Overview of C with its History

What Is C Programming?

C is a general-purpose programming language that helps computers follow instructions. Think of it like teaching a robot how to cook, clean, or play music—C tells the computer exactly what to do, step by step.

History of C Programming – Made Simple

1. The Starting Point: ALGOL → BCPL → B

- 1960s – ALGOL: A language used mostly in Europe. It introduced the idea of "structured programming"—like organizing your thoughts before writing an essay.
- 1967 – BCPL: Created by Martin Richards. It was used to write compilers (software that translates code into machine language).
- 1970 – B: Developed by Ken Thompson at Bell Labs. It was simpler than BCPL and used to build early versions of the UNIX operating system.

Analogy: If B was a basic toolkit, C was a full toolbox with screwdrivers, hammers, and measuring tape—ready to build complex machines.

2. Birth of C Language

- 1972 – Dennis Ritchie at Bell Labs improved B and created C.
- He added data types and structures, making C more powerful and flexible.
- C was designed to write operating systems—especially UNIX, which was later rewritten in C.

3. The Famous Book

- 1978 – "The C Programming Language" by Brian Kernighan and Dennis Ritchie.
- This book became the unofficial guide for learning C and is still respected today.
- This version is called K&R C (Kernighan & Ritchie C).

4. Standardization of C

To make sure C worked the same way everywhere, it was standardized:

Year	Version	Key Features
1989	ANSI C (C89)	Standardized syntax and libraries
1990	ISO C (C90)	International version of ANSI C
1999	C99	Added inline functions, variable-length arrays, // comments
2011	C11	Added multithreading, Unicode support

Year	Version	Key Features
2018	C17	Minor updates and bug fixes
2023	C23	Added nullptr, binary literals, modern hardware support

Why Was C Created?

- Computers used assembly language, which was hard to read and not portable.
- C was made to be simple, fast, and portable—so programs could run on different machines.
- It became the backbone of many systems, including UNIX, Windows, and even modern languages like C++, Java, and Python.

Impact of C

- C is like the grandparent of many modern languages.
- It's still used today in:
  - Operating systems
  - Embedded systems (like washing machines or smart TVs)
  - Game development
  - System-level programming

Importance of C

What Makes C So Important?

C is like the foundation of a building—strong, reliable, and essential for everything built on top. It's one of the oldest and most influential programming languages, and learning it gives you deep insight into how computers really work.

Key Reasons Why C Is Important

1. C Is the Mother of Many Languages

- Languages like C++, Java, Python, and JavaScript were inspired by C.
- If you understand C, learning these other languages becomes much easier.
- Analogy: If you learn how to drive a manual car (C), you'll find automatic cars (modern languages) much simpler.

2. Used to Build Operating Systems

- C was used to create the UNIX operating system, and parts of Windows, Linux, and MacOS are still written in C.
- It gives programmers direct access to hardware, which is essential for system-level programming.

3. Fast and Efficient

- C is a middle-level language—it balances low-level control (like assembly) with high-level readability.
- Programs written in C run very fast and use less memory, which is ideal for performance-critical applications.

4. Helps You Understand How Computers Work

- C teaches you about:
  - Memory management (how data is stored and accessed)
  - Pointers (how to directly access memory locations)
  - Data structures (how to organize data efficiently)
- These are the building blocks of computer science.

5. Widely Used in Embedded Systems

- Devices like washing machines, microwaves, smart TVs, and even medical equipment use C.
- It's perfect for programming microcontrollers and hardware-level tasks.

6. Minimal Libraries = Strong Fundamentals

- Unlike modern languages that do a lot for you, C requires you to write more code yourself.
- This helps you build strong programming skills and understand what's happening behind the scenes.

7. Still Relevant Today

- Even after 50+ years, C is still used in:
  - Operating systems
  - Game engines
  - Compilers
  - Databases
  - Network drivers

Summary Table: Why C Matters

Feature	Why It's Important
Low-level access	Direct control over memory and hardware
Speed and efficiency	Ideal for performance-critical applications
Foundation for other languages	Makes learning modern languages easier
System programming	Used in OS kernels and embedded systems
Strong fundamentals	Builds deep understanding of computer science

Structure of C Program

Basic Structure of a C Program

A C program is like a well-organized recipe. Each part has a purpose, and together they help the computer understand what to do. The structure usually includes these six main sections:

1. Documentation Section

- This part contains comments that describe what the program does.
- It's written for humans, not computers. The compiler ignores it.
- Helps others (or your future self) understand the purpose of the code.

Example:

```
// Program to calculate the area of a circle
/* Author: Ash
   Date: 14-09-2025 */
```

2. Pre-processor Section

- This section includes header files using #include.
- Header files contain ready-made functions like printf() for printing output.

Example:

```
#include <stdio.h>   // Standard input-output functions
#include <math.h>    // Math functions like sqrt(), pow()
```

3. Definition Section

- Used to define constants or macros using #define.
- Makes your code cleaner and easier to update.

Example:

```
#define PI 3.14159   // PI is now a constant used throughout the program
```

4. Global Declaration Section

- Variables or functions declared here can be used anywhere in the program.
- These are called global variables or global functions.

Example:

```
int radius;        // Global variable
float area;        // Global variable
```

5. Main Function Section

- This is the heart of the program—where execution starts.
- Every C program must have a main() function.
- Inside it, you write the actual logic: declare variables, take input, perform calculations, and show output.

Example:

```
int main() {
   radius = 5;
   area = PI * radius * radius;
   printf("Area of circle = %.2f", area);
   return 0;
}
```

6. Subprogram Section (User-Defined Functions)

- You can create your own functions to organize code better.
- These functions are called from main() when needed.

Example:

```
float calculateArea(int r) {
   return PI * r * r;
}
```

Summary Table

Section	Purpose
Documentation	Describes the program (comments)
Pre-processor	Includes libraries and macros
Definition	Defines constants and macros
Global Declaration	Declares variables/functions for global use
Main Function	Starts execution and contains core logic
Subprograms	User-defined functions for modular code

Character Set in C

What Is a Character Set in C?

A character set in C is the complete list of symbols, letters, and numbers that the C language understands and allows you to use when writing a program.

Think of it like the alphabet and punctuation you use when writing a sentence in English. In C, the character set includes everything you need to write code—like variable names, numbers, operators, and instructions.

Why Is It Important?

- It defines what characters are valid in a C program.
- Helps the compiler understand and translate your code correctly.
- Without a defined character set, the computer wouldn't know what symbols mean.

Types of Character Sets in C

C uses two main types of character sets:

1. Source Character Set

These are the characters you use while writing the code.

It includes:

a. Alphabets

- Uppercase letters: A to Z
- Lowercase letters: a to z
- Used for naming variables, functions, etc.

b. Digits

- Numbers: 0 to 9
- Used for numeric values and calculations.

c. Special Characters

Symbols used for operations and formatting:

```
+ - * / = < > ! & | ^ % ; , . : ? # ~
( ) [ ] { } " ' \ _
```

d. Whitespace Characters

Used for formatting code:

- Space ()
- Tab (\t)
- Newline (\n)
- Carriage return (\r)

These make the code readable but are ignored during execution.

e. Escape Sequences

Special codes that represent non-printable characters:

- \n → New line
- \t → Tab
- \\ → Backslash
- \' → Single quote
- \" → Double quote

f. Extended Characters

Characters beyond the basic ASCII set, like:

- é, ç, ¥
- Used for international or special symbols

2. Execution Character Set

These are the characters the computer uses when the program is running.

- While the source character set is about writing code,
- The execution character set is about how the code behaves during execution.

Summary Table

Category	Examples	Purpose
Alphabets	A–Z, a–z	Naming variables, functions
Digits	0–9	Numeric values
Special Characters	+, -, *, /, =, {}, (), ;, etc.	Operations and syntax
Whitespace	Space, Tab, Newline	Formatting code
Escape Sequences	\n, \t, \, ", '	Represent special characters
Extended Characters	é, ç, ¥	International/special symbols

Constants & Variables

What Are Variables?

A variable is like a container that holds data. You can change what's inside the container anytime during the program.

Think of it like:

A water bottle labeled “temperature.” You can pour in cold water, then hot water—the label stays the same, but the content changes.

Key Points:

- Variables store values that can change.
- You must declare a variable before using it.
- Each variable has a name, a type, and a value.

Example:

```
int age = 25;     // 'age' is a variable of type int with value 25
age = 30;         // Now the value of 'age' is changed to 30
```

What Are Constants?

A constant is like a locked container. Once you put something inside, you can't change it.

Think of it like:

A sealed jar labeled "PI" with the value 3.14. You can use it, but you can't open it and change the value.

Key Points:

- Constants store fixed values that do not change during the program.
- You can define constants using:
  - const keyword
  - #define preprocessor directive

Example using const:

```
const float PI = 3.14;   // PI is a constant and cannot be changed
```

Example using #define:

```
#define MAX_SIZE 100     // MAX_SIZE is a constant with value 100
```

Differences Between Variables and Constants

Feature	Variable	Constant
Value	Can change during program execution	Fixed and cannot be changed
Declaration	Using data type (e.g., int, float)	Using const or #define
Purpose	Store changing data	Store fixed reference values
Example	int score = 50;	const int maxScore = 100;

Why Are They Important?

- Variables help you store and update data like scores, age, temperature, etc.

- Constants help you define fixed values like PI, tax rates, or limits that should not be changed accidentally.

Identifiers & Keywords in C

What Are Keywords in C?

Keywords are special words that have a predefined meaning in the C language. You cannot use them for anything else—like naming variables or functions—because they are reserved by the language itself.

Think of it like:

Keywords are like traffic signs. A "STOP" sign always means stop—you can't change its meaning or use it as a street name.

Key Points:

- Keywords are built-in commands in C.
- They are always written in lowercase.
- You cannot use them as variable or function names.

Examples of Keywords:

- int – used to declare an integer variable
- return – used to return a value from a function
- if, else – used for decision-making
- for, while – used for loops
- float, char, void, break, continue, switch, case, struct, typedef, etc.

There are 32 keywords in standard C.

What Are Identifiers in C?

Identifiers are the names you give to things in your program—like variables, functions, arrays, etc.

Think of it like:

Identifiers are like labels you put on jars in your kitchen. You decide what to name them—"Sugar", "Salt", "Spices"—so you can find and use them easily.

Key Points:

- Identifiers are user-defined names.
- They help you identify and refer to variables, functions, etc.
- You can choose any name, but it must follow certain rules.

Rules for Naming Identifiers:

1. Can include letters (A–Z, a–z), digits (0–9), and underscores (_).
2. Must start with a letter or underscore, not a digit.
3. Cannot use keywords as identifiers.
4. Should be meaningful to make your code easy to understand.

Examples of Identifiers:

- total, count, main, calculateArea, student_name, x1, speed_limit

Code Example to Show Both

```
#include <stdio.h>

int main() {              // 'main' is an identifier
   int age = 25;          // 'int' is a keyword, 'age' is an identifier
   printf("Age: %d", age); // 'printf' is a keyword/function, 'age' is used again
   return 0;              // 'return' is a keyword
}
```

Summary Table

Feature	Keywords	Identifiers
Meaning	Predefined, reserved words	User-defined names
Purpose	Tell the compiler what to do	Label variables, functions, etc.
Examples	int, return, if, while	score, total, main, student
Can be changed	No	Yes
Can be reused	No	Yes

Data Type in C

What Are Data Types in C?

A data type tells the computer what kind of data a variable will store and how much memory it needs.

Think of it like:

Choosing the right container for your items—use a bottle for water, a box for books, and a wallet for money. In C, you choose the right data type to store numbers, letters, or decimal values.

Why Are Data Types Important?

- They help the computer understand the kind of data you're working with.
- They save memory by using just the right amount of space.

- They prevent errors by making sure you don't mix incompatible types.

Types of Data Types in C

C has three main categories of data types:

1. Primary (Basic) Data Types

These are the most commonly used types.

