C Programming Language

Mensi Mohamed Amine

Abstract

C is a powerful general-purpose programming language known for its speed and efficiency. It provides low-level memory access and is widely used in system software, embedded systems, and performance-critical applications.

1 Introduction

Developed in the early 1970s, C became the foundation for many modern languages. Its syntax simplicity and control over hardware make it essential for learning core programming concepts and building robust applications.

Contents

2 All C Programming Language Concepts

Here's a comprehensive overview of all major C programming language concepts:

2.1 Basics of C

- **History of C**: Developed by Dennis Ritchie at Bell Labs (1972)
- **Characteristics**: Procedural, portable, efficient, low-level access
- **Structure of a C program**: Preprocessor directives, main() function, statements

2.2 Data Types

- **Basic types**:
  - `int`: Integer values
  - `float`: Single-precision floating point
  - `double`: Double-precision floating point
  - `char`: Single character
- **Derived types**:
  - Arrays
  - Pointers
  - Structures
  - Unions
- **Type modifiers**: `signed`, `unsigned`, `short`, `long`
- **Type qualifiers**: `const`, `volatile`, `restrict`

2.3 Variables and Constants

- **Variable declaration and initialization**
- **Scope rules**: Local, global, block scope
- **Storage classes**: `auto`, `register`, `static`, `extern`
- **Constants**: Literals, #define, `const` keyword
- **Enumerations**: `enum` type

2.4 Operators

- **Arithmetic**: `+`, `-`, `*`, `/`, `%`
- **Relational**: `==`, `!=`, `>`, `<`, `>=`, `<=`
- **Logical**: , `||`, `!`
- **Bitwise**: `&`, `|`, `^`, `~`, `<<`, `>>`
- **Assignment**: `=`, `+=`, `-=`, etc.
- **Ternary**: `? :`
- **Special**: `sizeof()`, `,` operator
- **Operator precedence and associativity**

2.5 Control Flow

- **Decision making**:
  - `if`, `if-else`, `if-else if-else`
  - `switch-case`
- **Loops**:
  - `for`
  - `while`
  - `do-while`
- **Jump statements**:
  - `break`
  - `continue`
  - `goto`
  - `return`

2.6 Functions

- **Function declaration and definition**
- **Function prototypes**
- **Parameters**: Pass by value, pass by reference (using pointers)
- **Return values**
- **Recursion**
- **Main function**: `int main(int argc, char *argv[])`
- **Variable-length argument lists**: `stdarg.h`

2.7 Arrays

- **One-dimensional arrays**
- **Multi-dimensional arrays**
- **Array initialization**
- **Arrays and pointers relationship**
- **Passing arrays to functions**
- **Array bounds checking** (lack thereof)

2.8 Pointers

- **Pointer declaration and initialization**
- **Pointer arithmetic**
- **Pointers and arrays**
- **Pointers to pointers**
- **Pointers to functions**
- **Void pointers**
- **NULL pointer**
- **Dangling pointers**
- **Memory leaks**

2.9 Strings

- **String as character arrays**
- **String literals**
- **String handling functions**: `strlen()`, `strcpy()`, `strcat()`, `strcmp()`, etc.
- **Character handling functions**: `isalpha()`, `isdigit()`, etc.

2.10 Structures and Unions

- **Structure declaration and initialization**
- **Accessing structure members**
- **Arrays of structures**
- **Pointers to structures**
- **Structure padding and packing**
- **Unions**: Similar to structures but share memory
- **Typedef**: Creating type aliases

2.11 Dynamic Memory Management

- **Memory allocation functions**:
  - `malloc()`
  - `calloc()`
  - `realloc()`
  - `free()`
- **Memory leaks and dangling pointers**
- **Memory segmentation**

2.12 File Input/Output

- **File pointers**: `FILE *`
- **Opening/closing files**: `fopen()`, `fclose()`
- **Reading/writing**:
  - `fprintf()`, `fscanf()`
  - `fgets()`, `fputs()`
  - `fread()`, `fwrite()`
- **File positioning**: `fseek()`, `ftell()`, `rewind()`
- **Error handling**: `feof()`, `ferror()`

2.13 Preprocessor Directives

- #include
- #define (macros)
- #undef
- Conditional compilation:
  - #ifdef, #ifndef, #if, #else, #elif, #endif
- Predefined macros: __FILE__, __LINE__, __DATE__, __TIME__
- #pragma

2.14 Standard Library Functions

- **stdio.h**: Input/output functions
- **stdlib.h**: Memory allocation, conversion, etc.
- **string.h**: String manipulation
- **math.h**: Mathematical functions
- **time.h**: Date and time functions
- **ctype.h**: Character handling
- **stdarg.h**: Variable arguments
- **assert.h**: Diagnostics

2.15 Advanced Concepts

- **Command line arguments**
- **Bit fields**
- **Function pointers**
- **Variable-length arrays (VLAs)**
- **Multi-file programs**
- **Static libraries**
- **Inline assembly**
- **Linkage**: Internal vs. external
- **Callbacks**

2.16 Error Handling

- **errno.h**
- **perror()**
- **strerror()**
- **Exit statuses**
- **Assertions**

2.17 C Standards

- **K&R C** (original)
- **ANSI C (C89/C90)**
- **C99**
- **C11**
- **C17/C18**
- **C23** (upcoming)

2.18 Best Practices

- **Code formatting and style**
- **Commenting**
- **Error checking**
- **Portability considerations**
- **Security considerations** (buffer overflows, etc.)
- **Performance optimization**

3 Basics of C Programming Language

3.1 Introduction to C

C is a general-purpose, procedural programming language developed in 1972 by Dennis Ritchie at Bell Labs. It's one of the most widely used programming languages and forms the basis for many other languages like C++, Java, and Python.

3.2 Key Characteristics

- **Procedural Language**: Programs are divided into functions
- **Mid-level Language**: Combines features of both high-level and low-level languages
- **Portable**: C programs can run on different platforms with minimal changes
- **Efficient**: Provides direct access to memory and hardware
- **Structured**: Supports breaking down problems into smaller functions

3.3 Basic Structure of a C Program

```
#include <stdio.h>  // Preprocessor directive (header file inclusion)

int main() {       // Main function - program entry point
    // Program statements
    printf("Hello, World!\n");  // Print statement
    return 0;       // Return statement
}
```

3.4 Components Explained

- **Preprocessor Directives**
  - Begin with `#` (like `#include`, `#define`)
  - Processed before compilation
  - `#include` adds contents of header files (like `stdio.h` for input/output)
- **Main Function**
  - Every C program must have a `main()` function
  - Execution begins here
  - Return type is typically `int` (returns 0 for success)
- **Statements**
  - Instructions that perform actions
  - End with semicolon (`;`)
  - Example: `printf()` displays output
- **Comments**
  - Single-line: `// comment`
  - Multi-line: `/* comment */`

3.5 Basic Syntax Rules

- **Case Sensitivity**: C is case-sensitive (`main` $\neq$ `Main`)
- **Semicolons**: Terminate statements
- **Whitespace**: Ignored by compiler (except in strings)
- **Curly Braces**: {} define code blocks
- **Indentation**: Not required but improves readability

3.6 First Simple Program (Hello World)

```c
#include <stdio.h>

int main() {
    // Print Hello World message
    printf("Hello, World!\n");

    return 0;
}
```

3.7 Compilation Process

- **Write code** in a text editor (save as `.c` file)
- **Preprocessing**: Expands macros/includes
- **Compilation**: Converts to assembly code
- **Assembly**: Converts to machine code
- **Linking**: Combines with libraries to create executable

3.8 Basic Input/Output

Output: `printf()` function

```c
printf("Text to display");
printf("Number: %d", 42);  // %d for integers
```

Input: `scanf()` function

```c
int age;
scanf("%d", &age);  // & is address-of operator
```

3.9 Common Escape Sequences

- \`n`: New line
- \`t`: Tab
- \`textbackslash`: Backslash
- \`"`: Double quote
- \`0`: Null character

3.10 Why Learn C?

- Foundation for other languages
- Used in system programming (OS, compilers)
- Efficient and fast execution
- Gives low-level memory control
- Still widely used in embedded systems

This covers the fundamental basics of C programming. Would you like me to elaborate on any specific aspect or provide examples of any particular concept?

4 Data Types in C Programming Language

Data types in C define the type of data a variable can hold, the amount of memory it occupies, and the operations that can be performed on it. C supports various data types, categorized into:

1. **Primary (Primitive) Data Types**
2. **Derived Data Types**
3. **User-Defined Data Types**

—

4.1 Primary (Primitive) Data Types

These are the most basic data types in C.

Data Type	**Size (bytes)**	**Range**	**Format Specifier**	**Example**
`int`	2 or 4	-32,768 to 32,767 (2-byte) ¡br¿ -2,147,483,648 to 2,147,483,647 (4-byte)	%d	`int x = 10;`
`float`	4	1.2E-38 to 3.4E+38 (6-7 decimal digits)	%f	`float y = 3.14;`
`double`	8	2.3E-308 to 1.7E+308 (15 decimal digits)	%lf	`double z = 5.6789;`
`char`	1	-128 to 127 (signed) ¡br¿ 0 to 255 (unsigned)	%c	`char ch = 'A';`
`void`	0	No value	-	Used in functions and pointers

Table 1: Primary Data Types in C

4.1.1 Type Modifiers (Change Size/Range)

- `signed` (default for `int` and `char`)
- `unsigned` (only positive values)
- `short` (reduces size)
- `long` (increases size)

Examples:

```
unsigned int a = 40000;    // Only positive
short int b = 100;         // Smaller range
long int c = 1000000L;     // Larger range
```

—

4.2 Derived Data Types

These are built using primary data types.

