C Programming Notes

◆ Origins of C Language

- **1960s** – Early languages like **ALGOL**, **BCPL**, and **B** were developed.
    - **ALGOL** (Algorithmic Language) was influential in structuring programming.
    - **BCPL** (Basic Combined Programming Language) was created by Martin Richards in 1967 and was a precursor to C.
    - **B Language** was developed by Ken Thompson at Bell Labs as a simplified version of BCPL.

◆ Birth of C Language (1972)

- **Dennis Ritchie** at **Bell Labs** developed the **C programming language** in **1972**.
- C was created as an evolution of **B** to write the **Unix operating system**.
- It introduced features like:
    - Data types
    - Structures
    - Low-level memory access
    - Portability

◆ C and Unix (1973)

- Unix was **re-written in C**, making it one of the first operating systems written in a high-level language.
- This greatly improved its portability across machines.

◆ Standardization

- **1978** – **K&R C**: The first book, *"The C Programming Language"* by **Kernighan and Ritchie**, described the C language informally.
- **1983** – ANSI (American National Standards Institute) started standardizing C.
- **1989** – ANSI released **ANSI C (C89)**, the first official standard.

C Programming Notes

🔷 What is a Variable?

- A **variable** in C is a **named memory location** used to store a value that can be **changed during program execution**.
- It acts as a **container** for data.

✅ Example:

```
c
CopyEdit
int age = 25;
```

Here, `age` is a variable storing the value `25`.

◆ How to Create a Variable?

To create a variable in C:

```
c
CopyEdit
<data_type> <variable_name> = <initial_value>;
```

✅ Example:

```
c
CopyEdit
int marks = 90;
float price = 49.99;
char grade = 'A';
```

You can also declare without assigning a value:

```
c
CopyEdit
int score;
```

◆ Types of Variables in C

1. Based on Data Type:

Type	Description	Example
`int`	Integer numbers	`int a = 5;`
`float`	Decimal numbers	`float b = 4.5;`

C Programming Notes

Type	Description	Example
`char`	Single characters	`char c = 'A';`
`double`	Double-precision float	`double d = 10.1234;`

---

2. Based on Scope and Storage Class:

Type	Description
Local	Declared inside a function or block
Global	Declared outside all functions
Static	Retains value between function calls
Extern	Refers to a variable declared elsewhere
Register	Stored in CPU register for fast access

◆ Rules for Naming Variables

1. **Must begin with a letter (A–Z, a–z) or underscore _**
2. **Can contain letters, digits (0–9), and underscores**
3. **No spaces or special characters**
4. **Cannot use C keywords** (like `int`, `return`, `float`, etc.)
5. **Case-sensitive** (`age` and `Age` are different)
6. **Should be meaningful** (prefer `totalMarks` over `tm`)

✅ Valid names: `count`, `_num1`, `total_marks`
❌ Invalid names: `2num`, `float`, `my marks`

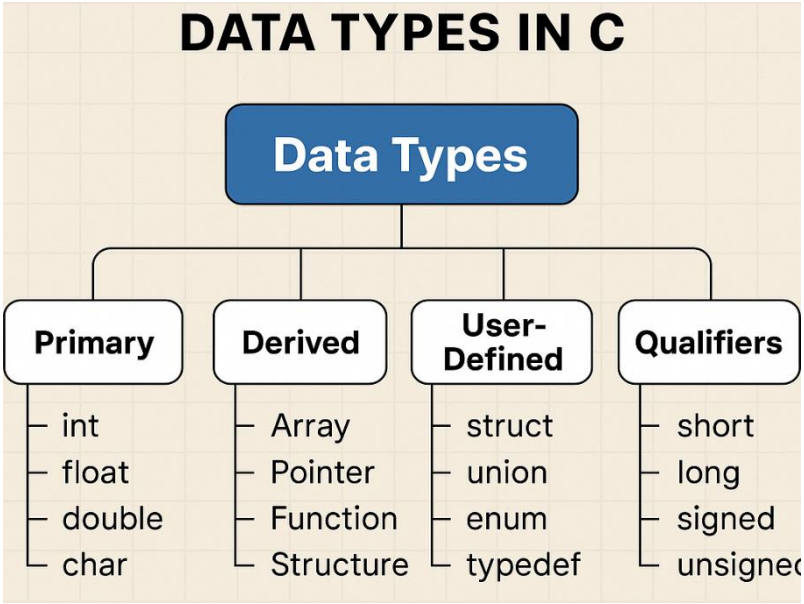
◆ Examples

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

int main() {
    int age = 20;          // integer variable
    float height = 5.9;    // float variable
    char grade = 'A';      // character variable

