

Programming in C

Class Handout

with Worked Programs

Data Types · Operators · Console I/O
Conditional Statements · Iteration Statements
Arrays · Strings · Pointers

Sreetam Ganguly

September 5, 2026

Copyright © 2026 Sreetam Ganguly.

This handout, together with every program listing it contains, is free material: you may redistribute it and/or modify it under the terms of the **GNU General Public License version 3** as published by the Free Software Foundation, or (at your option) any later version.

It is distributed in the hope that it will be useful, but **without any warranty**; without even the implied warranty of merchantability or fitness for a particular purpose. See the GNU General Public License for more details. You should have received a copy of the licence along with this document; if not, see <https://www.gnu.org/licenses/>.

Contents

1	Module 2 — Data Types, Console I/O and Operators	3
1.1	Data Types in C	3
1.1.1	Types of Data Types in C	3
1.2	Binary Representation	3
1.3	Allocation Size and Range	3
1.4	Console Input and Output	4
1.4.1	The printf() Function	4
1.4.2	The scanf() Function	4
1.5	Formatted Strings	5
1.6	Format Specifiers	6
1.7	Escape Sequences	6
1.8	Operators in C	8
1.9	Operands and Expressions	8
1.9.1	Operands	8
1.9.2	Expression	8
1.10	Unary, Binary and Ternary Operators	9
1.11	Types of Operators in C	9
1.11.1	A. Arithmetic Operators	9
1.11.2	B. Relational Operators	10
1.11.3	C. Logical Operators	10
1.11.4	D. Assignment Operators	10
1.11.5	E. Bitwise Operators	11
1.11.6	F. Increment Operator (++)	11
1.11.7	G. Decrement Operator (--)	12
1.11.8	H. Conditional (Ternary) Operator	12
1.12	Operator Precedence	12
2	Module 3 — Conditional Statements and Loops	15
	Part A: Conditional Statements	15
2.1	if Statement	15
2.2	if...else Statement	15
2.3	else if Ladder	16
2.4	Nested if	17
2.5	switch-case Statement	17
2.6	goto Statement	18
	Part B: Iteration Statements (Loops)	19
2.7	while Loop	19
2.8	do...while Loop	20
2.9	for Loop	21
2.10	break Statement	21
2.11	continue Statement	22
2.12	Nested Loops	22
3	Module 4 — Arrays and Strings	24
	Part A: Arrays	24
3.1	What is an Array?	24
3.2	Declaration and Initialization	24
3.3	Indexing	25
3.4	Memory Layout	25

3.5	Multidimensional Arrays	26
Part B:	Strings	27
3.6	What is a String?	27
3.7	Character Arrays vs Strings	27
3.8	Declaring and Initializing Strings	27
3.9	String Input and Output	28
3.10	Difference Between scanf() and fgets() in C	29
3.10.1	scanf()	29
3.10.2	fgets()	29
3.11	String Library Functions	31
3.11.1	(a) strlen()	32
3.11.2	(b) strcpy()	32
3.11.3	(c) strcat()	32
3.11.4	(d) strcmp()	33
4	Supplementary Programs	34
4.1	Multiplying Two Matrices	34
4.2	Swapping Two Array Elements — The Normal Way	35
4.3	Swapping the Same Two Elements — Using Pointers (Address Swap)	36
4.4	Train Signalling System Simulation	39

1 Module 2 — Data Types, Console I/O and Operators

1.1 Data Types in C

Definition

A **data type** specifies the type of data a variable can store. It also determines:

- The amount of memory allocated.
- The range of values that can be stored.
- The operations that can be performed on the data.

Example

```
1 int age = 20;  
2 float salary = 25000.50;  
3 char grade = 'A';
```

Here `int`, `float` and `char` are data types.

1.1.1 Types of Data Types in C

1. **Basic (Primitive) Data Types** — `int`, `char`, `float`, `double`, `void`
2. **Derived Data Types** — Arrays, Pointers, Functions
3. **User-defined Data Types** — `struct`, `union`, `enum`, `typedef`

1.2 Binary Representation

Definition

Computers understand only binary numbers (0 and 1). Therefore all data stored in memory is represented in binary form.

Example

Item	Representation
Decimal number	5
Binary representation	00000101
Character	'A'
ASCII value	65
Binary	01000001

1.3 Allocation Size and Range

The memory allocated to each data type determines the range of values it can store.

Data Type	Size (Typical)	Range	Format Specifier
char	1 byte	−128 to 127	%c
int	4 bytes	−2,147,483,648 to 2,147,483,647	%d
float	4 bytes	Approximately $\pm 3.4 \times 10^{38}$	%f
double	8 bytes	Approximately $\pm 1.7 \times 10^{308}$	%lf
void	No storage	No value	—

Note

The exact size may vary depending on the compiler and the operating system.

1.4 Console Input and Output

Console Input/Output allows the user to interact with the program using the keyboard and the monitor. The two most commonly used functions are:

- `printf()` — Displays output.
- `scanf()` — Accepts input.

1.4.1 The printf() Function

Definition

`printf()` is used to display text, variables and formatted output on the screen.

Syntax

```
1 printf("Format String", variable);
```

Example

```
1 #include <stdio.h>
2 int main()
3 {
4     int age = 20;
5     printf("Age = %d", age);
6     return 0;
7 }
```

Output

```
Age = 20
```

1.4.2 The scanf() Function

Definition

`scanf()` reads data entered by the user through the keyboard.

