



GANDHI SCHOOL OF ENGINEERING

Brahmapur, Odisha

Laboratory Manual

Programming in C Laboratory

Complete Laboratory Manual

B.Tech 1st Year · Semester 1 · Circuit Branch

Academic Year 2026-27



About this lab manual

This manual covers the **Programming in C and Data Structures Laboratory** (0-0-2). It contains 24 experiments, from basic input/output to stacks, queues and linked lists. For every experiment it gives the aim, **theory with a clear explanation**, the algorithm or method, a **complete program**, the **actual output** of the program (every program in this manual was compiled with GCC and run; the outputs shown are the real results), an explanation of how the program works, and viva-voce questions with answers.

No.	Experiment	Concepts
1	Input, output and arithmetic expressions	printf, scanf, operators, math.h
2	Operators and conditional expressions	swap, ternary operator
3	Roots of a quadratic equation	if-else ladder
4	Menu-driven calculator	switch
5	Digits of a number: sum, reverse, palindrome, Armstrong	while loop
6	Primes, factorial and Fibonacci series	for loop, nested loops
7	Pattern printing	nested loops
8	Array statistics and linear search	1-D arrays
9	Bubble sort and selection sort	sorting
10	Binary search	searching
11	Matrix transpose and multiplication	2-D arrays
12	String operations without library functions	strings
13	String library functions and sorting names	string.h
14	GCD and LCM; call by value and reference	functions
15	Factorial, Fibonacci and Tower of Hanoi	recursion
16	Pointer basics and pointer arithmetic	pointers
17	malloc, realloc and free	dynamic memory
18	Student records	structures
19	Writing and reading a file; counting characters, words and lines	files
20	Stack using an array	data structures
21	Circular queue using an array	data structures
22	Singly linked list	data structures



No.	Experiment	Concepts
23	Insertion sort and quick sort	sorting, recursion
24	Mini project: bank account system	modular programming

Steps of program development

- **Understand the problem** and identify the inputs, outputs and formulas.
- Write the **algorithm** (step-by-step method) or draw a **flowchart**.
- **Code** the program in C with meaningful variable names, comments and indentation.
- **Compile** (gcc prog.c -o prog -Wall -lm) and correct syntax errors.
- **Test** with several inputs, including special cases (zero, negative numbers, empty input, largest values), and correct logical errors.
- **Document** the program in the record: aim, algorithm, program, output and result.

Flowchart symbols

Symbol	Name	Use
Oval (terminal)	Start / Stop	Beginning and end of the program
Parallelogram	Input / Output	Reading or printing data
Rectangle	Process	Calculations, assignments
Diamond	Decision	A condition with Yes/No branches
Arrow	Flow line	Order of steps
Small circle	Connector	Joins parts of a flowchart



Contents

About this lab manual	2
Programming in C and Data Structures Laboratory	6
Experiment 1: Input, Output and Arithmetic Expressions	6
Experiment 2: Operators and Conditional Expressions	9
Experiment 3: Decision Making — Roots of a Quadratic Equation	11
Experiment 4: switch Statement — Menu-Driven Calculator	13
Experiment 5: Loops — Digits of a Number	15
Experiment 6: Prime Numbers, Factorial and Fibonacci Series	17
Experiment 7: Pattern Printing with Nested Loops	19
Experiment 8: One-Dimensional Arrays — Statistics and Linear Search	21
Experiment 9: Sorting — Bubble Sort and Selection Sort	23
Experiment 10: Binary Search	25
Experiment 11: Two-Dimensional Arrays — Matrix Operations	27
Experiment 12: Strings — Manual Operations	29
Experiment 13: String Library Functions	31
Experiment 14: User-Defined Functions — GCD, LCM and Call by Value/Reference .	34
Experiment 15: Recursion — Factorial, Fibonacci and Tower of Hanoi	37
Experiment 16: Pointers	40
Experiment 17: Dynamic Memory Allocation	42
Experiment 18: Structures — Student Records	45
Experiment 19: File Handling	48
Experiment 20: Stack using an Array	51
Experiment 21: Queue using an Array (Circular Queue)	54



Experiment 22: Singly Linked List	57
Experiment 23: Insertion Sort and Quick Sort	61
Experiment 24: Mini Project — Simple Bank Account System	64
Appendix: Compiling and Running C Programs	67



Programming in C and Data Structures Laboratory

24 experiments · Algorithm · Explanation · Tested programs · Real outputs · Viva

Experiment 1: Input, Output and Arithmetic Expressions

Aim

To write C programs that read data from the keyboard, evaluate arithmetic expressions and display the results: (a) area and circumference of a circle, (b) simple and compound interest, (c) temperature conversion.

Theory (with explanation)

Every C program is a collection of functions; execution starts at **main()**. The header **stdio.h** declares the input/output functions **printf()** (formatted output) and **scanf()** (formatted input).

- **Format specifiers** tell printf/scanf the type of data: %d (int), %f (float), %lf (double in scanf), %c (char), %s (string). %.2f prints two decimal places.
- scanf needs the **address** of the variable (&r), because it must store the value there.
- **Arithmetic operators:** + – * / %. Integer division truncates ($7/2 = 3$); at least one operand must be real for real division ($7.0/2 = 3.5$).
- **Math library (math.h):** pow(x, y), sqrt(x), fabs(x) etc. Link with -lm on Linux: gcc prog.c -lm.
- **#define** creates a symbolic constant (PI) that the preprocessor substitutes before compilation.

Algorithm (circle)

- Step 1: Start. Define $PI = 3.14159$.
- Step 2: Read the radius r.
- Step 3: $area = PI \times r \times r$; $circumference = 2 \times PI \times r$.
- Step 4: Print area and circumference. Step 5: Stop.

Program:

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

int main(void)
{
    float r, area, circum;
    printf("Enter the radius of the circle: ");
    scanf("%f", &r);
    area = PI * r * r;
    circum = 2 * PI * r;
    printf("Area = %.2f square units\n", area);
    printf("Circumference = %.2f units\n", circum);
    return 0;
}
```

Output (sample run):



```
Enter the radius of the circle: 7.5
Area = 176.71 square units
Circumference = 47.12 units
```

Explanation: the radius is read as a float. The expressions use the constant PI; the results are printed with two decimal places. For $r = 7.5$, $\text{area} = 3.14159 \times 56.25 = 176.71$.

Program:

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

int main(void)
{
    double p, r, t, si, ci;
    printf("Enter principal: ");
    scanf("%lf", &p);
    printf("Enter rate of interest (% per year): ");
    scanf("%lf", &r);
    printf("Enter time (years): ");
    scanf("%lf", &t);
    si = p * r * t / 100;
    ci = p * pow(1 + r / 100, t) - p;
    printf("Simple interest = Rs %.2f\n", si);
    printf("Compound interest = Rs %.2f\n", ci);
    return 0;
}
```

Output (sample run):

```
Enter principal: 10000
Enter rate of interest (% per year): 8.5
Enter time (years): 3
Simple interest = Rs 2550.00
Compound interest = Rs 2772.89
```

Explanation: simple interest = $PRT/100$. Compound interest = $P(1 + R/100)^T - P$, computed with `pow()` from `math.h`. To print a % sign inside `printf` we write `%%`.

Program:

```
#include <stdio.h>

int main(void)
{
    float f, c;
    printf("Enter temperature in Fahrenheit: ");
    scanf("%f", &f);
    c = (f - 32) * 5 / 9;
    printf("%.1f F = %.2f C\n", f, c);
    printf("In Kelvin: %.2f K\n", c + 273.15);
    return 0;
}
```

Output (sample run):

```
Enter temperature in Fahrenheit: 98.6
98.6 F = 37.00 C
In Kelvin: 310.15 K
```



Explanation: $C = (F - 32) \times 5/9$. Because f is a float, the whole expression is evaluated in real arithmetic. (Writing $5/9 * (f - 32)$ would give 0, since $5/9$ is integer division!)

Viva-voce questions

Q1. Why is & used in scanf but not in printf?

scanf must store a value, so it needs the address of the variable; printf only needs the value.

Q2. What is the output of printf("%d", 7/2)?

3, because both operands are integers (integer division truncates).

Q3. What is the difference between #define and const?

#define is a preprocessor text substitution with no type; const creates a typed read-only variable checked by the compiler.

Q4. What are header files?

Files (.h) containing declarations of library functions and macros, included with #include, e.g. stdio.h, math.h, string.h.

Experiment 2: Operators and Conditional Expressions

Aim

(a) To swap two numbers with and without a third variable. (b) To find the largest of three numbers using the conditional (ternary) operator.

Theory (with explanation)

- **Assignment** $a = b$ copies the value of b into a . To swap two values, a temporary variable holds one value while the other is copied.
- Swapping **without a third variable** uses arithmetic: $a = a + b$; $b = a - b$; $a = a - b$. (It can overflow for very large values; the XOR method $a \wedge = b$; $b \wedge = a$; $a \wedge = b$ avoids that for integers.)
- **Relational operators** ($<$, $>$, $<=$, $>=$, $==$, $!=$) give 1 (true) or 0 (false). **Logical operators**: $\&\&$ (and), $\|\|$ (or), $!$ (not).
- **Conditional operator** $\text{condition} ? \text{expr1} : \text{expr2}$ evaluates expr1 if the condition is true, else expr2 . It is a compact form of if-else.

Program:

```
#include <stdio.h>

int main(void)
{
    int a, b, temp;
    printf("Enter a: ");
    scanf("%d", &a);
    printf("Enter b: ");
    scanf("%d", &b);

    temp = a;          /* method 1: using a third variable */
    a = b;
    b = temp;
    printf("After swap (temp):    a = %d, b = %d\n", a, b);

    a = a + b;          /* method 2: without a third variable */
    b = a - b;
    a = a - b;
    printf("After swapping back:  a = %d, b = %d\n", a, b);
    return 0;
}
```

Output (sample run):

```
Enter a: 15
Enter b: 40
After swap (temp):    a = 40, b = 15
After swapping back:  a = 15, b = 40
```

Explanation: in method 2, with $a = 40$ and $b = 15$: $a = 55$, $b = 55 - 15 = 40$, $a = 55 - 40 = 15$, so the values are exchanged without extra memory.