    printf("Age: %d\n", age);
```

```
    printf("Height: %.1f\n", height);
    printf("Grade: %c\n", grade);

    return 0;
}
```

◆ What are Data Types?

- **Data types** in C define the **type of data** a variable can store.
- They determine:
  - The **size** of memory to allocate.
  - The **range** of values the variable can hold.
  - The **operations** allowed on the variable.



🩶 1. Primary (Fundamental) Data Types

Data Type	Size (in bytes)	Format Specifier	Example Value
`int`	2 or 4	`%d`	`10, -25`
`float`	4	`%f`	`3.14, -1.5`
`double`	8	`%lf`	`123.4567`
`char`	1	`%c`	`'A', 'z'`
`void`	0	-	No value (used for empty return type)

📌 *Note:* Size may vary depending on compiler/system (typically 4 bytes for `int` on 32/64-bit systems).

---

2. Derived Data Types

Type	Description	Example
Array	Collection of same data type	`int a[5];`
Pointer	Stores memory address	`int *p;`
Function	Block of reusable code	`int sum(int a, int b);`
Structure	Group of different types	`struct Student {...};`
Union	Memory shared between variables	`union Value {...};`

---

3. User-Defined Data Types

Type	Description	Example
`struct`	Groups multiple variables	`struct Book {...};`
`union`	Similar to struct, but shared memory	`union Data {...};`
`enum`	Used for enumerated constants	`enum Days {Sun, Mon};`
`typedef`	Gives alias to existing data type	`typedef int age;`

---

4. Qualifiers in C

Used to **modify the range or behavior** of base data types:

➤ Size Modifiers

Modifier	Used With	Description
`short`	`Int`	Smaller integer size
`long`	`int`, `double`	Larger size
`long long`	`Int`	Even larger integer range

➤ Sign Modifiers

C Programming Notes

Modifier	Description
`signed`	Can store positive and negative values
`unsigned`	Only positive values

✅ Example:

```
c
CopyEdit
unsigned int x = 300;
short int y = -10;
long double z = 12345.6789;
```

🔹 Data Type Summary Table

Type	Size (Bytes)*	Range (Approx.)	Format Specifier
`char`	1	-128 to 127	`%c`
`unsigned char`	1	0 to 255	`%c`
`int`	4	-2,147,483,648 to 2,147,483,647	`%d`
`unsigned int`	4	0 to 4,294,967,295	`%u`
`short int`	2	-32,768 to 32,767	`%hd`
`long int`	4 or 8	Larger range	`%ld`
`float`	4	~±3.4E±38 (7 digits precision)	`%f`
`double`	8	~±1.7E±308 (15 digits precision)	`%lf`
`long double`	10 or 12	More precision	`%Lf`

📌 *Size depends on system and compiler (e.g., Turbo C vs GCC).*

🔹 Examples

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

int main() {
    int age = 25;
    float weight = 65.5;
```

```
    char grade = 'A';
    double distance = 12345.6789;

    printf("Age = %d\n", age);
    printf("Weight = %.2f\n", weight);
    printf("Grade = %c\n", grade);
    printf("Distance = %.41f\n", distance);

    return 0;
}
```

◆ What is an Operator in C?

- An **operator** is a **symbol** that performs an **operation on one or more operands (variables, constants, or values)**.
- Example: `+`, `-`, `*`, `==`, `&&`

🔷 Types of Operators in C

Category	Description
1. Arithmetic Operators	Mathematical operations
2. Relational (Comparison)	Compare values
3. Logical Operators	Combine multiple conditions
4. Assignment Operators	Assign values
5. Increment / Decrement	Increase or decrease value
6. Conditional (Ternary)	Shortcut for if-else
7. Bitwise Operators	Operations at binary level
8. Special Operators	Miscellaneous (sizeof, pointer, etc.)