Syntax

```
1 scanf("Format Specifier", &variable);
```

The `&` (address-of operator) is used to store the input in the variable's memory location.

Example

```
1 #include <stdio.h>
2 int main()
3 {
4     int age;
5     printf("Enter Age: ");
6     scanf("%d", &age);
7     printf("Age = %d", age);
8     return 0;
9 }
```

Sample Output

```
Enter Age:  21
Age = 21
```

1.5 Formatted Strings

Definition

A **formatted string** is a string containing text along with format specifiers. It allows variables to be displayed or read in a specific format.

Example

```
1 int age = 20;
2 printf("Age = %d", age);
```

Here `"Age = %d"` is the formatted string, and `%d` is replaced by the value of `age`.

Multiple Variables

```
1 #include <stdio.h>
2 int main()
3 {
4     int age = 20;
5     float marks = 89.5;
6     printf("Age = %d\nMarks = %.1f", age, marks);
7     return 0;
8 }
```

Output

```
Age = 20
Marks = 89.5
```

1.6 Format Specifiers

Definition

Format specifiers are special symbols beginning with `%`, used in `printf()` and `scanf()` to specify the type of data.

Specifier	Data Type	Example
<code>%d</code>	Integer	25
<code>%f</code>	Float	12.5
<code>%lf</code>	Double	123.456
<code>%c</code>	Character	A
<code>%s</code>	String	Hello
<code>%u</code>	Unsigned Integer	100
<code>%x</code>	Hexadecimal	1A
<code>%o</code>	Octal	17
<code>%p</code>	Address (Pointer)	0x7ffd...

Example

```
1 #include <stdio.h>
2 int main()
3 {
4     int roll = 10;
5     float marks = 85.5;
6     char grade = 'A';
7     printf("Roll = %d\n", roll);
8     printf("Marks = %.1f\n", marks);
9     printf("Grade = %c\n", grade);
10    return 0;
11 }
```

Output

```
Roll = 10
Marks = 85.5
Grade = A
```

1.7 Escape Sequences

Definition

Escape sequences are special character combinations that begin with a backslash (`\`). They are used to format the output or to represent special characters.

Escape Sequence	Meaning	Example / Effect
<code>\n</code>	New line	Moves the cursor to the next line
<code>\t</code>	Horizontal tab	Adds tab space
<code>\\</code>	Backslash	Prints <code>\</code>
<code>\"</code>	Double quotation mark	Prints <code>"</code>
<code>\'</code>	Single quotation mark	Prints <code>'</code>
<code>\b</code>	Backspace	Moves the cursor one position back
<code>\r</code>	Carriage return	Moves the cursor to the beginning of the line
<code>\0</code>	Null character	Indicates the end of a string

Example 1: New Line

```
1 #include <stdio.h>
2 int main()
3 {
4     printf("Hello\nWorld");
5     return 0;
6 }
```

Output

```
Hello
World
```

Example 2: Tab Space

```
1 #include <stdio.h>
2 int main()
3 {
4     printf("Name\tAge");
5     return 0;
6 }
```

Output

```
Name    Age
```

Example 3: Printing Double Quotes

```
1 #include <stdio.h>
2 int main()
3 {
4     printf("\"C Programming\"");
5     return 0;
6 }
```

Output

```
"C Programming"
```

1.8 Operators in C

Definition

An **operator** is a special symbol that tells the compiler to perform a specific operation on one or more operands.

Example

```
1 int sum = 10 + 20;
```

- `+` → Operator
- `10` and `20` → Operands
- `10 + 20` → Expression

1.9 Operands and Expressions

1.9.1 Operands

Definition

Operands are the values or variables on which an operator performs an operation.

```
1 int a = 10;  
2 int b = 5;  
3 int c = a + b;
```

Here `a` and `b` are operands, and `+` is the operator.

1.9.2 Expression

Definition

An **expression** is a combination of operands, operators and constants that produces a value.

```
1 a + b  
2 x * y  
3 (a + b) * c
```

Types of Expressions

Type of Expression	Example
Arithmetic Expression	<code>a + b</code>
Relational Expression	<code>a > b</code>
Logical Expression	<code>a > b && b < c</code>
Assignment Expression	<code>a = 20;</code>

1.10 Unary, Binary and Ternary Operators

Unary Operator

A **unary operator** works on one operand.

```
1 ++a
2 --a
3 -a
```

Binary Operator

A **binary operator** works on two operands.

```
1 a + b
2 a * b
3 a > b
```

Ternary Operator

A **ternary operator** works on three operands.

Syntax

```
1 condition ? expression1 : expression2;
```

```
1 int max = (a > b) ? a : b;
```

If `a > b` is true then `max = a`; otherwise `max = b`.

Comparison Table

Operator Type	Operands	Example
Unary	One	<code>++a</code>
Binary	Two	<code>a + b</code>
Ternary	Three	<code>(a>b) ? a : b</code>

1.11 Types of Operators in C

1.11.1 A. Arithmetic Operators

Used to perform mathematical calculations.

Operator	Meaning	Example
+	Addition	<code>a+b</code>
-	Subtraction	<code>a-b</code>
*	Multiplication	<code>a*b</code>
/	Division	<code>a/b</code>
%	Modulus (Remainder)	<code>a%b</code>

```
1 #include <stdio.h>
2 int main()
3 {
4     int a = 10, b = 3;
5     printf("%d\n", a + b);
6     printf("%d\n", a - b);
7     printf("%d\n", a * b);
8     printf("%d\n", a / b);
9     printf("%d\n", a % b);
10    return 0;
11 }
```

1.11.2 B. Relational Operators

Used to compare two values.