Data Type	What It Stores	Example	Size	Format Specifier
int	Whole numbers (no decimals)	int age = 25;	4 bytes	%d
float	Decimal numbers	float price = 99.99;	4 bytes	%f
char	Single characters	char grade = 'A';	1 byte	%c
double	Large decimal numbers	double pi = 3.141592;	8 bytes	%lf
void	No value (used for functions)	void main()	0 bytes	—

2. Derived Data Types

These are built from basic types.

Type	Description	Example
array	Group of values of same type	int marks[5];
pointer	Stores memory address of a variable	int *p;
function	Block of code that performs a task	int add(int a, int b)

3. User-Defined Data Types

Created by the programmer to group different types together.

Type	Description	Example
struct	Combines different data types	struct Student { int id; char name[20]; };
union	Shares memory between members	union Data { int i; float f; };
enum	Defines named integer constants	enum Days { Mon, Tue, Wed };
typedef	Creates a new name for a data type	typedef int Number;

C Data Types with Size and Value Range

Data Type	Size (Bytes)	Value Range	Format Specifier
char	1	-128 to 127 (signed) or 0 to 255 (unsigned)	%c

Data Type	Size (Bytes)	Value Range	Format Specifier
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
unsigned short int	2	0 to 65,535	%hu
long int	8	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	%ld
unsigned long int	8	0 to 18,446,744,073,709,551,615	%lu
float	4	~1.2E-38 to ~3.4E+38 (approximate)	%f
double	8	~2.3E-308 to ~1.7E+308 (approximate)	%lf
long double	12–16	~3.4E-4932 to ~1.1E+4932 (approximate)	%Lf

Note: Actual size and range may vary slightly depending on the compiler and system architecture.

Real-Life Analogy

Real-Life Item	C Data Type	Reason
Age	int	Whole number
Price tag	float	Decimal value
Grade (A/B/C)	char	Single character
Pi value	double	Precise decimal
Empty box	void	No value

Summary

- Data types define what kind of data a variable can hold.
- They help with memory management, error prevention, and code clarity.
- Choosing the right data type is like choosing the right tool for a job.

Assignment Statement in C

What Is an Assignment Statement?

An assignment statement in C is used to store a value into a variable. It tells the computer:

"Put this value into this container (variable)."

Think of it like:

You have a jar labeled "age." If you write age = 25;, you're putting the number 25 inside the jar called "age."

Basic Syntax

variable_name = value;

- Left side: the variable (container)
- Right side: the value, another variable, or an expression (what you want to store)

Example:

```
int age;
age = 25;
```

Here, age is the variable, and 25 is the value being assigned.

Key Rules to Remember

1. Variable must be declared first:
2. int score; // declare
3. score = 100; // assign
4. The value must match the variable's data type:
    - You can't assign a decimal to an int without conversion.
    - Example:
    - int x = 5.5; // Incorrect
    - float x = 5.5; // Correct
5. Assignment overwrites previous value:
6. int marks = 80;
7. marks = 90; // Now marks is 90, not 80
8. Every statement ends with a semicolon (;).

Types of Assignment Statements

1. Simple Assignment

Assigns a direct value:

int a = 10;

2. Assignment from Another Variable

int x = 20;
int y = x; // y gets the value of x

3. Assignment from an Expression

int total = 5 + 3; // total becomes 8

4. Compound Assignment

Combines math and assignment:

int a = 5;
a += 3; // same as a = a + 3 → a becomes 8

Operator	Meaning	Example	Result
+=	Add and assign	a += 2;	a = a + 2
-=	Subtract and assign	a -= 2;	a = a - 2
*=	Multiply and assign	a *= 2;	a = a * 2
/=	Divide and assign	a /= 2;	a = a / 2
%=	Modulus and assign	a %= 2;	a = a % 2

Summary

Feature	Description
Purpose	To store a value in a variable
Symbol Used	= (assignment operator)
Must Match Types	Value must match the variable's data type
Can Use Expressions	You can assign results of calculations
Can Be Compound	Combine math and assignment for shorter code

Symbolic Constant

What Is a Symbolic Constant?

A symbolic constant is a name that represents a fixed value in your program. Instead of writing the value again and again, you give it a name and use that name wherever needed.

Think of it like: You write "PI" instead of typing 3.14159 every time. It's like using a shortcut label for a value that never changes.

Why Use Symbolic Constants?

- Makes your code easier to read
  (e.g., MAX_SPEED is clearer than 120)
- Makes your code easier to update
  (change the value in one place instead of many)
- Prevents accidental changes to important values

How to Define a Symbolic Constant in C

You use the #define directive, which is part of the pre-processor. It replaces the name with the value before the program is compiled.

Syntax:

#define CONSTANT_NAME value

Example:

#define PI 3.14159
#define MAX_STUDENTS 100
#define ERROR_MESSAGE "Invalid input"

Now, whenever you use PI, MAX_STUDENTS, or ERROR_MESSAGE in your code, the compiler will automatically replace them with their actual values.

Real-Life Analogy

Symbolic Constant	Real-Life Meaning
PI	Like using "π" instead of writing 3.14159 every time
MAX_SPEED	Like setting a speed limit sign on a road
ERROR_MESSAGE	Like printing the same warning label on every package

Rules for Symbolic Constants

1. Use uppercase letters by convention (e.g., MAX_VALUE)
2. No semicolon (;) at the end of #define
3. Place them at the top of your program
4. You cannot change their value later in the program

Example Program

```
#include <stdio.h>
#define PI 3.14159

int main() {
   float radius = 5.0;
```

```
    float area = PI * radius * radius;
    printf("Area of circle = %.2f", area);
    return 0;
}
```

Output:

Area of circle = 78.54

Summary Table

Feature	Description
Purpose	To give a name to a fixed value
Defined Using	#define directive
Value Can Change?	No – it stays constant
Used For	Readability, maintainability, safety
Example	#define PI 3.14159

What Is Input and Output in C?

- Input means giving data to the program (like typing your name or age).
- Output means showing results to the user (like printing your name or a message).

What Are Formatted I/O Functions?

Formatted I/O functions allow you to control how data is entered or displayed. You can choose the format—like showing numbers with two decimal places, printing text neatly, or reading specific types of data.

These functions are part of the stdio.h header file.

Common Formatted I/O Functions

1. printf() – For Output

Used to display data on the screen.

Syntax:

printf("format string", values);

Example:

```
int age = 25;
printf("Age is %d", age);  // %d is for integer
```

Format Specifiers:

Specifier	Data Type	Example Output
%d	Integer	25
%f	Float	3.14
%.2f	Float (2 decimals)	3.14
%c	Character	A
%s	String	Hello

2. scanf() – For Input

Used to take input from the user.

Syntax:

scanf("format string", &variables);

Note: Use & (address-of operator) to store the value in the variable.

Example:

```
int age;
scanf("%d", &age);  // Reads an integer from user
```

3. sprintf() – Print to String

Stores formatted output inside a string instead of printing it.

Example:

```
char message[50];
int x = 5;
sprintf(message, "Value is %d", x);  // message now holds "Value is 5"
```

4. sscanf() – Read from String

Reads formatted input from a string instead of the keyboard.

Example:

```
char data[] = "25";
int age;
sscanf(data, "%d", &age);  // age becomes 25
```

Real-Life Analogy

Function	Real-Life Example
printf()	Speaking to someone (giving output)
scanf()	Listening to someone (taking input)
sprintf()	Writing a message on paper
sscanf()	Reading a message from paper

Summary Table

Function	Purpose	Input/Output	Uses Format Specifiers
printf()	Display data	Output	Yes
scanf()	Take user input	Input	Yes
sprintf()	Store output in a string	Output	Yes
sscanf()	Read input from a string	Input	Yes

1. scanf() – Formatted Input Function

Definition:

scanf() is used to take formatted input from the user, like numbers, characters, or strings. You must tell it what type of data to expect using format specifiers like %d, %f, %c, %s.

Syntax:

scanf("format", &variable);

Note: Use & before variable names to store the input value.

Example:

```
int age;
scanf("%d", &age);  // Reads an integer and stores it in 'age'
```

2. getch() – Unformatted, Hidden Input

Definition:

getch() reads a single character from the keyboard without showing it on the screen. It doesn't wait for the Enter key.

Used when you want to take secret input like a password.

Header File:

#include <conio.h>

Example:

```
char ch;
ch = getch();  // Reads a character silently
```

3. getche() – Unformatted, Visible Input

Definition:

getche() reads a single character from the keyboard and displays it immediately on the screen. Like getch(), it doesn't wait for Enter.

Used when you want instant feedback while typing.

Header File:

#include <conio.h>

Example:

```
char ch;
ch = getche();  // Reads and displays the character
```

4. getchar() – Buffered Character Input

Definition:

getchar() reads a single character from the keyboard, but it waits for Enter before reading. It's a buffered input function.

Header File:

#include <stdio.h>

Example:

```
char ch;
ch = getchar();  // Reads one character after Enter is pressed
```

5. gets() – String Input Function

Definition:

gets() reads a whole line of text (string) from the user until Enter is pressed. It stores the input in a character array.

Note: gets() is unsafe and not recommended in modern C because it doesn't check buffer size. Use fgets() instead for safety.

Header File:

#include <stdio.h>

Example:

```
char name[50];
gets(name);  // Reads a full line of text into 'name'
```

Summary Table

Function	Reads What?	Shows Input?	Waits for Enter?	Use Case
scanf()	Formatted data	Yes	Yes	Numbers, characters, strings
getch()	One character	No	No	Hidden input (e.g., password)
getche()	One character	Yes	No	Instant visible input
getchar()	One character	Yes	Yes	Simple character input
gets()	Full line (string)	Yes	Yes	Reading sentences or names

Output Functions

1. printf() – Formatted Output Function

Definition:

printf() is used to display formatted output on the screen. You can print text, numbers, characters, and even results of calculations using format specifiers like %d, %f, %c, %s.

Syntax:

```
printf("format string", values);
```

Example:

```
int age = 25;
printf("Age is %d", age);  // Output: Age is 25
```

Format Specifiers:

Specifier	Meaning
%d	Integer

Specifier	Meaning
%f	Float/Decimal
%c	Character
%s	String

2. putch() – Character Output (Unformatted)

Definition:

putch() prints a single character to the screen. It's simple and direct, but not part of the standard C library—it's available in Turbo C or DOS-based compilers.

Not recommended for modern C programming.

Syntax:

putch(character);

Example:

putch('A'); // Output: A

3. putchar() – Standard Character Output

Definition:

putchar() is used to print one character at a time. It's part of the standard C library and is safer and more portable than putch().

Syntax:

putchar(character);

Example:

```
char ch = 'B';
putchar(ch);  // Output: B
```

You can also use it in loops to print multiple characters:

```
for (char ch = 'A'; ch <= 'Z'; ch++) {
    putchar(ch);
    putchar(' ');  // Adds space between letters
}
```

4. puts() – String Output Function

Definition:

puts() prints a whole string (text) to the screen and automatically moves to the next line after printing.

Syntax:

puts("Your message here");

Example:

```
char name[] = "Ash";
puts(name);  // Output: Ash (followed by a new line)
```

Unlike printf(), puts() does not support format specifiers like %d or %f.

Summary Table

Function	What It Prints	Supports Formatting	Adds New Line	Example Use
printf()	Text, numbers, etc	Yes (%d, %f, etc)	No	printf("Age: %d", age);
putch()	One character	No	No	putch('A');
putchar()	One character	No	No	putchar('B');
puts()	Whole string	No	Yes	puts("Hello");

What Are Operators and Expressions?

- An operator is a symbol that tells the computer to do a specific math operation (like + or -).
- An expression is a combination of variables, constants, and operators that produces a result.

Think of it like:

If you write 5 + 3, the + is the operator, and the whole thing is an expression that gives the result 8.

What Are Arithmetic Operators?

Arithmetic operators are used to perform basic math operations like addition, subtraction, multiplication, division, and finding remainders.