Data Type	**Description**	**Example**
Array	Collection of similar data types	`int arr[5] = {1, 2, 3, 4, 5};`
Pointer	Stores memory address	`int *ptr = &x;`
Function	Group of statements	`int sum(int a, int b);`

Table 2: Derived Data Types in C

Example:

```
int numbers[3] = {10, 20, 30};  // Array
int *ptr = &numbers[0];         // Pointer
```

—

Data Type	**Description**	**Example**
Structure (`struct`)	Groups different data types	
`struct Student { char name[50]; int age; }`		
Union (`union`)	Shares memory among members	
`union Data { int i; float f; }`		
Enumeration (`enum`)	Defines named integer constants	
`enum Week {Sun, Mon, Tue};`		
Typedef (`typedef`)	Creates an alias for a type	
`typedef int Integer;`		`Integer x = 5;`

Table 3: User-Defined Data Types in C

4.3 User-Defined Data Types

Created using `struct`, `union`, `enum`, and `typedef`.
Example:

```
struct Person {
    char name[30];
    int age;
};

struct Person p1 = {"Alice", 25};
```

—

4.4 Summary Table of Data Types

Category	**Data Types**
Primary	`int`, `float`, `double`, `char`, `void`
Derived	Arrays, Pointers, Functions
User-Defined	`struct`, `union`, `enum`, `typedef`

Table 4: Summary of Data Types in C

—

4.5 Key Points

- `int` is typically **4 bytes** (but can be **2 bytes** on older systems).
- `float` has **less precision** than `double`.
- `char` stores **ASCII values** (1 byte).
- `void` is used for **functions that return nothing** or **generic pointers** (`void*`).
- `sizeof()` operator checks the size of a data type (e.g., `sizeof(int)`).

5 Variables and Constants in C Programming

5.1 Variables

A **variable** is a named memory location that stores data which can be changed during program execution.

5.1.1 Rules for Naming Variables

- Must begin with a **letter** or `underscore (_)`.
- Can contain **letters**, **digits**, and `underscores`.
- Cannot use **C keywords** (`int`, `if`, `else`, etc.).
- **Case-sensitive** (`age` ≠ `Age`).

5.1.2 Variable Declaration & Initialization

```
int age;         // Declaration
age = 25;        // Assignment

float price = 99.99;  // Declaration + Initialization
char grade = 'A';
```

5.1.3 Types of Variables

1. **Local Variables** (Inside a function, accessible only within it)

```
void func() {
    int x = 10;  // Local variable
}
```

2. **Global Variables** (Outside all functions, accessible everywhere)

```
int globalVar = 100;  // Global variable

int main() {
    printf("%d", globalVar);  // Accessible
}
```

3. **Static Variables** (Retains value between function calls)

```
void counter() {
    static int count = 0;  // Static variable
    count++;
    printf("%d", count);
}
```

5.2 Constants

A **constant** is a fixed value that cannot be modified during execution.

5.2.1 Types of Constants

1. **Literal Constants** (Direct values)

```
100     // Integer constant
3.14    // Floating-point constant
'A'     // Character constant
"Hello" // String constant
```

2. `const` **Keyword** (Read-only variables)

```
const float PI = 3.14159;
// PI = 3.14;   Error (cannot modify)
```

3. **#define Preprocessor** (No memory allocation)

```
#define PI 3.14159
#define MAX 100
```

5.2.2 Constant Qualifiers

- `const` → Makes a variable read-only.
- `volatile` → Tells the compiler the variable may change unexpectedly (used in embedded systems).
- `register` → Suggests storing in a CPU register (compiler may ignore).

5.3 Storage Classes (Lifetime & Scope)

Storage Class	Keyword	Lifetime	Scope	Default Value
Automatic	`auto`	Function block	Local	Garbage
Register	`register`	Function block	Local	Garbage
Static	`static`	Entire program	Local/Global	Zero
External	`extern`	Entire program	Global	Zero

Table 5: Storage Classes in C

```
extern int globalVar;  // Declared elsewhere

int main() {
    auto int x = 10;    // Default (auto optional)
    register int y = 20; // Faster access (if possible)
    static int z = 30; // Retains value
}
```

5.4 Key Differences: Variables vs. Constants

Feature	Variables	Constants (`const`/`#define`)
Modifiable?	Yes	No
Memory Allocated?	Yes	`const`: Yes `#define`: No
Type Checking?	Yes	`const`: Yes `#define`: No
Scope Rules Apply?	Yes	`const`: Yes `#define`: No (preprocessor)

Table 6: Key Differences: Variables vs. Constants

5.5 Practical Example

```
#include <stdio.h>
#define MAX_AGE 100  // Macro constant

int globalCount = 0;  // Global variable

int main() {
    const float PI = 3.14;  // Constant variable
    int age = 25;           // Local variable

    printf("PI = %.2f\n", PI);
    printf("Age = %d\n", age);
    printf("Max Age = %d\n", MAX_AGE);

    return 0;
}
```

Output:

```
PI = 3.14
Age = 25
Max Age = 100
```

5.6 Summary

- **Variables** store changeable data (`int x = 10;`).
- **Constants** store fixed values (`const int y = 20;` or `#define Z 30`).
- **Storage classes** (`auto`, `static`, `register`, `extern`) control variable lifetime and scope.

6 Operators in C Programming

In the C programming language, **operators** are special symbols or keywords used to perform operations on variables and values. They are classified into several types based on the kind of operation they perform. Below is an overview:

6.1 Arithmetic Operators

Used for mathematical calculations:

Operator	Description
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division
`%`	Modulus (remainder)

6.2 Relational (Comparison) Operators

Used to compare values:

Operator	Description
`==`	Equal to
`!=`	Not equal to
`>`	Greater than
`<`	Less than
`>=`	Greater than or equal
`<=`	Less than or equal

6.3 Logical Operators

Used to combine conditional statements:

Operator	Description
`&&`	Logical AND
`||`	Logical OR
`!`	Logical NOT

6.4 Bitwise Operators

Operate on bits of integers:

Operator	Description
`&`	Bitwise AND
`|`	Bitwise OR
`^`	Bitwise XOR
`~`	Bitwise NOT
`<<`	Left shift
`>>`	Right shift

6.5 Assignment Operators

Assign values to variables:

Operator	Description
`=`	Assign
`+=`	Add and assign
`-=`	Subtract and assign
`*=`	Multiply and assign
`/=`	Divide and assign
`%=`	Modulus and assign

6.6 Increment and Decrement Operators

Used to increase or decrease a value by 1:

Operator	**Description**
`++`	Increment
`--`	Decrement

6.7 Conditional (Ternary) Operator

A shorthand for `if-else`:

```
condition ? expression_if_true : expression_if_false;
```

6.8 Special Operators

Operator	**Description**
`sizeof`	Returns size of data type
`&`	Address of a variable
`*`	Pointer dereference
`->`	Access structure member via pointer
`.`	Access structure member
`,`	Comma operator (evaluate multiple expressions)

7 Control Flow in C Programming Language

Control flow statements in C determine the order in which instructions are executed in a program. They allow you to make decisions, repeat code blocks, and manage program execution paths.

7.1 Decision-Making Statements

7.1.1 (a) `if` Statement

Executes a block of code if a condition is true.

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

Example:

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

7.1.2 (b) `if-else` Statement

Executes one block if true, another if false.

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

Example:

```c
int num = 10;
if (num % 2 == 0) {
    printf("Even number.\n");
} else {
    printf("Odd number.\n");
}
```

7.1.3 (c) `if-else if-else` Ladder

Checks multiple conditions sequentially.

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

Example:

```c
int marks = 85;
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");
}
```

7.1.4 (d) `switch-case` Statement

Efficient alternative to multiple `if-else` checks.

```
switch (expression) {
    case value1:
        // Code for value1
        break;
    case value2:
        // Code for value2
        break;
    default:
        // Code if no case matches
}
```

Example:

```
char grade = 'B';
switch (grade) {
    case 'A':
        printf("Excellent!\n");
        break;
    case 'B':
        printf("Good!\n");
        break;
    case 'C':
        printf("Average.\n");
        break;
    default:
        printf("Invalid grade.\n");
}
```

Note: `break` prevents ”fall-through” to the next case.

7.2 Looping Statements

7.2.1 (a) `for` Loop

Best for known iteration counts.

```
for (initialization; condition; update) {
    // Code to repeat
}
```

Example:

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

7.2.2 (b) `while` Loop

Repeats while a condition is true.

```
while (condition) {
    // Code to repeat
}
```

Example:

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

7.2.3 (c) `do-while` Loop

Executes at least once before checking the condition.

```
do {
    // Code to repeat
} while (condition);
```

Example:

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

7.3 Jump Statements

7.3.1 (a) `break`

Exits a loop or `switch` immediately.

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

7.3.2 (b) `continue`

Skips the current iteration and continues the loop.

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

7.3.3 (c) `goto`

Jumps to a labeled statement (avoid in modern code).

```
start:
    printf("Hello\n");
    goto end;
    printf("This won't print.\n");
end:
    printf("Done.\n");
```

7.3.4 (d) `return`

Exits a function and optionally returns a value.

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

7.4 Nested Control Flow

You can nest loops and conditions inside each other.