🔶 1. Arithmetic Operators

Operator	Description	Example	Result
`+`	Addition	`5 + 3`	`8`
`-`	Subtraction	`5 - 3`	`2`
`*`	Multiplication	`5 * 3`	`15`

Operator	Description	Example	Result
`/`	Division	`5 / 2`	`2` (int)
`%`	Modulus (remainder)	`5 % 2`	`1`

✅ Example:

```
c
CopyEdit
int a = 10, b = 3;
printf("%d", a % b);  // Output: 1
```

🔹 2. Relational Operators

Used to **compare** values. Returns **1 (true)** or **0 (false)**.

Operator	Meaning	Example
`==`	Equal to	`a == b`
`!=`	Not equal to	`a != b`
`>`	Greater than	`a > b`
`<`	Less than	`a < b`
`>=`	Greater or equal	`a >= b`
`<=`	Less or equal	`a <= b`

🔹 3. Logical Operators

Used to **combine conditions** (mostly in `if` statements).

Operator	Meaning	Example	Result
`&&`	Logical AND	`a > 0 && b < 10`	True if **both** are true
`	`	`	`	Logical OR
`!`	Logical NOT	`!(a == b)`	Inverts the condition

C Programming Notes

🔷 4. Assignment Operators

Used to **assign values** to variables.

Operator	Meaning	Example
`=`	Assign value	`a = 10`
`+=`	Add and assign	`a += 5` → `a = a + 5`
`-=`	Subtract and assign	`a -= 5`
`*=`	Multiply and assign	`a *= 2`
`/=`	Divide and assign	`a /= 2`
`%=`	Modulo and assign	`a %= 2`

🔷 5. Increment & Decrement Operators

Operator	Description	Example	Result
`++`	Increment by 1	`a++`	`a = a + 1`
`--`	Decrement by 1	`a--`	`a = a - 1`

✅ Types:

- **Postfix**: `a++` (use then increment)
- **Prefix**: `++a` (increment then use)

🔷 6. Conditional (Ternary) Operator

- **Shortcut for `if-else`.**
- ✅ Syntax:

```
c
CopyEdit
condition ? value_if_true : value_if_false;
```

✅ Example:

```
c
CopyEdit
```

```
int a = 5, b = 10;
int max = (a > b) ? a : b;
```

---

7. Bitwise Operators

Used to perform bit-level operations.

Operator	Name	Meaning
`&`	AND	`a & b`
`` ` ``	`` ` ``	OR
`^`	XOR	`a ^ b`
`~`	NOT	`~a` (invert bits)
`<<`	Left shift	`a << 1`
`>>`	Right shift	`a >> 1`

✓ Example:

```
c
CopyEdit
int a = 5, b = 3;
printf("%d", a & b);  // Output: 1 (binary AND)
```

---

8. Special Operators

Operator	Use
`sizeof`	Returns size of variable/type
`&`	Address-of (used with pointers)
`*`	Pointer dereferencing
`->`	Access structure member via pointer
`.`	Access structure member directly

✓ Example:

```
c
CopyEdit
```

```
int a = 10;
printf("%lu", sizeof(a));  // Output: 4 (on 32-bit system)
```

✅ Summary Table of C Operators

Type	Examples
Arithmetic	+, -, *, /, %
Relational	==, !=, <, >, <=, >=
Logical	&&, `
Assignment	=, +=, -=, etc.
Increment/Decrement	++, --
Ternary	? :
Bitwise	&, `
Special	sizeof, &, *, ., ->

🧪 Simple Practice Program

```
c
CopyEdit
#include <stdio.h>
int main() {
    int a = 5, b = 10;
    printf("Sum = %d\n", a + b);
    printf("Is a < b? %d\n", a < b);
    printf("Bitwise AND = %d\n", a & b);
    return 0;
}
```

📘 if, if-else, if-else if-else, and nested if-else

◆ 1. `if` Statement

✅Syntax:

```
c
CopyEdit
if (condition) {
    // Code to execute if condition is true
}
```

✅Example:

```
c
```

```
CopyEdit
int age = 20;
if (age >= 18) {
    printf("You are eligible to vote.\n");
}
```

◆ 2. `if-else` Statement

✓*Syntax:*

```
c
CopyEdit
if (condition) {
    // Code if condition is true
} else {
    // Code if condition is false
}
```

✓*Example:*

```
c
CopyEdit
int marks = 45;
if (marks >= 50) {
    printf("Pass\n");
} else {
    printf("Fail\n");
}
```

◆ 3. `if-else if-else` Ladder

Used when you have **multiple conditions** to check.