List of Arithmetic Operators:

Operator	Meaning	Example	Result
+	Addition	5 + 3	8
-	Subtraction	10 - 4	6
*	Multiplication	6 * 2	12
/	Division (quotient)	8 / 2	4
%	Modulus (remainder)	9 % 4	1

Note: % only works with integers, not with decimal numbers.

Types of Arithmetic Operators

1. Binary Operators

These work with two values (operands).

Example:

```
int a = 10, b = 3;
int sum = a + b;       // sum = 13
int product = a * b;   // product = 30
int quotient = a / b;  // quotient = 3
int remainder = a % b; // remainder = 1
```

2. Unary Operators

These work with one value.

Operator	Meaning	Example	Result
++	Increment (add 1)	a++	a = a + 1
--	Decrement (subtract 1)	a--	a = a - 1
+	Unary plus (no change)	+a	a
-	Unary minus (negation)	-a	-a

Example:

```
int a = 5;
a++;  // a becomes 6
--a;  // a becomes 5 again
```

What Is an Arithmetic Expression?

An arithmetic expression is a combination of numbers, variables, and arithmetic operators that gives a result.

Examples:

```
int result = 5 + 3 * 2;   // result = 11 (because * has higher priority)
float average = (a + b) / 2.0;
```

Operator Precedence (Which Comes First?)

When multiple operators are used, C follows priority rules:

Priority	Operators	Description
1	*, /, %	Multiplication, division, modulus
2	+, -	Addition, subtraction

Example:

```
int result = 5 + 3 * 2;  // result = 11, not 16
```

Because 3 * 2 is done first, then added to 5.

Summary Table

Concept	Description
Operator	Symbol that performs a math operation
Expression	Combination of values and operators
Binary Operators	Work with two values (+, -, *, /, %)
Unary Operators	Work with one value (++, --, +, -)
Precedence	Multiplication/division before addition/subtraction

What Are Relational Operators?

Relational operators are symbols used to compare two values. They help the computer decide whether one value is greater than, less than, equal to, or not equal to another.

Think of it like:

Comparing two exam scores to see who scored higher, or checking if someone's age is equal to 18.

Why Are They Important?

Relational operators are used in:

- Decision-making (if, else)
- Loops (while, for)
- Conditions that control how the program behaves

They always return a result:

- 1 means true
- 0 means false

List of Relational Operators in C

Operator	Meaning	Example	Result
==	Equal to	5 == 5	1 (true)
!=	Not equal to	5 != 3	1 (true)
>	Greater than	7 > 4	1 (true)
<	Less than	3 < 6	1 (true)
>=	Greater than or equal to	5 >= 5	1 (true)
<=	Less than or equal to	4 <= 5	1 (true)

Example Program

```
#include <stdio.h>

int main() {
   int a = 10, b = 5;

   printf("a == b: %d\n", a == b);   // false → 0
   printf("a != b: %d\n", a != b);   // true → 1
   printf("a > b: %d\n", a > b);     // true → 1
   printf("a < b: %d\n", a < b);     // false → 0
   printf("a >= b: %d\n", a >= b);   // true → 1
```

```
    printf("a <= b: %d\n", a <= b);   // false → 0

    return 0;
}
```

Real-Life Analogy

Situation	C Expression	Meaning
Is your age equal to 18?	age == 18	Checks if age is exactly 18
Is your score higher than 50?	score > 50	Checks if score is above 50
Is your password incorrect?	input != saved	Checks if input is not equal

Summary Table

Feature	Description
Purpose	To compare two values
Returns	1 for true, 0 for false
Used In	Conditions, loops, decision-making
Common Operators	==, !=, >, <, >=, <=

What Are Logical Operators?

Logical operators are symbols used to combine or reverse conditions in a program. They help the computer make decisions based on true or false values.

Think of it like:

You're checking if both your phone is charged and you have internet. If both are true, you can watch YouTube. Logical operators help check such combined conditions.

Why Are Logical Operators Important?

- Used in decision-making (if, while, for statements)
- Help check multiple conditions at once
- Return either:
  - 1 → True
  - 0 → False

Types of Logical Operators in C

C has three main logical operators:

1. && – Logical AND

Definition:

Returns true only if both conditions are true.

Syntax:

condition1 && condition2

Example:

```
int a = 5, b = 10;
if (a > 0 && b > 0) {
   printf("Both numbers are positive");
}
```

Truth Table:

a	b	a && b		a	b	a && b
false	false	false	→	0	0	0
false	true	false		0	1	0
true	false	false		1	0	0
true	true	true		1	1	1

2. || – Logical OR

Definition:

Returns true if at least one condition is true.

Syntax:

condition1 || condition2

Example:

```
int a = -1, b = 10;
if (a > 0 || b > 0) {
   printf("At least one number is positive");
}
```

Truth Table:

a	b	a || b		a	b	a || b
false	false	false	→	0	0	0
false	true	false		0	1	1
true	false	false		1	0	1
true	true	true		1	1	1

3. ! – Logical NOT

Definition:

Reverses the result of a condition:

- If true → becomes false
- If false → becomes true

Syntax:

!condition

Example:

```
int a = 0;
if (!a) {
   printf("a is zero");
}
```

Truth Table:

a	!a		a	!a
true	false	→	1	0
false	true		0	1

Real-Life Analogy

Situation	C Expression	Meaning
Both lights are on	light1 && light2	True only if both are on
Either door is open	door1 || door2	True if anyone is open
Not raining	! raining	True if it's not raining

Summary Table

Operator	Name	Meaning
&&	Logical AND	True if both conditions are true
||	Logical OR	True if at least one condition is true
!	Logical NOT	Reverses the truth value

What Are Bitwise Operators?

Bitwise operators work directly on the binary digits (bits) of numbers. Every number in a computer is stored as a series of 0s and 1s. Bitwise operators let you manipulate these bits to perform fast and low-level operations.

Think of it like:

If numbers are made of switches (ON = 1, OFF = 0), bitwise operators let you flip, combine, or shift those switches.

Why Are Bitwise Operators Important?

- They are used in system programming, embedded systems, and hardware control.
- Help in performance optimization and memory-efficient coding.
- Useful for tasks like encryption, graphics, and device drivers.

Common Bitwise Operators in C

Operator	Name	Description	Example
&	Bitwise AND	Returns 1 only if both bits are 1	5 & 3 = 1
|	Bitwise OR	Returns 1 if at least one bit is 1	5 | 3 = 7
^	Bitwise XOR	Returns 1 if bits are different	5 ^ 3 = 6
~	Bitwise NOT	Flips all bits (1 becomes 0, 0 becomes 1)	~5 = -6
<<	Left Shift	Shifts bits to the left (adds zeros on the right)	5 << 1 = 10
>>	Right Shift	Shifts bits to the right (removes bits from the right)	5 >> 1 = 2

How It Works (Using Binary)

Let's take two numbers:

- a = 5 → Binary: 0101
- b = 3 → Binary: 0011

Bitwise AND (a & b)

```
  0101
& 0011
= 0001 → Decimal: 1
```

Bitwise OR (a | b)

```
  0101
| 0011
= 0111 → Decimal: 7
```

Bitwise XOR (a ^ b)

```
  0101
^ 0011
= 0110 → Decimal: 6
```

Bitwise NOT (~a)

~0101 = 1010 (in 2's complement, becomes -6)

Left Shift (a << 1)

0101 becomes 1010 → Decimal: 10

Right Shift (a >> 1)

0101 becomes 0010 → Decimal: 2

Example Program

```
#include <stdio.h>

int main() {
    int a = 5, b = 3;

    printf("a & b = %d\n", a & b);
    printf("a | b = %d\n", a | b);
    printf("a ^ b = %d\n", a ^ b);
    printf("~a = %d\n", ~a);
    printf("a << 1 = %d\n", a << 1);
    printf("a >> 1 = %d\n", a >> 1);

    return 0;
}
```

Unary Operators in C

What Are Unary Operators?

Unary operators work with only one operand (one value). They perform operations like increasing, decreasing, flipping signs, or checking memory addresses.

Simple Analogy:

You press a button to increase the volume by 1. That button works on just one thing—your current volume level.

Common Unary Operators

Operator	Name	What It Does	Example	Result
+	Unary Plus	Returns the value as-is	+5	5
-	Unary Minus	Changes the sign	-5	-5
++	Increment	Adds 1 to the value	a++ or ++a	a = a + 1
--	Decrement	Subtracts 1 from the value	a-- or --a	a = a - 1
!	Logical NOT	Reverses true/false	!1	0
~	Bitwise NOT	Flips all bits (advanced use)	~5	-6
&	Address-of	Gets memory address of a variable	&a	Address
*	Pointer (Indirection)	Accesses value at a memory address	*p	Value at address
sizeof	Size Checker	Returns size of a data type or variable	sizeof(int)	4 (bytes)

Example:

```
int a = 5;
printf("a++ = %d\n", a++);  // Post-increment: prints 5, then a becomes 6
printf("++a = %d\n", ++a);  // Pre-increment: a becomes 7, then prints 7
```

Assignment Operators in C

What Are Assignment Operators?

Assignment operators are used to store a value into a variable. They assign values directly or through calculations.

Simple Analogy:

You write a number on a sticky note labeled “score.” That’s assigning a value to a variable.

Types of Assignment Operators

Operator	Meaning	Example	Same As
=	Simple assignment	a = 5	—
+=	Add and assign	a += 3	a = a + 3
-=	Subtract and assign	a -= 2	a = a - 2
*=	Multiply and assign	a *= 4	a = a * 4
/=	Divide and assign	a /= 2	a = a / 2
%=	Modulus and assign	a %= 3	a = a % 3
<<=	Left shift and assign	a <<= 1	a = a << 1
>>=	Right shift and assign	a >>= 1	a = a >> 1

Operator	Meaning	Example	Same As
&=	Bitwise AND and assign	a &= 2	a = a & 2
|=	Bitwise OR and assign	a |= 2	a = a | 2
^=	Bitwise XOR and assign	a ^= 2	a = a ^ 2

Example:

```
int a = 10;
a += 5;   // a becomes 15
a *= 2;   // a becomes 30
```

Summary Table

Concept	Unary Operators	Assignment Operators
Works On	One value	Two values (variable and value)
Purpose	Modify or inspect a single value	Store or update a value in a variable
Common Symbols	++, --, -, +, !, ~, &	=, +=, -=, *=, /=, %=
Use Case	Changing value, checking size	Assigning or updating values

Conditional Operator in C

What Is a Conditional Operator?

The conditional operator is also called the ternary operator because it works with three parts. It's a shortcut for writing simple if-else statements.

Syntax:

```
condition ? value_if_true : value_if_false;
```

Think of it like:

You ask: "Is it raining?" If yes, take an umbrella. If no, wear sunglasses.

Example:

```
int age = 18;
char *status = (age >= 18) ? "Adult" : "Minor";
printf("%s", status);  // Output: Adult
```

How It Works:

- If age >= 18 is true → status = "Adult"
- If false → status = "Minor"

Summary:

Part	Meaning
condition	What you're checking
?	Separates condition from result
:	Separates true and false values

Special Operators in C

Special operators perform unique tasks that don't fit into arithmetic or logical categories. Here are the most important ones:

1. sizeof Operator

Definition:

Returns the size (in bytes) of a data type or variable.

Example:

```
int a;
printf("%lu", sizeof(a));  // Output: 4 (on most systems)
```

Use Case:

Helps you understand how much memory a variable uses.

2. Comma , Operator

Definition:

Allows you to combine multiple expressions in one line. The last expression is the result.

Example:

```
int x;
x = (5, 10, 15);  // x becomes 15
```

Use Case:

Useful in loops or compact code.

3. Pointer Operators: & and *

Operator	Name	What It Does	Example
&	Address-of	Gives the memory address of a variable	&x
*	Dereference	Accesses the value at a memory address	*p

Example:

```
int x = 10;
int *p = &x;      // p stores address of x
printf("%d", *p);  // Output: 10 (value at address)
```

4. Member Access Operators: . and ->

Used with structures to access members.