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

7.5 Infinite Loops

Loops that run forever (use `break` to exit).

```
while (1) {
    printf("This runs forever!\n");
    break;  // Exit manually
}
```

7.6 Summary Table

Control Flow Type	**Syntax Example**	**Use Case**
`if`	`if (x > 0) { ... }`	Single condition check
`if-else`	`if (x > 0) { ... } else { ... }`	Binary decision
`if-else if`	`if (x > 0) { ... } else if (x < 0) { ... }`	Multiple conditions
`switch-case`	`switch (x) { case 1: ... }`	Multi-way branching
`for` loop	`for (int i=0; i<5; i++) { ... }`	Known iterations
`while` loop	`while (x < 5) { ... }`	Condition-based loop
`do-while` loop	`do { ... } while (x < 5);`	Loop with at least one execution
`break`	`if (x == 5) break;`	Exit loop/switch
`continue`	`if (x == 3) continue;`	Skip iteration
`goto`	`goto label;`	Jump to label (rarely used)

Table 7: Summary of Control Flow Statements

7.7 Practical Example: Number Guessing Game

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

int main() {
    srand(time(0));
    int secret = rand() % 100 + 1;  // Random number (1-100)
    int guess, attempts = 0;

    printf("Guess the number (1-100):\n");

    do {
        scanf("%d", &guess);
        attempts++;

        if (guess < secret) {
            printf("Too low! Try again.\n");
        } else if (guess > secret) {
            printf("Too high! Try again.\n");
        } else {
            printf("Correct! You took %d attempts.\n", attempts);
        }
    } while (guess != secret);
    return 0;
}
```

7.8 Key Takeaways

- `if-else` → Decision-making.
- `switch-case` → Efficient multi-way branching.
- `for`, `while`, `do-while` → Looping mechanisms.
- `break`, `continue`, `goto` → Control flow redirection.
- **Nested loops** → Handle complex iterations.

8 Functions in C Programming Language

Functions are the building blocks of C programs. They allow you to break code into reusable, modular components, improving readability and maintainability.

8.1 What is a Function?

A **function** is a self-contained block of code that performs a specific task. Key advantages:

- **Reusability** (Avoid code duplication)
- **Modularity** (Divide complex problems into smaller parts)
- **Abstraction** (Hide implementation details)

8.2 Function Syntax

8.2.1 Function Declaration (Prototype)

Tells the compiler about a function's existence before it's defined.

```
return_type function_name(parameters);
```

Example:

```
int add(int a, int b);  // Declaration
```

8.2.2 Function Definition

The actual implementation of the function.

```
return_type function_name(parameters) {
    // Function body
    return value;  // Optional
}
```

Example:

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

8.2.3 Function Call

Executes the function.

```
function_name(arguments);
```

Example:

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

8.3 Types of Functions

8.3.1 (a) Library Functions (Built-in)

Predefined in C headers (e.g., `printf()`, `scanf()`, `sqrt()`). Example:

```
#include <math.h>
double root = sqrt(25.0);  // Returns 5.0
```

8.3.2 (b) User-Defined Functions

Created by the programmer. Example:

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

8.4 Function Parameters & Return Types

Type	Description	Example
With Parameters & Return	Takes input, returns output	`int add(int a, int b) { return a + b; }`
With Parameters, No Return	Takes input, no output	`void printSum(int a, int b) { printf("%d", a + b); }`
No Parameters, With Return	No input, returns output	`int getRandom() { return rand() % 100; }`
No Parameters, No Return	Just executes code	`void sayHello() { printf("Hi!"); }`

8.5 Parameter Passing Methods

8.5.1 (a) Pass by Value

- Copies the value (original variable unchanged).

```
void increment(int x) {
    x++;  // Only modifies local copy
}

int main() {
    int a = 5;
    increment(a);  // 'a' remains 5
}
```

8.5.2 (b) Pass by Reference (Using Pointers)

- Modifies the original variable.

```
void increment(int *x) {
    (*x)++;  // Modifies original
}

int main() {
    int a = 5;
    increment(&a);  // 'a' becomes 6
}
```

8.6 Return Statement

- Exits the function and optionally returns a value. - **Void functions** don't need a `return`.

```
int max(int a, int b) {
    return (a > b) ? a : b;
}
```

8.7 Recursion (Function Calling Itself)

Example: Factorial Calculation

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

int main() {
    printf("5! = %d", factorial(5));  // Output: 120
}
```

8.8 Function Scope Rules

- **Local Variables**: Exist only inside the function. - **Global Variables**: Accessible everywhere (avoid overuse).

```c
int globalVar = 10;  // Global

void func() {
    int localVar = 5;  // Local
    printf("%d", globalVar + localVar);  // 15
}
```

8.9 Storage Classes in Functions

Class	Effect
`auto`	Default (local variables)
`static`	Retains value between calls
`register`	Suggests CPU register storage
`extern`	Links to global variable

Example (`static` variable):

```c
void counter() {
    static int count = 0;  // Remains in memory
    count++;
    printf("%d ", count);
}

int main() {
    counter();  // 1
    counter();  // 2
    counter();  // 3
}
```

8.10 Main Function

Every C program must have a `main()` function (entry point). Two standard forms:

```c
int main() { ... }          // No command-line args
int main(int argc, char *argv[]) { ... }  // With args
```

8.11 Practical Example: Calculator Program

```c
#include <stdio.h>

// Function declarations
float add(float a, float b);
float subtract(float a, float b);

int main() {
    float x = 10.5, y = 5.5;
    printf("Sum: %.2f\n", add(x, y));
    printf("Difference: %.2f\n", subtract(x, y));
    return 0;
}

// Function definitions
float add(float a, float b) {
    return a + b;
}

float subtract(float a, float b) {
    return a - b;
}
```

8.12 Common Mistakes

- Missing return statement (for non-void functions).
- Incorrect parameter types in declaration/definition.
- Unused variables (compiler warnings).
- Infinite recursion (no base case).

8.13 Summary Table

Concept	Example
Declaration	`int sum(int a, int b);`
Definition	`int sum(int a, int b) { return a + b; }`
Call	`int result = sum(2, 3);`
Pass by Value	`void func(int x) { ... }`
Pass by Reference	`void func(int *x) { ... }`
Recursion	`int fact(int n) { return n * fact(n-1); }`

8.14 Key Takeaways

- Functions **organize code** into logical blocks.
- Use **parameters** for input and **return** for output.
- **Pass by reference** modifies original variables.
- **Recursion** solves problems by self-calling.
- `main()` is the **entry point** of every C program.

9 Arrays in C Programming Language

Arrays are one of the most fundamental data structures in C, allowing you to store multiple values of the same type under a single variable name.

9.1 What is an Array?

An **array** is a **fixed-size collection** of elements of the **same data type**, stored in **contiguous memory locations**.

Key Properties

- **Homogeneous**: All elements are of the same type.
- **Fixed Size**: Size must be known at compile time.
- **Zero-Based Indexing**: First element is at index 0.
- **Contiguous Memory**: Elements are stored sequentially.

9.2 Array Declaration & Initialization

(a) Declaration

```
data_type array_name[array_size];
```

Example:

```
int numbers[5];  // Declares an array of 5 integers
```

(b) Initialization

```
data_type array_name[size] = {value1, value2, ..., valueN};
```

Example:

```
int numbers[5] = {10, 20, 30, 40, 50};  // Initialized array
```

Partial Initialization (Unspecified elements are set to 0):

```
int arr[5] = {1, 2};  // [1, 2, 0, 0, 0]
```

No Size Specified (Compiler infers size):

```
int arr[] = {1, 2, 3};  // Size = 3
```

9.3 Accessing Array Elements

- Elements are accessed using **indexing** (`[]`). - Index starts at 0 and goes up to `size - 1`.
Example:

```
int numbers[3] = {10, 20, 30};
printf("%d\n", numbers[0]);  // 10 (first element)
numbers[1] = 25;  // Modify second element
```

Warning: Accessing out-of-bound indices leads to **undefined behavior** (may crash or corrupt memory).

9.4 Types of Arrays

(a) One-Dimensional (1D) Array

A simple list of elements.

```c
int marks[5] = {85, 90, 78, 92, 88};
```

(b) Multi-Dimensional Arrays

2D Arrays (Matrix):

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

Accessing Elements:

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

3D Arrays:

```c
int cube[2][3][4];  // 2 layers, 3 rows, 4 columns
```

9.5 Passing Arrays to Functions

Arrays are **passed by reference** (modifications affect the original array).

(a) Passing 1D Array

```c
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
}

int main() {
    int nums[3] = {1, 2, 3};
    printArray(nums, 3);  // Output: 1 2 3
    return 0;
}
```

(b) Passing 2D Array

```c
void printMatrix(int mat[][3], int rows) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < 3; j++) {
            printf("%d ", mat[i][j]);
        }
        printf("\n");
    }
}

int main() {
    int mat[2][3] = {{1, 2, 3}, {4, 5, 6}};
    printMatrix(mat, 2);
    return 0;
}
```

9.6 Common Array Operations

(a) Traversal (Looping Through)

```
int arr[5] = {10, 20, 30, 40, 50};
for (int i = 0; i < 5; i++) {
    printf("%d ", arr[i]);
}
```

(b) Sum of Elements

```
int sum = 0;
for (int i = 0; i < 5; i++) {
    sum += arr[i];
}
printf("Sum = %d", sum);
```

(c) Finding Maximum Element

```
int max = arr[0];
for (int i = 1; i < 5; i++) {
    if (arr[i] > max) max = arr[i];
}
printf("Max = %d", max);
```

(d) Reversing an Array

```
int temp, start = 0, end = 4;
while (start < end) {
    temp = arr[start];
    arr[start] = arr[end];
    arr[end] = temp;
    start++;
    end--;
}
```

9.7 Relationship Between Arrays & Pointers

- **Array name** is a **pointer to the first element**.
- `arr[i]` is equivalent to `*(arr + i)`.

