.255

# Extended ACL – allow HTTP from specific subnet
access-list 100 permit tcp 192.168.1.0 0.0.0.255 any eq 80

# Allow SSH to specific server
access-list 101 permit tcp any 192.168.1.100 eq 22
```

Types of Firewalls

Type	How It Works	Example	Pros/Cons
Packet-Filtering	Checks IP, port, protocol	Cisco IOS ACLs	Fast but limited security
Stateful Inspection	Tracks connection state	Check Point	Better security, more resources
Proxy	Acts as middleman	Squid	Anonymity, can cache; may add latency
NGFW (Next-Gen)	Combines all + deep packet inspection	Palo Alto	Comprehensive but expensive
Software Firewall	Installed on device	Windows Defender	Easy install, device-level only
Hardware Firewall	Physical device	Fortinet FortiGate	Protects whole network, costly
Cloud Firewall	Cloud-hosted	AWS WAF	Scalable, no hardware needed

---

Honeypots and Trapdoors

Honeypots

A **decoy system** designed to attract attackers and study their behavior.

Types:

Type	Description	Example
Low-Interaction	Simulates few services, easy to set up	Honeyd

Type	Description	Example
High-Interaction	Full OS, more realistic, riskier	Kippo
Research	Academic study of attacks	—
Production	Detect real-world attacks	—

How Honeypots Work:
1. Attacker scans network
2. Honeypot simulates vulnerabilities
3. Attacker engages with honeypot
4. All activities are **logged and analyzed**
5. Real systems stay safe

Uses: Threat intelligence, intrusion detection, distraction

Trapdoors (Backdoors)

Hidden methods to **bypass normal security** of a system.

Type	Description
Legitimate	Built by developers for debugging/maintenance
Malicious	Inserted by attackers for unauthorized access

	Honeypot	Trapdoor
Purpose	Attract attackers, study behavior	Covert system access
Visibility	Visible to attackers (as bait)	Hidden from users
Use	Security research, detection	Often malicious

VPN (Virtual Private Network)

What is VPN?

Creates a **secure, encrypted tunnel** over the internet.
- Hides your real IP address
- Encrypts all traffic
- Bypasses geo-restrictions

How VPN Works

```
User Device → [Encrypt Data] → VPN Server → Internet
Internet → VPN Server → [Decrypt Data] → User Device
```

Types of VPN

Type	Use Case	Example
Remote Access VPN	Employee connects from home	Cisco AnyConnect
Site-to-Site VPN	Two offices connected securely	MPLS VPN
Client-to-Site VPN	Device connects to company network	OpenVPN

VPN Protocols

Protocol	Security	Speed	Use Case
PPTP	Weak	Fast	Low-security streaming
L2TP/IPSec	Good	Medium	Corporate communications
OpenVPN	Very Strong	Medium	Personal and business
IKEv2	Strong	Fast	Mobile devices
WireGuard	Excellent	Very Fast	Modern, privacy-focused

Network Intrusion Detection System (NIDS)

What is NIDS?

Monitors **network traffic** for suspicious activity and **alerts** administrators.
- **Passive** — only monitors, doesn't block
- Acts as an early warning system

How NIDS Works

1. **Packet Inspection** — examines network packets
2. **Signature-Based Detection** — compares to known attack patterns
3. **Anomaly-Based Detection** — flags unusual traffic
4. **Alerts** — notifies admin of threats

Types of IDS

Type	Monitors	Example
NIDS (Network)	Entire network traffic	Snort, Suricata
HIDS (Host)	Single device/host	OSSEC
Hybrid	Both network and host	—

Implementation Steps

1. Network assessment
2. Choose NIDS tool (Snort, Suricata, etc.)
3. Deploy sensors at strategic points
4. Configure detection rules
5. Set up monitoring/alerts
6. Regular updates

Network Intrusion Prevention System (NIPS)

NIPS vs NIDS

	NIDS	NIPS
Action	**Detects** and alerts	**Detects AND blocks**
Mode	Passive (out-of-band)	Active (inline)
Response	Alert administrator	Automatically blocks threat

How NIPS Works

- Placed **inline** with traffic (all traffic flows through it)
- Detects malicious activity → **immediately drops/blocks** it
- Uses signature + anomaly + behavior-based detection

NIPS Benefits

- Real-time threat prevention
- Automated response (no human needed immediately)
- Reduces risk of breaches
- Handles known and unknown (zero-day) threats

NIPS Challenges

- **False Positives** — blocks legitimate traffic
- **High resource demand** — needs powerful hardware
- **Encrypted traffic** — can't inspect HTTPS content
- **Evasion** — attackers can sometimes bypass it

Router Security

Common Threats to Routers

- **Weak passwords** — default/easy passwords
- **Outdated firmware** — unpatched vulnerabilities
- **Open ports** — entry points for attackers
- **DDoS attacks** — router gets overwhelmed

Router Security Checklist

- [ ] Change default admin username and password
- [ ] Use WPA3 encryption (WiFi)
- [ ] Disable WPS and remote management
- [ ] Keep firmware updated
- [ ] Set up guest network
- [ ] Enable firewall, restrict open ports
- [ ] Use MAC address filtering

- [ ] Monitor traffic and logs regularly

Switch Security

Common Threats to Switches

Attack	Description
MAC Flooding	Overloads switch with fake MACs → acts like a hub
VLAN Hopping	Exploits misconfiguration to access other VLANs
STP Attack	Manipulates Spanning Tree Protocol
ARP Spoofing	Sends false ARP messages to intercept traffic

Protection Measures

