C PROGRAMMING
LAB EXPERIMENTS

Experiment 1: Sum of Two Integers (Simple)

Aim:

To read two integers and print their sum.

Input Format:

- First line contains integer A
- Second line contains integer B

Output Format:

- Print the sum of A and B

Sample Input:

```
10

20
```

Sample Output:

```
30
```

Program:

```
#include <stdio.h>


int main() {

   int a, b;

   scanf("%d", &a);

   scanf("%d", &b);

   printf("%d", a + b);

   return 0;

}
```

Experiment 2: Difference of Two Integers (Simple)

Aim:

To read two integers and print their difference.

Sample Input:

```
15
8
```

Sample Output:

```
7
```

Program:

```c
#include <stdio.h>

int main() {
   int a, b;
   scanf("%d", &a);
   scanf("%d", &b);
   printf("%d", a - b);
   return 0;
}
```

Experiment 3: Area of a Circle (Simple)

Aim:

To calculate the area of a circle given its radius.

Input Format:

- One float value representing radius

Output Format:

- Print the area up to two decimal places

Sample Input:

```
7
```

Sample Output:

153.94

Program:

```
#include <stdio.h>

int main() {
    float r;
    scanf("%f", &r);
    printf("%.2f", 3.14159 * r * r);
    return 0;
}
```

Experiment 4: Simple Interest Calculation (Medium)

Aim:

To calculate simple interest using principal, rate, and time.

Input Format:

principal

rate

time

Output Format:

- Print simple interest up to two decimal places

Sample Input:

```
1000
5
2
```

Sample Output:

```
100.00
```

Program:

```
#include <stdio.h>


int main() {
    float p, r, t;
    scanf("%f", &p);
    scanf("%f", &r);
    scanf("%f", &t);
    printf("%.2f", (p * r * t) / 100);
    return 0;
}
```

Experiment 5: Basic Arithmetic Operations on Two Numbers (TOUGH – FULL)

Aim:

You are tasked with developing a C program that performs basic arithmetic operations (addition, subtraction, multiplication, and division) on two numbers.

The program should:

- Read two double-precision floating-point numbers
- Perform all four operations
- Display each result formatted to two decimal places

Assume the second number will never be zero.

Input Format:

enter number1: <value>

enter non zero number2: <value>

Output Format:

number1 operator number2 = result

(Printed for all four operations)

Sample Input:

```
enter number1: 12.50

enter non zero number2: 2.50
```

Sample Output:

```
12.50 + 2.50 = 15.00

12.50 - 2.50 = 10.00

12.50 * 2.50 = 31.25

12.50 / 2.50 = 5.00
```

Algorithm:

1. Start
2. Read first number
3. Read second number
4. Perform all arithmetic operations
5. Print results with two decimal places
6. Stop

Source Code:

calculation.c

Program:

```
#include <stdio.h>


int main() {
    double number1, number2;
    scanf("enter number1: %lf", &number1);

    scanf("enter non zero number2: %lf", &number2);


    printf("%.2lf + %.2lf = %.2lf\n", number1, number2, number1 + number2);

    printf("%.2lf - %.2lf = %.2lf\n", number1, number2, number1 - number2);

    printf("%.2lf * %.2lf = %.2lf\n", number1, number2, number1 * number2);

    printf("%.2lf / %.2lf = %.2lf\n", number1, number2, number1 / number2);


    return 0;

}
```

Result:

Experiment 6: Check Whether a Number is Even or Odd

(LONG DESCRIPTION)

Problem Description

Write a C program that reads an integer value and determines whether the given number is **even** or **odd**.

An integer is considered:

- **Even** if it is exactly divisible by 2
- **Odd** if it is not divisible by 2

The program must correctly handle:

- Positive integers
- Negative integers
- Zero

Zero should be treated as an even number.

Input Format

- A single integer N

Output Format

- Print Even if the number is even
- Print Odd if the number is odd

Sample Input

10

Sample Output

Even

Hidden Test Cases

- 0
- -7
- -100
- Large positive numbers

Constraints

-10^9 ≤ N ≤ 10^9

Program

```
#include <stdio.h>

int main() {
    int n;
    scanf("%d", &n);

    if (n % 2 == 0)
```

```
        printf("Even");
    else
        printf("Odd");

    return 0;
}
```

Experiment 7: Check Whether a Number is Positive, Negative, or Zero

(LONG DESCRIPTION)

Problem Description

Write a C program that reads an integer and determines whether the number is **positive**, **negative**, or **zero**.

The program must clearly distinguish between all three cases and print the appropriate output.

Input Format

- A single integer N

Output Format

- Print Positive if N > 0
- Print Negative if N < 0
- Print Zero if N == 0

Sample Input

-15

Sample Output

Negative

Hidden Test Cases

- 0
- 1
- -1
- Very large positive and negative numbers

Constraints

-10^9 ≤ N ≤ 10^9

Program

```
#include <stdio.h>

int main() {
    int n;
    scanf("%d", &n);

    if (n > 0)
        printf("Positive");
    else if (n < 0)
        printf("Negative");
    else
        printf("Zero");

    return 0;
}
```

Experiment 8: Find the Largest of Two Numbers

(SHORT)

Problem Description

Write a C program to find the largest of two given integers.

Sample Input

```
12
20
```

Sample Output

```
20
```

Program

```c
#include <stdio.h>

int main() {
   int a, b;
   scanf("%d", &a);
   scanf("%d", &b);

   if (a >= b)
      printf("%d", a);
   else
      printf("%d", b);

   return 0;
}
```

Experiment 9: Find the Largest of Three Numbers

(LONG DESCRIPTION)

Problem Description

Write a C program to find the largest among three integers using conditional statements.

The program should correctly handle:

- All values being different
- Two values being equal
- All values being equal
- Negative numbers

Input Format

A

B

C

Output Format

- Print the largest integer value

Sample Input

5

9

3

Sample Output

9

Hidden Test Cases

- 5 5 5
- -1 -2 -3
- 0 0 -1

Constraints

$-10^9 \leq A, B, C \leq 10^9$

Program

```
#include <stdio.h>


int main() {
    int a, b, c;
    scanf("%d %d %d", &a, &b, &c);


    if (a >= b && a >= c)
        printf("%d", a);
    else if (b >= a && b >= c)
        printf("%d", b);
    else
        printf("%d", c);


    return 0;
}
```

Experiment 10: Check Whether a Year is a Leap Year

(LONG DESCRIPTION)

Problem Description

Write a C program to determine whether a given year is a leap year.

A year is considered a leap year if:

- It is divisible by 4 and not divisible by 100 OR
- It is divisible by 400

The program must correctly handle century years.

Input Format

- A single integer representing the year

Output Format

- Print Leap Year or Not a Leap Year

Sample Input

2024

Sample Output

Leap Year

Hidden Test Cases

- 1900
- 2000
- 2100
- Large year values

Constraints

1 ≤ Year ≤ 9999

Program

```
#include <stdio.h>


int main() {
    int year;
    scanf("%d", &year);


    if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0)
```

```
        printf("Leap Year");
    else
        printf("Not a Leap Year");

    return 0;
}
```

Experiment 11: Simple Calculator Using switch

(LONG DESCRIPTION – TOUGH)

Problem Description

Develop a C program that performs basic arithmetic operations using the switch statement.