Example:

```
int arr[3] = {10, 20, 30};
int *ptr = arr;  // Points to arr[0]
printf("%d", *(ptr + 1));  // 20 (same as arr[1])
```

9.8 Dynamic Arrays (Using Pointers)

Since C doesn't support dynamic arrays natively, we use **pointers with** `malloc()`.

Example:

```
#include <stdlib.h>

int main() {
    int size = 5;
    int *dynArr = (int*)malloc(size * sizeof(int));  // Allocate memory

    if (dynArr == NULL) {
        printf("Memory allocation failed!");
        return 1;
    }
    // Use like a normal array
    for (int i = 0; i < size; i++) {
        dynArr[i] = i * 10;
    }

    free(dynArr);  // Release memory
    return 0;
}
```

9.9 Common Mistakes

- **Out-of-Bounds Access**

```
int arr[3] = {1, 2, 3};
printf("%d", arr[5]);  // Undefined behavior!
```

- **Missing Size in Declaration**

```
int arr[];  // Error: Size unknown
```

- **Assuming Array Size with `sizeof`**

```
int arr[5];
int size = sizeof(arr) / sizeof(arr[0]);  // Correct way
```

9.10 Practical Example: Finding Duplicates

```
#include <stdio.h>

int main() {
    int arr[] = {1, 2, 3, 2, 4, 1};
    int size = sizeof(arr) / sizeof(arr[0]);

    printf("Duplicate elements: ");
    for (int i = 0; i < size; i++) {
        for (int j = i + 1; j < size; j++) {
            if (arr[i] == arr[j]) {
                printf("%d ", arr[i]);
                break;
            }
        }
    }
    return 0;
}
```

Output:

```
Duplicate elements: 1 2
```

9.11 Summary Table

Concept	Example
Declaration	`int arr[5];`
Initialization	`int arr[3] = {1, 2, 3};`
Accessing	`arr[0] = 10;`
2D Array	`int mat[2][3] = {{1,2,3}, {4,5,6}};`
Pointer Relation	`int *ptr = arr;`
Dynamic Array	`int *arr = malloc(5 * sizeof(int));`

9.12 Key Takeaways

- Arrays store **multiple values** of the **same type**.
- Indexing starts at 0.
- **Passed by reference** to functions.
- **Dynamic arrays** use `malloc()` and `free()`.
- **Multi-dimensional arrays** represent matrices.

10 Pointers in C Programming

Pointers in C are a fundamental and powerful feature that allow you to directly work with memory addresses. A **pointer** is a variable that stores the *address of another variable*.

10.1 Declaration of Pointers

```
int *ptr;
```

- `int` is the data type of the variable the pointer will point to.
- `*` indicates that `ptr` is a pointer.

10.2 Assigning Address to a Pointer

```
int a = 10;
int *ptr = &a;  // & is the address-of operator
```

10.3 Dereferencing a Pointer

```
printf("%d", *ptr);  // *ptr gives the value stored at the address
```

- `*ptr` accesses the value stored at the address held by `ptr`.

10.4 Pointer Example

```
#include <stdio.h>

int main() {
    int a = 5;
    int *p = &a;

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

    return 0;
}
```

10.5 Pointer Arithmetic

You can perform arithmetic operations on pointers:

```
p++;  // moves to the next memory location of type int
```

10.6 Pointers and Arrays

Arrays and pointers are closely related:

```
int arr[3] = {10, 20, 30};
int *p = arr;

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

10.7 Pointer to Pointer

```
int a = 10;
int *p = &a;
int **pp = &p;

printf("%d", **pp);  // Outputs 10
```

10.8 Use Cases of Pointers

- Dynamic memory allocation
- Function arguments by reference
- Efficient array handling
- Data structures like linked lists, trees, etc.

11 Strings in C Programming Language

In C, strings are **arrays of characters** terminated by a **null character** (`\0`). Unlike higher-level languages, C does not have a built-in string type, so strings are handled as character arrays.

11.1 String Declaration & Initialization

11.1.1 (a) As a Character Array

```
char str[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

or (automatically null-terminated):

```
char str[] = "Hello";  // Size = 6 (includes '\0')
```

11.1.2 (b) Using a Pointer

```
char *str = "Hello";  // Stored in read-only memory (string literal)
```

Warning: Modifying a string literal (`char *str = "Hello";`) leads to **undefined behavior**.

11.2 String Input/Output

11.2.1 (a) Reading a String

- Using `scanf()` (stops at whitespace):

```
char name[20];
scanf("%s", name);  // Input: "John" → name = "John"
```

- Using `fgets()` (safer, reads spaces):

```
char sentence[50];
fgets(sentence, 50, stdin);  // Reads entire line
```

11.2.2 (b) Printing a String

```
printf("%s", str);  // Output: Hello
```

11.3 Common String Functions (`<string.h>`)

Function	**Usage**	**Example**
`strlen()`	Length of string	`strlen("Hello")` → `5`
`strcpy()`	Copy string	`strcpy(dest, src)`
`strcat()`	Concatenate strings	`strcat(str1, str2)`
`strcmp()`	Compare strings	`strcmp("A", "B")` → `<0`
`strstr()`	Find substring	`strstr("Hello", "ell")` → `"ello"`

Example:

```
#include <string.h>

char str1[10] = "Hello";
char str2[10] = "World";

strcat(str1, str2);  // str1 = "HelloWorld"
int len = strlen(str1);  // len = 10
```

11.4 String Manipulation

11.4.1 (a) Copying a String

```
char src[] = "Copy me!";
char dest[20];

strcpy(dest, src);  // dest = "Copy me!"
```

11.4.2 (b) Concatenation

```
char str1[20] = "Hello";
char str2[] = " World";

strcat(str1, str2);  // str1 = "Hello World"
```

11.4.3 (c) Comparing Strings

- `strcmp()` returns:
  - `0` if equal,
  - `<0` if the first string is smaller,
  - `>0` if the first string is larger.

```
if (strcmp(str1, str2) == 0) {
    printf("Strings are equal.");
}
```

11.5 Important Notes

- **Null Termination:** Always ensure strings end with \`0`.

  ```
  char str[5] = "Hello";  //  No space for '\0'
  ```

- **Buffer Overflow:** Avoid writing beyond array bounds.

  ```
  char str[5];
  strcpy(str, "HelloWorld");  //  Buffer overflow!
  ```

- **Dynamic Strings:** Use `malloc()` for resizable strings.

  ```
  char *str = (char*)malloc(10 * sizeof(char));
  strcpy(str, "Dynamic");
  free(str);  // Release memory
  ```

11.6 Practical Example: Reverse a String

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

void reverse(char *str) {
    int len = strlen(str);
    for (int i = 0; i < len / 2; i++) {
        char temp = str[i];
        str[i] = str[len - i - 1];
        str[len - i - 1] = temp;
    }
}

int main() {
    char str[] = "Hello";
```

```
    reverse(str);
    printf("%s", str);  // Output: "olleH"
    return 0;
}
```

11.7 Summary Table

Concept	Example
Declaration	`char str[] = "Hello";`
Null Termination	`str[5] = '';`
Length	`int len = strlen(str);`
Copy	`strcpy(dest, src);`
Concatenate	`strcat(str1, str2);`
Compare	`if (strcmp(s1, s2) == 0)`

11.8 Key Takeaways

- Strings in C are **null-terminated character arrays**.
- Use `<string.h>` for common operations.
- Always ensure **proper memory allocation** to avoid overflows.
- **Dynamic strings** can be created using `malloc()`.

12 Structures and Unions in C Programming Language

Structures (`struct`) and unions (`union`) are **user-defined data types** in C that allow grouping variables of different types under a single name.

12.1 Structures (`struct`)

12.1.1 What is a Structure?

A **structure** is a collection of **related variables** (of different types) under one name.

12.1.2 Syntax

```c
struct structure_name {
    data_type member1;
    data_type member2;
    // ...
};
```

12.1.3 Example: Defining a Structure

```c
struct Student {
    char name[50];
    int age;
    float marks;
};
```

12.1.4 Declaring Structure Variables

```c
// Method 1: During definition
struct Student {
    char name[50];
    int age;
} s1, s2;

// Method 2: After definition
struct Student s3;
```

12.1.5 Accessing Structure Members

Use the **dot (`.`) operator**:

```c
struct Student s1;
strcpy(s1.name, "Alice");
s1.age = 20;
s1.marks = 85.5;
```

12.1.6 Pointer to Structures

Use the **arrow (`->`) operator**:

```c
struct Student *ptr = &s1;
printf("Name: %s", ptr->name);  // Access via pointer
```

12.1.7 Nested Structures

A structure can contain another structure:

```c
struct Address {
    char city[50];
    int pincode;
};

struct Employee {
    char name[50];
    struct Address addr;  // Nested structure
};
```

12.1.8 Example: Using Structures

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

struct Person {
    char name[50];
    int age;
};

int main() {
    struct Person p1;
    strcpy(p1.name, "John");
    p1.age = 25;

    printf("Name: %s\nAge: %d", p1.name, p1.age);
    return 0;
}
```

Output:

```
Name: John
Age: 25
```

12.2 Unions (`union`)

12.2.1 What is a Union?

A **union** is similar to a structure but **shares the same memory location** for all its members. Only **one member** can be active at a time.