Operator	Use Case	Example
.	Access member using variable	student.age
->	Access member using pointer	ptr->age

Example:

```
struct Student {
   int age;
};
struct Student s1;
s1.age = 20;
printf("%d", s1.age);  // Output: 20
```

Summary Table

Operator	Type	Purpose
? :	Conditional	Short form of if-else
sizeof	Special	Finds memory size of data type
,	Special	Combines multiple expressions
&	Special	Gets memory address
*	Special	Accesses value at memory address
. / ->	Special	Access structure members

What Is an Arithmetic Expression?

An arithmetic expression is a combination of:

- Variables (like a, b)
- Constants (like 5, 10)
- Arithmetic operators (like +, -, *, /, %)

It performs a mathematical calculation and gives a result.

Example:

```
int result = a + b * c;
```

This expression adds a to the product of b and c.

What Is Operator Hierarchy?

Operator hierarchy (also called precedence) tells the compiler which operation to do first when there are multiple operators in an expression.

Think of it like:

Solving a math problem: you multiply before you add. For example, in 2 + 3 * 4, you do 3 * 4 first, then add 2.

Why Is It Important?

Without operator hierarchy, the computer wouldn't know which part of the expression to solve first. This could lead to wrong results.

Operator Precedence Table (Highest to Lowest)

Priority	Operators	Description	Associativity
1	()	Parentheses	Left to Right
2	*, /, %	Multiplication, Division, Modulus	Left to Right
3	+, -	Addition, Subtraction	Left to Right
4	=	Assignment	Right to Left

Associativity

When two operators have the same priority, associativity tells the compiler which direction to evaluate:

- Left to Right: Start from the left side
- Right to Left: Start from the right side

Example:

int x = 10 / 2 * 3; // Left to Right: (10 / 2) = 5 → 5 * 3 = 15

Examples of Arithmetic Expressions

1. Simple Expression

int result = 5 + 3;

2. Mixed Operators

int result = 5 + 3 * 2; // result = 5 + (3 * 2) = 11

3. Using Parentheses

int result = (5 + 3) * 2; // result = (8) * 2 = 16

4. Complex Expression

float x = a - b / 3 + c * 2 - 1;

Here, division and multiplication are done before addition and subtraction.

Summary Table

Concept	Description
Arithmetic Expression	Combination of values and operators
Operator Hierarchy	Rules that decide which operation comes first
Associativity	Direction of evaluation when priorities match
Parentheses	Used to change default order of operations

What Is an Arithmetic Expression?

An arithmetic expression is a combination of:

- Constants (like 5, 10)
- Variables (like a, b)
- Arithmetic operators (+, -, *, /, %)

It performs a calculation and gives a result.
Example:
x = a + b * c;

This expression calculates b * c first, then adds a, and stores the result in x.

What Is Evaluation of an Arithmetic Expression?

Evaluation means solving the expression step by step to get the final result. The computer follows specific rules to decide which part to solve first.

Key Rules for Evaluation

1. Operator Precedence

Some operators are solved before others.

Priority	Operators	Description
1	()	Parentheses
2	*, /, %	Multiplication, Division, Modulus
3	+, -	Addition, Subtraction

2. Associativity
When operators have the same priority, the computer follows a direction:

- Left to Right for +, -, *, /, %
- Right to Left for assignment (=)

Step-by-Step Example
Expression:
x = a - b / 3 + c * 2 - 1;

Given:

a = 10;
b = 12;
c = 2;

Step-by-Step Evaluation:

1. b / 3 → 12 / 3 = 4
2. c * 2 → 2 * 2 = 4
3. Now: x = 10 - 4 + 4 - 1
4. Left to right:
   - 10 - 4 = 6
   - 6 + 4 = 10
   - 10 - 1 = 9

Final Result:

x = 9;

Using Parentheses to Control Order

You can use parentheses to change the default order.

Example:

x = (a + b) * c;

Here, a + b is solved first, then multiplied by c.

Real-Life Analogy

Expression	Real-Life Meaning
price + tax	Add tax to price
(price + tax) * quantity	Total cost for multiple items
marks / total * 100	Calculate percentage

Summary Table

Concept	Description
Arithmetic Expression	Combination of values and operators
Evaluation	Step-by-step solving of the expression
Precedence	Which operator is solved first
Associativity	Direction of solving when priorities match
Parentheses	Used to control the order of operations

What Is Type Conversion?

Type conversion means changing a value from one data type to another. For example, converting an int to a float, or a char to an int.

Why Is It Needed?

- To match data types during calculations
- To avoid errors when storing values
- To make sure the result is accurate

Two Types of Type Conversion in C

1. Implicit Type Conversion (Automatic)

Definition:

The compiler automatically converts one data type to another when needed. This usually happens when combining different types in an expression.

Example:

```
int a = 10;
float b;
b = a;  // int is automatically converted to float
```

How It Works:

- Smaller types are converted to larger types
- char → int → float → double

Real-Life Analogy:

Pouring a small cup of water into a big bottle—no overflow, and it fits easily.

2. Explicit Type Casting (Manual)

Definition:

The programmer manually converts one data type to another using a casting operator.

Syntax:

```
(target_type) value;
```

Example:

```
float x = 5.75;
int y;
y = (int)x;  // Manually converts float to int → y = 5
```

How It Works:

- You decide what type you want
- Useful when you want to control rounding or precision

Real-Life Analogy:

Using a funnel to pour liquid into a smaller container—you control how much goes in.

Key Differences Between Type Conversion and Type Casting

Feature	Type Conversion (Implicit)	Type Casting (Explicit)
Who does it?	Compiler	Programmer
When it happens	Automatically during execution	Manually during coding
Needs casting syntax?	No	Yes ((type) before value)
Risk of data loss	Low (widening conversion)	Possible (narrowing conversion)
Example	float b = a;	int y = (int)x;

More Examples

Implicit Conversion:

```
char ch = 'A';     // ASCII value = 65
int num = ch;      // ch is converted to int → num = 65
```

Explicit Casting:

```
double d = 9.99;
int i = (int)d;    // i = 9 (decimal part is removed)
```

Summary Table

Concept	Description
Type Conversion	Automatic change of data type by compiler
Type Casting	Manual change of data type by programmer
Purpose	To match types and avoid errors
Risk	Casting may lose data; conversion is safer

What Is Decision Making in C?

In real life, we make decisions all the time:

- If it's raining, take an umbrella.
- If you score more than 40, you pass the exam.

In C programming, we use decision-making statements to let the computer make similar choices based on conditions. The most basic and commonly used decision-making tool is the if statement.

What Is an if Statement?

The if statement checks a condition.

- If the condition is true, it runs a block of code.
- If the condition is false, it skips that block.

Syntax:

```
if (condition) {
   // code to run if condition is true
}
```

Example:

```
int age = 20;

if (age >= 18) {
   printf("You are an adult.");
}
```

Explanation:
The program checks if age is greater than or equal to 18. If yes, it prints "You are an adult."

How It Works

- The condition inside ( ) is checked.
- If it's true (non-zero), the code inside { } runs.
- If it's false (zero), the code is skipped.

Real-Life Analogy

Situation	C Code Example	Meaning
If it's raining, take umbrella	if (raining)	Only take umbrella if raining is true
If marks > 40, pass exam	if (marks > 40)	Only pass if marks are above 40

Variations of if Statement

1. Simple if Statement

```
int temperature = 35;

if (temperature > 30) {
    printf("It's a hot day.");
}
```

2. if-else Statement

```
int marks = 45;

if (marks >= 40) {
    printf("You passed.");
} else {
    printf("You failed.");
}
```

3. Nested if Statement

```
int age = 65;

if (age >= 18) {
    if (age >= 60) {
        printf("Senior citizen.");
    } else {
        printf("Adult.");
    }
}
```

4. if-else if Ladder

```
int score = 85;

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

Summary Table

Statement Type	Purpose
if	Runs code only if condition is true
if-else	Chooses between two blocks
nested if	Checks multiple conditions inside another
if-else if ladder	Handles multiple conditions in sequence

What Is an if-else Statement?

Definition:

An if-else statement lets the program choose between two options based on a condition.

- If the condition is true, it runs one block of code.
- If the condition is false, it runs a different block.

Syntax:

```
if (condition) {
   // code if condition is true
} else {
   // code if condition is false
}
```

Example:

```
int marks = 45;

if (marks >= 40) {
   printf("You passed.");
} else {
   printf("You failed.");
}
```

Explanation:
The program checks if marks is 40 or more. If yes, it prints “You passed.” If not, it prints “You failed.”

What Is a Nested if Statement?

Definition:

A nested if statement means putting one if statement inside another. It’s used when you need to check multiple conditions step by step.

Syntax:

```
if (condition1) {
    if (condition2) {
        // code if both conditions are true
    } else {
        // code if condition1 is true but condition2 is false
    }
} else {
    // code if condition1 is false
}
```

Example:

```
int age = 65;

if (age >= 18) {
    if (age >= 60) {
        printf("Senior citizen.");
    } else {
        printf("Adult.");
    }
} else {
    printf("Minor.");
}
```

Explanation:

- First, it checks if age is 18 or more.
- If yes, it checks if age is 60 or more.
- Based on both checks, it prints “Senior citizen” or “Adult.”
- If age is less than 18, it prints “Minor.”

Real-Life Analogy

Situation	C Code Example	Meaning
If it’s raining, take umbrella	if (raining)	Only take umbrella if raining
If age ≥ 18, check if age ≥ 60	if (age >= 18) then if (age >= 60)	Decide between adult or senior

Summary Table

Statement Type	Purpose	Example Use Case
if-else	Choose between two options	Pass or fail based on marks
nested if	Check multiple conditions step by step	Age-based category (minor/adult/senior)

1. Else-If Ladder

What Is It?

An else-if ladder is used when you want to check multiple conditions one by one. The program checks each condition from top to bottom and executes the block of code for the first condition that is true.

Syntax:

```
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 none of the above conditions are true
}
```

Example:

```
int marks = 85;

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

Explanation:
The program checks each condition. Since marks is 85, it prints “Grade B” and skips the rest.

2. Switch Statement

What Is It?

A switch statement is used to test one variable or expression against multiple fixed values (called cases). It's cleaner than an else-if ladder when you're checking for exact matches.

Syntax:

```
switch (expression) {
   case value1:
      // code for value1
      break;

   case value2:
      // code for value2
      break;

   default:
      // code if no case matches
}
```

Example:

```
int day = 3;

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

Explanation:
Since day is 3, it matches case 3 and prints “Wednesday”.

3. Break Statement

What Is It?

The break statement is used to exit a switch case or a loop immediately. Without break, the program will continue checking the next cases even after a match is found (this is called “fall-through”).

Example Without Break:

```
int x = 2;
```

```
switch (x) {
   case 1:
      printf("One");
   case 2:
      printf("Two");
   case 3:
      printf("Three");
}
```

Output:

TwoThree

Explanation:
Since there's no break, it prints both "Two" and "Three".

Example With Break:

```
int x = 2;

switch (x) {
   case 1:
      printf("One");
      break;
   case 2:
      printf("Two");
      break;
   case 3:
      printf("Three");
      break;
}
```

Output:
Two

Summary Table

Concept	Purpose	Best Used When
Else-If Ladder	Checks multiple conditions in order	Conditions involve ranges or logic
Switch Statement	Matches one value against fixed cases	Conditions are exact matches
Break Statement	Stops execution of further cases or loops	Used inside switch or loops

1. goto Statement in C
What Is It?

The goto statement lets the program jump directly to another part of the code using a label. It's like saying:

"Skip everything and go straight to this point."

Syntax:

```
goto label;
...
label:
   // code to run
```

Example:

```
int n = 0;

if (n == 0)
   goto skip;

printf("This line will be skipped.");

skip:
printf("Jumped to this line.");
```

Explanation:
If n is 0, the program jumps to the label skip: and skips the line above it.

When to Use:

- To exit deeply nested loops
- For error handling
- Rarely used in modern programming because it can make code hard to read

2. Looping Statements in C

What Is Looping?

Looping means repeating a block of code multiple times until a condition is false. It helps avoid writing the same code again and again.

A. for Loop

Definition:

Used when you know how many times you want to repeat something.

Syntax:

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

Example:

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

Output:

1 2 3 4 5

Explanation:

- Start with i = 1
- Repeat while i <= 5
- Increase i by 1 each time

B. while Loop

Definition:

Used when you don't know how many times to repeat, but you have a condition to check.

Syntax:

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

Example:

```
int i = 1;

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

Output:

1 2 3 4 5

Explanation:

- Start with i = 1
- Keep repeating while i <= 5
- Increase i each time

Comparison Table

Feature	goto Statement	for Loop	while Loop
Purpose	Jump to a specific label	Repeat known number of times	Repeat while condition is true
Structure	Uses label and jump	Uses init, condition, update	Uses condition only
Readability	Can be confusing	Clear and structured	Simple and flexible
Best Use Case	Exiting nested loops	Counting, fixed repetitions	User input, unknown repeats

1. do-while Loop in C

What Is It?