Program:



```
#include <stdio.h>

int main(void)
{
    int a, b, c, big;
    printf("Enter three numbers: ");
    scanf("%d", &a);
    scanf("%d", &b);
    scanf("%d", &c);
    big = (a > b) ? ((a > c) ? a : c) : ((b > c) ? b : c);
    printf("Largest = %d\n", big);
    return 0;
}
```

Output (sample run):

```
Enter three numbers: 23
71
45
Largest = 71
```

Explanation: the outer condition compares a and b; the winner is then compared with c using a nested conditional operator.

Viva-voce questions

Q1. What is the difference between = and ==?

= is assignment; == compares two values for equality.

Q2. What is the value of 5 > 3 && 2 > 4?

0 (false), because the second condition is false.

Q3. What is short-circuit evaluation?

In a && b, b is not evaluated if a is false; in a || b, b is not evaluated if a is true.

Q4. What is the difference between i++ and ++i?

i++ uses the current value and then increments; ++i increments first and then uses the new value.

Experiment 3: Decision Making — Roots of a Quadratic Equation

Aim

To find the roots of a quadratic equation $ax^2 + bx + c = 0$ for all cases (real and distinct, real and equal, complex).

Theory (with explanation)

The nature of the roots depends on the **discriminant** $d = b^2 - 4ac$:

Discriminant	Nature of roots	Roots
$d > 0$	Real and distinct	$(-b \pm \sqrt{d})/2a$
$d = 0$	Real and equal	$-b/2a$
$d < 0$	Complex conjugate	$-b/2a \pm i\sqrt{(-d)}/2a$

The **if-else if-else ladder** checks conditions one by one and executes only the block of the first true condition. If $a = 0$ the equation is not quadratic, which is checked first.

Algorithm

- Step 1: Read a, b, c . If $a = 0$, print "not quadratic" and stop.
- Step 2: $d = b^2 - 4ac$.
- Step 3: If $d > 0$, $r1 = (-b + \sqrt{d})/2a$, $r2 = (-b - \sqrt{d})/2a$. Else if $d = 0$, $r1 = r2 = -b/2a$. Else real part = $-b/2a$, imaginary part = $\sqrt{(-d)}/2a$.
- Step 4: Print the roots and stop.

Program:

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

int main(void)
{
    double a, b, c, d, r1, r2, re, im;
    printf("Enter coefficients a, b, c: ");
    scanf("%lf %lf %lf", &a, &b, &c);
    if (a == 0) {
        printf("Not a quadratic equation.\n");
        return 0;
    }
    d = b * b - 4 * a * c;
    if (d > 0) {
        r1 = (-b + sqrt(d)) / (2 * a);
        r2 = (-b - sqrt(d)) / (2 * a);
        printf("Real and distinct roots: %.3f and %.3f\n", r1, r2);
    } else if (d == 0) {
        r1 = -b / (2 * a);
        printf("Real and equal roots: %.3f and %.3f\n", r1, r1);
    } else {
        re = -b / (2 * a);
        im = sqrt(-d) / (2 * a);
        printf("Complex roots: %.3f + i%.3f and %.3f - i%.3f\n", re, im, re, im);
    }
    return 0;
}
```

Output — run 1:



Enter coefficients a, b, c: 1 -5 6
Real and distinct roots: 3.000 and 2.000

Output — run 2:

Enter coefficients a, b, c: 1 2 5
Complex roots: -1.000 + i2.000 and -1.000 - i2.000

Output — run 3:

Enter coefficients a, b, c: 4 4 1
Real and equal roots: -0.500 and -0.500

Explanation: three sample runs are shown: $x^2 - 5x + 6 = 0$ ($d = 1$, roots 3 and 2), $x^2 + 2x + 5 = 0$ ($d = -16$, roots $-1 \pm 2i$) and $4x^2 + 4x + 1 = 0$ ($d = 0$, roots $-0.5, -0.5$).

Viva-voce questions

Q1. What is a dangling else problem?

In nested if statements without braces, an else pairs with the nearest unmatched if; braces should be used to make the intended pairing clear.

Q2. Why is it risky to compare floating-point numbers with ==?

Real numbers are stored approximately, so tiny rounding errors can make equal values appear unequal; compare $\text{fabs}(x - y) < \epsilon$ instead.

Q3. What does sqrt() return for a negative argument?

NaN (not a number), which is why the program handles $d < 0$ separately.

Experiment 4: switch Statement — Menu-Driven Calculator

Aim

To write a menu-driven calculator that performs addition, subtraction, multiplication, division and modulus using the switch statement.

Theory (with explanation)

The **switch** statement selects one of many blocks according to the value of an integer or character expression:

- Each **case** label must be a constant. Control jumps to the matching case and continues until a **break** (without break, execution "falls through" into the next case).
- The **default** block runs when no case matches.
- switch is clearer and often faster than a long if-else ladder when testing one variable against many constant values.

Program:

```
#include <stdio.h>

int main(void)
{
    double a, b;
    char op;
    printf("Enter first number: ");
    scanf("%lf", &a);
    printf("Enter operator (+ - * / %): ");
    scanf(" %c", &op);
    printf("Enter second number: ");
    scanf("%lf", &b);

    switch (op) {
        case '+': printf("%.2f + %.2f = %.2f\n", a, b, a + b); break;
        case '-': printf("%.2f - %.2f = %.2f\n", a, b, a - b); break;
        case '*': printf("%.2f * %.2f = %.2f\n", a, b, a * b); break;
        case '/':
            if (b == 0) printf("Error: division by zero\n");
            else printf("%.2f / %.2f = %.2f\n", a, b, a / b);
            break;
        case '%':
            if ((int)b == 0) printf("Error: modulus by zero\n");
            else printf("%d %% %d = %d\n", (int)a, (int)b, (int)a % (int)b);
            break;
        default:
            printf("Invalid operator\n");
    }
    return 0;
}
```

Output — run 1:

```
Enter first number: 84
Enter operator (+ - * / %): /
Enter second number: 5
84.00 / 5.00 = 16.80
```

Output — run 2:



```
Enter first number: 17
Enter operator (+ - * / %): %
Enter second number: 5
17 % 5 = 2
```

Output — run 3:

```
Enter first number: 9
Enter operator (+ - * / %): ^
Enter second number: 2
Invalid operator
```

Explanation: the space before %c in scanf(" %c", &op) skips the newline left in the input buffer by the previous scanf. The % operator works only on integers, so the operands are cast to int. The third run shows the default case.

Viva-voce questions

Q1. What happens if break is omitted in a case?

Control falls through and executes the statements of the following cases too.

Q2. Can a switch use float values or strings as case labels?

No; case labels must be integer (or character) constants.

Q3. What is the use of the default case?

It handles all values that do not match any case (like the final else of an if ladder).

Experiment 5: Loops — Digits of a Number

Aim

To find the sum of the digits and the reverse of a number and to check whether it is a palindrome and an Armstrong number.

Theory (with explanation)

A **loop** repeats a block of statements while a condition is true. C has three loops:

- **while (condition) { ... }** — entry-controlled; used when the number of repetitions is not known in advance.
- **do { ... } while (condition);** — exit-controlled; the body executes at least once.
- **for (init; condition; update) { ... }** — compact form for counting loops.

Extracting digits: $n \% 10$ gives the last digit and $n / 10$ removes it. Repeating until n becomes 0 visits every digit. The reverse is built as $rev = rev \times 10 + digit$.

- A **palindrome number** reads the same backwards (e.g. 12321).
- An **Armstrong number** equals the sum of its digits each raised to the power of the number of digits (e.g. $153 = 1^3 + 5^3 + 3^3$).

Program:

```
#include <stdio.h>

int main(void)
{
    long n, temp, rev = 0, sum = 0, arm = 0;
    int digit, count = 0, i;
    long p;
    printf("Enter a positive integer: ");
    scanf("%ld", &n);

    temp = n;
    while (temp > 0) {                /* count digits */
        count++;
        temp /= 10;
    }
    temp = n;
    while (temp > 0) {
        digit = temp % 10;           /* last digit */
        sum += digit;
        rev = rev * 10 + digit;
        p = 1;
        for (i = 0; i < count; i++) /* digit ^ count */
            p *= digit;
        arm += p;
        temp /= 10;                 /* remove last digit */
    }
    printf("Number of digits = %d\n", count);
    printf("Sum of digits    = %ld\n", sum);
    printf("Reverse        = %ld\n", rev);
    printf("%ld %s a palindrome\n", n, (rev == n) ? "is" : "is not");
    printf("%ld %s an Armstrong number\n", n, (arm == n) ? "is" : "is not");
    return 0;
}
```

Output — run 1:



```
Enter a positive integer: 12321
Number of digits = 5
Sum of digits    = 9
Reverse         = 12321
12321 is a palindrome
12321 is not an Armstrong number
```

Output — run 2:

```
Enter a positive integer: 153
Number of digits = 3
Sum of digits    = 9
Reverse         = 351
153 is not a palindrome
153 is an Armstrong number
```

Trace for n = 153: digits 3, 5, 1 are taken in turn; sum = 9, rev = 351, and $3^3 + 5^3 + 1^3 = 27 + 125 + 1 = 153$, so it is an Armstrong number.

Viva-voce questions

Q1. What is the difference between while and do-while?

while tests the condition before the body (may execute zero times); do-while tests after the body (executes at least once).

Q2. What are break and continue?

break exits the loop immediately; continue skips the rest of the current iteration and starts the next one.

Q3. What is an infinite loop? Give an example.

A loop whose condition never becomes false, e.g. while (1) { } or for (;;) { }; it is ended with break.

Experiment 6: Prime Numbers, Factorial and Fibonacci Series

Aim

(a) To print all prime numbers up to n. (b) To find the factorial of a number and print the first n terms of the Fibonacci series.

Theory (with explanation)

- A **prime number** has exactly two factors, 1 and itself. To test n, it is enough to check divisors from 2 up to $\sqrt{n}$: if n had a factor larger than $\sqrt{n}$, the matching factor would be smaller than $\sqrt{n}$.
- **Factorial:** $n! = 1 \times 2 \times \dots \times n$ ($0! = 1$). It grows very fast; $20!$ is the largest that fits in a 64-bit unsigned long long.
- **Fibonacci series:** each term is the sum of the two previous terms: 0, 1, 1, 2, 3, 5, 8 ...
- **Nested loops:** a loop inside another; the inner loop runs completely for each iteration of the outer loop.

Program:

```
#include <stdio.h>

int main(void)
{
    int n, i, j, isPrime, count = 0;
    printf("Print primes up to: ");
    scanf("%d", &n);
    for (i = 2; i <= n; i++) {
        isPrime = 1;
        for (j = 2; j * j <= i; j++) {
            if (i % j == 0) {
                isPrime = 0;
                break;
            }
        }
        if (isPrime) {
            printf("%d ", i);
            count++;
        }
    }
    printf("\nThere are %d primes up to %d.\n", count, n);
    return 0;
}
```

Output (sample run):

```
Print primes up to: 50
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
There are 15 primes up to 50.
```

Explanation: the inner loop stops as soon as a divisor is found (break), and runs only while $j \times j \leq i$, which makes the test fast.

Program:



```
#include <stdio.h>

int main(void)
{
    int n, i;
    unsigned long long fact = 1, a = 0, b = 1, c;
    printf("Enter n: ");
    scanf("%d", &n);
    for (i = 1; i <= n; i++)
        fact *= i;
    printf("%d! = %llu\n", n, fact);

    printf("First %d Fibonacci terms: ", n);
    for (i = 1; i <= n; i++) {
        printf("%llu ", a);
        c = a + b;
        a = b;
        b = c;
    }
    printf("\n");
    return 0;
}
```

Output (sample run):

```
Enter n: 10
10! = 3628800
First 10 Fibonacci terms: 0 1 1 2 3 5 8 13 21 34
```

Explanation: fact starts at 1 and is multiplied by 1, 2 ... n. For Fibonacci, a and b hold two consecutive terms; after printing a, they move one step forward.

Viva-voce questions

Q1. Why do we check divisors only up to $\sqrt{n}$?

Factors occur in pairs (d, n/d); one of each pair is at most $\sqrt{n}$, so if none is found up to $\sqrt{n}$, n is prime.

Q2. Is 1 a prime number?

No; a prime must have exactly two distinct factors.

Q3. What data type is used for large factorials and why?

unsigned long long (64 bits, up to about 1.8×10^{19}), since int overflows after 12!.

Experiment 7: Pattern Printing with Nested Loops

Aim

To print number and star patterns (right triangle, pyramid and Floyd's triangle) using nested loops.

Theory (with explanation)

In pattern programs the **outer loop controls the rows** and the **inner loops control what is printed in each row** (spaces and symbols). For a pyramid of n rows, row i has $(n - i)$ leading spaces and $(2i - 1)$ stars. Floyd's triangle prints consecutive natural numbers, row i having i numbers.

Program:

```
#include <stdio.h>

int main(void)
{
    int n, i, j, k = 1;
    printf("Enter number of rows: ");
    scanf("%d", &n);

    printf("\nRight triangle of numbers:\n");
    for (i = 1; i <= n; i++) {
        for (j = 1; j <= i; j++)
            printf("%d ", j);
        printf("\n");
    }

    printf("\nPyramid of stars:\n");
    for (i = 1; i <= n; i++) {
        for (j = 1; j <= n - i; j++)
            printf(" ");
        for (j = 1; j <= 2 * i - 1; j++)
            printf("*");
        printf("\n");
    }

    printf("\nFloyd's triangle:\n");
    for (i = 1; i <= n; i++) {
        for (j = 1; j <= i; j++)
            printf("%3d", k++);
        printf("\n");
    }
    return 0;
}
```

Output (sample run):



Enter number of rows: 5

Right triangle of numbers:

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

Pyramid of stars:

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

Floyd's triangle:

```
1
2 3
4 5 6
7 8 9 10
11 12 13 14 15
```

Explanation: for the pyramid with $n = 5$, row 3 prints 2 spaces and 5 stars. In Floyd's triangle the counter k is not reset, so numbering continues from row to row.

Viva-voce questions

Q1. How many times does the inner loop run in total for the right triangle of n rows?

$1 + 2 + \dots + n = n(n + 1)/2$ times.

Q2. What does %3d do in printf?

Prints the integer right-aligned in a field 3 characters wide.

Experiment 8: One-Dimensional Arrays — Statistics and Linear Search

Aim

To read n numbers into an array, find their sum, mean, maximum and minimum, and search for an element by linear search.

Theory (with explanation)

An **array** is a collection of elements of the same type stored in **contiguous memory locations** and accessed by an index: `int a[100]`; declares 100 integers `a[0] ... a[99]`. The index starts at 0, so the last element is `a[n - 1]`. C does not check array bounds — accessing `a[n]` or beyond is a serious error.

Linear search compares the key with each element from the start until it is found or the array ends. It works on unsorted data; its time complexity is $O(n)$ (n comparisons in the worst case).

Program:

```
#include <stdio.h>

int main(void)
{
    int a[100], n, i, key, pos = -1, max, min, sum = 0;
    printf("How many numbers? ");
    scanf("%d", &n);
    printf("Enter %d numbers: ", n);
    for (i = 0; i < n; i++)
        scanf("%d", &a[i]);

    max = min = a[0];
    for (i = 0; i < n; i++) {
        sum += a[i];
        if (a[i] > max) max = a[i];
        if (a[i] < min) min = a[i];
    }
    printf("Sum = %d, Mean = %.2f\n", sum, (float)sum / n);
    printf("Maximum = %d, Minimum = %d\n", max, min);

    printf("Enter element to search: ");
    scanf("%d", &key);
    for (i = 0; i < n; i++) {
        if (a[i] == key) {
            pos = i;
            break;
        }
    }
    if (pos >= 0)
        printf("%d found at position %d (index %d)\n", key, pos + 1, pos);
    else
        printf("%d not found\n", key);
    return 0;
}
```

Output (sample run):



```
How many numbers? 7
Enter 7 numbers: 45 12 78 34 89 23 56
Sum = 337, Mean = 48.14
Maximum = 89, Minimum = 12
Enter element to search: 34
34 found at position 4 (index 3)
```

Explanation: max and min are initialised to the first element (not to 0), so the program also works for negative numbers. The cast (float)sum gives a real mean.

Viva-voce questions

Q1. What is the index of the first element of an array?

0.

Q2. How is an array stored in memory?

In contiguous locations; the address of $a[i]$ is base address + $i \times \text{sizeof}(\text{element})$.

Q3. What is the time complexity of linear search?

$O(n)$ in the worst and average case, $O(1)$ in the best case.

Experiment 9: Sorting — Bubble Sort and Selection Sort

Aim

To sort an array of integers in ascending order using bubble sort and selection sort, showing the array after each pass.

Theory (with explanation)

Bubble sort: adjacent elements are compared and swapped if they are in the wrong order. After pass 1 the largest element "bubbles" to the last position; after pass k the last k elements are in place. A flag detects a pass with no swaps, which means the array is already sorted (best case $O(n)$). Worst case: $n(n - 1)/2$ comparisons, $O(n^2)$.

Selection sort: in pass i, the smallest element of the unsorted part $a[i \dots n - 1]$ is found and swapped with $a[i]$. It makes at most $n - 1$ swaps (useful when swapping is costly); comparisons are always $n(n - 1)/2$, $O(n^2)$.

Program:

```
#include <stdio.h>

void print(int a[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        printf("%4d", a[i]);
    printf("\n");
}

int main(void)
{
    int a[50], b[50], n, i, j, t, min, swapped;
    printf("Enter n: ");
    scanf("%d", &n);
    printf("Enter %d numbers: ", n);
    for (i = 0; i < n; i++) {
        scanf("%d", &a[i]);
        b[i] = a[i];
    }

    printf("\nBubble sort:\n");
    for (i = 0; i < n - 1; i++) {
        swapped = 0;
        for (j = 0; j < n - 1 - i; j++) {
            if (a[j] > a[j + 1]) {
                t = a[j]; a[j] = a[j + 1]; a[j + 1] = t;
                swapped = 1;
            }
        }
        printf("Pass %d:", i + 1);
        print(a, n);
        if (!swapped) break;
    }

    printf("\nSelection sort:\n");
    for (i = 0; i < n - 1; i++) {
        min = i;
        for (j = i + 1; j < n; j++)
            if (b[j] < b[min]) min = j;
    }
}
```



```
        t = b[i]; b[i] = b[min]; b[min] = t;
        printf("Pass %d:", i + 1);
        print(b, n);
    }
    return 0;
}
```

Output (sample run):

```
Enter n: 6
Enter 6 numbers: 64 25 12 22 11 90

Bubble sort:
Pass 1: 25 12 22 11 64 90
Pass 2: 12 22 11 25 64 90
Pass 3: 12 11 22 25 64 90
Pass 4: 11 12 22 25 64 90
Pass 5: 11 12 22 25 64 90

Selection sort:
Pass 1: 11 25 12 22 64 90
Pass 2: 11 12 25 22 64 90
Pass 3: 11 12 22 25 64 90
Pass 4: 11 12 22 25 64 90
Pass 5: 11 12 22 25 64 90
```

Explanation: in bubble sort the inner loop runs to $n - 1 - i$ because the last i elements are already sorted; the loop stops early when no swap occurs. In selection sort, after pass 1 the smallest element (11) is at $a[0]$.

Viva-voce questions

Q1. Which sorting algorithm is stable: bubble or selection?

Bubble sort is stable (equal elements keep their order); the usual selection sort is not.

Q2. What is the best-case complexity of bubble sort with a flag?

$O(n)$, when the array is already sorted (one pass with no swaps).

Q3. Name two faster sorting algorithms.

Quick sort and merge sort, $O(n \log n)$ on average.

Experiment 10: Binary Search

Aim

To search for an element in a sorted array using binary search (iterative).

Theory (with explanation)

Binary search works only on a **sorted** array. It compares the key with the middle element:

- If equal, the search is successful.
- If the key is smaller, the search continues in the left half; if larger, in the right half.

Each comparison halves the search range, so at most $\lfloor \log_2 n \rfloor + 1$ comparisons are needed: 1000 elements need at most 10 comparisons, while linear search may need 1000. Time complexity $O(\log n)$.

Program:

```
#include <stdio.h>

int main(void)
{
    int a[100], n, i, key, low, high, mid, steps = 0, found = 0;
    printf("Enter n: ");
    scanf("%d", &n);
    printf("Enter %d numbers in ascending order: ", n);
    for (i = 0; i < n; i++)
        scanf("%d", &a[i]);
    printf("Enter key: ");
    scanf("%d", &key);

    low = 0;
    high = n - 1;
    while (low <= high) {
        mid = (low + high) / 2;
        steps++;
        printf(" step %d: low=%d high=%d mid=%d a[mid]=%d\n",
            steps, low, high, mid, a[mid]);
        if (a[mid] == key) { found = 1; break; }
        else if (key < a[mid]) high = mid - 1;
        else low = mid + 1;
    }
    if (found)
        printf("%d found at position %d after %d comparisons\n",
            key, mid + 1, steps);
    else
        printf("%d not found (%d comparisons)\n", key, steps);
    return 0;
}
```

Output — run 1:

```
Enter n: 10
Enter 10 numbers in ascending order: 3 8 15 21 27 34 42 50 63 77
Enter key: 42
step 1: low=0 high=9 mid=4 a[mid]=27
step 2: low=5 high=9 mid=7 a[mid]=50
step 3: low=5 high=6 mid=5 a[mid]=34
step 4: low=6 high=6 mid=6 a[mid]=42
42 found at position 7 after 4 comparisons
```



Output — run 2:

```
Enter n: 10
Enter 10 numbers in ascending order: 3 8 15 21 27 34 42 50 63 77
Enter key: 10
  step 1: low=0 high=9 mid=4 a[mid]=27
  step 2: low=0 high=3 mid=1 a[mid]=8
  step 3: low=2 high=3 mid=2 a[mid]=15
10 not found (3 comparisons)
```

Explanation: the trace shows how low, high and mid change. For key 42 the range shrinks $10 \rightarrow 5 \rightarrow 2 \rightarrow 1$ element and the key is found in 4 comparisons (linear search would need 7). The unsuccessful search for 10 ends when low becomes greater than high.

Viva-voce questions

Q1. What is the precondition for binary search?

The array must be sorted.

Q2. What is the maximum number of comparisons for 1024 elements?

$\log_2 1024 + 1 = 11$.

Q3. Why is mid sometimes written as $\text{low} + (\text{high} - \text{low})/2$?

To avoid integer overflow of $\text{low} + \text{high}$ for very large indices.

Experiment 11: Two-Dimensional Arrays — Matrix Operations

Aim

To add and multiply two matrices and find the transpose of a matrix.

Theory (with explanation)

A **2-D array** `int a[3][3]` is a table of rows and columns stored row by row (row-major order). Element `a[i][j]` is in row `i` and column `j`.

- **Addition:** possible only for matrices of the same order: $c[i][j] = a[i][j] + b[i][j]$.
- **Multiplication:** A $(m \times n) \times B (n \times p)$ is defined only if columns of A = rows of B; the result C is $m \times p$ with $c[i][j] = \sum_k a[i][k] \times b[k][j]$. Three nested loops are needed.
- **Transpose:** $t[j][i] = a[i][j]$ (rows become columns).

Program:

```
#include <stdio.h>

int main(void)
{
    int a[10][10], b[10][10], c[10][10];
    int m, n, p, q, i, j, k;
    printf("Enter rows and columns of A: ");
    scanf("%d %d", &m, &n);
    printf("Enter elements of A row-wise: ");
    for (i = 0; i < m; i++)
        for (j = 0; j < n; j++)
            scanf("%d", &a[i][j]);
    printf("Enter rows and columns of B: ");
    scanf("%d %d", &p, &q);
    printf("Enter elements of B row-wise: ");
    for (i = 0; i < p; i++)
        for (j = 0; j < q; j++)
            scanf("%d", &b[i][j]);

    printf("\nTranspose of A:\n");
    for (j = 0; j < n; j++) {
        for (i = 0; i < m; i++)
            printf("%5d", a[i][j]);
        printf("\n");
    }

    if (n != p) {
        printf("Multiplication not possible (columns of A != rows of B)\n");
        return 0;
    }
    for (i = 0; i < m; i++)
        for (j = 0; j < q; j++) {
            c[i][j] = 0;
            for (k = 0; k < n; k++)
                c[i][j] += a[i][k] * b[k][j];
        }
    printf("\nProduct A x B (%d x %d):\n", m, q);
    for (i = 0; i < m; i++) {
        for (j = 0; j < q; j++)
            printf("%5d", c[i][j]);
    }
}
```

```
        printf("\n");
    }
    if (m == p && n == q)
        printf("Addition possible\n");
    else
        printf("Addition not possible (orders differ)\n");
    return 0;
}
```

Output (sample run):

```
Enter rows and columns of A: 2 3
Enter elements of A row-wise: 1 2 3 4 5 6
Enter rows and columns of B: 3 2
Enter elements of B row-wise: 7 8 9 10 11 12

Transpose of A:
 1  4
 2  5
 3  6

Product A x B (2 x 2):
 58  64
139 154
Addition not possible (orders differ)
```

Explanation: for A (2×3) and B (3×2), $c[0][0] = 1 \times 7 + 2 \times 9 + 3 \times 11 = 58$ and the product is 2×2 . Addition is not possible because the orders are different. (For two matrices of the same order, the addition loop is $c[i][j] = a[i][j] + b[i][j]$.)

Viva-voce questions**Q1. How is a 2-D array stored in memory in C?**

In row-major order: all elements of row 0, then row 1, and so on.

Q2. What is the condition for multiplying two matrices?

The number of columns of the first must equal the number of rows of the second.

Q3. How many multiplications are needed to multiply two $n \times n$ matrices by this method?

n^3 .

Experiment 12: Strings — Manual Operations

Aim

To find the length of a string, reverse it, check whether it is a palindrome, and count vowels, consonants, digits and spaces — without using string library functions.

Theory (with explanation)

A **string** in C is an array of characters terminated by the **null character '\0'**. "HELLO" occupies 6 bytes: H E L L O \0. Functions find the end of a string by searching for '\0'.

- `fgets(s, size, stdin)` reads a whole line including spaces (safer than `gets`, which has been removed from the language because it can overflow the buffer). `fgets` keeps the newline, which we remove.
- Characters are small integers (ASCII codes): 'A' = 65, 'a' = 97, '0' = 48; so `ch - 'a'` gives the position of a letter, and `ch - 32` converts lower to upper case.

Program:

```
#include <stdio.h>

int main(void)
{
    char s[100], r[100];
    int len = 0, i, v = 0, c = 0, d = 0, sp = 0, pal = 1;
    char ch;
    printf("Enter a string: ");
    fgets(s, sizeof s, stdin);

    while (s[len] != '\0' && s[len] != '\n')    /* length */
        len++;
    s[len] = '\0';

    for (i = 0; i < len; i++)                    /* reverse */
        r[i] = s[len - 1 - i];
    r[len] = '\0';

    for (i = 0; i < len; i++) {
        ch = s[i];
        if (ch >= 'A' && ch <= 'Z') ch = ch + 32;    /* to lower case */
        if (ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o' || ch == 'u') v++;
        else if (ch >= 'a' && ch <= 'z') c++;
        else if (ch >= '0' && ch <= '9') d++;
        else if (ch == ' ') sp++;
    }
    for (i = 0; i < len / 2; i++)                /* palindrome (case-sensitive) */
        if (s[i] != s[len - 1 - i]) { pal = 0; break; }

    printf("Length    = %d\n", len);
    printf("Reverse    = %s\n", r);
    printf("Vowels = %d, Consonants = %d, Digits = %d, Spaces = %d\n", v, c, d, sp);
    printf("The string %s a palindrome\n", s, pal ? "is" : "is not");
    return 0;
}
```

Output — run 1:



```
Enter a string: Madam in Eden 2023
Length      = 18
Reverse     = 3202 nedE ni madaM
Vowels = 5, Consonants = 6, Digits = 4, Spaces = 3
The string is not a palindrome
```

Output — run 2:

```
Enter a string: racecar
Length      = 7
Reverse     = racecar
Vowels = 3, Consonants = 4, Digits = 0, Spaces = 0
The string is a palindrome
```

Explanation: the length loop stops at '\0' (or at the newline kept by fgets, which is then replaced by '\0'). The reverse copies characters from the end. Each character is classified by comparing its ASCII range.

Viva-voce questions

Q1. What is the null character and why is it needed?

'\0' (ASCII 0) marks the end of a string, so functions know where the string stops.

Q2. What is the difference between 'A' and "A"?

'A' is a single character (1 byte); "A" is a string of two bytes, 'A' and '\0'.

Q3. Why is gets() not used?

It does not check the buffer size and can overflow the array; it was removed in C11. fgets() is used instead.



Experiment 13: String Library Functions

Aim

To use the standard string functions `strlen`, `strcpy`, `strcat`, `strcmp`, `strrev`-equivalent and `toupper`, and to sort a list of names alphabetically.

Theory (with explanation)

The header **`string.h`** provides ready-made string functions:

Function	Purpose
<code>strlen(s)</code>	Number of characters before '\0'
<code>strcpy(d, s)</code>	Copies s into d (d must be large enough)
<code>strcat(d, s)</code>	Appends s to the end of d
<code>strcmp(a, b)</code>	Returns 0 if equal, negative if a comes before b, positive if after (dictionary order)
<code>strncpy</code> , <code>strncat</code> , <code>strncmp</code>	Versions limited to n characters (safer)
<code>strchr(s, c)</code> , <code>strstr(s, t)</code>	Find a character or a substring

`ctype.h` provides `toupper()`, `tolower()`, `isalpha()`, `isdigit()`, `isspace()`. Sorting strings uses `strcmp` to compare and `strcpy` (through a temporary array) to swap.

Program:



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

int main(void)
{
    char first[30], last[30], full[70], up[70], names[10][30], t[30];
    int n, i, j;

    printf("Enter first name: ");
    scanf("%29s", first);
    printf("Enter last name: ");
    scanf("%29s", last);

    strcpy(full, first);          /* full = first */
    strcat(full, " ");           /* full = first + " " */
    strcat(full, last);          /* full = first + last */
    printf("Full name: %s (length %zu)\n", full, strlen(full));

    for (i = 0; full[i]; i++)
        up[i] = toupper((unsigned char)full[i]);
    up[i] = '\0';
    printf("Upper case: %s\n", up);
    printf("strcmp(\"%s\", \"%s\") = %d\n", first, last, strcmp(first, last));

    printf("How many names to sort? ");
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        printf("Name %d: ", i + 1);
        scanf("%29s", names[i]);
    }
    for (i = 0; i < n - 1; i++)
        for (j = 0; j < n - 1 - i; j++)
            if (strcmp(names[j], names[j + 1]) > 0) {
                strcpy(t, names[j]);
                strcpy(names[j], names[j + 1]);
                strcpy(names[j + 1], t);
            }
    printf("Sorted names:");
    for (i = 0; i < n; i++)
        printf(" %s", names[i]);
    printf("\n");
    return 0;
}
```

Output (sample run):

```
Enter first name: Ravi
Enter last name: Kumar
Full name: Ravi Kumar (length 10)
Upper case: RAVI KUMAR
strcmp("Ravi", "Kumar") = 7
How many names to sort? 4
Name 1: Priya
Name 2: Amit
Name 3: Sneha
Name 4: Deepak
Sorted names: Amit Deepak Priya Sneha
```



Explanation: strcmp("Ravi", "Kumar") is positive because 'R' comes after 'K'. Names are sorted with bubble sort, using strcmp to compare and strcpy to swap (arrays cannot be assigned with =).

Viva-voce questions

Q1. Why can we not compare strings with ==?

== compares the addresses of the arrays, not their contents; strcmp compares character by character.

Q2. What does strcmp return?

0 if equal, a negative value if the first string is smaller (earlier in dictionary order), a positive value if larger.

Q3. What is the risk in strcpy and strcat?

They do not check the destination size and can overflow the buffer; strncpy/strncat or snprintf are safer.



Experiment 14: User-Defined Functions – GCD, LCM and Call by Value/Reference

Aim

To write functions to find the GCD and LCM of two numbers, and to demonstrate call by value and call by reference with a swap function.

Theory (with explanation)

A **function** is a self-contained block that performs a task. It has a **declaration (prototype)**, a **definition** and **calls**. Functions avoid repetition, make programs modular and easier to test.

- **Call by value:** a copy of the argument is passed; changes inside the function do not affect the caller's variable.
- **Call by reference (using pointers):** the address is passed; the function changes the caller's variable through the pointer.

Euclid's algorithm: $\text{GCD}(a, b) = \text{GCD}(b, a \bmod b)$, and $\text{GCD}(a, 0) = a$. It is very fast.
 $\text{LCM}(a, b) = a \times b / \text{GCD}(a, b)$.

Program:

```
#include <stdio.h>

int gcd(int a, int b);           /* prototypes */
void swapByValue(int x, int y);
void swapByReference(int *x, int *y);

int main(void)
{
    int a, b, g;
    printf("Enter two numbers: ");
    scanf("%d", &a);
    scanf("%d", &b);
    g = gcd(a, b);
    printf("GCD = %d, LCM = %d\n", g, a / g * b);

    swapByValue(a, b);
    printf("After swapByValue:      a = %d, b = %d (unchanged)\n", a, b);
    swapByReference(&a, &b);
    printf("After swapByReference: a = %d, b = %d (swapped)\n", a, b);
    return 0;
}

int gcd(int a, int b)           /* Euclid's algorithm */
{
    int r;
    while (b != 0) {
        r = a % b;
        a = b;
        b = r;
    }
    return a;
}

void swapByValue(int x, int y)  /* works on copies */
{
    int t = x; x = y; y = t;
}

void swapByReference(int *x, int *y) /* works on originals */
{
    int t = *x; *x = *y; *y = t;
}
```

Output (sample run):

```
Enter two numbers: 84
120
GCD = 12, LCM = 840
After swapByValue:      a = 84, b = 120 (unchanged)
After swapByReference: a = 120, b = 84 (swapped)
```

Trace of Euclid's algorithm for (84, 120): (84, 120) → (120, 84) → (84, 36) → (36, 12) → (12, 0), so GCD = 12 and LCM = $84 \times 120 / 12 = 840$. Writing $a / g * b$ instead of $a * b / g$ avoids overflow for large numbers.

Viva-voce questions

Q1. What is a function prototype?



A declaration giving the function's name, return type and parameter types before it is used, so the compiler can check calls.

Q2. What is the difference between actual and formal parameters?

Actual parameters are the values passed in the call; formal parameters are the variables in the function definition that receive them.

Q3. How can a function return more than one value?

Through pointer parameters (call by reference), a structure, or an array.



Experiment 15: Recursion — Factorial, Fibonacci and Tower of Hanoi

Aim

To write recursive functions for the factorial of a number, the nth Fibonacci number and the Tower of Hanoi puzzle.

Theory (with explanation)

A **recursive function** calls itself. Every recursive function needs:

- a **base case** that stops the recursion (e.g. $0! = 1$), and
- a **recursive case** that reduces the problem towards the base case ($n! = n \times (n - 1)!$).

Each call gets its own copy of local variables on the **call stack**; when the base case is reached, the calls return in reverse order. Without a base case the recursion never ends and the stack overflows.

Tower of Hanoi: move n discs from peg A to peg C using B, never placing a larger disc on a smaller one. Recursive idea: move $n - 1$ discs $A \rightarrow B$, move the largest disc $A \rightarrow C$, move $n - 1$ discs $B \rightarrow C$. It needs $2^n - 1$ moves.

Program:



```
#include <stdio.h>

long fact(int n)
{
    if (n <= 1) return 1;           /* base case */
    return n * fact(n - 1);        /* recursive case */
}

int fib(int n)
{
    if (n <= 1) return n;          /* fib(0) = 0, fib(1) = 1 */
    return fib(n - 1) + fib(n - 2);
}

int moves = 0;
void hanoi(int n, char from, char via, char to)
{
    if (n == 0) return;
    hanoi(n - 1, from, to, via);
    printf(" Move disc %d from %c to %c\n", n, from, to);
    moves++;
    hanoi(n - 1, via, from, to);
}

int main(void)
{
    int n, i;
    printf("Enter n: ");
    scanf("%d", &n);
    printf("%d! = %ld\n", n + 5, fact(n + 5));
    printf("Fibonacci numbers fib(0) to fib(%d):", n + 7);
    for (i = 0; i <= n + 7; i++)
        printf(" %d", fib(i));
    printf("\nTower of Hanoi with %d discs:\n", n);
    hanoi(n, 'A', 'B', 'C');
    printf("Total moves = %d (2^%d - 1)\n", moves, n);
    return 0;
}
```

Output (sample run):

```
Enter n: 3
8! = 40320
Fibonacci numbers fib(0) to fib(10): 0 1 1 2 3 5 8 13 21 34 55
Tower of Hanoi with 3 discs:
Move disc 1 from A to C
Move disc 2 from A to B
Move disc 1 from C to B
Move disc 3 from A to C
Move disc 1 from B to A
Move disc 2 from B to C
Move disc 1 from A to C
Total moves = 7 (2^3 - 1)
```

How fact(4) is evaluated: $\text{fact}(4) = 4 \times \text{fact}(3) = 4 \times 3 \times \text{fact}(2) = 4 \times 3 \times 2 \times \text{fact}(1) = 4 \times 3 \times 2 \times 1 = 24$. The output for $n = 3$ shows $8! = 40320$, the first eleven Fibonacci numbers and the 7 moves of a 3-disc Tower of Hanoi.

Viva-voce questions



Q1. What is a base case? What happens without it?

The condition at which the function returns without calling itself. Without it the recursion continues until the stack overflows (program crash).

Q2. Compare recursion and iteration.

Recursion gives short, clear code for naturally recursive problems but uses extra stack memory and function-call time; iteration is usually faster and uses less memory.

Q3. Why is recursive Fibonacci slow?

It recomputes the same values many times (exponential time, about 1.6^n calls); an iterative loop or memoisation computes it in $O(n)$.

Experiment 16: Pointers

Aim

To understand pointers: address and dereference operators, pointer arithmetic, and accessing an array through a pointer.

Theory (with explanation)

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

- `int *p;` declares `p` as a pointer to `int`. `p = &x;` stores the address of `x` in `p`.
- `*p` (dereference) gives the value at that address; `*p = 10` changes `x`.
- **Pointer arithmetic:** `p + 1` points to the next element of its type, i.e. the address increases by `sizeof(int)` bytes (usually 4), not by 1.
- The **name of an array** is the address of its first element: `a == &a[0]`, and `a[i] == *(a + i)`.
- A **NULL pointer** points nowhere; dereferencing it (or an uninitialised pointer) is an error.
- **Pointer to pointer:** `int pp = &p;` then `pp` is `x`.