12.2.2 Syntax

```c
union union_name {
    data_type member1;
    data_type member2;
    // ...
};
```

12.2.3 Example: Defining a Union

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

12.2.4 Memory Allocation in Unions

- The **size of a union** is equal to its **largest member**.
- All members **share the same memory space**.

12.2.5 Accessing Union Members

```
union Data data;
data.i = 10;        // Stores integer
data.f = 3.14;      // Overwrites integer
strcpy(data.str, "Hello");  // Overwrites float
```

12.2.6 Example: Using Unions

```
#include <stdio.h>

union Number {
    int i;
    float f;
};

int main() {
    union Number num;
    num.i = 10;
    printf("Integer: %d\n", num.i);

    num.f = 3.14;
    printf("Float: %.2f\n", num.f);  // Overwrites 'i'
    return 0;
}
```

Output:

```
Integer: 10
Float: 3.14
```

12.3 Key Differences: `struct` vs `union`

Feature	Structure (`struct`)	Union (`union`)
Memory Allocation	Each member has separate memory	All members share memory
Size	Sum of all members	Size of the largest member
Access	All members can be accessed at once	Only one member active at a time
Use Case	When all members are needed	When only one member is needed

12.4 Practical Applications

12.4.1 (a) Structures

- **Database records** (name, age, salary).
- **Complex data** (coordinates, dates).
- **Linked lists, trees, graphs**.

12.4.2 (b) Unions

- **Memory-efficient storage** (only one field needed).
- **Type conversion** (same memory, different interpretations).
- **Hardware register access**.

12.5 Typedef with Structures/Unions

`typedef` creates an **alias** for a structure/union:

```
typedef struct {
    char name[50];
    int age;
} Person;

Person p1;  // No need to write 'struct' every time
```

12.6 Example: Employee Database

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

typedef struct {
    char name[50];
    int emp_id;
    float salary;
} Employee;

int main() {
    Employee emp1;
    strcpy(emp1.name, "Alice");
    emp1.emp_id = 101;
    emp1.salary = 50000.0;

    printf("Employee: %s\nID: %d\nSalary: %.2f",
           emp1.name, emp1.emp_id, emp1.salary);
    return 0;
}
```

Output:

```
Employee: Alice
ID: 101
Salary: 50000.00
```

12.7 Common Mistakes

- **Forgetting `struct` keyword** (if `typedef` not used):

```
Student s1;  //  Error (unless typedef is used)
struct Student s1;  //  Correct
```

- **Memory overflow in unions**:

```
union Data d;
d.i = 1000;
printf("%s", d.str);  //  Undefined behavior
```

- **Comparing structures directly**:

```
if (s1 == s2) { ... }  //  Not allowed (compare members individually)
```

12.8 Summary Table

Concept	Structure Example	Union Example
Definition	`struct Student { ... };`	`union Data { ... };`
Declaration	`struct Student s1;`	`union Data d1;`
Member Access	`s1.age = 20;`	`d1.i = 10;`
Pointer Access	`ptr->age = 20;`	`ptr->i = 10;`
Size	Sum of all members	Largest member

12.9 Key Takeaways

- **Structures** group different data types under one name.
- **Unions** share memory among members (only one active at a time).
- `typedef` simplifies structure/union usage.
- **Arrow (`->`)** operator is used for pointer access.

13 Dynamic Memory Management in C Programming

Dynamic memory management allows programs to **allocate** and **free** memory at runtime, enabling flexible data structures like linked lists, trees, and dynamic arrays. C provides four key functions for this in `<stdlib.h>`:

13.1 Core Functions

Function	Purpose	Syntax	Example
`malloc()`	Allocates raw memory (uninitialized)	`void* malloc(size_t size);`	`int* arr = (int*)malloc(5 * sizeof(int));`
`calloc()`	Allocates & initializes memory to 0	`void* calloc(size_t num, size_t size);`	`int* arr = (int*)calloc(5, sizeof(int));`
`realloc()`	Resizes previously allocated memory	`void* realloc(void *ptr, size_t new_size);`	`arr = (int*)realloc(arr, 10 * sizeof(int));`
`free()`	Releases allocated memory	`void free(void *ptr);`	`free(arr);`

13.2 Detailed Explanation

13.2.1 (a) `malloc()` (Memory Allocation)

- Allocates **uninitialized** memory.
- Returns `void*` (must be typecast).
- **Fails** if memory is exhausted (returns `NULL`).

```c
int *arr = (int*)malloc(5 * sizeof(int));  // Allocates space for 5 integers
if (arr == NULL) {
    printf("Memory allocation failed!");
    exit(1);
}
```

13.2.2 (b) `calloc()` (Contiguous Allocation)

- Allocates **zero-initialized** memory.
- Takes **number of elements** and **size per element**.

```c
int *arr = (int*)calloc(5, sizeof(int));  // Allocates & initializes to 0
if (arr == NULL) {
    printf("Memory allocation failed!");
    exit(1);
}
```

13.2.3 (c) `realloc()` (Reallocation)

- Resizes existing memory block.
- **Preserves** old data (if new size ¿ old size).
- May **move** the block to a new location.

```c
arr = (int*)realloc(arr, 10 * sizeof(int));  // Expands to 10 integers
if (arr == NULL) {
    printf("Memory reallocation failed!");
    exit(1);
}
```

13.2.4 (d) `free()` (Memory Deallocation)

- Releases memory to prevent **memory leaks**.
- **Always free** dynamically allocated memory.

```c
free(arr);  // Release memory
arr = NULL; // Prevents dangling pointer
```

13.3 Key Concepts

13.3.1 (a) Memory Leaks

Memory leaks occur when allocated memory is **not freed**.

```c
int *leak = (int*)malloc(5 * sizeof(int));
// Forgot to free(leak) → Memory leak!
```

13.3.2 (b) Dangling Pointers

Dangling pointers are pointers that **reference freed memory**.

```c
int *ptr = (int*)malloc(sizeof(int));
free(ptr);
printf("%d", *ptr);  //  Undefined behavior (dangling pointer)
```

13.3.3 (c) Best Practices

- **Check** `NULL` after allocation (`malloc/calloc/realloc`).
- **Free memory** when no longer needed.
- **Avoid dangling pointers** by setting to `NULL` after `free()`.
- **Use** `realloc` carefully (may return a new pointer).

13.4 Dynamic Arrays Example

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

int main() {
    int n;
    printf("Enter array size: ");
    scanf("%d", &n);

    // Allocate memory
    int *arr = (int*)malloc(n * sizeof(int));
    if (arr == NULL) {
        printf("Memory allocation failed!");
        return 1;
    }

    // Input values
    printf("Enter %d integers: ", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    // Print values
    printf("Array elements: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    // Free memory
    free(arr);
    arr = NULL;

    return 0;
}
```

Output:

```
Enter array size: 3
Enter 3 integers: 10 20 30
Array elements: 10 20 30
```

13.5 Dynamic Structures Example

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

typedef struct {
    char name[50];
    int age;
} Person;

int main() {
    Person *p = (Person*)malloc(sizeof(Person));
    if (p == NULL) {
        printf("Memory allocation failed!");
        return 1;
    }

    strcpy(p->name, "Alice");
    p->age = 25;

    printf("Name: %s\nAge: %d", p->name, p->age);

    free(p);
    p = NULL;

    return 0;
}
```

Output:

```
Name: Alice
Age: 25
```

13.6 Common Mistakes

Mistake	Explanation	Fix
No `NULL` check	`malloc` may fail	`if (ptr == NULL) { ... }`
Memory leak	Forgetting `free()`	Always `free(ptr)`
Double free	Calling `free()` twice	Set `ptr = NULL` after freeing
Dangling pointer	Using freed memory	Set `ptr = NULL` after `free()`

13.7 Summary Table

Function	Use Case	Initialized?	Resizable?
`malloc()`	General allocation	No	No
`calloc()`	Array allocation	Yes (zeros)	No
`realloc()`	Resize memory	No	Yes
`free()`	Release memory	-	-

13.8 Key Takeaways

- `malloc`/`calloc` allocate memory at runtime.
- `realloc` adjusts memory size.
- `free` prevents memory leaks.
- Always check for `NULL` after allocation.
- Avoid dangling pointers by setting to `NULL` after `free()`.

14 File Input/Output (I/O) in C Programming

File handling in C allows programs to **read from** and **write to** files on the disk. The `<stdio.h>` library provides functions for file operations.

14.1 File Operations Overview

Operation	Function	Description
Open	`fopen()`	Opens a file
Close	`fclose()`	Closes a file
Read	`fscanf()`, `fgets()`, `fread()`	Reads data from a file
Write	`fprintf()`, `fputs()`, `fwrite()`	Writes data to a file
Positioning	`fseek()`, `ftell()`, `rewind()`	Moves the file pointer

14.2 Opening & Closing Files

14.2.1 (a) `fopen()`

```
FILE *fopen(const char *filename, const char *mode);
```

Modes:

Mode	**Description**
`"r"`	Read (file must exist)
`"w"`	Write (creates/overwrites file)
`"a"`	Append (creates if doesn't exist)
`"r+"`	Read & write (file must exist)
`"w+"`	Read & write (creates/overwrites)
`"a+"`	Read & append (creates if needed)
`"rb"`, `"wb"`, `"ab"`	Binary file modes

Example:

```
FILE *file = fopen("data.txt", "r");
if (file == NULL) {
    printf("Error opening file!");
    exit(1);
}
```

14.2.2 (b) `fclose()`

```
int fclose(FILE *file);
```

Example:

```
fclose(file);  // Always close files!
```

14.3 Reading from Files

14.3.1 (a) `fscanf()` (Formatted Read)