A do-while loop is a type of loop that always runs at least once, even if the condition is false. This is because the condition is checked after the loop body runs.

Think of it like:

You taste food before deciding if it needs more salt. That's how a do-while loop works—it runs first, then checks.

Syntax:

```
do {
   // code to run
} while (condition);
```

Example:

```
int i = 1;

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

Output:

1 2 3 4 5

Explanation:

- The loop prints numbers from 1 to 5.
- Even if the condition was false at the start, the loop would still run once.

Key Features:

- Exit-controlled loop (condition checked after execution)
- Runs at least once
- Useful when you want to guarantee one-time execution

2. Jump Statements in Loops

Jump statements are used to change the normal flow of loops. They help you exit, skip, or jump to specific parts of the loop.

A. break Statement

Purpose:
Used to exit the loop immediately, even if the condition is still true.

Example:

```
for (int i = 1; i <= 5; i++) {
    if (i == 3)
        break;
    printf("%d ", i);
}
```

Output:

1 2

Explanation:
The loop stops when i == 3.

B. continue Statement

Purpose:
Used to skip the current iteration and move to the next one.

Example:

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

Output:

1 2 4 5

Explanation:
When i == 3, it skips printing and continues with the next number.

C. goto Statement

Purpose:
Used to jump to a labeled part of the program.

Example:

```
int x = 1;

if (x == 1)
    goto skip;

printf("This will not print.");

skip:
printf("Jumped to this line.");
```

Output:

```
Jumped to this line.
```

Explanation:
The program jumps directly to the label skip:.

Summary Table

Statement	Purpose	Best Use Case
do-while	Runs loop at least once	Menu systems, retry prompts
break	Exits loop immediately	Stop when condition is met
continue	Skips current loop iteration	Skip unwanted values
goto	Jumps to a labeled part of code	Rare use; avoid unless necessary

What Is an Array?

An array is a collection of variables stored together under one name. All the values in an array are of the same type, like all integers or all strings. Think of it like a row of boxes, each holding one value.

What Is a One-Dimensional Array?

A one-dimensional array is the simplest type of array. It's just a single row of elements.

Example:

[10, 20, 30, 40, 50]

This array has 5 elements. Each element has a position called an index, starting from 0.

Index	Value
0	10
1	20
2	30
3	40
4	50

How to Create a One-Dimensional Array (in C/C++ style)

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

- int means the array stores integers.
- numbers is the name of the array.
- [5] means it has 5 elements.
- {10, 20, 30, 40, 50} are the values stored.

Accessing Elements

You use the index to get a value:

```
numbers[2];  // This gives you 30
```

Remember: Index starts from 0, not 1.

Changing Values

You can update a value like this:

numbers[3] = 100; // Now the array becomes [10, 20, 30, 100, 50]

Looping Through an Array

You can use a loop to go through each item:

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

This prints: 10 20 30 100 50

Why Use Arrays?

- To store multiple values in one place.
- To organize data neatly.
- To process many items easily using loops.

Real-Life Analogy

Think of an array like a train with compartments. Each compartment holds one passenger (value), and you can find them by their seat number (index).

Declaration of Array

What Is an Array in C?

An array is a way to store multiple values of the same type using a single name. Instead of creating many variables, you can use one array to hold all the values.

Why Use Arrays?

Imagine you want to store marks of 5 students. You could do:

int mark1, mark2, mark3, mark4, mark5;

But that's messy. Instead, use an array:

int marks[5];

Now you have one variable marks that holds 5 values.

How to Declare an Array in C

Basic Syntax:

data_type array_name[size];

- data_type: Type of data (like int, float, char)
- array_name: Name of the array
- size: Number of elements the array can hold

Example:

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

This creates 5 boxes to store integers:

numbers[0], numbers[1], numbers[2], numbers[3], numbers[4]

Declaring and Initializing Together

You can also give values while declaring:

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

If you don't mention the size, C will count the values:

int numbers[] = {10, 20, 30, 40, 50}; // Size is 5

Accessing Array Elements

Use the index (starting from 0):

printf("%d", numbers[2]); // Prints 30

Changing Array Values

You can update values like this:

numbers[3] = 100; // Now numbers[3] is 100

Summary

Concept	Example	Meaning
Declare array	int marks[5];	Creates space for 5 integers
Initialize array	int marks[5] = {10,20,30,40,50};	Sets values during declaration
Access element	marks[2]	Gets the 3rd value (index starts at 0)
Change value	marks[2] = 99;	Updates the 3rd value to 99

Initialization of Arrays in C

Initialization means giving values to the array when you create it.

Ways to Initialize:

a) Manual Initialization

You can assign values one by one:

```
int marks[5];
marks[0] = 10;
marks[1] = 20;
marks[2] = 30;
marks[3] = 40;
marks[4] = 50;
```

b) Initialization at Declaration

You can give all values at once:

```
int marks[5] = {10, 20, 30, 40, 50};
```

c) Letting C Count the Size

If you don't mention the size, C will count the values:

```
int marks[] = {10, 20, 30, 40, 50};  // Size is 5
```

d) Partial Initialization

If you give fewer values, the rest become 0:

```
int marks[5] = {10, 20};  // Remaining values are 0
```

Memory Representation of Arrays

When you create an array, C stores it in contiguous memory locations. That means all elements are placed side by side in memory.

Example:

```
int marks[5] = {10, 20, 30, 40, 50};
```

Let's say each int takes 4 bytes. Then memory looks like this:

Address	Value	Index
1000	10	marks[0]
1004	20	marks[1]
1008	30	marks[2]

Address	Value	Index
1012	40	marks[3]
1016	50	marks[4]

- Each value is stored at a different memory address.
- The gap between addresses is 4 bytes (because int is 4 bytes).

Why Is Memory Representation Important?

- Helps in understanding how arrays work internally.
- Useful when working with pointers.
- Helps in debugging and optimizing programs.

Summary

Concept	Meaning
Initialization	Giving values to array elements
Manual	Assigning values one by one
At declaration	Giving all values at once
Partial	Giving some values; rest become 0
Memory layout	Array elements stored side by side in memory
Address spacing	Depends on data type size (e.g., 4 bytes for int)

What Is a Two-Dimensional Array?

A two-dimensional array is like a table or a grid with rows and columns. It's used to store data in a structured format, like marks of students in different subjects.

Think of it like a matrix or a spreadsheet.

Basic Syntax of Declaration

data_type array_name[rows][columns];

- data_type: Type of data (like int, float, char)
- array_name: Name of the array
- rows: Number of rows
- columns: Number of columns

Example: Declaring a 2D Array

int marks[3][4];

This means:

- You have 3 rows (maybe 3 students)
- You have 4 columns (maybe 4 subjects)
- Total elements = 3 × 4 = 12

Visual Representation

Here's how marks[3][4] looks in memory:

Row 0: marks[0][0] marks[0][1] marks[0][2] marks[0][3]
Row 1: marks[1][0] marks[1][1] marks[1][2] marks[1][3]
Row 2: marks[2][0] marks[2][1] marks[2][2] marks[2][3]

Each element is accessed using two indices:

- First index → Row number
- Second index → Column number

Declaring and Initializing Together

You can also give values while declaring:

```
int marks[2][3] = {
    {10, 20, 30},
    {40, 50, 60}
};
```

This creates:

marks[0][0] = 10 marks[0][1] = 20 marks[0][2] = 30
marks[1][0] = 40 marks[1][1] = 50 marks[1][2] = 60

Accessing Elements

To get a value:

```
printf("%d", marks[1][2]);  // Prints 60
```

To change a value:

```
marks[0][1] = 99;  // Now marks[0][1] is 99
```

Summary Table

Concept	Example	Meaning
Declare 2D array	int table[3][4];	3 rows, 4 columns

Concept	Example	Meaning
Initialize at once	int table[2][2] = {{1,2},{3,4}};	Sets values during declaration
Access element	table[1][0]	Gets value from 2nd row, 1st column
Change value	table[0][1] = 99;	Updates value in 1st row, 2nd column

Initialization of Two-Dimensional Arrays

Initialization means giving values to the array when you create it.

Syntax:

```
data_type array_name[rows][columns] = {
   {value1, value2, value3},
   {value4, value5, value6}
};
```

Example:

```
int marks[2][3] = {
   {10, 20, 30},
   {40, 50, 60}
};
```

This creates a table like this:

Row	Column 0	Column 1	Column 2
0	10	20	30
1	40	50	60

You can also initialize partially:

```
int marks[2][3] = {
   {10},      // Row 0: 10, 0, 0
   {40, 50}    // Row 1: 40, 50, 0
};
```

If you don't give all values, the rest become 0.

Memory Representation of Two-Dimensional Arrays

In C, a 2D array is stored in row-major order. That means:

- First row is stored completely
- Then second row

- Then third row, and so on

Example:

```
int marks[2][3] = {
    {10, 20, 30},
    {40, 50, 60}
};
```

Let's say each int takes 4 bytes. Then memory looks like this:

Address	Value	Element
1000	10	marks[0][0]
1004	20	marks[0][1]
1008	30	marks[0][2]
1012	40	marks[1][0]
1016	50	marks[1][1]
1020	60	marks[1][2]

So even though it looks like a table, in memory it's stored as a long line of values.

Accessing and Changing Values

To access:

```
printf("%d", marks[1][2]);  // Prints 60
```

To change:

```
marks[0][1] = 99;  // Now marks[0][1] is 99
```

Summary

Concept	Meaning
Initialization	Giving values to array elements
Full Initialization	All values given in rows and columns
Partial Initialization	Missing values become 0
Memory Layout	Stored row by row in continuous memory
Accessing	Use two indices: array[row][column]

What Is a Function?

A function is a block of code that performs a specific task.

Think of it like a machine: You give it input, it does something, and gives you output.

Instead of writing the same code again and again, you can write it once in a function and use it whenever needed.

Why Use Functions?

- Makes your code organized
- Avoids repeating the same code
- Makes programs easier to read and manage
- Helps in breaking big problems into smaller parts

Basic Structure of a Function

```
return_type function_name(parameters) {
   // code to do something
   return value;
}
```

Example:

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

This function:

- Is named add
- Takes two inputs: a and b
- Adds them and returns the result

Types of Functions

1. Library Functions
   Already built into C (like printf(), scanf())
2. User-Defined Functions
   You create them yourself (like add() above)

How to Use a Function

You call a function when you want to use it.

Example:

```
int result = add(5, 3);  // Calls the add function
printf("%d", result);    // Prints 8
```

Summary Table

Part	Meaning
Function Name	Name of the task (e.g., add)
Parameters	Inputs to the function
Return Type	Type of result it gives back (e.g., int)
Function Body	Code that does the work
Function Call	Tells the function to run

What Is a Function Prototype?

A function prototype tells the compiler:

- The name of the function
- The type of value it returns
- The types of inputs (parameters) it takes

It's like giving the compiler a preview of the function before it's used.

Syntax:

return_type function_name(parameter_type1, parameter_type2, ...);

Example:

int add(int a, int b);

This means:

- The function is named add
- It takes two int inputs
- It returns an int value

Why Use a Prototype?

- Helps the compiler understand the function before it's used
- Avoids errors when calling the function
- Makes your code organized and readable

What Is a Function Call?

A function call is when you use the function in your program.

You give it the required inputs (arguments), and it gives you the result.

Syntax:

function_name(arguments);

Example:

int result = add(5, 3); // Calls the add function

This sends 5 and 3 to the add function, and stores the result in result.

Complete Example

```
#include <stdio.h>

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

int main() {
    int x = 10, y = 20;
    int sum = add(x, y);  // Function Call
    printf("Sum = %d", sum);
    return 0;
}

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

Flow:

1. Prototype tells the compiler about add
2. Call sends values to add
3. Definition does the work and returns the result

Summary Table

Term	Meaning	Example
Function Prototype	Declares the function before use	int add(int, int);
Function Call	Uses the function in the code	add(5, 3);
Function Definition	Actual code that performs the task	int add(int a, int b) {...}

What Does "Passing Arguments" Mean?