Program:

```
#include <stdio.h>

int main(void)
{
    int x = 25, *p, **pp;
    int a[5] = {10, 20, 30, 40, 50};
    int *q, i, sum = 0;

    p = &x;
    pp = &p;
    printf("x = %d, *p = %d, **pp = %d\n", x, *p, **pp);
    *p = 100; /* changes x through p */
    printf("After *p = 100, x = %d\n", x);
    printf("sizeof(int) = %zu bytes\n", sizeof(int));

    q = a; /* same as q = &a[0] */
    printf("Byte difference between &a[1] and &a[0] = %ld\n",
        (long)((char *) (q + 1) - (char *) q));
    for (i = 0; i < 5; i++) {
        printf("a[%d] = %d, *(q + %d) = %d\n", i, a[i], i, *(q + i));
        sum += *(q + i);
    }
    printf("Sum using pointer = %d\n", sum);

    q = &a[4]; /* traverse backwards */
    printf("Reverse:");
    while (q >= a)
        printf(" %d", *q--);
    printf("\n");
    return 0;
}
```

Output (sample run):



```
x = 25, *p = 25, **pp = 25
After *p = 100, x = 100
sizeof(int) = 4 bytes
Byte difference between &a[1] and &a[0] = 4
a[0] = 10, *(q + 0) = 10
a[1] = 20, *(q + 1) = 20
a[2] = 30, *(q + 2) = 30
a[3] = 40, *(q + 3) = 40
a[4] = 50, *(q + 4) = 50
Sum using pointer = 150
Reverse: 50 40 30 20 10
```

Explanation: changing *p changes x because p holds the address of x. q + 1 is 4 bytes ahead of q (size of int), which is why *(q + i) reaches a[i]. Actual addresses differ from machine to machine, so the program prints differences instead of raw addresses.

Viva-voce questions

Q1. What is a pointer?

A variable that holds the memory address of another variable.

Q2. What is the output of *(a + 2) if a = {10, 20, 30}?

30, the element at index 2.

Q3. What is a dangling pointer?

A pointer that still holds the address of memory that has been freed or of a local variable that no longer exists.

Q4. What is a void pointer?

A generic pointer (void *) that can hold the address of any type; it must be cast before dereferencing.



Experiment 17: Dynamic Memory Allocation

Aim

To allocate memory at run time for n numbers using malloc, enlarge it with realloc, and release it with free.

Theory (with explanation)

Arrays declared as `int a[100]` have a **fixed size** decided at compile time; if fewer elements are used, memory is wasted, and more cannot be stored. **Dynamic memory allocation** takes memory from the **heap** while the program runs (`stdlib.h`):

Function	Purpose
<code>malloc(size)</code>	Allocates size bytes; contents are garbage; returns NULL if it fails
<code>calloc(n, size)</code>	Allocates $n \times \text{size}$ bytes initialised to zero
<code>realloc(p, newsize)</code>	Changes the size of a block (may move it), keeping the old contents
<code>free(p)</code>	Releases the block; p should then be set to NULL

Forgetting to free memory causes a **memory leak**; using memory after freeing it is undefined behaviour.

Program:

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

int main(void)
{
    int *a, n, m, i, max;
    printf("How many numbers? ");
    scanf("%d", &n);
    a = (int *)malloc(n * sizeof(int));
    if (a == NULL) {
        printf("Memory not available\n");
        return 1;
    }
    printf("Enter %d numbers: ", n);
    for (i = 0; i < n; i++)
        scanf("%d", a + i);

    printf("How many more numbers? ");
    scanf("%d", &m);
    int *tmp = (int *)realloc(a, (n + m) * sizeof(int));
    if (tmp == NULL) { free(a); return 1; }
    a = tmp;
    printf("Enter %d more numbers: ", m);
    for (i = n; i < n + m; i++)
        scanf("%d", &a[i]);
    n += m;

    max = a[0];
    printf("Array of %d elements:", n);
    for (i = 0; i < n; i++) {
        printf(" %d", a[i]);
        if (a[i] > max) max = a[i];
    }
    printf("\nLargest = %d\n", max);
    free(a);
    a = NULL;
    printf("Memory released.\n");
    return 0;
}
```

Output (sample run):

```
How many numbers? 5
Enter 5 numbers: 12 7 25 3 18
How many more numbers? 2
Enter 2 more numbers: 40 1
Array of 7 elements: 12 7 25 3 18 40 1
Largest = 40
Memory released.
```

Explanation: malloc allocates exactly $n \times \text{sizeof(int)}$ bytes. realloc's result is stored in a temporary pointer first, so that the original block is not lost if realloc fails. The memory is freed at the end.

Viva-voce questions**Q1. What is the difference between malloc and calloc?**

malloc takes one argument (bytes) and leaves memory uninitialised; calloc takes two (count, size) and initialises it to zero.



Q2. What is a memory leak?

Memory allocated dynamically but never freed, so it cannot be reused while the program runs.

Q3. Where is dynamically allocated memory taken from?

From the heap (local variables are on the stack).



Experiment 18: Structures — Student Records

Aim

To store the records of n students (roll number, name, marks in three subjects) using an array of structures, calculate totals and percentages, assign grades and display the list sorted by total marks.

Theory (with explanation)

A **structure** groups variables of different types under one name:

- `struct student { int roll; char name[30]; float marks[3]; };` defines a new type.
- Members are accessed with the **dot operator** (`s.roll`) or, through a pointer, with the **arrow operator** (`p->roll`, same as `(*p).roll`).
- Structures can be assigned as a whole (`s1 = s2`), passed to functions and stored in arrays.
- **typedef** gives a type a shorter name: `typedef struct student Student;`
- A **union** is similar, but all members share the same memory (only one is valid at a time), so its size is that of the largest member.

Program:

```
#include <stdio.h>

typedef struct {
    int roll;
    char name[30];
    float marks[3];
    float total, percent;
    char grade;
} Student;

char gradeOf(float p)
{
    if (p >= 90) return 'O';
    if (p >= 80) return 'E';
    if (p >= 70) return 'A';
    if (p >= 60) return 'B';
    if (p >= 50) return 'C';
    return 'F';
}

int main(void)
{
    Student s[20], t;
    int n, i, j;
    printf("Number of students: ");
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        printf("Roll, name, marks in 3 subjects: ");
        scanf("%d %29s %f %f %f", &s[i].roll, s[i].name,
            &s[i].marks[0], &s[i].marks[1], &s[i].marks[2]);
        s[i].total = s[i].marks[0] + s[i].marks[1] + s[i].marks[2];
        s[i].percent = s[i].total / 3;
        s[i].grade = gradeOf(s[i].percent);
    }
    for (i = 0; i < n - 1; i++) /* sort by total, descending */
        for (j = 0; j < n - 1 - i; j++)
            if (s[j].total < s[j + 1].total) {
                t = s[j]; s[j] = s[j + 1]; s[j + 1] = t;
            }
    printf("\n%-5s %-10s %7s %8s %6s\n", "Roll", "Name", "Total", "Percent", "Grade");

    for (i = 0; i < n; i++)
        printf("%-5d %-10s %7.1f %8.2f %6c\n", s[i].roll, s[i].name,
            s[i].total, s[i].percent, s[i].grade);
    return 0;
}
```

Output (sample run):

```
Number of students: 3
Roll, name, marks in 3 subjects: 101 Anita 78 85 92
Roll, name, marks in 3 subjects: 102 Bikash 65 58 70
Roll, name, marks in 3 subjects: 103 Chandan 88 91 79

Roll  Name      Total  Percent  Grade
103   Chandan    258.0   86.00    E
101   Anita      255.0   85.00    E
102   Bikash     193.0   64.33    B
```

Explanation: each student is one structure variable. Whole structures are swapped with `t = s[j]` while sorting. The grade letters follow the usual O, E, A, B, C, F scheme.



Viva-voce questions

Q1. What is the difference between a structure and an array?

An array holds elements of the same type accessed by index; a structure holds members of different types accessed by name.

Q2. What is the difference between a structure and a union?

In a structure every member has its own memory (size $\geq$ sum of members); in a union all members share one memory area (size = largest member).

Q3. When is the arrow operator used?

To access a member through a pointer to a structure: `p->roll`.



Experiment 19: File Handling

Aim

To create a text file, write data into it, read it back, and count the number of characters, words and lines in the file.

Theory (with explanation)

Data in variables is lost when the program ends; **files** store data permanently on disk. In C a file is accessed through a pointer of type **FILE *** (stdio.h).

Function	Purpose
fopen(name, mode)	Opens a file; modes "r" (read), "w" (write, creates/truncates), "a" (append), "r+", "w+"; returns NULL on failure
fprintf / fscanf	Formatted write / read
fputc / fgetc	Write / read one character; fgetc returns EOF at the end of the file
fputs / fgets	Write / read a string (line)
fclose(fp)	Closes the file and flushes the buffer

Always check that fopen did not return NULL, and close every file you open.

Program:

```
#include <stdio.h>

int main(void)
{
    FILE *fp;
    int ch, chars = 0, words = 0, lines = 0, inWord = 0;

    fp = fopen("notes.txt", "w");
    if (fp == NULL) {
        printf("Cannot create file\n");
        return 1;
    }
    fprintf(fp, "Gandhi School of Engineering\n");
    fprintf(fp, "C programming lab\n");
    fprintf(fp, "File handling in C is easy\n");
    fclose(fp);
    printf("Data written to notes.txt\n\nContents of the file:\n");

    fp = fopen("notes.txt", "r");
    if (fp == NULL) {
        printf("Cannot open file\n");
        return 1;
    }
    while ((ch = fgetc(fp)) != EOF) {
        putchar(ch);
        chars++;
        if (ch == '\n') lines++;
        if (ch == ' ' || ch == '\n' || ch == '\t')
            inWord = 0;
        else if (!inWord) {
            inWord = 1;
            words++;
        }
    }
    fclose(fp);
    printf("\nCharacters = %d, Words = %d, Lines = %d\n", chars, words, lines);
    return 0;
}
```

Output (sample run):

```
Data written to notes.txt

Contents of the file:
Gandhi School of Engineering
C programming lab
File handling in C is easy