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

```
1 if(a > b)
2 {
3     printf("a is greater");
4 }
```

1.11.3 C. Logical Operators

Used to combine multiple conditions.

Operator	Meaning
&&	Logical AND
	Logical OR
!	Logical NOT

```
1 if(age > 18 && age < 60)
```

1.11.4 D. Assignment Operators

Used to assign values to variables.

Operator	Example	Meaning
=	a=5	Assign
+=	a+=2	a=a+2
-=	a-=2	a=a-2
=	a=2	a=a*2
/=	a/=2	a=a/2
%=	a%=2	a=a%2

1.11.5 E. Bitwise Operators

Operate on binary bits.

Operator	Meaning
&	Bitwise AND
	Bitwise OR
^	Bitwise XOR
~	Bitwise NOT
<<	Left Shift
>>	Right Shift

Example

```

1 5 & 3
2
3 Binary:
4 5 = 0101
5 3 = 0011
6 -----
7      0001 = 1

```

1.11.6 F. Increment Operator (++)

Increases a variable by 1.

- **Pre-increment** `++a;` — first increases the value, then uses it.
- **Post-increment** `a++;` — first uses the value, then increases it.

```

1 int a = 5;
2 printf("%d", ++a);

```

Output

6

1.11.7 G. Decrement Operator (--)

Decreases a variable by 1.

- **Pre-decrement** `-a;`
- **Post-decrement** `a--;`

```
1 int a = 5;  
2 printf("%d", --a);
```

Output

4

1.11.8 H. Conditional (Ternary) Operator

The conditional operator is a shorthand form of the `if-else` statement.

Syntax

```
1 condition ? expression1 : expression2;
```

```
1 int a = 20, b = 10;  
2 int max = (a > b) ? a : b;  
3 printf("%d", max);
```

Output

20

1.12 Operator Precedence

Definition

Operator precedence determines the order in which operators are evaluated in an expression. If two operators have the same precedence, **associativity** determines the order of evaluation.

Common Operator Precedence (Highest to Lowest)

Precedence	Operators
1 (Highest)	()
2	++, --, !
3	*, /, %
4	+, -
5	«, »
6	<, <=, >, >=
7	==, !=
8	&
9	^
10	
11	&&
12	
13	?:
14	=, +=, -=, *=, /=, %=

Example 1

```
int result = 5 + 3 * 2;
```

Multiplication has higher precedence: $3 \times 2 = 6$, then $5 + 6 = 11$.

Output

11

Example 2

```
int result = (5 + 3) * 2;
```

Parentheses are evaluated first: $5 + 3 = 8$, then $8 \times 2 = 16$.

Output

16

Frequently Asked Questions — Module 2

1. What is a data type? Explain its different categories in C.
2. What is binary representation? Explain with an example.
3. Differentiate between `printf()` and `scanf()`.
4. What is a formatted string? Give an example.
5. Explain common format specifiers used in C.
6. What are escape sequences? List any five with suitable examples.
7. Explain the size and range of basic data types in C.
8. What is an operator? Explain operands and expressions with examples.

9. Differentiate between unary, binary and ternary operators.
10. Explain arithmetic, relational, logical, assignment and bitwise operators with examples.
11. What is the difference between pre-increment and post-increment operators?
12. Explain the conditional (ternary) operator with syntax and example.
13. What is operator precedence? Why are parentheses important in expressions?
14. Write a program using arithmetic, relational and logical operators.

2 Module 3 — Conditional Statements and Loops

Part A: Conditional Statements

Conditional statements are used to make decisions in a program. They execute different blocks of code depending on whether a condition is true or false.

2.1 if Statement

Definition

The `if` statement executes a block of code only if the given condition is true.

Syntax

```
1 if(condition)
2 {
3     // statements
4 }
```

Flow Diagram

```
Condition
|
True ----> Execute Statements
|
False ----> Skip Statements
```

Example

```
1 #include <stdio.h>
2 int main()
3 {
4     int age = 20;
5     if(age >= 18)
6     {
7         printf("Eligible to Vote");
8     }
9     return 0;
10 }
```

Output

Eligible to Vote

2.2 if...else Statement

Definition

The `if...else` statement executes one block if the condition is true and another block if the condition is false.

Syntax

```
1 if(condition)
2 {
```

```
3 // True block
4 }
5 else
6 {
7 // False block
8 }
```

Example

```
1 #include <stdio.h>
2 int main()
3 {
4     int age = 16;
5     if(age >= 18)
6     {
7         printf("Eligible to Vote");
8     }
9     else
10    {
11        printf("Not Eligible");
12    }
13    return 0;
14 }
```

Output

Not Eligible

2.3 else if Ladder

Definition

The `else if` ladder is used to check multiple conditions.

Syntax

```
1 if(condition1)
2 {
3 }
4 else if(condition2)
5 {
6 }
7 else
8 {
9 }
```

Example

```
1 #include <stdio.h>
2 int main()
3 {
4     int marks = 82;
5     if(marks >= 90)
6         printf("Grade A");
7     else if(marks >= 75)
8         printf("Grade B");
9     else if(marks >= 50)
10        printf("Grade C");
```

```
11     else
12         printf("Fail");
13     return 0;
14 }
```

Output

Grade B

2.4 Nested if

Definition

A **nested if** means an `if` statement inside another `if` statement.