The program should:

- Read two integers
- Read an arithmetic operator (+, -, *, /)
- Perform the corresponding operation
- Display the result

Assume valid input and no division by zero.

Input Format

enter first number: <int>

enter second number: <int>

enter operator: <char>

Output Format

result = <value>

Sample Input

enter first number: 10

```
enter second number: 5

enter operator: *
```

Sample Output

```
result = 50
```

Hidden Test Cases

- All operators
- Negative values
- Large integers

Constraints

-10^9 ≤ A, B ≤ 10^9

Program

```
#include <stdio.h>

int main() {
    int a, b;
    char op;

    scanf("enter first number: %d", &a);
    scanf("enter second number: %d", &b);
    scanf("enter operator: %c", &op);

    switch (op) {
        case '+': printf("result = %d", a + b); break;
        case '-': printf("result = %d", a - b); break;
        case '*': printf("result = %d", a * b); break;
        case '/': printf("result = %d", a / b); break;
    }

    return 0;
}
```

Experiment 12: Print Numbers from 1 to N

(LONG DESCRIPTION)

Problem Description

Write a C program that reads a positive integer N and prints all numbers from **1 to N** in increasing order.

The program must use a looping statement to generate the sequence.

The program should correctly handle:

- Small values of N
- Large values of N
- The case when N is equal to 1

Input Format

- A single integer N

Output Format

- Print numbers from 1 to N, each separated by a space

Sample Input

5

Sample Output

```
1 2 3 4 5
```

Hidden Test Cases

- 1
- Large values like 10000
- Boundary values

Constraints

1 ≤ N ≤ 10^5

Program

```
#include <stdio.h>

int main() {
    int n, i;
    scanf("%d", &n);

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

    return 0;
}
```

Experiment 13: Find the Sum of First N Natural Numbers

(LONG DESCRIPTION)

Problem Description

Write a C program to calculate the sum of the first N natural numbers using a loop.

Natural numbers start from 1.

The program must correctly compute the sum even for large values of N.

Input Format

- A single integer N

Output Format

- Print the sum of the first N natural numbers

Sample Input

10

Sample Output

55

Hidden Test Cases

- 1
- Large values like 100000
- Values close to integer limits

Constraints

1 ≤ N ≤ 10^5

Program

```
#include <stdio.h>

int main() {
    int n, i;
    long long sum = 0;
    scanf("%d", &n);


    for (i = 1; i <= n; i++) {
        sum += i;
    }


    printf("%lld", sum);
    return 0;
}
```

Experiment 14: Print Multiplication Table of a Number

(SHORT)

Problem Description

Write a C program to print the multiplication table of a given number.

Sample Input

```
5
```

Sample Output

```
5 x 1 = 5

5 x 2 = 10

5 x 3 = 15

5 x 4 = 20

5 x 5 = 25

5 x 6 = 30

5 x 7 = 35

5 x 8 = 40

5 x 9 = 45

5 x 10 = 50
```

Program

```
#include <stdio.h>


int main() {
    int n, i;
    scanf("%d", &n);


    for (i = 1; i <= 10; i++) {
        printf("%d x %d = %d\n", n, i, n * i);
    }


    return 0;
}
```

Experiment 15: Find the Factorial of a Number

(LONG DESCRIPTION)

Problem Description

Write a C program to find the factorial of a given non-negative integer using a loop.

Factorial of a number N is defined as:

N! = N × (N-1) × (N-2) × ... × 1

The program must:

- Handle the case when N = 0
- Avoid overflow for large values by using appropriate data types

Input Format

- A single integer N

Output Format

- Print the factorial of N

Sample Input

5

Sample Output

120

Hidden Test Cases

- 0
- 1
- Larger values like 15

Constraints

0 ≤ N ≤ 20

Program

```
#include <stdio.h>

int main() {
    int n, i;
    long long fact = 1;

    scanf("%d", &n);

    for (i = 1; i <= n; i++) {
        fact *= i;
    }
    printf("%lld", fact);
    return 0;
}
```

Experiment 16: Reverse a Given Number

(LONG DESCRIPTION)

Problem Description

Write a C program to reverse a given integer.

The program must:

- Extract digits one by one
- Construct the reversed number
- Handle numbers ending with zero

Input Format

- A single integer N

Output Format

- Print the reversed number

Sample Input

1204

Sample Output

4021

Hidden Test Cases

- Numbers ending with zero
- Single-digit numbers

- Large numbers

Constraints

0 ≤ N ≤ 10^9

Program

```
#include <stdio.h>

int main() {
    int n, rev = 0;

    scanf("%d", &n);

    while (n != 0) {
        rev = rev * 10 + (n % 10);
        n /= 10;
    }

    printf("%d", rev);
    return 0;
}
```

Experiment 17: Check Whether a Number is a Palindrome

(LONG DESCRIPTION – TOUGH)

Problem Description

Write a C program to check whether a given number is a **palindrome**.

A number is called a palindrome if it remains the same when its digits are reversed.

The program must:

- Reverse the number
- Compare it with the original number
- Print the correct result

Input Format

- A single integer N

Output Format

- Print Palindrome or Not Palindrome

Sample Input

121

Sample Output

Palindrome

Hidden Test Cases

- Single-digit numbers
- Numbers with trailing zeros
- Large palindromes

Constraints

0 ≤ N ≤ 10^9

Program

```
#include <stdio.h>

int main() {
   int n, temp, rev = 0;
```

```
    scanf("%d", &n);
    temp = n;

    while (n != 0) {
        rev = rev * 10 + (n % 10);
        n /= 10;
    }

    if (rev == temp)
        printf("Palindrome");
    else
        printf("Not Palindrome");

    return 0;
}
```

Experiment 18: Read and Display Elements of an Array
(LONG DESCRIPTION)

Problem Description
In many real-world programs, data is not handled as a single value but as a collection of values. Arrays are used in C to store multiple values of the same data type under a single name.

Write a C program that reads an integer value N representing the number of elements,

followed by N integer values, and stores them in an array. The program should then display all the elements of the array in the same order in which they were entered.

The program must correctly handle cases where:

- The number of elements is small or large
- Elements include positive numbers, negative numbers, and zero

- Input size reaches the maximum allowed limit

This program helps in understanding array declaration, input traversal, and output traversal using loops.