```
int fscanf(FILE *file, const char *format, ...);
```

Example:

```
int num;
fscanf(file, "%d", &num);  // Reads an integer
```

14.3.2 (b) `fgets()` (Line-by-Line Read)

```c
char *fgets(char *str, int n, FILE *file);
```

Example:

```c
char line[100];
fgets(line, 100, file);  // Reads a line (max 99 chars + '\0')
```

14.3.3 (c) `fread()` (Binary Read)

```c
size_t fread(void *ptr, size_t size, size_t count, FILE *file);
```

Example:

```c
int arr[5];
fread(arr, sizeof(int), 5, file);  // Reads 5 integers
```

14.4 Writing to Files

14.4.1 (a) `fprintf()` (Formatted Write)

```c
int fprintf(FILE *file, const char *format, ...);
```

Example:

```c
fprintf(file, "Value: %d\n", 42);  // Writes formatted text
```

14.4.2 (b) `fputs()` (String Write)

```c
int fputs(const char *str, FILE *file);
```

Example:

```c
fputs("Hello, File!\n", file);
```

14.4.3 (c) `fwrite()` (Binary Write)

```c
size_t fwrite(const void *ptr, size_t size, size_t count, FILE *file);
```

Example:

```c
int arr[5] = {1, 2, 3, 4, 5};
fwrite(arr, sizeof(int), 5, file);  // Writes 5 integers
```

14.5 File Positioning

14.5.1 (a) `fseek()` (Move File Pointer)

```c
int fseek(FILE *file, long offset, int origin);
```

Origin Values:

- SEEK_SET → Start of file

- `SEEK_CUR` → Current position
- `SEEK_END` → End of file

Example:

```c
fseek(file, 10, SEEK_SET);  // Moves to 10th byte
```

14.5.2 (b) `ftell()` (Current Position)

```c
long ftell(FILE *file);
```

Example:

```c
long pos = ftell(file);  // Gets current position
```

14.5.3 (c) `rewind()` (Reset Position)

```c
void rewind(FILE *file);
```

Example:

```c
rewind(file);  // Goes back to start
```

14.6 Error Handling

14.6.1 (a) `feof()` (End-of-File Check)

```c
int feof(FILE *file);
```

Example:

```c
while (!feof(file)) {
    // Read until end
}
```

14.6.2 (b) `ferror()` (Error Check)

```c
int ferror(FILE *file);
```

Example:

```c
if (ferror(file)) {
    printf("File error occurred!");
}
```

14.7 Text vs Binary Files

Feature	**Text File**	**Binary File**
Content	Human-readable	Raw bytes
Modes	`"r"`, `"w"`, `"a"`	`"rb"`, `"wb"`, `"ab"`
Usage	Config files, logs	Images, databases

14.8 Practical Examples

14.8.1 (a) Copying a File

```c
#include <stdio.h>

int main() {
    FILE *src = fopen("source.txt", "r");
    FILE *dest = fopen("dest.txt", "w");
    char ch;

    if (src == NULL || dest == NULL) {
        printf("File error!");
        return 1;
    }

    while ((ch = fgetc(src)) != EOF) {
        fputc(ch, dest);
    }

    fclose(src);
    fclose(dest);
    return 0;
}
```

14.8.2 (b) Reading/Writing Structures

```c
#include <stdio.h>

typedef struct {
    char name[50];
    int age;
} Person;

int main() {
    FILE *file = fopen("people.dat", "wb");
    Person p = {"Alice", 25};

    if (file == NULL) {
        printf("File error!");
        return 1;
    }

    fwrite(&p, sizeof(Person), 1, file);
    fclose(file);
    return 0;
}
```

14.9 Common Mistakes

Mistake	Fix
Not checking `fopen()`	Always check `if (file == NULL)`
Forgetting `fclose()`	Leads to memory leaks
Using `feof()` **incorrectly**	Check after read, not before
Binary vs text mode confusion	Use `"rb"`/`"wb"` for binary files

14.10 Summary Table

Task	Function
Open	`fopen()`
Close	`fclose()`
Read	`fscanf()`, `fgets()`, `fread()`
Write	`fprintf()`, `fputs()`, `fwrite()`
Position	`fseek()`, `ftell()`, `rewind()`
Error Check	`feof()`, `ferror()`

14.11 Key Takeaways

- Use `fopen()` with the correct **mode** (`"r"`, `"w"`, `"a"`, etc.).
- Always **check if the file opened successfully**.
- **Close files** with `fclose()` to avoid leaks.
- `fseek()` and `ftell()` help navigate files.
- **Binary files** (`"rb"`, `"wb"`) handle raw data.

15 Preprocessor Directives in C Programming

Preprocessor directives are **special commands** that instruct the C preprocessor (which runs before compilation) to modify the source code. They begin with # and are not terminated with a semicolon.

15.1 What is the Preprocessor?

- Runs **before compilation**.
- Processes #directives.
- Performs **text substitution**, **file inclusion**, and **conditional compilation**.

15.2 Common Preprocessor Directives

Directive	Description	Example
#include	Inserts another file	#include ¡stdio.h¿
#define	Defines a macro	#define PI 3.14159
#undef	Removes a macro	#undef PI
#ifdef / #ifndef	Conditional compilation	#ifdef DEBUG
#if, #elif, #else	Conditional checks	#if VERSION == 2
#pragma	Compiler-specific commands	#pragma once
#error	Forces a compilation error	#error ”Missing config!”

15.3 #include (File Inclusion)

Inserts the contents of another file.

15.3.1 (a) Standard Library Headers

```
#include <stdio.h>  // Searches in system directories
```

15.3.2 (b) User-Defined Headers

```
#include "myheader.h"  // Searches in current directory first
```

15.4 #define (Macros)

15.4.1 (a) Object-like Macros (Constants)

```
#define PI 3.14159
#define MAX 100
```

Usage:

```
double area = PI * radius * radius;
```

15.4.2 (b) Function-like Macros

```
#define SQUARE(x) ((x) * (x))
```

Usage:

```
int result = SQUARE(5);  // Expands to ((5) * (5)) → 25
```

Pitfall: Always use parentheses to avoid unexpected behavior.

```c
#define BAD_SQ(x) x * x
int val = BAD_SQ(2 + 3);  // Expands to 2 + 3 * 2 + 3 → 11 (not 25!)
```

15.5 Conditional Compilation

15.5.1 (a) #ifdef / #ifndef (Check if Defined)

```c
#define DEBUG  // Comment out to disable debug code

#ifdef DEBUG
    printf("Debug mode: x=%d\n", x);
#endif
```

15.5.2 (b) #if, #elif, #else (Conditional Checks)

```c
#define VERSION 2

#if VERSION == 1
    printf("Running v1\n");
#elif VERSION == 2
    printf("Running v2\n");
#else
    printf("Unknown version\n");
#endif
```

15.5.3 (c) #undef (Remove a Macro)

```c
#define TEMP 42
#undef TEMP  // TEMP no longer defined
```

15.6 Predefined Macros

Macro	Description	Example Output
__DATE__	Current date (string)	"May 12 2023"
__TIME__	Current time (string)	"14:30:45"
__FILE__	Current filename (string)	"main.c"
__LINE__	Current line number (int)	42
__STDC__	1 if ANSI C compliant	1

Example:

```c
printf("File: %s, Line: %d\n", __FILE__, __LINE__);
```

15.7 #pragma (Compiler-Specific Directives)

- Used for **compiler-specific** features.
- Common uses:
  - #pragma once → Ensures header is included only once.
  - #pragma pack(1) → Disables struct padding.

```c
#pragma once  // Prevents double-inclusion
```

15.8 #error (Force Compilation Error)

```c
#ifndef VERSION
    #error "VERSION not defined!"
#endif
```

Output:

```
error: #error "VERSION not defined!"
```

15.9 Advanced Macro Techniques

15.9.1 (a) Stringizing (#)

Converts a macro argument to a string.

```c
#define STR(x) #x
printf("%s", STR(Hello));  // Output: "Hello"
```

15.9.2 (b) Token Pasting (##)

Combines two tokens.

```c
#define CONCAT(a, b) a##b
int xy = 10;
printf("%d", CONCAT(x, y));  // Output: 10
```

15.10 Practical Examples

15.10.1 (a) Debugging Macros

```c
#define DEBUG 1

#if DEBUG
    #define LOG(msg) printf("[DEBUG] %s\n", msg)
#else
    #define LOG(msg)
#endif

int main() {
    LOG("Starting program");  // Only prints if DEBUG=1
    return 0;
}
```

15.10.2 (b) Platform-Specific Code