Syntax

```
1 if(condition1)
2 {
3     if(condition2)
4     {
5     }
6 }
```

Example

```
1 #include <stdio.h>
2 int main()
3 {
4     int age = 20;
5     int citizen = 1;
6     if(age >= 18)
7     {
8         if(citizen == 1)
9         {
10             printf("Eligible to Vote");
11         }
12     }
13     return 0;
14 }
```

Output

Eligible to Vote

2.5 switch-case Statement

Definition

The `switch-case` statement is used to select one block of code from multiple choices.

Syntax

```
1 switch(expression)
2 {
3     case valuel:
```

```
4     statements;
5     break;
6 case value2:
7     statements;
8     break;
9 default:
10    statements;
11 }
```

Example

```
1 #include <stdio.h>
2 int main()
3 {
4     int day = 2;
5     switch(day)
6     {
7         case 1:
8             printf("Monday");
9             break;
10        case 2:
11            printf("Tuesday");
12            break;
13        case 3:
14            printf("Wednesday");
15            break;
16        default:
17            printf("Invalid Day");
18    }
19    return 0;
20 }
```

Output

Tuesday

Advantages

- Easy to read.
- Faster than multiple if-else statements in many cases.
- Suitable for menu-driven programs.

2.6 goto Statement

Definition

The `goto` statement transfers program control to a labeled statement.

Syntax

```
1 goto label;
2 label:
3     statements;
```

Example

```
1 #include <stdio.h>
2 int main()
3 {
4     printf("Start\n");
5     goto end;
6     printf("This will not execute\n");
7 end:
8     printf("End");
9     return 0;
10 }
```

Output

```
Start
End
```

Note

The `goto` statement should be avoided in large programs because it makes code difficult to read and maintain.

Part B: Iteration Statements (Loops)

Loops are used to repeat a block of code multiple times.

2.7 while Loop

Definition

The `while` loop executes the block of code as long as the condition is true.

Syntax

```
1 while(condition)
2 {
3     statements;
4 }
```

Flow

```
Condition
|
True
|
Statements
|
Back to Condition
False
|
Exit
```

Example

```
1 #include <stdio.h>
2 int main()
3 {
```

```
4  int i = 1;
5  while(i <= 5)
6  {
7      printf("%d ", i);
8      i++;
9  }
10 return 0;
11 }
```

Output

1 2 3 4 5

2.8 do...while Loop

Definition

The `do-while` loop executes the block **at least once**, even if the condition is false.

Syntax

```
1  do
2  {
3      statements;
4  }
5  while(condition);
```

Example

```
1  #include <stdio.h>
2  int main()
3  {
4      int i = 1;
5      do
6      {
7          printf("%d ", i);
8          i++;
9      }
10     while(i <= 5);
11     return 0;
12 }
```

Output

1 2 3 4 5

Difference Between while and do-while

While	Do-While
Condition checked first	Condition checked after execution
May execute zero times	Executes at least once

2.9 for Loop

Definition

The `for` loop is used when the number of iterations is known in advance.

Syntax

```
1 for(initialization; condition; increment)
2 {
3     statements;
4 }
```

Example

```
1 #include <stdio.h>
2 int main()
3 {
4     int i;
5     for(i = 1; i <= 5; i++)
6     {
7         printf("%d ", i);
8     }
9     return 0;
10 }
```

Output

1 2 3 4 5

2.10 break Statement

Definition

The `break` statement immediately terminates the nearest loop or switch statement.

Example

```
1 #include <stdio.h>
2 int main()
3 {
4     int i;
5     for(i = 1; i <= 10; i++)
6     {
7         if(i == 6)
8             break;
9         printf("%d ", i);
10    }
11    return 0;
12 }
```

Output

1 2 3 4 5

2.11 continue Statement

Definition

The `continue` statement skips the current iteration and moves to the next iteration of the loop.

Example

```
1 #include <stdio.h>
2 int main()
3 {
4     int i;
5     for(i = 1; i <= 5; i++)
6     {
7         if(i == 3)
8             continue;
9         printf("%d ", i);
10    }
11    return 0;
12 }
```

Output

1 2 4 5

2.12 Nested Loops

Definition

A **nested loop** means one loop inside another loop. The outer loop controls how many times the inner loop runs; the inner loop completes all its iterations every time the outer loop executes once.

Syntax

```
1 for(initialization1; condition1; increment1)
2 {
3     for(initialization2; condition2; increment2)
4     {
5         // Statements
6     }
7 }
```

Program

```
1 #include <stdio.h>
2 int main()
3 {
4     int i, j;
5     for(i = 1; i <= 4; i++)
6     {
7         for(j = 1; j <= 4; j++)
8         {
9             printf("* ");
10        }
11        printf("\n");
12    }
```

```
13     return 0;  
14 }
```

Output

```
* * * *  
* * * *  
* * * *  
* * * *
```

Explanation

- The outer loop runs 4 times, so there are 4 rows.
- The inner loop runs 4 times for each row, printing *.
- After each row, `\n` moves to the next line.