Input Format

- First line contains an integer N (number of elements)
- Second line contains N integers separated by space

Output Format

- Print all array elements separated by a space

Sample Input

5

10 20 30 40 50

Sample Output

10 20 30 40 50

Hidden Test Cases

- N = 1
- N = maximum allowed size
- All elements are zero
- Mix of positive and negative numbers

Constraints

1 ≤ N ≤ 1000

Program

```
#include <stdio.h>

int main() {
    int n, i;
    int arr[1000];

    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

Experiment 19: Find the Sum and Average of Array Elements
(LONG DESCRIPTION)

Problem Description
Arrays are often used to store numerical data that needs to be analyzed. One of the most common operations performed on arrays is finding the sum and average of the stored elements.

Write a C program that reads an integer N representing the number of elements in an array, followed by N integer values. The program should calculate the sum of all array elements and also compute the average value.

The program must:

- Use a loop to traverse the array
- Accumulate the sum correctly
- Compute the average using floating-point division

This problem tests the student's understanding of array traversal, accumulation logic, and type conversion.

Input Format

- First line contains integer N
- Second line contains N integers

Output Format

- Print the sum
- Print the average up to two decimal places

Sample Input

```
4
10 20 30 40
```

Sample Output

```
100
25.00
```

Hidden Test Cases

- All elements are the same
- Array contains negative values
- Large values that increase the sum

Constraints

1 ≤ N ≤ 1000

Program

```
#include <stdio.h>

int main() {
    int n, i;
    int arr[1000];
    int sum = 0;

    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
        sum += arr[i];
    }
    printf("%d\n", sum);
    printf("%.2f", (float)sum / n);

    return 0;
}
```

Experiment 20: Find the Largest Element in an Array
(SHORT)

Problem Description
Write a C program to find the largest element in a given array of integers.

Sample Input

```
5
2 8 1 6 4
```

Sample Output

```
8
```

Program

```
#include <stdio.h>

int main() {
    int n, i;
    int arr[1000], max;

    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    max = arr[0];
    for (i = 1; i < n; i++) {
        if (arr[i] > max)
            max = arr[i];
    }

    printf("%d", max);
    return 0;
}
```

Experiment 21: Find the Smallest Element in an Array
(SHORT)

Problem Description
Write a C program to find the smallest element in a given array.

Sample Input

```
5
2 8 1 6 4
```

Sample Output

```
1
```

Program

```
#include <stdio.h>

int main() {
    int n, i;
    int arr[1000], min;

    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    min = arr[0];

    for (i = 1; i < n; i++) {

        if (arr[i] < min)

            min = arr[i];

    }


    printf("%d", min);

    return 0;

}
```

Experiment 22: Linear Search in an Array
(LONG DESCRIPTION)

Problem Description
Searching is one of the most fundamental operations performed on data collections. Linear search is the simplest searching technique where each element of the array is compared one by one with the target value.

Write a C program that reads an integer N, followed by N array elements, and another integer key to be searched. The program should check whether the key is present in the array. If found, print the position (index + 1). If not found, print an appropriate message.

The program must correctly handle:

- Key present at the beginning, middle, or end
- Key not present in the array
- Arrays with duplicate values

This program strengthens understanding of loops, condition checking, and early termination.

Input Format

- First line contains integer N
- Second line contains N integers
- Third line contains integer key

Output Format

- Print position of the element if found

- Otherwise print Element not found

Sample Input

```
5
10 20 30 40 50
30
```

Sample Output

```
3
```

Hidden Test Cases

- Key not present
- Key present multiple times
- Key is the first or last element

Constraints

1 ≤ N ≤ 1000

Program

```c
#include <stdio.h>

int main() {
    int n, i, key;
    int arr[1000];
    int found = 0;

    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    scanf("%d", &key);
    for (i = 0; i < n; i++) {
        if (arr[i] == key) {
            printf("%d", i + 1);
            found = 1;
            break;
        }
    }

    if (!found)
        printf("Element not found");

    return 0;
}
```

Experiment 23: Reverse an Array
(LONG DESCRIPTION – TOUGH)

Problem Description
Reversing an array is a common operation that requires careful index manipulation. In this problem, the elements of the array must be rearranged such that the first element becomes the last, the second becomes the second last, and so on.

Write a C program that reads an integer N followed by N integer values stored in an array. The program should reverse the array **in place** and then display the reversed array.

The program must:

- Use index-based swapping
- Not use an additional array
- Correctly handle even and odd sized arrays

This problem tests logical thinking, array indexing, and loop control.

Input Format

- First line contains integer N

- Second line contains N integers

Output Format

- Print the reversed array

Sample Input

5

1 2 3 4 5

Sample Output

5 4 3 2 1

Hidden Test Cases

- N = 1
- N = 2
- Large arrays
- Arrays with duplicate values

Constraints

1 ≤ N ≤ 1000

Program

```
#include <stdio.h>

int main() {
    int n, i, temp;
    int arr[1000];

    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    for (i = 0; i < n / 2; i++) {
        temp = arr[i];
        arr[i] = arr[n - i - 1];
        arr[n - i - 1] = temp;
    }

    for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

Experiment 24: Read and Display a String
(LONG DESCRIPTION)

Problem Description
In C programming, a string is represented as an array of characters terminated by a special character called the null character (\\0). Understanding how to read and display strings is essential before performing any string manipulation operations.

Write a C program that reads a string entered by the user and displays the same string as output. The program should be able to handle strings containing spaces, which means standard input functions that stop at whitespace should not be used.

The program must correctly:

- Read an entire line of text including spaces
- Store the string in a character array
- Display the string exactly as entered

This problem introduces safe string input handling and reinforces the concept of null-terminated character arrays.

Input Format

- A single line containing a string (may include spaces)

Output Format

- Print the entered string

Sample Input

C programming language

Sample Output

C programming language

Hidden Test Cases

- String with only one character
- String containing multiple spaces
- Empty string input

Constraints

String length ≤ 100

Program

```
#include <stdio.h>


int main() {
    char str[101];
    fgets(str, sizeof(str), stdin);
    printf("%s", str);
    return 0;
}
```

Experiment 25: Find the Length of a String
(LONG DESCRIPTION)

Problem Description
Finding the length of a string is a common operation used in many applications such as validation, searching, and formatting. The length of a string is defined as the number of characters present in the string excluding the null terminator.

Write a C program that reads a string and calculates its length without using the built-in strlen() function. The program should traverse the string character by character until the null character is encountered and count the number of characters.

This problem strengthens understanding of:

- Character arrays
- Loop traversal
- The role of the null terminator

Input Format

- A single line containing a string

Output Format

- Print the length of the string as an integer

Sample Input

Hello

Sample Output

5

Hidden Test Cases

- Empty string
- String with spaces
- Long strings

Constraints

String length ≤ 100

Program

```
#include <stdio.h>

int main() {
   char str[101];
   int length = 0;

   fgets(str, sizeof(str), stdin);

   while (str[length] != '\0' && str[length] != '\n') {
       length++;
   }

   printf("%d", length);
   return 0;
}
```

Experiment 26: Copy One String to Another
(SHORT)

Problem Description
Write a C program to copy one string into another string.

Sample Input

```
Hello World
```

Sample Output

```
Hello World
```

Program

```
#include <stdio.h>

int main() {
    char src[101], dest[101];
    int i = 0;

    fgets(src, sizeof(src), stdin);

    while (src[i] != '\0') {
        dest[i] = src[i];
        i++;
    }
    dest[i] = '\0';
    printf("%s", dest);
    return 0;
}
```

Experiment 27: Compare Two Strings
(LONG DESCRIPTION)

Problem Description
String comparison is used to determine whether two strings are equal or which one is lexicographically greater. In C, strings cannot be compared using relational operators directly and must be compared character by character.

Write a C program that reads two strings and compares them without using the built-in strcmp() function. The program should check each corresponding character and determine whether the strings are equal or not.