When you use a function, you often want to give it some input values. These inputs are called arguments.

For example:

add(5, 3); // 5 and 3 are arguments

What Is "Call by Value"?

Call by value means:

- You send a copy of the value to the function.
- The original value does not change.

The function works with the copy, not the original.

Simple Example

```
#include <stdio.h>

void change(int x) {
    x = 100;
    printf("Inside function: x = %d\n", x);
}

int main() {
    int a = 50;
    change(a);
    printf("Outside function: a = %d\n", a);
    return 0;
}
```

Output:

```
Inside function: x = 100
Outside function: a = 50
```

What Happened:

- a = 50 is passed to change()
- Inside the function, x becomes 100
- But a in main() stays 50

Because only a copy of a was sent, the original a didn't change.

Key Points

Concept	Meaning
Call by Value	Sends a copy of the value
Original Value	Does not change

Concept	Meaning
Safe	Function cannot affect the original variable
Common Use	Used when you don't want to modify inputs

Real-Life Analogy

Imagine you give someone a photocopy of your document. They can write on it, tear it, or change it—but your original document stays safe.

That's exactly how call by value works.

What Is "Call by Reference"?

Call by reference means:

- You send the address of a variable to a function.
- The function can change the original value.

Instead of sending a copy (like in call by value), you send a link to the actual variable.

Why Use Call by Reference?

- To modify the original values inside a function.
- To save memory (especially with large data).
- To return multiple values from a function.

Real-Life Analogy

Imagine you give someone the key to your house (reference). They can go inside and change things.

In call by value, you give them a photo of your house. They can draw on it, but your real house stays the same.

How to Use Call by Reference in C

You use pointers to pass the address of variables.

Example:

```
#include <stdio.h>

void change(int *x) {
   *x = 100;
}
```

```
int main() {
    int a = 50;
    change(&a);
    printf("a = %d\n", a);
    return 0;
}
```

Output:

a = 100

What Happened:

- a = 50 is declared in main()
- &a sends the address of a to change()
- *x = 100 changes the value at that address
- So a becomes 100

Key Concepts

Term	Meaning
&a	Address of variable a
int *x	Pointer to an integer
*x = 100	Change the value at the address

Summary

Feature	Call by Value	Call by Reference
What is passed	Copy of value	Address of variable
Can change original?	No	Yes
Uses pointers?	No	Yes
Memory efficient	Less	More

What Is a Recursive Function?

A recursive function is a function that calls itself to solve a problem.

It breaks a big problem into smaller versions of the same problem, and keeps solving until it reaches a stopping point (called the base case).

Real-Life Example

Imagine you are climbing down stairs one by one.

- You say: "I'll take one step, then ask myself to take the rest."

- You keep doing this until you reach the last step.

Structure of a Recursive Function

```
return_type function_name(parameters) {
   if (base_condition) {
      return result;  // stopping point
   } else {
      return function_name(smaller_problem);  // recursive call
   }
}
```

Example: Factorial Using Recursion

Factorial of a number n is:

$n! = n \times (n-1) \times (n-2) \times \ldots \times 1$

Code:

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

How It Works:

- factorial(3) → 3 * factorial(2)
- factorial(2) → 2 * factorial(1)
- factorial(1) → 1 * factorial(0)
- factorial(0) → returns 1 (base case)

Then it goes back up:

- factorial(1) = 1 * 1 = 1
- factorial(2) = 2 * 1 = 2
- factorial(3) = 3 * 2 = 6

Key Parts of Recursion

Part	Meaning
Base Case	The stopping condition (no more recursion)
Recursive Call	Function calls itself with smaller input

Part	Meaning
Stack Memory	Each call is stored in memory until it ends

Important Notes

- Always write a base case to avoid infinite loops.
- Recursion uses more memory because each call is stored.
- It's useful for problems like:
  - Factorial
  - Fibonacci series
  - Tower of Hanoi
  - Tree and graph traversal

Summary

Concept	Explanation
Recursive Function	A function that calls itself
Base Case	Stops the recursion
Recursive Call	Solves smaller version of the problem
Example	factorial(n) = n * factorial(n - 1)

What Is a String?

In C, a string is a collection of characters stored together.
It's used to store words, sentences, names, etc.

Technically, a string is just a character array that ends with a null character '\0'.

1. Declaration of Strings

You declare a string using a character array.

Syntax:

```
char name[size];
```

Example:

```
char name[10];
```

This creates space for 10 characters.
You can store a word like "Hello" in it.

2. Initialization of Strings

There are two main ways to initialize a string.

a) Using Double Quotes

char name[] = "Hello";

- C automatically adds '\0' at the end.
- So it stores: 'H', 'e', 'l', 'l', 'o', '\0'

b) Using Individual Characters

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

- You must manually add the '\0' character.
- Without it, C won't know where the string ends.

Memory Representation

Let's see how "Hello" is stored in memory:

Index	Value
0	'H'
1	'e'
2	'l'
3	'l'
4	'o'
5	'\0'

The '\0' is very important—it marks the end of the string.

Accessing String Characters

You can access each character using its index:

printf("%c", name[1]); // Prints 'e'

Summary Table

Concept	Example	Meaning
Declare string	char name[10];	Creates space for 10 characters
Initialize (quoted)	char name[] = "Hello";	Stores string with automatic '\0'
Initialize (manual)	char name[6] = {'H','e','l','l','o','\0'};	Manual setup
Access character	name[2]	Gets the 3rd character

What Is String I/O?

String I/O means:

- Input: Taking a string from the user.
- Output: Showing the string on the screen.

In C, strings are handled using character arrays, and we use special functions to read and print them.

1. Taking String Input

a) Using scanf()

```
char name[20];
scanf("%s", name);
```

- This reads a string from the user.
- It stops at space, so it only reads one word.

Example: If user types: John Smith
Only John is stored in name.

b) Using gets() (Not recommended in modern C)

```
char name[50];
gets(name);
```

- Reads the whole line, including spaces.
- But it's unsafe (can cause memory problems), so not recommended.

c) Using fgets() (Better and safer)

```
char name[50];
fgets(name, 50, stdin);
```

- Reads up to 49 characters (leaves space for \0)
- Reads the whole line, including spaces
- Safe and preferred

2. Showing String Output

a) Using printf()

```
printf("%s", name);
```

- Prints the string stored in name

b) Using puts()

puts(name);

- Prints the string and automatically adds a new line at the end

Complete Example

```
#include <stdio.h>

int main() {
   char name[50];

   printf("Enter your name: ");
   fgets(name, 50, stdin);  // Input

   printf("Hello, ");
   puts(name);              // Output

   return 0;
}
```

Summary Table

Function	Purpose	Reads Spaces?	Safe to Use?
scanf()	Reads one word	No	Yes
gets()	Reads full line	Yes	No (unsafe)
fgets()	Reads full line	Yes	Yes (preferred)
printf()	Prints string	—	Yes
puts()	Prints with newline	—	Yes

What Is an Array of Strings?

An array of strings means:

- A group of multiple strings stored together.
- Each string is stored as a character array.
- All strings are stored in a 2D character array.

Think of it like a list of words or names.

Real-Life Example

Imagine you want to store the names of 5 students:

- "Amit"
- "Ravi"
- "Neha"
- "Simran"
- "Karan"

Instead of creating 5 separate string variables, you can use one array of strings.

How to Declare an Array of Strings

Syntax:

```
char array_name[number_of_strings][maximum_length];
```

Example:

```
char names[5][20];
```

This means:

- You can store 5 strings.
- Each string can be up to 19 characters (1 space is for '\0').

How to Initialize an Array of Strings

a) Manual Initialization

```
strcpy(names[0], "Amit");
strcpy(names[1], "Ravi");
strcpy(names[2], "Neha");
strcpy(names[3], "Simran");
strcpy(names[4], "Karan");
```

You need to use strcpy() from <string.h> to copy strings.

b) Direct Initialization

```
char names[5][20] = {
   "Amit",
   "Ravi",
   "Neha",
   "Simran",
   "Karan"
};
```

This is easier and cleaner.

Accessing Strings

You can access each string using its index:

```
printf("%s", names[2]);  // Prints "Neha"
```

You can also loop through all strings:

```
for (int i = 0; i < 5; i++) {
    printf("%s\n", names[i]);
}
```

Memory Representation

Each string is stored in a row of a 2D character array.

Example: char names[3][10] = {"Cat", "Dog", "Cow"};

Memory layout:

Row	Characters Stored
0	'C', 'a', 't', '\0'
1	'D', 'o', 'g', '\0'
2	'C', 'o', 'w', '\0'

Each row is a separate string.

Summary Table

Concept	Example	Meaning
Declare array	char names[5][20];	5 strings, each up to 19 characters
Initialize directly	{"Amit", "Ravi", ...}	Stores strings in rows
Access string	names[2]	Gets the 3rd string
Print string	printf("%s", names[i]);	Shows the string on screen

What Is String Length?

String length means:

- The number of characters in a string.
- It does not count the special ending character '\0' (null character).

For example:

```
char name[] = "Hello";
```

The length of this string is 5 (H, e, l, l, o)

Which Function Is Used?

In C, we use the strlen() function to find the length of a string.

It is part of the <string.h> library.