Characters = 74, Words = 13, Lines = 3
```

Explanation: ch is declared int (not char) so that it can hold EOF (−1) as well as every character. A word is counted when a non-space character follows a space (the inWord flag). The newline characters are included in the character count.

Viva-voce questions**Q1. What is the difference between "w" and "a" modes?**

"w" creates a new file or erases an existing one; "a" appends data at the end of an existing file (creating it if needed).



Q2. What is EOF?

A constant (usually -1) returned by input functions at the end of a file.

Q3. Why should a file be closed?

To flush buffered data to disk and release the file handle; unclosed files may lose data.

Q4. What is the difference between text and binary files?

Text files store characters (human-readable); binary files store data in memory format (read/written with fread/fwrite), which is compact and faster.



Experiment 20: Stack using an Array

Aim

To implement a stack of integers using an array with push, pop, peek and display operations, handling overflow and underflow.

Theory (with explanation)

A **stack** is a linear data structure in which insertion (**push**) and deletion (**pop**) take place at one end called the **top**. It follows **LIFO** (last in, first out) — like a pile of plates.

- In an array implementation, $\text{top} = -1$ means the stack is empty. push: if $\text{top} = \text{MAX} - 1$ the stack is full (**overflow**); else $a[++\text{top}] = x$. pop: if $\text{top} = -1$ the stack is empty (**underflow**); else return $a[\text{top}--]$.
- All operations take $O(1)$ time.
- **Applications:** function calls (call stack), undo in editors, expression conversion and evaluation (infix to postfix), balancing parentheses, recursion, backtracking.

Program:



```
#include <stdio.h>
#define MAX 5

int stack[MAX], top = -1;

void push(int x)
{
    if (top == MAX - 1) printf("Stack overflow\n");
    else { stack[++top] = x; printf("%d pushed\n", x); }
}

void pop(void)
{
    if (top == -1) printf("Stack underflow\n");
    else printf("%d popped\n", stack[top--]);
}

void display(void)
{
    int i;
    if (top == -1) { printf("Stack is empty\n"); return; }
    printf("Stack (top to bottom):");
    for (i = top; i >= 0; i--) printf(" %d", stack[i]);
    printf("\n");
}

int main(void)
{
    int choice, x;
    do {
        printf("1.Push 2.Pop 3.Display 4.Peek 5.Exit  Choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1: printf("Value: "); scanf("%d", &x); push(x); break;
            case 2: pop(); break;
            case 3: display(); break;
            case 4:
                if (top == -1) printf("Stack is empty\n");
                else printf("Top element = %d\n", stack[top]);
                break;
            case 5: printf("Bye\n"); break;
            default: printf("Invalid choice\n");
        }
    } while (choice != 5);
    return 0;
}
```

Output (sample run):



```
1.Push 2.Pop 3.Display 4.Peek 5.Exit  Choice: 1
Value: 10
10 pushed
1.Push 2.Pop 3.Display 4.Peek 5.Exit  Choice: 1
Value: 20
20 pushed
1.Push 2.Pop 3.Display 4.Peek 5.Exit  Choice: 1
Value: 30
30 pushed
1.Push 2.Pop 3.Display 4.Peek 5.Exit  Choice: 3
Stack (top to bottom): 30 20 10
1.Push 2.Pop 3.Display 4.Peek 5.Exit  Choice: 2
30 popped
1.Push 2.Pop 3.Display 4.Peek 5.Exit  Choice: 3
Stack (top to bottom): 20 10
1.Push 2.Pop 3.Display 4.Peek 5.Exit  Choice: 4
Top element = 20
1.Push 2.Pop 3.Display 4.Peek 5.Exit  Choice: 2
20 popped
1.Push 2.Pop 3.Display 4.Peek 5.Exit  Choice: 2
10 popped
1.Push 2.Pop 3.Display 4.Peek 5.Exit  Choice: 2
Stack underflow
1.Push 2.Pop 3.Display 4.Peek 5.Exit  Choice: 5
Bye
```

Explanation: 10, 20 and 30 are pushed; the display shows 30 on top. Popping returns 30 first (LIFO). After three pops, a fourth pop reports underflow.

Viva-voce questions

Q1. What is LIFO?

Last in, first out: the element inserted last is removed first.

Q2. What is stack overflow and underflow?

Overflow: pushing onto a full stack. Underflow: popping from an empty stack.

Q3. Give two applications of a stack.

Evaluating postfix expressions and managing function calls (recursion); also undo operations and parenthesis checking.



Experiment 21: Queue using an Array (Circular Queue)

Aim

To implement a circular queue using an array with insert (enqueue), delete (dequeue) and display operations.

Theory (with explanation)

A **queue** is a linear data structure where insertion takes place at the **rear** and deletion at the **front**. It follows **FIFO** (first in, first out) — like a line of people at a ticket counter.

In a simple array queue, the space freed by deletions at the front cannot be reused. A **circular queue** solves this by wrapping the indices round: $\text{rear} = (\text{rear} + 1) \% \text{MAX}$. Here a **count** of elements is kept, so the queue is full when $\text{count} = \text{MAX}$ and empty when $\text{count} = 0$.

Applications: CPU and printer scheduling, buffering (keyboard, network packets), breadth-first search, call-centre systems.

Program:



```
#include <stdio.h>
#define MAX 4

int q[MAX], front = 0, rear = -1, count = 0;

void enqueue(int x)
{
    if (count == MAX) { printf("Queue is full\n"); return; }
    rear = (rear + 1) % MAX;
    q[rear] = x;
    count++;
    printf("%d inserted\n", x);
}

void dequeue(void)
{
    if (count == 0) { printf("Queue is empty\n"); return; }
    printf("%d deleted\n", q[front]);
    front = (front + 1) % MAX;
    count--;
}

void display(void)
{
    int i, idx;
    if (count == 0) { printf("Queue is empty\n"); return; }
    printf("Queue (front to rear):");
    for (i = 0, idx = front; i < count; i++, idx = (idx + 1) % MAX)
        printf(" %d", q[idx]);
    printf("\n");
}

int main(void)
{
    int ch, x;
    do {
        printf("1.Insert 2.Delete 3.Display 4.Exit  Choice: ");
        scanf("%d", &ch);
        switch (ch) {
            case 1: printf("Value: "); scanf("%d", &x); enqueue(x); break;
            case 2: dequeue(); break;
            case 3: display(); break;
            case 4: printf("Bye\n"); break;
            default: printf("Invalid choice\n");
        }
    } while (ch != 4);
    return 0;
}
```

Output (sample run):



```
1.Insert 2.Delete 3.Display 4.Exit  Choice: 1
Value: 5
5 inserted
1.Insert 2.Delete 3.Display 4.Exit  Choice: 1
Value: 15
15 inserted
1.Insert 2.Delete 3.Display 4.Exit  Choice: 1
Value: 25
25 inserted
1.Insert 2.Delete 3.Display 4.Exit  Choice: 3
Queue (front to rear): 5 15 25
1.Insert 2.Delete 3.Display 4.Exit  Choice: 2
5 deleted
1.Insert 2.Delete 3.Display 4.Exit  Choice: 1
Value: 35
35 inserted
1.Insert 2.Delete 3.Display 4.Exit  Choice: 3
Queue (front to rear): 15 25 35
1.Insert 2.Delete 3.Display 4.Exit  Choice: 4
Bye
```

Explanation: the first element inserted (5) is the first deleted (FIFO). The modulus operator makes the indices wrap round, so the array slots are reused.

Viva-voce questions

Q1. What is FIFO?

First in, first out: the element inserted first is removed first.

Q2. Why is a circular queue better than a linear queue?

It reuses the empty slots at the front, so the queue does not report "full" while space is available.

Q3. What is a deque and a priority queue?

A deque allows insertion and deletion at both ends; a priority queue removes the element with the highest priority first.



Experiment 22: Singly Linked List

Aim

To implement a singly linked list with insertion at the beginning and end, deletion of a given value, searching, counting and display.

Theory (with explanation)

A **linked list** is a collection of **nodes**, each containing data and a pointer to the next node. The first node is pointed to by **head**; the last node's next pointer is NULL.

Array	Linked list
Fixed size, contiguous memory	Dynamic size; nodes anywhere in memory
Random access $a[i]$ in $O(1)$	Sequential access, $O(n)$
Insertion/deletion needs shifting, $O(n)$	Insertion/deletion by changing pointers, $O(1)$ once the position is known
No extra memory	Extra memory for pointers

Insertion at the beginning: create a node, set `newnode->next = head`, then `head = newnode`. **Deletion:** find the node and its previous node; set `prev->next = cur->next` and free `cur`.

Program:



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

struct node {
    int data;
    struct node *next;
};

struct node *head = NULL;

struct node *create(int x)
{
    struct node *n = (struct node *)malloc(sizeof(struct node));
    n->data = x;
    n->next = NULL;
    return n;
}

void insertBegin(int x)
{
    struct node *n = create(x);
    n->next = head;
    head = n;
}

void insertEnd(int x)
{
    struct node *n = create(x), *p = head;
    if (head == NULL) { head = n; return; }
    while (p->next != NULL) p = p->next;
    p->next = n;
}

void deleteValue(int x)
{
    struct node *cur = head, *prev = NULL;
    while (cur != NULL && cur->data != x) { prev = cur; cur = cur->next; }
    if (cur == NULL) { printf("%d not found\n", x); return; }
    if (prev == NULL) head = cur->next; /* deleting the first node */
    else prev->next = cur->next;
}
```



```
    free(cur);
    printf("%d deleted\n", x);
}

void display(void)
{
    struct node *p = head;
    int count = 0;
    printf("List: ");
    while (p != NULL) { printf("%d -> ", p->data); p = p->next; count++; }
    printf("NULL    (%d nodes)\n", count);
}

int search(int x)
{
    int pos = 1;
    struct node *p;
    for (p = head; p != NULL; p = p->next, pos++)
        if (p->data == x) return pos;
    return -1;
}

int main(void)
{
    insertEnd(20);
    insertEnd(30);
    insertBegin(10);
    insertEnd(40);
    display();
    printf("30 is at position %d\n", search(30));
    deleteValue(10);
    deleteValue(30);
    deleteValue(99);
    display();
    while (head != NULL) {                /* free all nodes */
        struct node *t = head;
        head = head->next;
        free(t);
    }
    return 0;
}
```

Output (sample run):

```
List: 10 -> 20 -> 30 -> 40 -> NULL    (4 nodes)
30 is at position 3
10 deleted
30 deleted
99 not found
List: 20 -> 40 -> NULL    (2 nodes)
```

Explanation: the list is built as $10 \rightarrow 20 \rightarrow 30 \rightarrow 40$. Deleting 10 (the first node) changes head; deleting 30 links 20 directly to 40; deleting 99 reports "not found". All nodes are freed at the end.