Comparison of Loops

Loop	Condition Checked	Executes at Least Once?	Best Used When
while	Before execution	No	Number of iterations is unknown
do-while	After execution	Yes	Loop must execute at least once
for	Before execution	No	Number of iterations is known

Frequently Asked Questions — Module 3

1. What is the difference between `if`, `if-else`, and `else if`?
2. Explain nested `if` with an example.
3. Explain the `switch-case` statement. Why is `break` used?
4. What is the purpose of the `goto` statement? Why is it generally discouraged?
5. Differentiate between `while`, `do-while`, and `for` loops.
6. Explain the difference between `break` and `continue` with examples.
7. What is a nested loop? Write a program to print a simple pattern using nested loops.

3 Module 4 — Arrays and Strings

Part A: Arrays

3.1 What is an Array?

Definition

An **array** is a collection of elements of the same data type stored in contiguous (continuous) memory locations. Each element is accessed using an index.

Advantages

- Stores multiple values using one variable name.
- Easy to access and process data using loops.
- Reduces the number of variables in a program.

Example

```
1 int marks[5];
```

This creates an array named `marks` that can store 5 integers.

3.2 Declaration and Initialization

Declaration

Syntax

```
1 data_type array_name[size];
```

```
1 int marks[5];
```

This declares an array of 5 integer elements.

Initialization — Method 1: During Declaration

```
1 int marks[5] = {80, 85, 90, 75, 88};
```

Method 2: Without Specifying Size

```
1 int marks[] = {80, 85, 90, 75, 88};
```

The compiler automatically determines the array size.

Method 3: Assign Values Later

```
1 int marks[5];  
2 marks[0] = 80;  
3 marks[1] = 85;  
4 marks[2] = 90;  
5 marks[3] = 75;  
6 marks[4] = 88;
```

Example Program

```

1 #include <stdio.h>
2 int main()
3 {
4     int marks[5] = {80, 85, 90, 75, 88};
5     printf("%d", marks[2]);
6     return 0;
7 }

```

Output

90

3.3 Indexing

Definition

Each element of an array has an **index** (position). Indexing always starts from 0.

```

1 int marks[5] = {80, 85, 90, 75, 88};

```

Index	Value
0	80
1	85
2	90
3	75
4	88

Accessing Elements

```

1 printf("%d", marks[0]);

```

Output

80

3.4 Memory Layout

Definition

Array elements are stored in continuous (contiguous) memory locations.

```

1 int marks[5] = {80, 85, 90, 75, 88};

```

Memory Representation

Index	0	1	2	3	4
Value	80	85	90	75	88
Memory	1000	1004	1008	1012	1016

(Assuming an `int` occupies 4 bytes.)

Important Points

- Elements are stored one after another.
- The address of the next element = Previous address + Size of data type.

3.5 Multidimensional Arrays

Definition

A **multidimensional array** is an array containing one or more arrays. The most common type is a 2D array (matrix).

Declaration

```
1 int marks[3][4];
```

This creates 3 rows and 4 columns.

Initialization

```
1 int marks[2][3] =  
2 {  
3     {10,20,30},  
4     {40,50,60}  
5 };
```

Example Program

```
1 #include <stdio.h>  
2 int main()  
3 {  
4     int a[2][3] =  
5     {  
6         {1,2,3},  
7         {4,5,6}  
8     };  
9     printf("%d", a[1][2]);  
10    return 0;  
11 }
```

Output

6

Memory Representation

```
1 2 3  
4 5 6
```

Part B: Strings

3.6 What is a String?

Definition

A **string** is a sequence of characters ending with a null character (`\0`).

```
1 char name[] = "Alice";
```

Internally stored as

```
A l i c e \0
```

3.7 Character Arrays vs Strings

Character Array	String
Collection of characters	Sequence of characters ending with <code>\0</code>
May or may not contain <code>\0</code>	Always ends with <code>\0</code>
Not treated as a complete string	Treated as a complete string
Example: <code>{ 'H', 'i' }</code>	Example: <code>"Hi"</code>

Example

```
1 char ch[2] = {'H', 'i'};
```

This is **not** a proper string because there is no null character.

```
1 char str[] = "Hi";
```

Stored as `'H' 'i' '\0'` — this **is** a valid string.

3.8 Declaring and Initializing Strings

Method 1

```
1 char name[10] = "Alice";
```

Method 2

```
1 char name[] = "Alice";
```

Method 3

```
1 char name[6] = {'A', 'l', 'i', 'c', 'e', '\0'};
```

Example Program

```
1 #include <stdio.h>
2 int main()
3 {
4     char name[] = "Alice";
5     printf("%s", name);
6     return 0;
7 }
```

Output

Alice

3.9 String Input and Output

Output using printf()

```
1 char name[] = "Alice";
2 printf("%s", name);
```

Output

Alice

Input using scanf()

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

Input

Alice

Output

Alice

Note

`scanf("%s", name)` reads only **one word** (it stops at a space).

Input using fgets()

```
1 char name[30];
2 fgets(name, sizeof(name), stdin);
```

Input

Alice Johnson

Output

Alice Johnson

`fgets()` can read spaces and is generally safer than `scanf("%s", ...)`.

3.10 Difference Between scanf() and fgets() in C

Both `scanf()` and `fgets()` are used to take input from the user, but they work differently.

3.10.1 scanf()

Definition

`scanf()` reads input according to the format specifier.

Syntax

```
1 scanf("%s", name);
```

Example

```
1 #include <stdio.h>
2 int main()
3 {
4     char name[20];
5     printf("Enter your name: ");
6     scanf("%s", name);
7     printf("Name: %s", name);
8     return 0;
9 }
```

Input

Alice Johnson

Output

Name: Alice

Why?