The program must:

- Compare strings character by character
- Stop comparison when a mismatch is found or the end of strings is reached
- Produce correct output even if strings differ in length

Input Format

- First line contains the first string
- Second line contains the second string

Output Format

- Print Equal if both strings are the same
- Print Not Equal otherwise

Sample Input

Hello

Hello

Sample Output

Equal

Hidden Test Cases

- Strings with different lengths
- Strings with same prefix but different endings
- Case-sensitive comparisons

Constraints

String length ≤ 100

Program

```
#include <stdio.h>

int main() {
   char str1[101], str2[101];
   int i = 0, equal = 1;

   fgets(str1, sizeof(str1), stdin);
   fgets(str2, sizeof(str2), stdin);

   while (str1[i] != '\0' || str2[i] != '\0') {
      if (str1[i] != str2[i]) {
         equal = 0;
         break;
      }
      i++;
   }

   if (equal)
      printf("Equal");
   else
      printf("Not Equal");
```

```
    return 0;
}
```

kand. The string should be reversed such that the first character becomes the last and the last character becomes the first.

Write a C program that reads a string and reverses it without using built-in reverse functions. The program should modify the string by swapping characters from the beginning and end.

The program must:

- Correctly identify the length of the string
- Swap characters in place
- Handle strings with spaces

Input Format

- A single line containing a string

Output Format

- Print the reversed string

Sample Input

programming

Sample Output

gnimmargorp

Hidden Test Cases

- Single-character string
- Palindrome strings
- Strings containing spaces

Constraints

String length ≤ 100

Program

```
#include <stdio.h>


int main() {
    char str[101];
    int i, j;
    char temp;

    fgets(str, sizeof(str), stdin);

    for (j = 0; str[j] != '\0' && str[j] != '\n'; j++);

    for (i = 0; i < j / 2; i++) {
        temp = str[i];
        str[i] = str[j - i - 1];
        str[j - i - 1] = temp;
    }

    printf("%s", str);
    return 0;
}
```

Experiment 29: Check Whether a String is a Palindrome
(LONG DESCRIPTION – TOUGH)

Problem Description
A string is said to be a palindrome if it reads the same forwards and backwards. Checking for palindromes is commonly used in validation, pattern recognition, and algorithmic problems.

Write a C program that reads a string and checks whether it is a palindrome. The program should reverse the string and compare it with the original string to determine the result.

The program must:

- Ignore the newline character introduced during input
- Correctly compare characters
- Work for both even and odd length strings

Input Format

- A single line containing a string

Output Format

- Print Palindrome if the string is a palindrome
- Print Not Palindrome otherwise

Sample Input

madam

Sample Output

Palindrome

Hidden Test Cases

- Single-character strings
- Strings with spaces
- Non-palindrome strings

Constraints

String length ≤ 100

Program

```
#include <stdio.h>

int main() {
    char str[101];
    int i, j, flag = 1;

    fgets(str, sizeof(str), stdin);

    for (j = 0; str[j] != '\0' && str[j] != '\n'; j++);

    for (i = 0; i < j / 2; i++) {
        if (str[i] != str[j - i - 1]) {
            flag = 0;
            break;

        }

    }



    if (flag)

        printf("Palindrome");

    else

        printf("Not Palindrome");



    return 0;

}
```

Experiment 30: Simple Function to Add Two Numbers
(SHORT)

Problem Description
Write a C program that uses a user-defined function to add two integers and display the result.

Sample Input

```
10

20
```

Sample Output

```
30
```

Program

```
#include <stdio.h>



int add(int a, int b) {

    return a + b;

}



int main() {
    int x, y;

    scanf("%d %d", &x, &y);

    printf("%d", add(x, y));

    return 0;

}
```

Experiment 31: Function to Find the Maximum of Two Numbers
(SHORT)

Problem Description
Write a C program that uses a function to find the maximum of two integers.

Sample Input

```
15

9
```

Sample Output

```
15
```

Program

```
#include <stdio.h>


int max(int a, int b) {

    return (a > b) ? a : b;

}


int main() {

    int x, y;

    scanf("%d %d", &x, &y);

    printf("%d", max(x, y));

    return 0;

}
```

Experiment 32: Function to Find Factorial of a Number
(LONG DESCRIPTION)

Problem Description
Functions are often used to isolate a specific computation so that it can be reused multiple times in a program. One such computation is finding the factorial of a number, which is frequently used in mathematical and combinatorial problems.

Write a C program that uses a user-defined function to calculate the factorial of a given non-negative integer. The factorial function should receive the number as a parameter, compute the factorial using an iterative approach, and return the result to the calling function.

The program must:

- Correctly handle the case when the input value is 0
- Use an appropriate data type to avoid overflow
- Separate the logic of computation from input and output

This experiment reinforces the concept of function definition, function calling, parameter passing, and return values.

Input Format

- A single integer N

Output Format

- Print the factorial of N

Sample Input

5

Sample Output

120

Hidden Test Cases

- 0
- 1
- Larger values like 15

Constraints

$0 \leq N \leq 20$

Program

```
#include <stdio.h>


long long factorial(int n) {
   long long fact = 1;
   int i;
   for (i = 1; i <= n; i++) {
      fact *= i;
   }
   return fact;
}


int main() {
   int n;
   scanf("%d", &n);
   printf("%lld", factorial(n));
   return 0;
}
```

Experiment 33: Demonstrate Call by Value
(LONG DESCRIPTION)

Problem Description
In C, function arguments are passed using the call by value mechanism. This means that a copy of the actual argument is passed to the function, and any changes made to the parameter inside the function do not affect the original variable.

Write a C program to demonstrate the concept of call by value using a function that attempts to swap two numbers. The program should clearly show that the values of the variables in the calling function remain unchanged after the function call.

This experiment helps students understand:

- How data is passed to functions
- Why changes inside a function may not reflect outside

- The importance of memory separation between caller and callee

Input Format

- Two integers A and B

Output Format

- Print the values of A and B before and after the function call

Sample Input

10 20

Sample Output

```
Before swapping: 10 20

After swapping: 10 20
```

Hidden Test Cases

- Equal values
- Negative values

Constraints

-10^9 ≤ A, B ≤ 10^9

Program

```
#include <stdio.h>


void swap(int a, int b) {

   int temp = a;

   a = b;

   b = temp;

}


int main() {

   int x, y;

   scanf("%d %d", &x, &y);
```

```
    printf("Before swapping: %d %d\n", x, y);

    swap(x, y);

    printf("After swapping: %d %d", x, y);


    return 0;

}
```

Experiment 34: Demonstrate Call by Reference
(LONG DESCRIPTION)

Problem Description
Unlike call by value, call by reference allows a function to modify the actual variables passed to it. In C, this is achieved using pointers. Understanding call by reference is crucial for working with arrays, dynamic memory, and complex data structures.

Write a C program that swaps two numbers using call by reference. The function should receive the addresses of the variables and modify their values directly.

This experiment emphasizes:

- Pointer usage in functions
- Difference between value and reference passing
- Real-world need for call by reference

Input Format

- Two integers A and B

Output Format

- Print the values before and after swapping

Sample Input

10 20

Sample Output

Before swapping: 10 20

After swapping: 20 10

Hidden Test Cases

- Negative numbers
- Zero values

Constraints

-10^9 ≤ A, B ≤ 10^9

Program

```
#include <stdio.h>

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int x, y;
    scanf("%d %d", &x, &y);

    printf("Before swapping: %d %d\n", x, y);
    swap(&x, &y);
    printf("After swapping: %d %d", x, y);

    return 0;
}
```

Experiment 35: Recursive Function to Find Fibonacci Series
(LONG DESCRIPTION – TOUGH)

Problem Description
Recursion is a technique where a function calls itself to solve a problem by breaking it into

smaller subproblems. While recursion can simplify certain problems conceptually, it must be used carefully to avoid excessive memory usage.