```c
#ifdef _WIN32
    #define OS "Windows"
#elif __linux__
    #define OS "Linux"
#else
    #define OS "Unknown"
#endif

printf("Running on %s\n", OS);
```

15.11 Common Pitfalls

Mistake	Fix
Missing parentheses in macros	#define SQ(x) ((x)*(x))
Forgetting #endif	Always close #if blocks
Overusing macros	Prefer `const`/`inline` where possible
Circular includes	Use #pragma once or include guards

15.12 Summary Table

Directive	Use Case
#include	Insert headers/files
#define	Define constants/macros
#ifdef / #ifndef	Check if macro exists
#if, #elif, #else	Conditional compilation
#pragma	Compiler-specific control
#error	Force compilation error

15.13 Key Takeaways

- Preprocessor runs **before compilation**.
- #include inserts files, #define creates macros.
- **Conditional compilation** (#ifdef, #if) enables platform-specific code.
- **Macros** are powerful but risky (always use parentheses).
- #pragma and #error handle special cases.

16 Standard Library Functions in C Programming

The C Standard Library provides a rich collection of built-in functions for common tasks like I/O, string handling, math operations, memory management, and more. These functions are declared in **header files** (e.g., `<stdio.h>`, `<string.h>`).

16.1 Key Header Files & Their Functions

16.1.1 `<stdio.h>` (Input/Output)

Function	Description	Example
`printf()`	Prints formatted output	`printf("Hello, %d", 42);`
`scanf()`	Reads formatted input	`scanf("%d", &num);`
`fopen()`	Opens a file	`FILE *f = fopen("file.txt", "r");`
`fclose()`	Closes a file	`fclose(f);`
`fgets()`	Reads a line from a file	`fgets(buffer, 100, f);`
`fputs()`	Writes a string to a file	`fputs("Hello", f);`

16.1.2 `<string.h>` (String Manipulation)

Function	Description	Example
`strlen()`	Returns string length	`int len = strlen("Hello");`
`strcpy()`	Copies a string	`strcpy(dest, src);`
`strcat()`	Concatenates strings	`strcat(str1, str2);`
`strcmp()`	Compares two strings	`if (strcmp(s1, s2) == 0) {...}`
`strstr()`	Finds a substring	`char *pos = strstr("Hello", "ell");`

16.1.3 `<stdlib.h>` (Memory & Utilities)

Function	Description	Example
`malloc()`	Allocates memory	`int *arr = malloc(5 * sizeof(int));`
`free()`	Frees memory	`free(arr);`
`atoi()`	Converts string to `int`	`int num = atoi("123");`
`rand()`	Generates a random number	`int r = rand() % 100;`
`exit()`	Terminates the program	`exit(1);`

16.1.4 `<math.h>` (Mathematical Operations)

Function	Description	Example
`sqrt()`	Square root	`double s = sqrt(25.0);`
`pow()`	Power function	`double p = pow(2, 3);`
`sin()`, `cos()`, `tan()`	Trigonometric functions	`double val = sin(3.14);`
`ceil()`, `floor()`	Rounding functions	`double c = ceil(3.2);`

16.1.5 `<ctype.h>` (Character Handling)

Function	Description	Example
`isalpha()`	Checks if a character is alphabetic	`if (isalpha(c)) {...}`
`isdigit()`	Checks if a character is a digit	`if (isdigit(c)) {...}`
`toupper()`	Converts to uppercase	`char uc = toupper('a');`
`tolower()`	Converts to lowercase	`char lc = tolower('A');`

16.1.6 `<time.h>` (Time & Date)

Function	Description	Example
`time()`	Gets current time	`time_t now = time(NULL);`
`ctime()`	Converts time to string	`printf("%s", ctime(&now));`
`difftime()`	Computes time difference	`double sec = difftime(end, start);`

16.2 Practical Examples

16.2.1 Example 1: String Manipulation

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

int main() {
    char str1[20] = "Hello";
    char str2[20] = "World";

    strcat(str1, " ");
    strcat(str1, str2);  // str1 = "Hello World"

    printf("%s (Length: %zu)\n", str1, strlen(str1));
    return 0;
}
```

Output:

```
Hello World (Length: 11)
```

16.2.2 Example 2: Dynamic Memory Allocation

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

int main() {
    int *arr = (int*)malloc(5 * sizeof(int));
    if (arr == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    for (int i = 0; i < 5; i++) {
        arr[i] = i * 10;
    }

    free(arr);
    return 0;
}
```

16.2.3 Example 3: Math Operations

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

int main() {
    double x = 2.0;
    double y = pow(x, 3);  // 8.0
    printf("Cube of %.2f = %.2f\n", x, y);
    return 0;
}
```

Output:

```
Cube of 2.00 = 8.00
```

16.3 Key Takeaways

- `<stdio.h>`: For input/output operations (`printf`, `scanf`, file handling).
- `<string.h>`: For string manipulation (`strcpy`, `strcat`, `strlen`).
- `<stdlib.h>`: For memory management (`malloc`, `free`) and utilities (`atoi`, `rand`).
- `<math.h>`: For mathematical functions (`sqrt`, `pow`, `sin`).

- `<ctype.h>`: For character checks (`isalpha`, `isdigit`).
- `<time.h>`: For time/date operations (`time`, `ctime`).

16.4 Common Pitfalls

Mistake	**Fix**
Forgetting \0 in strings	Ensure strings are null-terminated.
Memory leaks	Always `free()` allocated memory.
Buffer overflow	Use `strncpy()` instead of `strcpy()`.
Ignoring return values	Check if `malloc()` or `fopen()` succeeded.

16.5 Summary Table of Key Functions

Header	**Key Functions**
`<stdio.h>`	`printf`, `scanf`, `fopen`, `fclose`
`<string.h>`	`strlen`, `strcpy`, `strcat`, `strcmp`
`<stdlib.h>`	`malloc`, `free`, `atoi`, `rand`
`<math.h>`	`sqrt`, `pow`, `sin`, `ceil`
`<ctype.h>`	`isalpha`, `isdigit`, `toupper`
`<time.h>`	`time`, `ctime`, `difftime`

16.6 When to Use Which?

- **Need input/output?** → `<stdio.h>`
- **Working with strings?** → `<string.h>`
- **Dynamic memory?** → `<stdlib.h>`
- **Math operations?** → `<math.h>`
- **Character checks?** → `<ctype.h>`
- **Time/date handling?** → `<time.h>`

17 Advanced Concepts in C Programming

Here are some **advanced concepts in C programming** that go beyond the basics and are essential for mastering systems-level programming.

1. Dynamic Memory Allocation

C allows you to allocate memory at runtime using functions from `<stdlib.h>`:

Function	Purpose
`malloc()`	Allocates memory (uninitialized)
`calloc()`	Allocates and initializes to zero
`realloc()`	Reallocates memory
`free()`	Frees allocated memory

```c
int *arr = (int *)malloc(10 * sizeof(int));
if (arr == NULL) {
    // Handle allocation failure
}
free(arr);  // Prevent memory leaks
```

2. Function Pointers

Function pointers allow calling functions dynamically.

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

void (*func_ptr)() = greet;
func_ptr();  // Calls greet
```

Function pointers are useful for callbacks, dynamic dispatch, and plug-in architectures.

3. Structures and Unions

Used to group related variables:

```c
struct Person {
    char name[50];
    int age;
};
```

Unions share memory:

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

4. Bit Manipulation

Useful for efficient, low-level programming:

```c
int a = 5;
a = a << 1;  // Multiply by 2 using left shift
```

Includes: `&`, `|`, `^`, `~`, `<<`, `>>`.

5. Preprocessor Directives

Executed before compilation:

- `#define` – Constants or macros
- `#include` – File inclusion
- `#ifdef`, `#ifndef`, `#endif` – Conditional compilation

```
#define PI 3.14
#ifdef DEBUG
    printf("Debug mode\n");
#endif
```

6. Macros vs. Inline Functions

Macros are simple substitutions:

```
#define SQUARE(x) ((x)*(x))
```

For safety and clarity, prefer **inline functions** (C99 and later).

7. Recursion

Functions that call themselves, e.g., for factorial:

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

8. File Handling

Work with files using standard I/O:

```
FILE *fp = fopen("file.txt", "r");
fgets(buffer, 100, fp);
fclose(fp);
```

9. Memory Management Issues

Common pitfalls:

- Dangling pointers
- Memory leaks
- Buffer overflows

Use tools like **Valgrind** to detect and fix issues.

10. Multithreading (with POSIX Threads)

C uses libraries like `pthread` for multithreading:

```
#include <pthread.h>
pthread_create(&thread_id, NULL, my_function, NULL);
```

11. `volatile` and `const` Keywords

- `const` – Prevents modification
- `volatile` – Prevents compiler optimizations; used in hardware programming or multithreading

18 Error Handling in C Programming

Error handling in C is primarily done through **return values, global variables (`errno`), and custom error-handling mechanisms**. Unlike modern languages with exceptions, C relies on manual checks and propagation of error states.

18.1 Common Error Handling Techniques

18.1.1 Return Values (Most Common Approach)

Functions indicate errors by returning special values (typically `-1`, `NULL`, or a custom error code).

Example: File Handling

```c
#include <stdio.h>

int main() {
    FILE *file = fopen("nonexistent.txt", "r");

    if (file == NULL) {
        perror("Error opening file"); // Prints: "Error opening file: No such file or directory"
        return 1; // Exit with error code
    }

    fclose(file);
    return 0; // Success
}
```

Example: Custom Function with Error Codes

```c
#define ERROR_INVALID_INPUT -1
#define ERROR_OUT_OF_MEMORY -2

int compute(int a, int b) {
    if (a < 0 || b < 0) {
        return ERROR_INVALID_INPUT;
    }
    return a + b;
}

int main() {
    int result = compute(-5, 10);
    if (result == ERROR_INVALID_INPUT) {
        printf("Invalid input!\n");
    }
    return 0;
}
```

18.1.2 Global `errno` Variable (`<errno.h>`)

- Used by standard library functions to indicate errors.
- Must be checked **immediately** after a function fails.

Example: Checking `errno`

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

int main() {
    FILE *file = fopen("missing.txt", "r");
    if (file == NULL) {
        printf("Error %d: %s\n", errno, strerror(errno));
        // Output: "Error 2: No such file or directory"
    }
    return 0;
}
```

Common `errno` Values

Value	Macro	Description
1	`EPERM`	Operation not permitted
2	`ENOENT`	No such file/directory
12	`ENOMEM`	Out of memory
13	`EACCES`	Permission denied

18.1.3 Custom Error Handling with `setjmp`/`longjmp` (`<setjmp.h>`)

- Allows ”jumping” back to a saved program state (like a limited form of exceptions).
- **Use sparingly** (can make code hard to debug).