✓*Syntax:*

```
c
CopyEdit
if (condition1) {
    // Code if condition1 is true
} else if (condition2) {
    // Code if condition2 is true
} else if (condition3) {
    // Code if condition3 is true
} else {
    // Code if all conditions are false
}
```

✓*Example:*

```
c
CopyEdit
int num = 0;
if (num > 0) {
    printf("Positive number\n");
} else if (num < 0) {
    printf("Negative number\n");
```

```
} else {
    printf("Zero\n");
}
```

◆ 4. Nested `if-else` Statement

Placing an `if` or `if-else` block **inside another `if` or `else` block**.

✓Syntax:

```
c
CopyEdit
if (condition1) {
    if (condition2) {
        // Code if both condition1 and condition2 are true
    } else {
        // Code if condition1 is true and condition2 is false
    }
} else {
    // Code if condition1 is false
}
```

✓Example:

```
c
CopyEdit
int age = 25;
char gender = 'M';

if (age >= 18) {
    if (gender == 'M') {
        printf("Adult Male\n");
    } else {
        printf("Adult Female\n");
    }
} else {
    printf("Minor\n");
}
```

◆ 🔍 Comparison Table

Structure	Use When	Can Handle Multiple Conditions?	Else Optional?
`if`	Single condition	✗	✓
`if-else`	Two-way branching	✗	✗
`if-else if`	Multiple conditions	✓	✓
`nested if-else`	Conditions inside conditions	✓	✓

✅ Practice Example: Grading System

```
c
CopyEdit
int marks = 75;

if (marks >= 90) {
    printf("Grade A\n");
} else if (marks >= 75) {
    printf("Grade B\n");
} else if (marks >= 60) {
    printf("Grade C\n");
} else {
    printf("Fail\n");
}
```

📘 for, while, do-while, and switch Statements

🔹 1. `for` Loop

✅ Purpose:

Used when the number of **iterations is known** beforehand.

✅ Syntax:

```
c
CopyEdit
for (initialization; condition; increment/decrement) {
    // Code to execute
}
```

✅ Example:

```
c
CopyEdit
for (int i = 1; i <= 5; i++) {
    printf("%d ", i);
}
```

🔸 Output: `1 2 3 4 5`

🔹 2. `while` Loop

✅ Purpose:

Used when the number of iterations is **not known in advance** and the loop needs to run **as long as a condition is true**.

✅ Syntax:

```
c
CopyEdit
while (condition) {
    // Code to execute
}
```

✅ Example:

```
c
CopyEdit
int i = 1;
while (i <= 5) {
    printf("%d ", i);
    i++;
}
```

🔸 Output: `1 2 3 4 5`

🔹 3. `do-while` Loop

✅ Purpose:

Same as `while`, **but executes at least once**, even if the condition is false.

✅ Syntax:

```
c
CopyEdit
do {
    // Code to execute
} while (condition);
```

✅ Example:

```
c
CopyEdit
```

```
int i = 1;
do {
    printf("%d ", i);
    i++;
} while (i <= 5);
```

🧵 **Output:** `1 2 3 4 5`

🔼 🔁 Loop Comparison Table

Feature	`for` loop	`while` loop	`do-while` loop
Condition Check	At beginning	At beginning	At end
Runs At Least Once?	❌ No	❌ No	✅ Yes
Use When	Iterations known	Iterations unknown	Must run once

🔼 4. `switch` Statement

✅ Purpose:

Used for **multi-way branching**, replacing long `if-else-if` chains when checking the **value of a single variable**.

✅ Syntax:

```
c
CopyEdit
switch (expression) {
    case value1:
        // Code
        break;
    case value2:
        // Code
        break;
    ...
    default:
        // Code
}
```

✅ Example:

```
c
CopyEdit
int day = 3;
switch (day) {
```

```
    case 1:
        printf("Monday");
        break;
    case 2:
        printf("Tuesday");
        break;
    case 3:
        printf("Wednesday");
        break;
    default:
        printf("Invalid Day");
}
```

Output: `Wednesday`

Key Points About `switch`

- Works only with **integer, `char`, and `enum` types**.
- `break` stops execution after the matching case.
- `default` is optional and runs when no case matches.
- **Fall-through** happens if `break` is omitted.