Write a C program that uses a recursive function to generate the Fibonacci series up to a given number of terms. The Fibonacci series is defined as follows:

- The first two terms are 0 and 1
- Each subsequent term is the sum of the previous two terms

The program should:

- Use recursion to compute Fibonacci numbers
- Display the series correctly
- Handle small values of N properly

This experiment builds a strong understanding of recursive calls, base conditions, and function call stacks.

Input Format

- A single integer N representing the number of terms

Output Format

- Print the Fibonacci series up to N terms

Sample Input

5

Sample Output

0 1 1 2 3

Hidden Test Cases

- 1
- 2
- Larger values of N

Constraints

1 ≤ N ≤ 20

Program

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

```
int fibonacci(int n) {
    if (n == 0)
        return 0;
    if (n == 1)
        return 1;
    return fibonacci(n - 1) + fibonacci(n - 2);
}

int main() {
    int n, i;
    scanf("%d", &n);

    for (i = 0; i < n; i++) {
        printf("%d ", fibonacci(i));
    }

    return 0;
}
```

Experiment 36: Introduction to Pointers – Display Value and Address
(LONG DESCRIPTION)

Problem Description
In C programming, every variable stored in memory has an associated memory address. A pointer is a variable that stores the memory address of another variable. Understanding this relationship between a variable, its address, and a pointer is essential before moving on to advanced topics such as arrays, dynamic memory allocation, and data structures.

Write a C program that declares an integer variable, assigns a value to it, and then uses a pointer to store the address of that variable. The program should display the value of the variable, the address of the variable, and the value accessed using the pointer.

This program helps students understand:

- How memory addresses are accessed using the address-of operator

- How pointers store addresses
- How dereferencing works to access the value stored at an address

Input Format

- A single integer value

Output Format

- Print the value of the variable
- Print the address of the variable
- Print the value using the pointer

Sample Input

```
10
```

Sample Output

```
Value: 10
Address: <address>
Pointer Value: 10
```

Hidden Test Cases

- Negative values
- Zero

Constraints

-10^9 ≤ N ≤ 10^9

Program

```
#include <stdio.h>

int main() {
    int x;
    int *ptr;

    scanf("%d", &x);
    ptr = &x;
    printf("Value: %d\n", x);
    printf("Address: %p\n", (void*)&x);
    printf("Pointer Value: %d", *ptr);

    return 0;
}
```

Experiment 37: Swap Two Numbers Using Pointers
(LONG DESCRIPTION)

Problem Description
Swapping two variables is a common task in programming. When functions are involved, swapping values requires the use of pointers so that the actual variables in memory are modified. This program demonstrates how pointers enable call by reference behavior in C.

Write a C program that reads two integers and swaps their values using pointers. The program should clearly show the values before swapping and after swapping.

This experiment reinforces:

- Pointer parameters in functions
- Dereferencing pointers to modify values
- Difference between call by value and call by reference

Input Format

- Two integers A and B

Output Format

- Print values before swapping
- Print values after swapping

Sample Input

```
10 20
```

Sample Output

```
Before swapping: 10 20
```

After swapping: 20 10

Hidden Test Cases

- Equal values
- Negative numbers
- Zero values

Constraints

-10^9 ≤ A, B ≤ 10^9

Program

```
#include <stdio.h>

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int x, y;
    scanf("%d %d", &x, &y);

    printf("Before swapping: %d %d\n", x, y);
    swap(&x, &y);
    printf("After swapping: %d %d", x, y);

    return 0;
}
```

Experiment 38: Access Array Elements Using Pointers
(SHORT)

Problem Description
Write a C program to access and display array elements using pointers instead of array indexing.

Sample Input

```
5

1 2 3 4 5
```

Sample Output

```
1 2 3 4 5
```

Program

```
#include <stdio.h>


int main() {

    int n, i;

    int arr[1000];

    int *ptr;


    scanf("%d", &n);

    for (i = 0; i < n; i++) {

        scanf("%d", &arr[i]);

    }


    ptr = arr;

    for (i = 0; i < n; i++) {

        printf("%d ", *(ptr + i));

    }


    return 0;
```

}

Experiment 39: Pointer Arithmetic Demonstration
(LONG DESCRIPTION)

Problem Description
Pointer arithmetic allows pointers to move through memory locations based on the size of the data type they point to. This is especially important when working with arrays, where pointer increments move to the next element automatically.

Write a C program that demonstrates pointer arithmetic by iterating through an array using pointer increment operations. The program should display each element along with its corresponding memory address.

This experiment helps students understand:

- How pointer increment works
- Why pointer arithmetic depends on data type size
- Memory layout of arrays

Input Format

- Integer N followed by N integers

Output Format

- Print each element and its address

Sample Input

3

10 20 30

Sample Output

10 <address>

20 <address>

30 <address>

Hidden Test Cases

- N = 1
- Large arrays

Constraints

1 ≤ N ≤ 1000

Program

```
#include <stdio.h>

int main() {
   int n, i;
   int arr[1000];
   int *ptr;

   scanf("%d", &n);
   for (i = 0; i < n; i++) {
      scanf("%d", &arr[i]);
   }

   ptr = arr;
   for (i = 0; i < n; i++) {
      printf("%d %p\n", *(ptr + i), (void*)(ptr + i));
   }

   return 0;
}
```

Experiment 40: Pointers and Functions
(LONG DESCRIPTION)

Problem Description
Pointers are often passed to functions to allow the function to access and modify data stored in the calling function. This is especially useful when working with arrays and large data structures.

Write a C program that passes an array to a function using pointers and calculates the sum of the array elements inside the function.

This experiment demonstrates:

- Passing arrays as pointers
- Processing data inside functions
- Returning computed results

Input Format

- Integer N
- N integers

Output Format

- Print the sum of array elements

Sample Input

```
4

10 20 30 40
```

Sample Output

```
100
```

Hidden Test Cases

- Single element array
- Large arrays

Constraints

1 ≤ N ≤ 1000

Program

```
#include <stdio.h>


int sumArray(int *arr, int n) {

    int i, sum = 0;

    for (i = 0; i < n; i++) {

        sum += *(arr + i);
    }
    return sum;
}

int main() {
    int n, arr[1000];
    scanf("%d", &n);

    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    printf("%d", sumArray(arr, n));
    return 0;
}
```

Experiment 41: Pointer to Pointer
(LONG DESCRIPTION – TOUGH)

Problem Description
A pointer to a pointer stores the address of another pointer. This concept is important in advanced applications such as dynamic memory allocation, multidimensional arrays, and complex data structures.

Write a C program that demonstrates the use of a pointer to a pointer by storing the address of a pointer and accessing the original variable through double dereferencing.