```
Port Security       → Limit MAC addresses per port
DHCP Snooping      → Prevents rogue DHCP servers
Dynamic ARP Inspection (DAI) → Validates ARP packets
BPDU Guard         → Protects Spanning Tree Protocol
Private VLANs      → Isolates devices within same VLAN
```

Switch Security Checklist

- [ ] Change default credentials
- [ ] Secure unused ports and VLANs
- [ ] Enable port security and BPDU guard
- [ ] Implement DHCP snooping and DAI
- [ ] Segment network with VLANs
- [ ] Use 802.1X for port-based authentication
- [ ] Monitor logs using SNMP

Proxy Server

What is a Proxy?

An **intermediary** between client and destination server.

```
Client → Proxy Server → Internet
Internet → Proxy Server → Client
```

Types of Proxy

Type	Description	Use Case
Forward Proxy	In front of clients, forwards their requests	Content filtering, anonymity
Reverse Proxy	In front of servers, handles requests for them	Load balancing, security

Type	Description	Use Case
Transparent Proxy	Client doesn't know it exists	Caching, content filtering

Proxy Configurations

- **IP Address + Port** — basic setup on client
- **Authentication** — username/password for access
- **Caching** — stores frequently accessed content
- **ACLs** — control which sites/content can be accessed
- **SSL/TLS** — intercept and inspect encrypted traffic

Popular Proxy Tools

Tool	Type	Use
Squid	Forward Proxy	Web caching and filtering
Nginx	Reverse Proxy	Load balancing, performance
HAProxy	Load Balancer + Proxy	High-performance TCP/HTTP

Load Balancers

What is a Load Balancer?

Distributes incoming traffic across **multiple servers** to prevent overload.

Types

Type	Works At	Description
Layer 4	Transport (TCP/UDP)	Fast, basic routing by IP/port
Layer 7	Application (HTTP)	Smart routing by content, headers, cookies
Hardware	Physical device	High performance, expensive
Software	Virtual/cloud	Flexible, cost-effective

Load Balancing Algorithms

Algorithm	How It Works
Round Robin	Sends requests to each server in turn
Least Connections	Sends to server with fewest active connections
Weighted Round Robin	Powerful servers get more traffic
IP Hash	Same client always goes to same server
Least Response Time	Sends to fastest responding server

Popular Tools

- **HAProxy** — open-source, powerful
- **Nginx** — web server + load balancer

- **AWS ELB** — cloud load balancer
- **Azure Load Balancer** — Microsoft cloud

IPv6 and IPv6 Security

IPv4 vs IPv6

	IPv4	IPv6
Address Length	32 bits	128 bits
Address Pool	~4.3 billion	3.4×10^{38} (massive!)
Notation	Decimal (192.168.1.1)	Hex (2001:0db8:85a3::8a2e:0370:7334)
Security	Optional IPSec	**Built-in IPSec**
NAT Required?	Yes (addresses ran out)	No

Why Move to IPv6?

- **IPv4 exhaustion** — ran out of addresses
- **Better security** — IPSec built-in
- **Improved performance** — efficient routing
- **No NAT** — simpler network design

IPv6 Security Features

- **IPSec** — mandatory, provides encryption + authentication
- **End-to-end encryption** — direct device communication
- **No broadcast** — uses multicast/anycast (reduces DDoS risk)
- **SLAAC** — auto-configuration (but can be a security risk)

IPv6 Security Checklist

- [ ] Enable IPSec for encrypted communication
- [ ] Use Secure Neighbor Discovery (SEND)
- [ ] Configure firewalls for IPv6 (not just IPv4!)
- [ ] Disable unnecessary tunneling
- [ ] Use IPv6-aware IDS/IPS
- [ ] Monitor IPv6 traffic logs

Secure Forwarding in Overlay Networks

What is an Overlay Network?

A **virtual network** built ON TOP of a physical network.
- **Examples:** VPN, CDN, P2P networks
- Data travels through **logical links** that may span multiple physical links

What is Secure Forwarding?

Ensuring data travels between nodes **without being intercepted, modified, or compromised**.

Four objectives:
1. **Confidentiality** — data is encrypted
2. **Integrity** — data is not tampered with
3. **Authentication** — verify sender/receiver identity
4. **Availability** — ensure traffic can flow

Key Techniques for Secure Forwarding

Technique	How It Helps
Encryption (TLS, IPSec, DTLS)	Ensures confidentiality
Forwarding Path Authentication	Verifies route integrity
Overlay Isolation (VLANs)	Separates traffic
Traffic Filtering (ACLs/firewalls)	Blocks unauthorized forwarding

Common Threats in Overlay Networks

Threat	Description	Mitigation
Eavesdropping	Attacker intercepts traffic	Encryption (IPSec/TLS)
Man-in-the-Middle	Traffic rerouted through malicious node	Path authentication
DoS Attack	Overloads network nodes	Rate-limiting, load balancing
Routing Attacks	Manipulates routing tables	Secure routing protocols

Role of SDN (Software-Defined Networking)

- **Centralized control** — easier to manage security policies
- **Dynamic updates** — automatically updates forwarding rules
- **Better path control** — enforces encrypted paths

📝 QUICK REVISION CHEATSHEET

Most Important Code Snippets

File Handling (Very Likely in Exam!)