Sample Program Using All

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

int main() {
    int i = 1;

    // for loop
    for (i = 1; i <= 3; i++) {
        printf("For Loop: %d\n", i);
    }

    // while loop
    i = 1;
    while (i <= 3) {
        printf("While Loop: %d\n", i);
        i++;
    }

    // do-while loop
    i = 1;
    do {
        printf("Do-While Loop: %d\n", i);
        i++;
    } while (i <= 3);
```

```
    // switch statement
    int choice = 2;
    switch (choice) {
        case 1: printf("Option 1 selected\n"); break;
        case 2: printf("Option 2 selected\n"); break;
        default: printf("Invalid option\n");
    }

    return 0;
}
```

📃 Arrays, Multidimensional Arrays & Strings

🔹 1. Arrays in C

✅ What is an Array?

- An **array** is a **collection of elements of the same data type** stored in **contiguous memory locations**.
- Elements are accessed using **index**, starting from **0**.

✅ Syntax:

```
c
CopyEdit
data_type array_name[size];
```

✅ Example:

```
c
CopyEdit
int numbers[5] = {10, 20, 30, 40, 50};
```

✅ Accessing Elements:

```
c
CopyEdit
printf("%d", numbers[2]);  // Outputs 30
```

✅ Looping through Array:

```
c
CopyEdit
for (int i = 0; i < 5; i++) {
    printf("%d ", numbers[i]);
}
```

✅ Memory Representation:

If `numbers[5] = {10, 20, 30, 40, 50}`
Memory block:

```
nginx
CopyEdit
Index →   0   1   2   3   4
Value →  10  20  30  40  50
```

🔷 2. Multidimensional Arrays

✅ What is a Multidimensional Array?

- An array with **more than one index** (rows and columns).
- Most commonly used: **2D Arrays** (like a table).

✅ Syntax:

```
c
CopyEdit
data_type array_name[rows][columns];
```

✅ Example:

```
c
CopyEdit
int matrix[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
};
```

✅ Accessing 2D Array Elements:

```
c
CopyEdit
printf("%d", matrix[1][2]);  // Outputs 6
```

C Programming Notes

✅ Looping through 2D Array:

```
c
CopyEdit
for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 3; j++) {
        printf("%d ", matrix[i][j]);
    }
    printf("\n");
}
```

🔷 3. Strings in C

✅ What is a String?

- A **string** is a **1D array of characters** ending with a **null character** `'\0'`.
- Used to store text (words, sentences).

✅ Syntax:

```
c
CopyEdit
char name[10] = "John";
```

This is internally stored as:
`{'J', 'o', 'h', 'n', '\0'}`

✅ Declaring and Reading Strings:

```
c
CopyEdit
char city[20];
scanf("%s", city);  // Input: Mumbai
```

⚠ Note: `scanf()` reads **only one word** (stops at space). Use `fgets()` for full line.

✅ Printing Strings:

```
c
CopyEdit
printf("City: %s", city);
```

C Programming Notes

✅ String Functions (from `<string.h>`)

Function	Description	Example
`strlen(str)`	Returns string length	`strlen("Hello")` → 5
`strcpy(dest, src)`	Copies one string to another	`strcpy(a, b);`
`strcat(str1, str2)`	Concatenates strings	`strcat(a, b);`
`strcmp(str1, str2)`	Compares two strings	`strcmp(a, b)` returns 0 if equal

🧪 Sample Program: Strings & Arrays

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

int main() {
    char name[20];
    printf("Enter your name: ");
    scanf("%s", name);  // Try: Alice

    printf("Welcome, %s!\n", name);
    printf("Length of name: %d\n", strlen(name));
    return 0;
}
```

✅ Key Differences Summary

Feature	Array	String (Character Array)
Stores	Similar type data	Characters + null character `\0`
Type	Any data type	Only `char` type
Null Terminator	Not required	Required
Header File Needed	❌ (no extra)	✅ `<string.h>` for functions

C Programming Notes

🔷 1. What is a Function?

- A **function** is a **block of code** that performs a specific task.
- It helps to **divide a program** into smaller, manageable pieces.
- Promotes **code reuse**, **readability**, and **modularity**.