Syntax of strlen()

int length = strlen(string_name);

- string_name: The name of the string
- strlen() returns the number of characters in the string (excluding '\0')

Example Program

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

int main() {
   char word[] = "Computer";
   int len = strlen(word);
   printf("Length of the string is: %d", len);
   return 0;
}
```

Output:

Length of the string is: 8

How It Works

- The string "Computer" has 8 characters.
- strlen() counts each character until it reaches '\0'.
- It returns 8.

Important Points

Feature	Description
Counts characters	Only visible characters, not '\0'
Returns integer	Gives back the length as an int
Needs <string.h>	You must include this header file
Works with strings	Only works with character arrays (strings)

Real-Life Analogy

Think of a string as a word written on paper.
strlen() is like counting the number of letters in that word—not the space after it, just the letters.

1. strcpy() – Copy a String

Purpose: Copies one string into another.

Syntax:

strcpy(destination, source);

Example:

char name1[20];
char name2[] = "Amit";
strcpy(name1, name2);

Now name1 becomes "Amit".

Important:

- Make sure destination has enough space.
- It copies the entire string including the '\0'.

2. strcmp() – Compare Two Strings

Purpose: Compares two strings character by character.

Syntax:

strcmp(string1, string2);

Returns:

- 0 → if both strings are equal
- <0 → if string1 is less than string2
- >0 → if string1 is greater than string2

Example:

```
char a[] = "Apple";
char b[] = "Banana";

if (strcmp(a, b) == 0) {
   printf("Strings are equal");
} else {
   printf("Strings are different");
```

}

3. strcat() – Concatenate (Join) Strings

Purpose: Adds one string to the end of another.

Syntax:

strcat(destination, source);

Example:

```
char first[20] = "Hello ";
char second[] = "World";
strcat(first, second);
```

Now first becomes "Hello World".

Important:

- destination must have enough space to hold both strings.

4. strlen() – Find Length of a String

Purpose: Counts the number of characters in a string (excluding '\0').

Syntax:

int length = strlen(string);

Example:

```
char word[] = "Computer";
int len = strlen(word);  // len = 8
```

Summary Table

Function	Purpose	Example
strcpy()	Copy one string	strcpy(a, b);
strcmp()	Compare two strings	strcmp(a, b);
strcat()	Join two strings	strcat(a, b);
strlen()	Find string length	strlen(a);

Real-Life Analogy

- strcpy() → Like copying a name from one paper to another.

- strcmp() → Like checking if two names are the same.
- strcat() → Like joining two words to make a full sentence.
- strlen() → Like counting the number of letters in a word.

What Is a Substring?

A substring is a part of a string.

For example, in the string "Hello World":

- "Hello" is a substring
- "World" is a substring
- "lo Wo" is also a substring

Why Search for a Substring?

You might want to:

- Check if a word exists inside a sentence
- Find a keyword in a paragraph
- Look for a name in a list

Which Function Is Used?

In C, we use the strstr() function to search for a substring.

It is part of the <string.h> library.

Syntax of strstr()

char *strstr(const char *main_string, const char *substring);

- main_string: The full string you want to search in
- substring: The part you are looking for
- Returns: A pointer to the first occurrence of the substring, or NULL if not found

Simple Example

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

int main() {
   char text[] = "Welcome to Chandigarh";
   char *found;

   found = strstr(text, "Chandigarh");
```

```
    if (found != NULL) {
        printf("Substring found: %s\n", found);
    } else {
        printf("Substring not found.\n");
    }

    return 0;
}
```

Output:

Substring found: Chandigarh

How It Works

- strstr() looks inside text for "Chandigarh"
- If it finds it, it returns the pointer to where it starts
- If not, it returns NULL

Important Notes

Feature	Description
Case-sensitive	"Hello" ≠ "hello"
Returns pointer	Not just true/false, but location in the string
Needs <string.h>	Must include this header

Real-Life Analogy

Imagine you're reading a book and want to find the word "magic."
You scan each page until you find it.
That's exactly what strstr() does—it searches the string for the word you want.

What Is a Pointer in C?

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

Real-Life Analogy:

Think of a pointer like a sticky note with a house address written on it. The sticky note doesn't hold the people (data), it just tells you where they live (memory location).

Declaring a Pointer in C

Basic Syntax:

data_type *pointer_name;

Example:

int *p;

This means:

- int → The pointer will point to an integer.
- *p → p is a pointer (the * means "pointer to").
- So p can store the address of an integer variable.

Step-by-Step Example

```
int a = 10;     // Normal integer variable
int *p;         // Declaring a pointer to int
p = &a;         // Assigning address of 'a' to pointer 'p'
```

What's Happening?

- a holds the value 10.
- &a gives the address of a.
- p now holds that address.

Memory View:

Variable	Value	Meaning
a	10	Actual data
p	&a	Address of a

Accessing Value Using Pointer

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

- *p means “go to the address stored in p and get the value”.
- This is called dereferencing.

Summary Table

Term	Meaning
*	Declares a pointer or dereferences it
&	Gets the address of a variable
int *p;	Declares a pointer to an integer
p = &a;	Stores address of a in pointer p
*p	Gets the value at the address stored in p

What Does "Initializing a Pointer" Mean?

Initializing a pointer means giving it a starting value — usually the address of a variable.

Real-Life Analogy:

Imagine a pointer as a GPS device. To use it, you must first enter a location (address). Without an address, the GPS is useless. Similarly, a pointer must be given a memory address to work properly.

Syntax of Pointer Initialization

data_type *pointer_name = &variable_name;

Example:

```
int a = 5;
int *p = &a;
```

Explanation:

- a is a normal integer variable with value 5.
- &a means “address of a”.
- p is a pointer to an integer, and it is initialized with the address of a.

Step-by-Step Breakdown

1. Declare a variable:
2. int a = 5;

   This creates a variable a and stores the value 5.

3. Declare a pointer:
4. int *p;

   This creates a pointer p that can store the address of an integer.

5. Initialize the pointer:
6. p = &a;

   Now p holds the address of a.

Accessing Value Through Pointer

printf("%d", *p); // Output: 5

- *p means “go to the address stored in p and get the value”.
- This is called dereferencing.

Common Mistake: Uninitialized Pointer

int *p;
printf("%d", *p); // □ Dangerous: p is not initialized

- This is like using a GPS without entering a location — it may crash or give garbage results.
- Always initialize a pointer before using it.

Safe Initialization Options

Method	Example	Description
Point to a variable	int *p = &a;	Most common and safe
Set to NULL	int *p = NULL;	Means “no address yet” — safe placeholder
Dynamic memory allocation	int *p = malloc(sizeof(int));	Used in advanced topics like arrays and structures

Summary Table

Term	Meaning
*	Declares or dereferences a pointer
&	Gets the address of a variable
int *p = &a;	Initializes pointer p with address of a
*p	Gets the value at the address stored in p
NULL	Special value meaning “no address”

What Does "Accessing Address" Mean in C?

In C programming, every variable is stored somewhere in memory, and that location has a unique address. Accessing the address means finding out where the variable lives in memory.

Real-Life Analogy:

Imagine each variable is a house, and the address tells you where that house is located. You can use the address to visit the house or send something to it.

How to Access the Address of a Variable

Syntax:

&variable_name

The & symbol means “address of”.

Example:

```
int a = 10;
printf("%p", &a);
```

- a is a variable holding the value 10.
- &a gives the address of a.
- %p is used in printf to display an address.

Step-by-Step Breakdown

1. Declare a variable:
2. int a = 10;
3. Access its address:
4. &a
5. Print the address:

6. printf("Address of a: %p", &a);

Why Accessing Address Is Useful

- To use pointers (pointers store addresses).
- To pass variables by reference to functions.
- To understand memory layout and debugging.

Example with Pointer

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

printf("Value of a: %d\n", a);
printf("Address of a: %p\n", &a);
printf("Address stored in p: %p\n", p);
printf("Value at address p: %d\n", *p);
```

Explanation:

- p stores the address of a.
- *p accesses the value at that address (which is 10).

Summary Table

Expression	Meaning
&a	Address of variable a
p = &a	Pointer p stores address of a
*p	Value at the address stored in p
%p	Format specifier to print address

What Does “Accessing Value Using Pointers” Mean?

When you use a pointer in C, you're working with the address of a variable. But sometimes, you want to get the actual value stored at that address. This is called dereferencing the pointer.

Real-Life Analogy:

Imagine a pointer as a house address written on a slip of paper. If you want to know who lives there (the value), you go to that address. Dereferencing is like opening the door and seeing who's inside.

Syntax to Access Value Using Pointer

*pointer_name

The * symbol is used to dereference the pointer — it means "go to the address stored in the pointer and get the value".

Step-by-Step Example

```
int a = 10;       // Step 1: Declare a variable
int *p = &a;      // Step 2: Declare and initialize a pointer with address of 'a'
printf("%d", *p); // Step 3: Access value using pointer
```

Explanation:

- a holds the value 10.
- &a gives the address of a.
- p stores that address.
- *p gives the value stored at that address — which is 10.

Visual Representation

Variable	Value	Address (Example)
a	10	0x100
p	0x100 (address of a)	—

- *p → Go to address 0x100 → Find value 10

Why Is This Useful?

- You can indirectly access and modify variables.
- Helps in function arguments (pass by reference).
- Used in arrays, structures, and dynamic memory.

Modifying Value Using Pointer

```
*p = 20;
printf("%d", a);  // Output: 20
```

- This changes the value of a through the pointer p.

Summary Table

Expression	Meaning
*p	Value at the address stored in p
p = &a	Pointer p stores address of variable a
*p = 20	Change value at the address to 20

Pointer & Arrays

What Is an Array?

An array is a group of variables of the same type stored together in memory.

Example:

int numbers[3] = {10, 20, 30};

This creates an array with 3 integers:

- numbers[0] = 10
- numbers[1] = 20
- numbers[2] = 30

What Is a Pointer?

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

Example:

int a = 5;
int *p = &a;

Here, p stores the address of a.

How Are Pointers and Arrays Connected?

In C, the name of an array is actually a pointer to its first element.

Example:

int numbers[3] = {10, 20, 30};

- numbers is the name of the array.
- numbers is also a pointer to numbers[0].

So:

```
printf("%d", *numbers);       // Output: 10
printf("%d", *(numbers + 1)); // Output: 20
```

Step-by-Step Explanation

1. Array Declaration

```
int numbers[3] = {10, 20, 30};
```

2. Pointer to Array

```
int *p = numbers;
```

This means:

- p points to the first element of the array (numbers[0]).

3. Accessing Values Using Pointer

```
printf("%d", *p);       // Output: 10
printf("%d", *(p + 1)); // Output: 20
printf("%d", *(p + 2)); // Output: 30
```

- *p → value at first element
- *(p + 1) → value at second element
- *(p + 2) → value at third element

Visual Representation

Index	Value	Address (Example)
0	10	0x100
1	20	0x104
2	30	0x108

- numbers or p → points to 0x100
- *(p + 1) → goes to 0x104 and gets 20

Summary Table

Concept	Meaning
numbers	Array name, also pointer to first element

Concept	Meaning
*numbers	Value of first element
*(numbers + i)	Value at index i
int *p = numbers;	Pointer p points to array start
*(p + i)	Value at index i using pointer

What Are User-Defined Data Types?

Definition:
User-defined data types are custom data types created by the programmer using existing basic types (like int, float, char). They help organize and manage complex data more effectively.

Real-Life Analogy:

Think of basic types (int, float, char) as bricks. User-defined types are like custom-shaped blocks made by combining bricks to build something meaningful — like a house, a car, or a student record.

Types of User-Defined Data Types in C

1. struct (Structure)

Definition:
A struct is used to group different types of variables under one name.

Purpose:
To represent real-world objects with multiple properties.

Example:

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

Usage:

```
struct Student s1;
s1.age = 18;
```

Analogy:
A student form with fields for name, age, and marks.

2. union

Definition:
A union is similar to a struct, but all members share the same memory location. Only one member can hold a value at a time.

Purpose:
To save memory when only one value is needed at a time.

Example:

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

Usage:

```
union Data d;
d.i = 10;
```

Analogy:
A single container that can hold either water or oil — but not both at the same time.

3. enum (Enumeration)

Definition:
An enum is used to define a set of named integer constants.

Purpose:
To make code more readable and meaningful.

Example:

```
enum Day { MON, TUE, WED, THU, FRI };
```

Usage:

```
enum Day today;
today = WED;
```

Analogy:
Days of the week labeled with names instead of numbers.

4. typedef

Definition:
typedef is used to create a new name (alias) for an existing data type.

Purpose:
To simplify complex type declarations.

Example:

typedef unsigned int uint;

Usage:

uint age = 25;

Analogy:
Giving a nickname to a long name — like calling "Jonathan" as "Jon".

Summary Table

Type	Definition	Purpose	Example Use
struct	Groups different types of variables	Represent real-world objects	struct Student
union	Shares memory among members	Save memory when only one value is used	union Data
enum	Defines named integer constants	Improve readability	enum Day
typedef	Creates a new name for an existing type	Simplify complex declarations	typedef int Marks;

What Is a Structure in C?

Definition:

A structure in C is a user-defined data type that allows you to group different types of variables under one name. These grouped variables are called members of the structure.

In simple words:
A structure is like a custom container that holds different kinds of data together — like numbers, characters, and strings — all in one place.

Real-Life Analogy

Imagine you're designing a student ID card. It contains:

- Name (text)
- Age (number)
- Marks (decimal)

Instead of creating separate variables for each student, you create a structure called Student that holds all these details together.

Syntax of Structure

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

Example:

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

This structure defines a Student with:

- name → character array (string)
- age → integer
- marks → float

How to Use a Structure

Declare a structure variable:

```
struct Student s1;
```

Assign values:

```
strcpy(s1.name, "Amit");
s1.age = 18;
s1.marks = 85.5;
```

Access values:

```
printf("Name: %s", s1.name);
printf("Age: %d", s1.age);
printf("Marks: %.2f", s1.marks);
```

Key Features of Structures

- Can hold multiple types of data together

- Each member is accessed using dot notation (e.g., s1.age)
- Useful for creating records, like students, employees, books, etc.

Summary Table

Term	Meaning
struct	Keyword to define a structure
StructureName	Name of your custom data type
member	Individual variable inside the structure
. (dot operator)	Used to access structure members
struct Student s1;	Declares a variable of type Student

Advantages of Structures in C

Definition
A structure is a user-defined data type that lets you group different types of variables under one name. It helps organize related data into a single unit.

Why Use Structures?

Here are the key advantages:

1. Groups Related Data Together

Explanation:
Instead of creating separate variables for each detail, you can combine them into one structure.

Example:

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

Analogy:
Like a folder that holds all documents related to one student — name, age, and marks.
2. Improves Code Readability
Explanation:
Using structures makes your code cleaner and easier to understand, especially when dealing with complex data.

Example:

s1.age = 18; // Clear that this is the age of student s1

Analogy:
Labels on boxes make it easier to find what's inside.

3. Makes Data Management Easier

Explanation:
You can manage multiple records (like many students or employees) using arrays of structures.

Example:

struct Student class[50]; // 50 students in one array

Analogy:
Like keeping a list of all students in one notebook.

4. Supports Real-World Modeling

Explanation:
Structures help you model real-world objects like books, cars, employees, etc.

Example:

```
struct Car {
   char model[30];
   int year;
   float price;
};
```

Analogy:
Just like describing a car with its model, year, and price — all in one place.

5. Allows Nested Structures

Explanation:
You can place one structure inside another to represent more detailed data.

Example:

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

```
struct Student {
    char name[50];
    struct Date dob;  // Date of birth
};
```

Analogy:
Like putting a small box inside a bigger box to organize things better.

6. Used in Functions and File Handling

Explanation:
Structures can be passed to functions and used to store data in files.

Example:

```
void printStudent(struct Student s);
```

Analogy:
Like sending a complete form to another department for processing.

Summary Table

Advantage	Explanation
Groups related data	Combines different types under one name
Improves readability	Makes code easier to understand
Easier data management	Handles multiple records with arrays
Real-world modeling	Represents objects like books, cars, etc.
Nested structures	Organizes complex data with sub-structures
Function and file support	Can be used in functions and file operations

Declaring Structure Variables in C

What Is a Structure?

A structure is a user-defined data type in C that lets you group different types of variables under one name. These grouped variables are called members of the structure.

What Is a Structure Variable?

A structure variable is an actual variable created from a structure type. It holds real data for the structure's members.

Real-Life Analogy:

Imagine a form template called Student. Declaring a structure variable is like filling out one copy of that form for a specific student — say, Amit.

Step-by-Step: How to Declare Structure Variables

Step 1: Define the Structure

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

This creates a blueprint called Student with three members:

- name (string)
- age (integer)
- marks (float)

Step 2: Declare Structure Variables

You can declare structure variables in two ways:

Method 1: After the structure definition

```
struct Student s1, s2;
```

- s1 and s2 are two separate structure variables.
- Each holds its own name, age, and marks.

Method 2: Inside the structure definition

```
struct Student {
   char name[50];
   int age;
   float marks;
} s1, s2;
```

- This defines the structure and declares variables s1 and s2 at the same time.

Accessing Structure Members

Use the dot operator (.) to access members:

```
s1.age = 18;
s1.marks = 85.5;
strcpy(s1.name, "Amit");
```

To display values:

```
printf("Name: %s\n", s1.name);
printf("Age: %d\n", s1.age);
printf("Marks: %.2f\n", s1.marks);
```

Summary Table

Concept	Meaning
struct	Keyword to define a structure
Student	Name of the structure
s1, s2	Structure variables (actual data holders)
. (dot operator)	Used to access individual members of the structure
s1.age = 18;	Assigns value to the age member of s1

Accessing Structure Members in C

What Is a Structure?

Definition:
A structure is a user-defined data type that groups different types of variables (called members) under one name.

Example:

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

This structure has three members:

- name → string
- age → integer
- marks → float

What Does "Accessing Structure Members" Mean?

Definition:
Accessing structure members means getting or changing the value of individual fields (members) inside a structure variable.

Real-Life Analogy:

Think of a structure as a form filled out for a student. Each field (name, age, marks) is a member. Accessing a member is like reading or writing in one of those fields.

How to Access Structure Members

Use the dot operator (.) to access members of a structure.

Syntax:

structure_variable.member_name

Step-by-Step Example

Step 1: Define the structure

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

Step 2: Declare a structure variable

```
struct Student s1;
```

Step 3: Assign values to members

```
strcpy(s1.name, "Amit");  // Assign name
s1.age = 18;              // Assign age
s1.marks = 85.5;          // Assign marks
```

Step 4: Access and display values

```
printf("Name: %s\n", s1.name);
printf("Age: %d\n", s1.age);
printf("Marks: %.2f\n", s1.marks);
```

Summary Table

Expression	Meaning
s1.name	Accesses the name member of s1
s1.age	Accesses the age member of s1
s1.marks	Accesses the marks member of s1
. (dot operator)	Used to access structure members

Bonus Tip: Array of Structures

You can also access members in an array of structures:

```
struct Student class[2];

strcpy(class[0].name, "Amit");
class[0].age = 18;
class[0].marks = 85.5;

strcpy(class[1].name, "Sara");
class[1].age = 19;
class[1].marks = 90.0;
```

Structure Member Initialization in C

What Is a Structure?

Definition:
A structure is a user-defined data type that allows you to group different types of variables (called members) under one name.

Example:

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

This structure has three members:

- name → string
- age → integer
- marks → float

What Is Structure Member Initialization?

Definition:
Structure member initialization means giving starting values to the members of a structure variable when it is created.

Real-Life Analogy:

Think of a structure as a blank student form. Initializing members is like filling in the form with actual details — name, age, and marks.

Ways to Initialize Structure Members

Method 1: Manual Assignment (After Declaration)

```
struct Student s1;

strcpy(s1.name, "Amit");
s1.age = 18;
s1.marks = 85.5;
```

- Each member is assigned a value one by one.
- Use strcpy() for string members like name.

Method 2: Initialization at Declaration (Using Braces)

```
struct Student s2 = {"Sara", 19, 90.0};
```

- All members are initialized in order.
- This method is short and clean.

Method 3: Using Designated Initializers (C99 and later)

```
struct Student s3 = {.marks = 88.5, .name = "Ravi", .age = 20};
```

- You can assign values in any order using member names.
- Useful when you want to skip or reorder fields.

Important Notes

- If you skip some members during initialization, they get default values:
  - int → 0
  - float → 0.0
  - char → '\0' (null character)
- Always use strcpy() for string members (character arrays), not =.

Summary Table

Method	Description	Example
Manual Assignment	Assign values one by one	s1.age = 18;
Initialization at Declaration	Use braces to set all values at once	struct Student s2 = {"Sara", 19, 90.0};
Designated Initializers	Use member names to assign values	struct Student s3 = {.age = 20, .name = "Ravi"};

Array of Structures in C

What Is a Structure?

Definition:
A structure is a user-defined data type that groups different types of variables (called members) under one name.

Example:

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

This structure holds details about one student.

What Is an Array of Structures?

Definition:
An array of structures means creating multiple structure variables using a single array. Each element of the array holds a complete set of structure members.

Real-Life Analogy:

Imagine a classroom with 50 students. Instead of creating 50 separate student records, you create one array that holds all 50 student forms — each form is a structure.

Why Use Array of Structures?

- To store and manage multiple records (students, employees, books, etc.)
- To easily access and update data using loops
- To organize complex data in a clean and readable way

Step-by-Step Example

Step 1: Define the Structure

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

Step 2: Declare an Array of Structures

```
struct Student class[3];  // Array for 3 students
```

Step 3: Assign Values to Each Element

```
strcpy(class[0].name, "Amit");
class[0].age = 18;
class[0].marks = 85.5;

strcpy(class[1].name, "Sara");
class[1].age = 19;
class[1].marks = 90.0;

strcpy(class[2].name, "Ravi");
class[2].age = 20;
class[2].marks = 88.0;
```

Step 4: Access Values Using a Loop

```
for (int i = 0; i < 3; i++) {
    printf("Name: %s\n", class[i].name);
    printf("Age: %d\n", class[i].age);
    printf("Marks: %.2f\n\n", class[i].marks);
}
```

Summary Table

Concept	Meaning
struct Student	Defines the structure type
class[3]	Array of 3 structure variables
class[i].name	Accesses name of the i-th student
Looping through array	Helps manage multiple records easily

Union in C

Definition of Union

A union is a user-defined data type in C that allows multiple variables to share the same memory location. You can store different types of data in a union, but only one member can hold a value at a time.

Real-Life Analogy

Imagine a single locker that can hold either a book, a laptop, or a bag — but only one item at a time. If you put a book in, the laptop and bag must stay out. That locker is like a union — it has space for different things, but only one can be stored at a time.

Syntax of Union

```
union UnionName {
   data_type member1;
   data_type member2;
   ...
};
```

Example:

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

This union can hold:

- an integer i
- a float f
- a character ch
  But only one of them at a time.

Declaring Union Variables

```
union Data d1;
```

This creates a union variable d1 that can store either an int, a float, or a char.

Assigning and Accessing Values

```
d1.i = 10;
printf("Integer: %d\n", d1.i);

d1.f = 5.5;
printf("Float: %.2f\n", d1.f);
```

Important Note:

After assigning d1.f, the value of d1.i becomes unreliable because both share the same memory.

Key Features of Union

- All members share the same memory space
- Only one member can hold a valid value at a time
- Saves memory when you need to store only one type of data at a time

Summary Table

Feature	Description
union keyword	Used to define a union
Shared memory	All members use the same memory location
One value at a time	Only one member can hold a valid value
Saves memory	Useful when storing one value from many options

Structure vs Union in C

1. Definition

Structure (struct)

A structure is a user-defined data type that groups different types of variables together.
Each member gets its own separate memory space.

Union (union)

A union is a user-defined data type where all members share the same memory space.
Only one member can hold a valid value at a time.

2. Real-Life Analogy

Structure

Imagine a toolbox with separate compartments for a hammer, screwdriver, and wrench.
You can store all tools at the same time.

Union

Imagine a single drawer that can hold either a hammer or a screwdriver or a wrench —
but only one item at a time.

3. Memory Allocation

Structure

Each member gets its own memory.
Total memory = sum of all member sizes.

Union

All members share the same memory.
Total memory = size of the largest member only.

4. Usage Purpose

Structure

Use when you need to store multiple values at the same time.

Union

Use when you need to store only one value at a time, and want to save memory.

5. Example Code

Structure:

```
struct Student {
    int age;
    float marks;
};
```

Union:

```
union Data {
```

```
    int i;
    float f;
};
```

6. Accessing Members

Structure

You can access and use all members independently.

Union

You can access only one member at a time.
Changing one member overwrites the others.

7. Summary Table

Feature	Structure (struct)	Union (union)
Memory Allocation	Separate memory for each member	Shared memory for all members
Total Memory Used	Sum of all member sizes	Size of the largest member only
Value Storage	All members can hold values together	Only one member holds value at a time
Member Access	All members accessible independently	Only one member valid at a time
Use Case	Store multiple values simultaneously	Save memory when storing one value
Keyword	struct	union