`scanf("%s", name)` stops reading when it encounters the first whitespace (space, tab or newline). So Alice Johnson becomes Alice.

3.10.2 fgets()

Definition

`fgets()` reads an entire line of text, including spaces, until:

- a newline (`\n`) is encountered,
- the specified number of characters is reached, or
- the end of the file (EOF) is reached.

Syntax

```
1 fgets(name, sizeof(name), stdin);
```

- `name` → Character array to store the input.

- `sizeof(name)` → Maximum number of characters to read.
- `stdin` → Standard input (keyboard).

Example

```
1 #include <stdio.h>
2 int main()
3 {
4     char name[20];
5     printf("Enter your name: ");
6     fgets(name, sizeof(name), stdin);
7     printf("Name: %s", name);
8     return 0;
9 }
```

Input

Alice Johnson

Output

Name: Alice Johnson

Why is fgets() Safer?

Suppose you declare `char name[10];` — the array can store 9 characters plus the null character (`\0`).

Using scanf():

```
1 scanf("%s", name);
```

If the user enters Christopher, this input is longer than the array. `scanf()` keeps writing characters beyond the array size, which can cause **buffer overflow** (writing outside the allocated memory). This may lead to crashes or unpredictable program behavior.

Using fgets():

```
1 fgets(name, sizeof(name), stdin);
```

If the user enters Christopher, `fgets()` reads at most `sizeof(name) - 1` characters and automatically adds the null character. For `char name[10];` it stores Christoph and ignores the remaining characters. So it prevents writing beyond the array, making it safer.

Example Comparison — Using scanf()

```
1 #include <stdio.h>
2 int main()
3 {
4     char name[30];
5     printf("Enter Name: ");
6     scanf("%s", name);
7     printf("%s", name);
8     return 0;
9 }
```

Input

John Smith

Output

John

Example Comparison — Using fgets()

```

1 #include <stdio.h>
2 int main()
3 {
4     char name[30];
5     printf("Enter Name: ");
6     fgets(name, sizeof(name), stdin);
7     printf("%s", name);
8     return 0;
9 }

```

Input

John Smith

Output

John Smith

Easy Way to Remember**scanf("%s")**

Reads one word

Stops at the first space

Can cause buffer overflow if input is too long

fgets()

Reads a whole line

Continues reading including spaces

Reads only up to the specified buffer size, making it safer

Exam Tip

- Use `scanf("%s")` when you want to read a single word (e.g., a username).
- Use `fgets()` when you want to read a complete sentence or line, or when you want to limit the number of characters read for safer input handling.

3.11 String Library Functions

String functions are declared in the `<string.h>` header file.

```

1 #include <string.h>

```

3.11.1 (a) strlen()

Definition

Returns the length of a string (excluding `\0`).

```
1 #include <stdio.h>
2 #include <string.h>
3 int main()
4 {
5     char str[] = "Hello";
6     printf("%lu", strlen(str));
7     return 0;
8 }
```

Output

5

3.11.2 (b) strcpy()

Definition

Copies one string into another.

```
1 char s1[20];
2 char s2[] = "Hello";
3 strcpy(s1, s2);
4 printf("%s", s1);
```

Output

Hello

3.11.3 (c) strcat()

Definition

Concatenates (joins) two strings.

```
1 char s1[20] = "Hello ";
2 char s2[] = "World";
3 strcat(s1, s2);
4 printf("%s", s1);
```

Output

Hello World

3.11.4 (d) strcmp()

Definition

Compares two strings.

```
1 char s1[] = "Apple";  
2 char s2[] = "Apple";  
3 printf("%d", strcmp(s1, s2));
```

Output

0

0 means both strings are equal.

Important Questions — Module 4

1. What is an array? Explain declaration and initialization with an example.
2. What is array indexing? Why does indexing start from 0?
3. Explain the memory layout of an array.
4. What is a multidimensional array? Write a program to access elements of a 2D array.
5. Differentiate between a character array and a string.
6. How do you declare and initialize a string in C?
7. Explain different methods of string input and output.
8. Write the purpose and syntax of `strlen()`, `strcpy()`, `strcat()`, and `strcmp()`.
9. Write detailed notes on `fgets()`. Difference between `scanf()` and `fgets()`.

4 Supplementary Programs

About this Section

The three programs below apply the ideas from Modules 2–4 (arrays, loops, pointers) to slightly bigger, self-contained examples. To keep them beginner friendly, **no user-defined functions are used** — every program does all of its work inside `main()`.

4.1 Multiplying Two Matrices

Definition

Matrix multiplication combines a 2D array (rows $\times$ columns) with nested `for` loops. To multiply matrix `A` (size $m \times n$) with matrix `B` (size $n \times p$), each element of the result `C[i][j]` is the sum of products of the corresponding row of `A` and column of `B`. This program hard-codes two 2×2 matrices for simplicity.

```

1  #include <stdio.h>
2
3  int main()
4  {
5      /* Two 2 x 2 matrices to multiply */
6      int a[2][2] = { {1, 2},
7                      {3, 4} };
8
9      int b[2][2] = { {5, 6},
10                     {7, 8} };
11
12     int c[2][2]; /* result matrix */
13     int i, j, k;
14
15     /* Compute C = A x B */
16     for(i = 0; i < 2; i++)
17     {
18         for(j = 0; j < 2; j++)
19         {
20             c[i][j] = 0; /* start each cell at 0 */
21             for(k = 0; k < 2; k++)
22             {
23                 c[i][j] = c[i][j] + a[i][k] * b[k][j];
24             }
25         }
26     }
27
28     /* Print matrix A */
29     printf("Matrix A:\n");
30     for(i = 0; i < 2; i++)
31     {
32         for(j = 0; j < 2; j++)
33         {
34             printf("%d ", a[i][j]);
35         }
36         printf("\n");
37     }
38
39     /* Print matrix B */
40     printf("\nMatrix B:\n");
41     for(i = 0; i < 2; i++)