The program must clearly show:

- Value of the variable
- Value accessed using a pointer
- Value accessed using a pointer to a pointer

Input Format

- A single integer value

Output Format

- Print the value using variable, pointer, and pointer to pointer

Sample Input

```
25
```

Sample Output

```
25
25
25
```

Hidden Test Cases

- Zero
- Negative values

Constraints

-10^9 ≤ N ≤ 10^9

Program

```c
#include <stdio.h>

int main() {
   int x;
   int *p;
   int **pp;

   scanf("%d", &x);

   p = &x;
   pp = &p;

   printf("%d\n", x);
```

```
    printf("%d\n", *p);

    printf("%d", **pp);


    return 0;

}
```

Experiment 42: Allocate Memory Using malloc and Store Values
(LONG DESCRIPTION)

Problem Description
In C, dynamic memory allocation allows a program to request memory during execution using functions such as malloc. Unlike static arrays, dynamically allocated memory can be sized based on user input, making programs more flexible and memory-efficient.

Write a C program that reads an integer N, dynamically allocates memory for N integers using malloc, stores N integer values in the allocated memory, and then displays those values.

The program must:

- Allocate memory only after reading the input size
- Check whether memory allocation is successful
- Access dynamically allocated memory using pointers
- Display all values correctly

This program helps students understand how memory is requested from the heap and accessed through pointers.

Input Format

- First line contains integer N
- Second line contains N integers

Output Format

- Print the N integers separated by space

Sample Input

```
5

10 20 30 40 50
```

Sample Output

```
10 20 30 40 50
```

Hidden Test Cases

- N = 1
- Large value of N
- All values are zero

Constraints

1 ≤ N ≤ 1000

Program

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

int main() {
    int n, i;
    int *arr;

    scanf("%d", &n);

    arr = (int *)malloc(n * sizeof(int));
    if (arr == NULL) {
        return 0;
    }

    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    free(arr);

    return 0;

}
```

Experiment 43: Allocate and Initialize Memory Using calloc
(LONG DESCRIPTION)

Problem Description
The calloc function is used to allocate memory dynamically and initialize all allocated bytes to zero. This is useful when the program requires a clean memory block without any garbage values.

Write a C program that dynamically allocates memory for N integers using calloc, reads values into the allocated memory, and displays the elements.

This experiment demonstrates:

- Difference between malloc and calloc
- Automatic initialization of memory
- Safe access to heap memory

Input Format

- Integer N
- N integers

Output Format

- Print all elements stored in memory

Sample Input

```
3

4 5 6
```

Sample Output

```
4 5 6
```

Hidden Test Cases

- N = 0 (no elements)

- All elements are zero

Constraints

0 ≤ N ≤ 1000

Program

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

int main() {
    int n, i;
    int *arr;

    scanf("%d", &n);

    arr = (int *)calloc(n, sizeof(int));
    if (arr == NULL) {
        return 0;
    }

    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

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

Experiment 44: Resize Memory Using realloc
(SHORT)

Problem Description
Write a C program to demonstrate resizing of dynamically allocated memory using realloc.

Sample Input

```
2

10 20

4

30 40
```

Sample Output

```
10 20 30 40
```

Program

```
#include <stdio.h>

#include <stdlib.h>


int main() {

   int *arr;

   int n1, n2, i;


   scanf("%d", &n1);

   arr = (int *)malloc(n1 * sizeof(int));


   for (i = 0; i < n1; i++) {

      scanf("%d", &arr[i]);

   }


   scanf("%d", &n2);
```

```
    arr = (int *)realloc(arr, (n1 + n2) * sizeof(int));


    for (i = n1; i < n1 + n2; i++) {
        scanf("%d", &arr[i]);
    }


    for (i = 0; i < n1 + n2; i++) {
        printf("%d ", arr[i]);
    }


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

Experiment 45: Find the Sum of Elements Using Dynamic Memory
(LONG DESCRIPTION)

Problem Description
Dynamic memory allocation is often combined with computation functions. This program focuses on using dynamically allocated memory to store input values and then perform calculations on those values.

Write a C program that dynamically allocates memory for N integers, stores the values, calculates the sum of all elements, and displays the result.

This experiment emphasizes:

- Safe dynamic allocation
- Loop-based traversal
- Proper memory deallocation

Input Format

- Integer N
- N integers

Output Format

- Print the sum of the elements

Sample Input

```
4
5 10 15 20
```

Sample Output

```
50
```

Hidden Test Cases

- Single element
- Large values

Constraints

1 ≤ N ≤ 1000

Program

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

int main() {
    int n, i;
    int *arr;
    int sum = 0;

    scanf("%d", &n);
    arr = (int *)malloc(n * sizeof(int));

    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
        sum += arr[i];
    }
    printf("%d", sum);


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

Experiment 46: Memory Deallocation Using free
(LONG DESCRIPTION)

Problem Description
Memory allocated dynamically using malloc or calloc remains allocated until it is explicitly released using the free function. Failure to release memory leads to memory leaks, which can degrade system performance.

Write a C program that demonstrates proper allocation and deallocation of memory. The program should allocate memory, use it, and then release it using free.

This experiment teaches:

- Importance of freeing memory
- Correct order of allocation and deallocation
- Safe programming practices

Input Format

- Integer N
- N integers

Output Format

- Print the stored elements

Sample Input

3

1 2 3

Sample Output

1 2 3

Hidden Test Cases

- N = 1
- Large N

Constraints

1 ≤ N ≤ 1000

Program

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

int main() {
   int n, i;
   int *arr;

   scanf("%d", &n);
   arr = (int *)malloc(n * sizeof(int));

   for (i = 0; i < n; i++) {
      scanf("%d", &arr[i]);
   }

   for (i = 0; i < n; i++) {
      printf("%d ", arr[i]);
   }

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

Experiment 47: Detect and Avoid Memory Leak
(LONG DESCRIPTION – TOUGH)

Problem Description
A memory leak occurs when dynamically allocated memory is not released after use. Over time, memory leaks can cause programs to consume excessive memory and crash.

Write a C program that dynamically allocates memory, uses it, and then ensures that all allocated memory is properly freed before program termination. The program should demonstrate correct memory management practices.

This experiment reinforces:

- Identifying memory leaks
- Proper placement of free
- Writing safe and efficient C programs

Input Format

- Integer N
- N integers

Output Format

- Print all elements

Sample Input

2

7 8

Sample Output

7 8

Hidden Test Cases

- Repeated execution scenarios
- Large memory allocation

Constraints

1 ≤ N ≤ 1000

Program

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

```
#include <stdlib.h>

int main() {
    int n, i;
    int *arr;

    scanf("%d", &n);
    arr = (int *)malloc(n * sizeof(int));

    if (arr == NULL) {
        return 0;
    }

    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    free(arr);
    arr = NULL;

    return 0;
}
```

Experiment 48: Define and Display Student Details Using Structure (LONG DESCRIPTION)

Problem Description
In real-world applications, data related to an entity is often composed of multiple attributes of different data types. For example, a student has a roll number, a name, and marks. Using separate variables for each attribute becomes inefficient and unmanageable as the program grows. Structures in C allow grouping related data of different data types under a single name.

Write a C program that defines a structure named Student containing fields for roll number, name, and marks. The program should read the details of one student and display them in a clear format.

The program must correctly:

- Define a structure with multiple data types
- Read string and numeric data into structure members
- Access and display structure members using the dot operator

This experiment helps students understand how structures model real-world entities in C.