✅ Syntax of a Function

```
c
CopyEdit
return_type function_name(parameters) {
    // Body of function
    return value;
}
```

✅ Example:

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

🔷 2. Types of Functions in C

✅ Based on Declaration and Use

Type	Description	Example
Library Functions	Predefined in header files	`printf()`, `scanf()`, `strlen()`
User-Defined Functions	Created by the user	`int add(int a, int b)`

✅ Based on Arguments and Return Type

Type	Syntax Example	Description
No argument, no return value	`void greet(void)`	Does not take input or return output
Argument, no return value	`void print(int a)`	Takes input, but does not return anything

C Programming Notes

Type	Syntax Example	Description
No argument, return value	`int getAge(void)`	Returns output, but takes no input
Argument and return value	`int sum(int a, int b)`	Takes input and returns result

✅ Example: Function with Arguments and Return Value

```
c
CopyEdit
int square(int n) {
    return n * n;
}
```

🔷 3. Function Declaration vs Definition vs Call

- **Declaration (Prototype)**: Tells the compiler about the function

  ```
  c
  CopyEdit
  int sum(int, int);  // Function prototype
  ```

- **Definition**: Contains the actual code

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

- **Function Call**: Executes the function

  ```
  c
  CopyEdit
  int result = sum(10, 5);
  ```

🔷 4. Recursive Function

✅ What is Recursion?

- A function that **calls itself** is called **recursive**.
- Used in problems like factorial, Fibonacci, etc.

✅ Syntax:

```
c
CopyEdit
return_type function_name() {
    if (condition)
        return value;
    else
        return function_name();  // Recursive call
}
```

✅ Example: Factorial using Recursion

```
c
CopyEdit
int factorial(int n) {
    if (n == 0 || n == 1)
        return 1;
    else
        return n * factorial(n - 1);
}
```

⚠ Important:

- Recursive functions must have a **base condition** to avoid infinite recursion.

🔼 5. Call by Value vs Call by Reference

✅ 1. Call by Value

- **Copies the actual value** into the function.
- Changes made inside the function do **not affect** the original variable.

✅Example:

```
c
CopyEdit
void change(int a) {
    a = a + 10;
}
```

✅ 2. Call by Reference

- **Passes the address** of the variable.
- Changes made inside the function **affect** the original variable.

✓*Example using pointers:*

```
c
CopyEdit
void change(int *a) {
    *a = *a + 10;
}
```

🔍 Difference Summary

Feature	Call by Value	Call by Reference
What is passed?	Actual value	Address of variable
Original changed?	✗ No	✓ Yes
Uses pointer?	✗ No	✓ Yes

🧪 Sample Program: Add & Modify using Function

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

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

void modify(int *x) {
    *x = *x + 100;
}

int main() {
    int a = 10, b = 20, val = 50;
    add(a, b);

    modify(&val);  // Call by reference
    printf("Modified value = %d\n", val);

    return 0;
}
```

C Programming Notes

◆ What is a Pointer?

A **pointer** is a **variable that stores the memory address** of another variable.

📌 Think of a pointer as a "reference" or "arrow" pointing to another value in memory.

◆ Why Use Pointers?

- Efficient **memory management**
- **Function arguments** (pass by reference)
- **Dynamic memory allocation**
- To work with **arrays**, **strings**, **structures**, etc.

◆ Syntax of a Pointer

```
c
CopyEdit
data_type *pointer_name;
```

📌 `*` indicates it's a pointer
📌 `data_type` is the type of variable it points to

◆ Example:

```
c
CopyEdit
int a = 10;
int *p;

p = &a;  // p stores address of a

printf("Value of a: %d\n", a);
printf("Address of a: %p\n", &a);
printf("Value stored in p: %p\n", p);
printf("Value pointed by p: %d\n", *p);  // Dereferencing
```

◆ Key Operators with Pointers

Operator	Meaning	Example
`&`	Address-of operator	`&a`
`*`	Dereference operator (value at address)	`*p`

◆ Pointer Declaration and Initialization

```
c
CopyEdit
int *ptr;       // Declaration
int num = 20;
ptr = &num;     // Initialization
```

◆ Pointer to Pointer (Double Pointer)

```
c
CopyEdit
int a = 10;
int *p = &a;
int **q = &p;

printf("%d", **q);  // Output: 10
```

◆ Pointer with Arrays

```
c
CopyEdit
int arr[] = {10, 20, 30};
int *p = arr;  // Same as &arr[0]

printf("%d", *(p + 1));  // Output: 20
```

📌 `arr[i]` is same as `*(arr + i)`

◆ Pointer with Functions

📌 Pass by Value:

```
c
CopyEdit
void change(int a) {
    a = 100;
}
```

📌 *Pass by Reference using Pointers:*