```

```

42     {
43         for(j = 0; j < 2; j++)
44         {
45             printf("%d ", b[i][j]);
46         }
47         printf("\n");
48     }
49
50     /* Print result matrix C */
51     printf("\nProduct Matrix (A x B):\n");
52     for(i = 0; i < 2; i++)
53     {
54         for(j = 0; j < 2; j++)
55         {
56             printf("%d ", c[i][j]);
57         }
58         printf("\n");
59     }
60
61     return 0;
62 }

```

Output

Matrix A:

1 2
3 4

Matrix B:

5 6
7 8

Product Matrix (A x B):

19 22
43 50**Explanation**

- The outer two loops (`i` , `j`) walk over every cell of the result matrix `c` .
- The innermost loop (`k`) computes the dot product of row `i` of `a` with column `j` of `b` .
- `c[i][j]` is reset to `0` before each dot product is accumulated.

4.2 Swapping Two Array Elements — The Normal Way**Definition**

Swapping two array elements means exchanging their values using a **temporary variable**. This is the direct, value-based approach: the actual numbers stored at the two index positions are copied around.

```

1  #include <stdio.h>
2
3  int main()
4  {
5      int arr[5] = {10, 20, 30, 40, 50};
6      int i;

```

```
7   int temp;
8
9   printf("Array before swapping:\n");
10  for(i = 0; i < 5; i++)
11  {
12      printf("%d ", arr[i]);
13  }
14  printf("\n");
15
16  /* Swap the element at index 0 with the element at index 4
17   using an ordinary temporary variable */
18  temp = arr[0];
19  arr[0] = arr[4];
20  arr[4] = temp;
21
22  printf("\nArray after swapping arr[0] and arr[4]:\n");
23  for(i = 0; i < 5; i++)
24  {
25      printf("%d ", arr[i]);
26  }
27  printf("\n");
28
29  return 0;
30 }
```

Output

```
Array before swapping:
10 20 30 40 50

Array after swapping arr[0] and arr[4]:
50 20 30 40 10
```

Explanation

- `temp` borrows the value of `arr[0]` so it is not lost.
- `arr[0]` is overwritten with `arr[4]`.
- `arr[4]` is overwritten with the value saved in `temp`.
- Three assignment statements, in that exact order, are what make a clean swap possible.

4.3 Swapping the Same Two Elements — Using Pointers (Address Swap)

Definition

Instead of moving values, this version stores the **addresses** of the two array elements in two pointer variables and swaps through those pointers. Conceptually the values still move, but the code is written in terms of "where the data lives" rather than the data itself — this is the idea behind `swap-by-address`, which is how real `scanf()`-style and call-by-reference operations work in C.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int arr[5] = {10, 20, 30, 40, 50};
6     int i;
```

```

7   int temp;
8
9   /* Pointers that store the ADDRESS of the two elements
10      we want to swap, instead of their values */
11   int *p1 = &arr[0];
12   int *p2 = &arr[4];
13
14   printf("Array before swapping:\n");
15   for(i = 0; i < 5; i++)
16   {
17       printf("%d ", arr[i]);
18   }
19   printf("\n");
20
21   printf("\nAddress stored in p1 (points to arr[0]): %p\n", (void *)p1);
22   printf("Address stored in p2 (points to arr[4]): %p\n", (void *)p2);
23
24   /* Swap the VALUES found at those two addresses.
25      We dereference the pointers with * to reach the
26      actual int stored in the array. */
27   temp = *p1;
28   *p1 = *p2;
29   *p2 = temp;
30
31   printf("\nArray after swapping through pointers:\n");
32   for(i = 0; i < 5; i++)
33   {
34       printf("%d ", arr[i]);
35   }
36   printf("\n");
37
38   return 0;
39 }

```

Output

Array before swapping:
10 20 30 40 50

Address stored in p1 (points to arr[0]): 0x7ffd3a2c1a70
Address stored in p2 (points to arr[4]): 0x7ffd3a2c1a80

Array after swapping through pointers:
50 20 30 40 10

Note

The actual addresses printed by `%p` will be different every time you run the program — they depend on where the operating system places the array in memory. What matters is that `p1` and `p2` always point at `arr[0]` and `arr[4]`.

Explanation

- `int *p1 = &arr[0];` makes `p1` store the **address** of `arr[0]`, not the value 10.
- `*p1` (the **dereference** operator) means "the value at the address stored in `p1`" — this is how we reach back into the array through the pointer.

- The three-line swap (`temp = *p1; *p1 = *p2; *p2 = temp;`) is exactly the same pattern as the normal swap, except every access goes through a pointer first.
- The end result on the array is identical to the previous program — only the mechanism used to reach the data is different.

Normal Swap vs Pointer Swap

Normal Swap	Pointer (Address) Swap
Works directly on <code>arr[0], arr[4]</code>	Works on <code>*p1, *p2</code>
Temporary variable stores a value	Temporary variable still stores a value, but it is reached via a pointer
No addresses involved in the logic	Explicitly stores and uses memory addresses
Simpler for beginners	Mirrors how call-by-reference and <code>scanf()</code> work internally

4.4 Train Signalling System Simulation

Definition

This program models a very simple single-track railway line as an **array buffer** of fixed size, where each index of the array represents one **block** of track. A value of `0` means a block is empty, and any other number represents the ID of the train currently occupying that block. On every simulated time-tick, each train moves one block forward, and the program prints:

- the current position of every block in the buffer (the "track view"), and
- the signal — **RED** or **GREEN** — for every block, based on a simple rule: a block shows **RED** if it is occupied (or if the very next block ahead is occupied, to prevent a collision), and **GREEN** otherwise.

No functions are used — the whole simulation runs inside `main()` using only arrays and loops.