Input Format

- Integer representing roll number
- String representing student name
- Float representing marks

Output Format

- Display roll number, name, and marks on separate lines

Sample Input

```
101

Arun

85.5
```

Sample Output

```
Roll Number: 101

Name: Arun

Marks: 85.50
```

Hidden Test Cases

- Name containing spaces
- Marks with decimal values
- Boundary roll numbers

Constraints

Roll number > 0

Name length ≤ 50

Program

```
#include <stdio.h>

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

int main() {
    struct Student s;

    scanf("%d", &s.roll);
    scanf(" %[^\n]", s.name);
    scanf("%f", &s.marks);

    printf("Roll Number: %d\n", s.roll);
    printf("Name: %s\n", s.name);
    printf("Marks: %.2f", s.marks);

    return 0;
```

}

Experiment 49: Array of Structures
(LONG DESCRIPTION)

Problem Description
Often, programs need to store details of multiple entities of the same type, such as student records. Using an array of structures allows storing multiple structured records efficiently and accessing them using indices.

Write a C program that reads details of N students using an array of structures and displays all the stored records.

The program must:

- Use an array of structures
- Read and store multiple records
- Display each record correctly

This experiment builds confidence in handling collections of structured data.

Input Format

- Integer N
- For each student: roll number, name, marks

Output Format

- Display details of all students

Sample Input

```
2

1

Anu

78.5

2

Ravi

82
```

Sample Output

```
1 Anu 78.50
2 Ravi 82.00
```

Hidden Test Cases

- N = 1
- Maximum allowed records
- Same marks for multiple students

Constraints

1 ≤ N ≤ 50

Program

```
#include <stdio.h>

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

int main() {
    int n, i;
    struct Student s[50];

    scanf("%d", &n);

    for (i = 0; i < n; i++) {
        scanf("%d", &s[i].roll);
        scanf(" %[^\n]", s[i].name);
        scanf("%f", &s[i].marks);
    }
    for (i = 0; i < n; i++) {
        printf("%d %s %.2f\n", s[i].roll, s[i].name, s[i].marks);
    }

    return 0;
}
```

Experiment 50: Pass Structure to a Function (SHORT)

Problem Description
Write a C program to pass a structure to a function and display its contents.

Sample Input

```
10
Ramesh
90
```

Sample Output

```
10 Ramesh 90.00
```

Program

```
#include <stdio.h>

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

void display(struct Student s) {
    printf("%d %s %.2f", s.roll, s.name, s.marks);
}


int main() {
    struct Student s;


    scanf("%d", &s.roll);
    scanf(" %[^\n]", s.name);
    scanf("%f", &s.marks);


    display(s);
    return 0;
}
```

Experiment 51: Nested Structures
(LONG DESCRIPTION)

Problem Description
Nested structures are structures that contain other structures as members. They are used when data naturally has a hierarchical relationship, such as a student having an address with multiple components.

Write a C program that defines a structure for Address and another structure for Student that includes Address as a member. The program should read and display complete student details.

This experiment helps students understand:

- Structure nesting
- Accessing nested structure members
- Real-world data representation

Input Format

- Roll number
- Name
- City

- Pin code

Output Format

- Display all details

Sample Input

```
12
Kiran
Chennai
600001
```

Sample Output

```
12 Kiran Chennai 600001
```

Hidden Test Cases

- City names with spaces
- Large pin codes

Constraints

Pin code must be numeric

Program

```c
#include <stdio.h>

struct Address {
   char city[50];
   int pin;
};

struct Student {
   int roll;
   char name[50];
   struct Address addr;
};
```

```
int main() {
    struct Student s;

    scanf("%d", &s.roll);
    scanf(" %[^\n]", s.name);
    scanf(" %[^\n]", s.addr.city);
    scanf("%d", &s.addr.pin);

    printf("%d %s %s %d", s.roll, s.name, s.addr.city, s.addr.pin);
    return 0;
}
```

Experiment 52: Union Demonstration
(SHORT)

Problem Description
Write a C program to demonstrate the use of a union and display its size.

Sample Input

```
10
```

Sample Output

```
10
```

Program

```
#include <stdio.h>

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

```
int main() {
    union Data d;
    scanf("%d", &d.i);
    printf("%d", d.i);
    return 0;
}
```

Experiment 53: Difference Between Structure and Union
(LONG DESCRIPTION – TOUGH)

Problem Description
Structures and unions are both user-defined data types in C, but they differ significantly in memory allocation and usage. While structures allocate separate memory for each member, unions share the same memory location for all members.

Write a C program that defines both a structure and a union with the same members and displays their sizes. This helps in understanding how memory is allocated differently for structures and unions.

This experiment reinforces:

- Memory allocation concepts
- Efficient memory usage
- Practical differences between structures and unions

Input Format

- No input

Output Format

- Print size of structure
- Print size of union

Sample Output

```
Size of structure: <value>

Size of union: <value>
```

Hidden Test Cases

- Different compiler architectures

Constraints

- Compiler dependent sizes

Program

```
#include <stdio.h>

struct DataStruct {
   int i;
   float f;
};

union DataUnion {
   int i;
   float f;
};

int main() {
   printf("Size of structure: %lu\n", sizeof(struct DataStruct));
   printf("Size of union: %lu", sizeof(union DataUnion));
   return 0;
}
```

Experiment 54: Write Data to a File
(LONG DESCRIPTION)

Problem Description
In many applications, data must be stored permanently so that it can be accessed even after the program terminates. Files provide a way to store data on secondary storage. Writing data to a file is one of the most fundamental file operations in C.

Write a C program that opens a file in write mode and writes a string entered by the user into the file. If the file already exists, its contents should be overwritten. If the file does not exist, it should be created automatically.

The program must:

- Open the file using the appropriate file mode
- Check whether the file is opened successfully
- Write data into the file using file handling functions
- Close the file properly after writing

This experiment helps students understand file pointers, file opening modes, and safe file writing practices.

Input Format

- A single line containing a string

Output Format

- Print Data written to file successfully

Sample Input

Welcome to C programming

Sample Output

Data written to file successfully

Hidden Test Cases

- Empty string
- String with spaces
- Long strings

Constraints

String length ≤ 200

Program

```
#include <stdio.h>


int main() {
    FILE *fp;
    char str[201];
    fgets(str, sizeof(str), stdin);


    fp = fopen("data.txt", "w");
    if (fp == NULL) {
        return 0;
    }


    fputs(str, fp);
    fclose(fp);


    printf("Data written to file successfully");
    return 0;
}
```

Experiment 55: Read Data from a File
(LONG DESCRIPTION)

Problem Description
Reading data from a file allows programs to retrieve stored information for processing or display. This experiment focuses on opening a file in read mode and reading its contents safely.

Write a C program that opens a file in read mode and displays its contents on the screen. The program should read the file line by line until the end of the file is reached.

The program must:

- Open the file using read mode
- Read file contents safely
- Display the content exactly as stored
- Close the file after reading

Input Format

- File data.txt must already exist

Output Format

- Print the contents of the file

Sample File Content

Welcome to C programming

Sample Output

Welcome to C programming

Hidden Test Cases

- File with multiple lines
- File containing numbers and symbols

Constraints

- File must exist

Program

```
#include <stdio.h>

int main() {
   FILE *fp;
   char ch;

   fp = fopen("data.txt", "r");
   if (fp == NULL) {
      return 0;
   }

   while ((ch = fgetc(fp)) != EOF) {
      putchar(ch);
   }

   fclose(fp);
```

```
    return 0;
}
```

Experiment 56: Append Data to an Existing File
(SHORT)

Problem Description
Write a C program to append text to an existing file without overwriting its contents.

Sample Input