```
c
CopyEdit
void change(int *p) {
    *p = 100;
}

int main() {
    int a = 50;
    change(&a);      // Passing address
    printf("%d", a); // Output: 100
}
```

◆ Pointers and Strings

```
c
CopyEdit
char *str = "Hello";
printf("%c", *(str + 1));  // Output: e
```

◆ Pointer Arithmetic

You can perform arithmetic like:

Operation	Meaning
`ptr + 1`	Move to next element
`ptr - 1`	Move to previous element
`++ptr / --ptr`	Increment/decrement address

◆ Null Pointer

```
c
CopyEdit
int *p = NULL;
```

- A pointer with **no address** assigned
- Used for **safe programming**

◆ Dangling Pointer

A pointer pointing to a memory location that has been **freed or deleted**.

```
c
CopyEdit
int *p;
{
    int a = 10;
    p = &a;
}
// Now p is a dangling pointer
```

◆ Void Pointer

Can store the address of **any data type**.

```
c
CopyEdit
void *ptr;
int a = 5;
ptr = &a;
```

◆ Summary Table

Concept	Example
Basic Pointer	`int *p = &a;`
Pointer Dereferencing	`*p`
Address of a Variable	`&a`
Pointer to Pointer	`int **q = &p;`
Array & Pointer	`*(arr + i)`
Void Pointer	`void *p;`
NULL Pointer	`int *p = NULL;`

✅ Practice Example: Swap Using Pointers

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

```
}

int main() {
    int a = 5, b = 10;
    swap(&a, &b);
    printf("a = %d, b = %d", a, b);  // Output: a = 10, b = 5
}
```

C Programming Notes

📄 File Handling

🔹 1. What is File Handling?

In C, **file handling** is a mechanism to **create, open, read, write, and close files** stored on disk. It allows data to **persist** beyond program execution.

📌 Files are used for:

- Storing large data
- Input/output operations
- Data sharing across programs

🔹 2. File Types in C

File Type	Description
Text files	Contain readable characters (`.txt`)
Binary files	Contain encoded data (`.dat`, `.bin`)

🔹 3. File Operations in C

Operation	Function
Create/Open	`fopen()`
Read	`fscanf()`, `fgets()`, `fread()`
Write	`fprintf()`, `fputs()`, `fwrite()`
Close	`fclose()`
Other	`fseek()`, `ftell()`, `rewind()`

🔹 4. File Pointer

```
c
CopyEdit
FILE *fp;
```

Used to point to and manage a file.

🔹 5. Opening a File – `fopen()`

```
c
CopyEdit
fp = fopen("file_name", "mode");
```

◆ Modes in `fopen()`:

Mode	Description
`"r"`	Open file for reading (must exist)
`"w"`	Open file for writing (creates new or overwrites)
`"a"`	Open file for appending (adds at end)
`"r+"`	Open for reading and writing
`"w+"`	Create for read/write, overwrites
`"a+"`	Read/append, creates if not exist
`"rb/wb/ab"`	Binary versions of modes

🔹 6. Closing a File – `fclose()`

```
c
CopyEdit
fclose(fp);
```

- Always close the file after use to **save data** and **free memory**.

🔹 7. Writing to a File

Using `fprintf()` (Formatted):

```
c
CopyEdit
FILE *fp = fopen("data.txt", "w");
fprintf(fp, "Hello, File!");
fclose(fp);
```

Using `fputs()` (String):

```
c
CopyEdit
fputs("Writing string", fp);
```

🔹 8. Reading from a File

Using `fscanf()` (Formatted):

```
c
CopyEdit
int age;
fscanf(fp, "%d", &age);
```

Using `fgets()` (Line by line):

```
c
CopyEdit
char str[100];
fgets(str, 100, fp);
```

🔹 9. File Check (Null Check)

Always check if file is opened successfully:

```
c
CopyEdit
if (fp == NULL) {
    printf("File not found or error opening file!");
    return 1;
}
```

🔹 10. File Positioning Functions

Function	Purpose
`fseek()`	Move file pointer to specific position
`ftell()`	Returns current file position
`rewind()`	Moves pointer to beginning

◆ Example: Using `fseek()` and `ftell()`