Viva-voce questions

Q1. What is a self-referential structure?



A structure that contains a pointer to a structure of the same type, e.g. struct node {
int data; struct node *next; }.

Q2. What are the advantages of a linked list over an array?

Dynamic size and easy insertion/deletion without shifting elements.

Q3. What are doubly and circular linked lists?

In a doubly linked list each node has pointers to both the next and previous nodes; in a circular list the last node points back to the first.



Experiment 23: Insertion Sort and Quick Sort

Aim

To sort an array using insertion sort and quick sort and to compare the number of comparisons made.

Theory (with explanation)

Insertion sort: the array is divided into a sorted part (initially $a[0]$) and an unsorted part. Each new element is inserted into its correct place in the sorted part by shifting larger elements one position right (like arranging playing cards in hand). Best case $O(n)$ (already sorted), worst case $O(n^2)$. Efficient for small or nearly sorted data.

Quick sort (divide and conquer): choose a **pivot, partition** the array so that elements smaller than the pivot are on its left and larger ones on its right, then recursively sort the two parts. Average $O(n \log n)$; worst case $O(n^2)$ when the pivot is always the smallest or largest element.

Program:

```
#include <stdio.h>

long cmpIns = 0, cmpQuick = 0;

void insertionSort(int a[], int n)
{
    int i, j, key;
    for (i = 1; i < n; i++) {
        key = a[i];
        j = i - 1;
        while (j >= 0 && (++cmpIns, a[j] > key)) {
            a[j + 1] = a[j];          /* shift right */
            j--;
        }
        a[j + 1] = key;
    }
}

int partition(int a[], int low, int high)
{
    int pivot = a[high], i = low - 1, j, t;    /* last element as pivot */
    for (j = low; j < high; j++) {
        cmpQuick++;
        if (a[j] < pivot) {
            i++;
            t = a[i]; a[i] = a[j]; a[j] = t;
        }
    }
    t = a[i + 1]; a[i + 1] = a[high]; a[high] = t;
    return i + 1;
}

void quickSort(int a[], int low, int high)
{
    if (low < high) {
        int p = partition(a, low, high);
        quickSort(a, low, p - 1);
        quickSort(a, p + 1, high);
    }
}

int main(void)
{
    int a[50], b[50], n, i;
    printf("Enter n: ");
    scanf("%d", &n);
    printf("Enter %d numbers: ", n);
    for (i = 0; i < n; i++) { scanf("%d", &a[i]); b[i] = a[i]; }
    insertionSort(a, n);
    quickSort(b, 0, n - 1);
    printf("Insertion sort:");
    for (i = 0; i < n; i++) printf(" %d", a[i]);
    printf("    (%ld comparisons)\n", cmpIns);
    printf("Quick sort:    ");
    for (i = 0; i < n; i++) printf(" %d", b[i]);
    printf("    (%ld comparisons)\n", cmpQuick);
    return 0;
}
```

Output (sample run):



```
Enter n: 8
Enter 8 numbers: 38 27 43 3 9 82 10 55
Insertion sort: 3 9 10 27 38 43 55 82    (17 comparisons)
Quick sort:     3 9 10 27 38 43 55 82    (16 comparisons)
```

Explanation: both methods give the same sorted order. The comparison counters show the work done by each algorithm for this input; for large random inputs quick sort needs far fewer comparisons.

Viva-voce questions

Q1. What is the principle of quick sort?

Divide and conquer: partition around a pivot and sort the two parts recursively.

Q2. When does quick sort show its worst case?

When the pivot is always the smallest or largest element (e.g. an already sorted array with the last element as pivot).

Q3. Why is insertion sort good for nearly sorted data?

Each element needs only a few comparisons and shifts, so the time is close to $O(n)$.

Experiment 24: Mini Project — Simple Bank Account System

Aim

To combine structures, arrays, functions and menus in a small application that creates accounts, deposits and withdraws money, and displays balances.

Theory (with explanation)

Real programs are built by **modular design**: the problem is divided into functions, each doing one job (create, deposit, withdraw, display). Data about each account is kept in a structure, and the accounts are stored in an array. **Input validation** (e.g. no withdrawal larger than the balance, account must exist) makes the program robust.

Program:

```
#include <stdio.h>

typedef struct {
    int accNo;
    char name[30];
    double balance;
} Account;

Account acc[50];
int n = 0;

int find(int no)
{
    int i;
    for (i = 0; i < n; i++)
        if (acc[i].accNo == no) return i;
    return -1;
}

void create(void)
{
    printf("Account no, name, opening balance: ");
    scanf("%d %29s %lf", &acc[n].accNo, acc[n].name, &acc[n].balance);
    printf("Account %d created for %s\n", acc[n].accNo, acc[n].name);
    n++;
}

void deposit(void)
{
    int no, i; double amt;
    printf("Account no and amount: ");
    scanf("%d %lf", &no, &amt);
    if ((i = find(no)) < 0) { printf("No such account\n"); return; }
    acc[i].balance += amt;
    printf("Deposited %.2f. New balance = %.2f\n", amt, acc[i].balance);
}

void withdraw(void)
{
    int no, i; double amt;
```



```
printf("Account no and amount: ");
scanf("%d %lf", &no, &amt);
if ((i = find(no)) < 0) { printf("No such account\n"); return; }
if (amt > acc[i].balance) {
    printf("Insufficient balance (%.2f)\n", acc[i].balance);
    return;
}
acc[i].balance -= amt;
printf("Withdrawn %.2f. New balance = %.2f\n", amt, acc[i].balance);
}

void show(void)
{
    int i;
    printf("%-8s %-10s %10s\n", "AccNo", "Name", "Balance");
    for (i = 0; i < n; i++)
        printf("%-8d %-10s %10.2f\n", acc[i].accNo, acc[i].name, acc[i].balance);
}

int main(void)
{
    int ch;
    do {
        printf("1.Create 2.Deposit 3.Withdraw 4.Show 5.Exit  Choice: ");
        scanf("%d", &ch);
        switch (ch) {
            case 1: create(); break;
            case 2: deposit(); break;
            case 3: withdraw(); break;
            case 4: show(); break;
            case 5: printf("Thank you\n"); break;
            default: printf("Invalid choice\n");
        }
    } while (ch != 5);
    return 0;
}
```

Output (sample run):

```
1.Create 2.Deposit 3.Withdraw 4.Show 5.Exit  Choice: 1
Account no, name, opening balance: 5001 Rohit 2500
Account 5001 created for Rohit
1.Create 2.Deposit 3.Withdraw 4.Show 5.Exit  Choice: 1
Account no, name, opening balance: 5002 Meena 10000
Account 5002 created for Meena
1.Create 2.Deposit 3.Withdraw 4.Show 5.Exit  Choice: 2
Account no and amount: 5001 1500
Deposited 1500.00. New balance = 4000.00
1.Create 2.Deposit 3.Withdraw 4.Show 5.Exit  Choice: 3
Account no and amount: 5002 12000
Insufficient balance (10000.00)
1.Create 2.Deposit 3.Withdraw 4.Show 5.Exit  Choice: 3
Account no and amount: 5002 4000
Withdrawn 4000.00. New balance = 6000.00
1.Create 2.Deposit 3.Withdraw 4.Show 5.Exit  Choice: 4
AccNo    Name      Balance
5001     Rohit     4000.00
5002     Meena     6000.00
1.Create 2.Deposit 3.Withdraw 4.Show 5.Exit  Choice: 5
Thank you
```



Explanation: each menu option calls a separate function. The withdrawal of 12,000 from an account holding 10,000 is refused; the next withdrawal of 4,000 succeeds.

Viva-voce questions

Q1. What is modular programming?

Dividing a program into independent functions (modules), each performing one task, which makes it easier to write, test and maintain.

Q2. Why are global variables used for acc and n here, and what is the disadvantage?

So that all functions can access the accounts easily; the disadvantage is that any function can change them, making errors harder to trace. Passing them as parameters is better in large programs.

Q3. How could the accounts be saved permanently?

By writing the array to a file (fwrite or fprintf) before exiting and reading it back at the start.



Appendix: Compiling and Running C Programs

Using GCC on Linux (or MinGW on Windows)

- Save the program as prog.c. Compile: gcc prog.c -o prog (add -lm when math.h functions are used; add -Wall to see all warnings).
- Run: ./prog on Linux, prog.exe on Windows.
- **Stages of compilation:** preprocessing (#include, #define) → compilation (to assembly) → assembly (to object code .o) → linking (with libraries) → executable.

Types of errors

Error	When detected	Example
Syntax (compile-time) error	By the compiler	Missing semicolon, undeclared variable
Linker error	While linking	Undefined reference to sqrt (forgot -lm), misspelt function name
Run-time error	While running	Division by zero, segmentation fault (bad pointer, array out of bounds)
Logical error	Wrong output; not detected automatically	Using < instead of <=, integer division 5/9

General viva questions

Q1. Who developed C and when?

Dennis Ritchie at Bell Laboratories in 1972, to write the UNIX operating system.

Q2. Why is C called a middle-level language?

It has high-level features (functions, structures) and low-level features (pointers, bit operations, direct memory access).

Q3. What are tokens in C?

The smallest units of a program: keywords, identifiers, constants, strings, operators and punctuators.

Q4. What is the difference between a compiler and an interpreter?

A compiler translates the whole program into machine code before running it; an interpreter translates and executes one statement at a time.

Q5. What is the size of int, char, float and double on a typical 64-bit system?

4, 1, 4 and 8 bytes respectively (sizeof operator gives the exact size).

Q6. What is a segmentation fault?

A run-time error caused by accessing memory the program is not allowed to use, e.g. dereferencing NULL or going far beyond array bounds.