Example

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

jmp_buf save_state;

void risky_function() {
    if (some_error) {
        longjmp(save_state, 1); // Jump back with error code
    }
}

int main() {
    if (setjmp(save_state) == 0) {
        risky_function(); // First execution
    } else {
        printf("An error occurred!\n");
    }
    return 0;
}
```

18.1.4 Assertions (`<assert.h>`)

- Used for debugging (disabled if `NDEBUG` is defined).
- Crashes the program if a condition is false.

Example

```c
#include <assert.h>

int divide(int a, int b) {
    assert(b != 0); // Crash if b == 0
    return a / b;
}
```

18.2 Best Practices for Error Handling

- Always check return values of functions that can fail (`malloc`, `fopen`, etc.).
- Use `perror()` or `strerror(errno)` for human-readable errors.
- Document error codes in function headers.
- Avoid silent failures—propagate errors to the caller.
- Clean up resources (close files, free memory) before exiting.

18.3 Example: Robust File Copy with Error Handling

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

int main() {
    FILE *src = fopen("source.txt", "r");
    if (!src) {
        perror("Failed to open source file");
        return 1;
    }

    FILE *dest = fopen("dest.txt", "w");
    if (!dest) {
        perror("Failed to open destination file");
        fclose(src); // Clean up
        return 1;
    }

    char buffer[1024];
    while (fgets(buffer, sizeof(buffer), src)) {
        if (fputs(buffer, dest) == EOF) {
            perror("Write error");
            fclose(src);
            fclose(dest);
            return 1;
        }
    }

    if (ferror(src)) {
        perror("Read error");
    }

    fclose(src);
    fclose(dest);
    return 0;
}
```

18.4 Common Pitfalls

Mistake	Fix
Ignoring return values	Always check `if (func() == ERROR)`.
Not resetting `errno`	Set `errno = 0` before calling functions that may set it.
Memory leaks on error	Use `goto` or cleanup sections before returning.
Overusing `assert`	Only for debugging, not production error handling.

18.5 Summary Table

Technique	When to Use	Pros	Cons
Return Values	Most cases	Simple, explicit	Manual propagation
`errno`	Standard library errors	Rich error codes	Not thread-safe by default
`setjmp/longjmp`	Rare cases (e.g., parsers)	Non-local jumps	Hard to debug
Assertions	Debugging	Catches bugs early	Disabled in release builds

18.6 Key Takeaways

- C **lacks exceptions**, so errors must be handled manually.
- `errno` + `perror()` is useful for system errors.
- **Always clean up resources** (files, memory) on failure.
- **Assertions** help catch bugs during development.

19 C Standards in C Programming Language

The C programming language has evolved through several standardized versions, each introducing new features, optimizations, and clarifications. Below is a breakdown of the major C standards:

19.1 K&R C (1978)

- **Original C** defined by Brian Kernighan and Dennis Ritchie in "The C Programming Language" (K&R book).
- **No official standard**, just a de facto reference.
- **Features**:
  - Basic syntax (`if`, `while`, `for`, `switch`).
  - No function prototypes (implicit `int` return type).
  - No `void` or `const` keywords.

Example (K&R-style function):

```
main() {  // Implicit 'int' return
    printf("Hello, K&R C!\n");
}
```

19.2 ANSI C / C89 / C90 (1989-1990)

- First **official standardization** by ANSI (1989) and ISO (1990).
- **Key additions**:
  - Function prototypes (`int func(void);`).
  - `void`, `const`, `volatile` keywords.
  - Standard library (`<stdio.h>`, `<stdlib.h>`, etc.).
  - Preprocessor enhancements (`elif`, `error`).

Example (C89-compliant code):

```
#include <stdio.h>

int main(void) {  // Explicit 'int' and 'void'
    const int x = 10;  // 'const' keyword
    printf("%d\n", x);
    return 0;
}
```

19.3 C99 (1999)

- **Major update** with significant new features.
- **Key additions**:
  - Inline functions (`inline` keyword).
  - Variable-length arrays (VLAs).
  - `//` single-line comments.
  - `long long int` and `bool` (`<stdbool.h>`).
  - Designated initializers (`struct Point p = { .x = 1, .y = 2 };`).

Example (C99 features):

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

inline int square(int x) { return x * x; }  // Inline function

int main() {
    bool flag = true;  // Boolean type
    int arr[5] = { [0] = 1, [4] = 2 };  // Designated initializer
    printf("%d\n", square(5));
    return 0;
}
```

19.4 C11 (2011)

- Focused on **multithreading** and **safety**.
- **Key additions**:
  - Thread support (`<threads.h>`).
  - Atomic operations (`<stdatomic.h>`).
  - Type-generic macros (`_Generic`).
  - Bounds-checking functions (`fopen_s`, `scanf_s`).

Example (C11 multithreading):

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

int run(void *arg) {
    printf("Thread running!\n");
    return 0;
}

int main() {
    thrd_t thread;
    thrd_create(&thread, run, NULL);
    thrd_join(thread, NULL);
    return 0;
}
```

19.5 C17 / C18 (2018)

- **Minor revision** (bug fixes and clarifications).
- No new features; officially called **C17** (ISO) and **C18** (ANSI).
- **Key clarifications**:
  - Removed VLAs (made optional).
  - Fixed undefined behaviors.

19.6 C23 (Upcoming, 2024)

- **Expected features**:
  - `#elifdef` and `#elifndef` (simplified conditional compilation).
  - `constexpr`-like `consteval` functions.
  - Enhanced `nullptr` (similar to C++).
  - Digit separators (`1’000’000`).

Example (C23 digit separators):

```
int million = 1'000'000;  // Improved readability
```

19.7 Comparison of C Standards

Standard	**Year**	**Key Features**
K&R C	1978	Original C (no prototypes, implicit `int`)
ANSI C (C89/C90)	1989-1990	Standardization, `void`, `const`
C99	1999	`//` comments, VLAs, `inline`, `bool`
C11	2011	Threads, atomics, `_Generic`
C17/C18	2018	Bug fixes, removed mandatory VLAs
C23	2024	#`elifdef`, digit separators, `consteval`

19.8 How to Specify a Standard in Compilers?

Most compilers let you choose a standard:

- **GCC/Clang**:
  - `gcc -std=c99` # Compile as C99
  - `gcc -std=c11` # Compile as C11
- **MSVC (Visual Studio)**: Use `/std:c11` in project settings.

19.9 Why Do Standards Matter?

- **Portability**: Code behaves consistently across compilers.
- **Modern Features**: Access to newer language improvements.
- **Safety**: Avoid deprecated/undefined behaviors.

19.10 Which Standard Should You Use?

- **Embedded Systems**: Often stick to **C89/C99** (limited compiler support).
- **Modern Applications**: **C11** or **C17** (better threading and safety).
- **Future Projects**: **C23** (once widely supported).

Key Takeaways

- **K&R C** → Original, pre-standard C.
- **C89/C90** → First standardization (`void`, `const`).
- **C99** → Major update (VLAs, `inline`, `//` comments).
- **C11** → Multithreading and atomics.
- **C17/C18** → Minor fixes (no new features).
- **C23** → Upcoming (digit separators, #`elifdef`).

20 Best Practices in C Programming

Here are some widely accepted **best practices in C programming** to help write clean, efficient, and maintainable code.

20.1 Use Meaningful Variable and Function Names

Descriptive names improve readability.

```c
int total_marks;    // Good
int tm;             // Bad
```

20.2 Keep Code Modular

Break code into small, focused functions. Use header files (`.h`) for declarations, and `.c` files for definitions.

20.3 Avoid Magic Numbers

Use `#define` or `const` for named constants.

```c
#define MAX_STUDENTS 100
```

20.4 Comment and Document Your Code

Explain *why*, not just *what*.

```c
// Use binary search for efficiency on sorted data
```

20.5 Always Initialize Variables

Uninitialized variables may lead to undefined behavior.

```c
int x = 0;  // Always initialize
```

20.6 Check Return Values

Always verify success/failure of critical functions.

```c
FILE *fp = fopen("data.txt", "r");
if (fp == NULL) {
    perror("File open failed");
}
```

20.7 Manage Memory Properly

Every allocation should be paired with a `free` to avoid memory leaks.

20.8 Use `const` Correctly

Helps avoid unintended modifications and aids compiler optimizations.

```c
void printMessage(const char *msg);
```

20.9 Limit Use of Global Variables

Favor local variables to improve modularity and maintainability.

20.10 Prefer Pre-increment (`++i`) Over Post-increment (`i++`)

Pre-increment can be slightly more efficient when return value is not used.

20.11 Handle Errors Gracefully

Don't just exit; use meaningful return values and error messages.

20.12 Use Tools and Warnings

Compile with:

```
-Wall -Wextra
```

Use analysis tools:

- **Valgrind** – memory errors
- **Lint** – code style
- **GDB** – debugging

20.13 Follow a Consistent Coding Style

Use consistent indentation and brace style:

```c
if (x > 0) {
    printf("Positive\n");
}
```

20.14 Use Header Guards in `.h` Files

Prevents double inclusion:

```c
#ifndef MY_HEADER_H
#define MY_HEADER_H
// Declarations
#endif
```

20.15 Learn About Undefined Behavior

C has many undefined behaviors—be cautious with:

- Uninitialized memory
- Invalid pointer access
- Out-of-bounds array access