```
c
CopyEdit
fseek(fp, 0, SEEK_END); // move to end
long size = ftell(fp);  // get current position
```

C Programming Notes

🔷 11. Binary File Functions

Function	Description
`fread()`	Read binary data
`fwrite()`	Write binary data

✅ Example:

```
c
CopyEdit
struct Student {
    int id;
    char name[20];
};

struct Student s = {1, "John"};
fwrite(&s, sizeof(s), 1, fp);
```

📝 Example Program: Write and Read

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

int main() {
    FILE *fp;
    char str[100];

    // Writing to file
    fp = fopen("example.txt", "w");
    fprintf(fp, "Hello C File Handling");
    fclose(fp);

    // Reading from file
    fp = fopen("example.txt", "r");
    fgets(str, 100, fp);
    printf("Data from file: %s", str);
    fclose(fp);

    return 0;
}
```

🔚 Summary

Concept	Key Function
File Pointer	`FILE *fp;`
Open File	`fopen()`

C Programming Notes

Concept	Key Function
Write File	`fprintf()`, `fputs()`
Read File	`fscanf()`, `fgets()`
Close File	`fclose()`
File Position	`fseek()`, `ftell()`

Structures & Unions

◆ What is a Structure in C?

A **structure** is a **user-defined data type** in C that allows grouping of variables of **different data types** under a single name.

📌 Think of it like a container that holds multiple variables together (e.g., ID, name, salary of an employee).

✅ Syntax of Structure

```
c
CopyEdit
struct StructureName {
    data_type member1;
    data_type member2;
    ...
};
```

✅ Example: Structure Declaration & Usage

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

struct Student {
    int id;
    char name[20];
    float marks;
};

int main() {
    struct Student s1;

    s1.id = 101;
    strcpy(s1.name, "Alice");
    s1.marks = 87.5;

    printf("ID: %d\nName: %s\nMarks: %.2f", s1.id, s1.name, s1.marks);

    return 0;
}
```

C Programming Notes

◆ Initializing Structure

```
c
CopyEdit
struct Student s1 = {101, "Bob", 78.5};
```

◆ Accessing Members

```
c
CopyEdit
s1.id
s1.name
```

◆ Structure with Array

```
c
CopyEdit
struct Student students[3];
```

◆ Array inside Structure

```
c
CopyEdit
struct Book {
    char title[50];
    int pages;
};
```

◆ Structure with Functions

```
c
CopyEdit
void printStudent(struct Student s) {
    printf("%d %s %.2f", s.id, s.name, s.marks);
}
```

◆ Nested Structures

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

```
struct Employee {
    int id;
    struct Date doj;  // Nested structure
};
```

What is a Union?

A **union** is similar to a structure but with a key difference:
🔁 All **members share the same memory location**.

- Only **one member can hold a value at a time**.
- Saves memory when not all members are needed at once.

✅ Syntax of Union

```
c
CopyEdit
union UnionName {
    data_type member1;
    data_type member2;
};
```

✅ Example: Union Declaration & Usage

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

union Data {
    int i;
    float f;
    char str[20];
};

int main() {
    union Data d;

    d.i = 10;
    printf("d.i = %d\n", d.i);

    d.f = 3.14;  // Overwrites i
    printf("d.f = %.2f\n", d.f);

    strcpy(d.str, "C Language");  // Overwrites f
    printf("d.str = %s\n", d.str);

    return 0;
}
```

C Programming Notes

📌 Only the last assigned member will retain correct value.

🔍 Structure vs Union

Feature	Structure	Union
Memory	Separate memory for each member	Shared memory for all members
Size	Sum of all member sizes	Size of largest member
Store multiple values	Yes	No (one at a time)
Use	Full data needed at once	Memory-saving, one data at a time

---

◆ Example Size Comparison

```
c
CopyEdit
struct A {
    int a;      // 4 bytes
    float b;    // 4 bytes
};              // Total: 8 bytes

union B {
    int a;      // 4 bytes
    float b;    // 4 bytes
};              // Total: 4 bytes (shared)
```

? Summary Table

Concept	Keyword	Memory	Use Case
Structure	`struct`	Each member gets own memory	Store multiple related fields
Union	`union`	Shared among all members	Optimize memory use

? Practice Suggestion:

1. Create a structure `Employee` with ID, Name, Salary.
2. Use union `Value` with members: `int i`, `float f`, `char c`. Try assigning all three and print values.
3. Use structure array to store and display multiple student details.

~~~End of document~~~