```c
#include<stdio.h>

int main() {
    FILE *fp = fopen("test.txt", "w");
    if (fp == NULL) { perror("Error"); return 1; }

    fprintf(fp, "Hello World\n");
    fclose(fp);
```

```
    fp = fopen("test.txt", "r");
    char line[100];
    fgets(line, 100, fp);
    printf("%s", line);
    fclose(fp);
    return 0;
}
```

Dynamic Memory (Very Likely!)

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

int main() {
    int n = 5;
    int *arr = (int*) malloc(n * sizeof(int));

    for (int i = 0; i < n; i++) arr[i] = i * 10;
    for (int i = 0; i < n; i++) printf("%d ", arr[i]);

    free(arr);  // ALWAYS free!
    return 0;
}
```

Recursion — Factorial

```
int factorial(int n) {
    if (n == 0 || n == 1) return 1;  // base case
    return n * factorial(n - 1);      // recursive case
}
```

Pointer Basics

```
int x = 10;
int *ptr = &x;      // ptr holds address of x
*ptr = 20;          // change x via pointer
printf("%d", x);    // 20
```

Structure with Pointer

```
struct Point { int x, y; };
struct Point p = {3, 4};
struct Point *ptr = &p;
printf("%d%d", ptr->x, ptr->y);  // 3 4
```

Key Terms Glossary

Term	One-Line Definition
Pointer	Variable that stores memory address
Dereferencing	Getting value at pointer's address using `*`
NULL pointer	Pointer that points to nothing (safe)
Stack	Memory for local variables (auto-managed)
Heap	Memory for dynamic allocation (manual)
malloc	Allocate memory at runtime
free	Release allocated memory
Recursion	Function calling itself
Base case	Condition that stops recursion
Thread	Smallest execution unit in a process
Mutex	Lock that allows only 1 thread in critical section
Race condition	Bug when threads access shared data unsafely
Firewall	Security tool filtering network traffic
NIDS	Detects intrusions passively (alerts only)
NIPS	Prevents intrusions actively (blocks)
VPN	Encrypted tunnel over internet
Honeypot	Decoy system to attract/study attackers
ACL	Rules that allow/deny network access
IPv6	Next-gen IP with 128-bit addresses
Proxy	Intermediary between client and server

🎯 Exam Tips

1. **Part A short questions:** Write definition + 1-line example
2. **Part B 6-mark questions:** Write definition + working + code/diagram
3. **Part C 8-mark questions:** Write detailed explanation + compare/contrast table + code + diagram

Topics Most Likely to Come in Exam

🔥 **HIGH PRIORITY:**
- Pointers (all types, pointer arithmetic)
- File handling (fopen, fread, fwrite, fclose)
- Dynamic memory allocation (malloc, calloc, free)
- Recursion (types, memory allocation)
- Structures (typedef, nested, bit fields)
- Threads (pthread_create, mutex, condition variables)

⭐ **MEDIUM PRIORITY:**
- Storage classes (auto, extern, static, register)
- Type qualifiers (const, volatile, restrict)

- NIDS vs NIPS
- VPN types and protocols
- Firewall types

📖 **GOOD TO KNOW:**
- Preprocessor directives
- Macros (types and examples)
- Honeypots vs Trapdoors
- Load balancing algorithms
- IPv6 security features