```

1  #include <stdio.h>
2
3  int main()
4  {
5      /* The track is modelled as a buffer of 8 blocks (index 0 to 7).
6       * 0 means the block is empty.
7       * Any other number is the ID of the train sitting in that block. */
8      int track[8] = {0, 0, 0, 0, 0, 0, 0, 0};
9
10     int totalTicks = 6; /* how many time-steps to simulate */
11     int tick;
12     int i;
13
14     /* Two trains enter the track at the start:
15      * Train 1 enters at block 0, Train 2 enters at block 2 */
16     track[0] = 1; /* Train 1 */
17     track[2] = 2; /* Train 2 */
18
19     printf("==== TRAIN SIGNALLING SYSTEM SIMULATION =====\n");
20     printf("Track has 8 blocks, indexed 0 to 7.\n");
21     printf("0 = empty block, N = block occupied by Train N\n\n");
22
23     for(tick = 1; tick <= totalTicks; tick++)
24     {
25         printf("-----\n");
26         printf("Tick %d\n", tick);
27
28         /* Move every train forward by ONE block.
29          * We must scan from the LAST block down to the FIRST
30          * block, so that a train does not immediately collide
31          * with / overwrite the train that just moved out of the
32          * block ahead of it during the SAME tick. */
33         for(i = 7; i >= 1; i--)
34         {
35             if(track[i - 1] != 0 && track[i] == 0)
36             {
37                 track[i] = track[i - 1]; /* train moves into block i */
38                 track[i - 1] = 0; /* block it left becomes empty */
39             }
40         }
41
42         /* A train that reaches the last block simply leaves the
43          * track buffer on the following tick (single-line exit). */
44         if(track[7] != 0)

```

```

45     {
46         printf("Train %d has left the track from block 7.\n", track[7]);
47         track[7] = 0;
48     }
49
50     /* Print the current position of every block (the buffer) */
51     printf("Block:   ");
52     for(i = 0; i < 8; i++)
53     {
54         printf("%2d ", i);
55     }
56     printf("\n");
57
58     printf("Train:   ");
59     for(i = 0; i < 8; i++)
60     {
61         if(track[i] == 0)
62             printf(" . ");
63         else
64             printf("T%d ", track[i]);
65     }
66     printf("\n");
67
68     /* Print the signal for every block:
69        RED    -> block is occupied, OR the block right ahead
70                of it is occupied (to avoid a rear-end collision)
71        GREEN -> block and the block ahead are both clear */
72     printf("Signal: ");
73     for(i = 0; i < 8; i++)
74     {
75         if(track[i] != 0)
76         {
77             printf("RED ");
78         }
79         else if(i < 7 && track[i + 1] != 0)
80         {
81             printf("RED ");
82         }
83         else
84         {
85             printf("GRN ");
86         }
87     }
88     printf("\n");
89 }
90
91 printf("-----\n");
92 printf("Simulation finished after %d ticks.\n", totalTicks);
93
94 return 0;
95 }

```

Output

```

===== TRAIN SIGNALLING SYSTEM SIMULATION =====
Track has 8 blocks, indexed 0 to 7.
0 = empty block, N = block occupied by Train N

```

```

-----
Tick 1
Block:  0 1 2 3 4 5 6 7

```

```

Train:  T1 .  T2 .  .  .  .  .
Signal:  RED RED RED RED GRN GRN GRN GRN

(... further ticks follow the same pattern as the two trains
advance one block per tick, each showing RED on its own block
and on the block immediately behind it ...)

-----
Simulation finished after 6 ticks.

```

Explanation

- `track[8]` is the **array buffer** representing the physical line; its index is the block number and its value is which train (if any) occupies that block.
- Trains are advanced with a single `for` loop that walks **backwards** through the buffer (`i = 7` down to `i = 1`); moving backwards stops a train from "catching up" and overwriting another train within the same tick.
- A train that walks off the end of the buffer (block 7) is treated as having left the track and the block is cleared to `0`.
- The **signal rule** is intentionally simple and mirrors real block signalling: a signal is **RED** if a train is standing on that block *or* on the very next block ahead (so a following train cannot be waved into a block that leads straight into another train), and **GREEN** otherwise.
- Everything — movement, printing the buffer, and computing signals — is done with simple `for` loops and `if` conditions directly inside `main()`, using only the ideas covered in Modules 2–4.

Note

Try it yourself: change `totalTicks`, the starting positions of the two trains, or add a third train (e.g. `track[5] = 3;`) and trace through the signal output by hand before running the program.