```
This line is appended
```

Sample Output

```
Data appended successfully
```

Program

```
#include <stdio.h>


int main() {
    FILE *fp;
    char str[201];


    fgets(str, sizeof(str), stdin);


    fp = fopen("data.txt", "a");
    if (fp == NULL) {
        return 0;
    }


    fputs(str, fp);
    fclose(fp);


    printf("Data appended successfully");
    return 0;
}
```

Experiment 57: Copy Contents from One File to Another
(LONG DESCRIPTION)

Problem Description
File copying is a common operation where the contents of one file are duplicated into another file. This operation is useful for backup, duplication, and data migration purposes.

Write a C program that reads the contents of a source file and writes them into a destination file character by character. The program should ensure that all data is copied accurately.

The program must:

- Open the source file in read mode
- Open the destination file in write mode
- Copy each character from source to destination
- Close both files properly

Input Format

- Source file must exist

Output Format

- Print File copied successfully

Sample Output

File copied successfully

Hidden Test Cases

- Empty source file
- Source file with multiple lines

Constraints

- Source file must exist

Program

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

```
int main() {
    FILE *fs, *fd;
    char ch;

    fs = fopen("source.txt", "r");
    fd = fopen("destination.txt", "w");

    if (fs == NULL || fd == NULL) {
        return 0;
    }

    while ((ch = fgetc(fs)) != EOF) {
        fputc(ch, fd);
    }

    fclose(fs);
    fclose(fd);

    printf("File copied successfully");
    return 0;
}
```

Experiment 58: Count Number of Characters in a File
(SHORT)

Problem Description
Write a C program to count the number of characters present in a file.

Sample Output

```
Total characters: <count>
```

Program

```
#include <stdio.h>


int main() {
    FILE *fp;
    char ch;
    int count = 0;


    fp = fopen("data.txt", "r");
    if (fp == NULL) {
        return 0;
    }


    while ((ch = fgetc(fp)) != EOF) {
        count++;
    }


    fclose(fp);
    printf("Total characters: %d", count);
    return 0;
}
```

Experiment 59: Count Words and Lines in a File
(LONG DESCRIPTION – TOUGH)

Problem Description
Analyzing file content is an important task in many applications such as text processing, log analysis, and data validation. Counting the number of words and lines in a file requires careful handling of characters such as spaces and newline characters.

Write a C program that opens a file and counts:

- Total number of characters

- Total number of words
- Total number of lines

A word is defined as a sequence of characters separated by spaces or newlines.

The program must correctly handle:

- Multiple spaces
- Multiple lines
- Empty lines

This experiment strengthens understanding of file reading, condition checking, and character classification.

Input Format

- File must exist

Output Format

Characters: <count>

Words: <count>

Lines: <count>

Sample Output

Characters: 30

Words: 5

Lines: 2

Hidden Test Cases

- File with only one word
- File with empty lines
- File with trailing spaces

Constraints

- File must exist

Program

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

```
int main() {
    FILE *fp;
    char ch;
    int characters = 0, words = 0, lines = 0;
    int inWord = 0;

    fp = fopen("data.txt", "r");
    if (fp == NULL) {
        return 0;
    }

    while ((ch = fgetc(fp)) != EOF) {
        characters++;

        if (ch == '\n')
            lines++;

        if (ch == ' ' || ch == '\n' || ch == '\t') {
            inWord = 0;
        } else if (!inWord) {
            inWord = 1;
            words++;
        }
    }

    fclose(fp);

    printf("Characters: %d\n", characters);
    printf("Words: %d\n", words);

    printf("Lines: %d", lines);


    return 0;

}
```

Experiment 61: Command Line Argument Based Arithmetic Processor
(HARD – PLACEMENT LEVEL)

Problem Description
In C programming, command line arguments allow a program to receive inputs directly from the command line at the time of execution. These inputs are passed to the program through the parameters of the main function as argc (argument count) and argv (argument vector). Unlike programs that use scanf, command line–based programs are commonly used in system utilities, automation tools, and scripting environments.

Write a C program that performs arithmetic operations using command line arguments only. The program should accept exactly three command line arguments:

- First operand (integer)
- Arithmetic operator (+, -, *, /)
- Second operand (integer)

The program must not prompt the user for input or use scanf. All values must be taken strictly from the command line. The program should validate the number of arguments, identify the operator correctly, and perform the corresponding operation.

Special care must be taken to handle error conditions such as:

- Incorrect number of command line arguments
- Invalid operator symbols
- Division by zero
- Non-numeric operand values

If any invalid condition is detected, the program should print Invalid Input.

This experiment is designed to test deep understanding of program execution flow, argument parsing, defensive programming, and real-world usage of C programs outside interactive environments.

Input Format
Program execution format:

program_name <integer1> <operator> <integer2>

Output Format

- Print the result of the arithmetic operation
- If input is invalid, print Invalid Input

Sample Execution

./calc 10 + 5

Sample Output

15

Another Sample Execution

./calc 20 / 4

Sample Output

5

Hidden Test Cases

- Less than or more than three arguments
- Operator other than + - * /
- Division by zero
- Negative operands
- Very large integer values
- Non-numeric strings as operands

Constraints

-10^9 ≤ operands ≤ 10^9

Program

```
#include <stdio.h>

#include <stdlib.h>


int main(int argc, char *argv[]) {

    int a, b;

    char op;
    if (argc != 4) {
        printf("Invalid Input");
        return 0;
    }

    a = atoi(argv[1]);
    op = argv[2][0];
    b = atoi(argv[3]);

    if (op == '+') {
        printf("%d", a + b);
    } else if (op == '-') {
        printf("%d", a - b);
    } else if (op == '*') {
        printf("%d", a * b);
    } else if (op == '/') {
        if (b == 0) {
            printf("Invalid Input");
        } else {
            printf("%d", a / b);
        }
    } else {
        printf("Invalid Input");
    }

    return 0;
}
```

Experiment 62: Identify and Fix Undefined Behavior in Pointer and Array Usage (HARD – PLACEMENT / DEBUGGING LEVEL)

Problem Description
In C programming, a program may compile successfully but still produce unpredictable results at runtime. Such behavior is called undefined behavior and is one of the most dangerous aspects of the language. Undefined behavior can arise due to invalid memory access, uninitialized variables, incorrect pointer usage, array index violations, or misuse of operators. These issues are difficult to detect because the program may appear to work correctly for some inputs and fail silently for others.

In this experiment, you are given a logically simple task, but the constraints force you to reason carefully about memory safety, initialization, and execution order. The objective is to write a C program that reads an integer N, stores N integers in an array, and computes the sum of the elements. However, the challenge lies in ensuring that the program avoids all forms of undefined behavior.

The program must be written in such a way that:

- No variable is used before initialization
- No array index goes out of bounds
- No pointer points to invalid or freed memory
- No garbage values affect the computation
- The result is correct for all valid inputs, including edge cases

This experiment tests whether the student truly understands how C executes at runtime rather than just writing syntactically correct code. Many placement questions are based on identifying why a program produces unexpected output, and this problem is designed to build that level of reasoning.

Input Format

- First line contains an integer N
- Second line contains N integers

Output Format

- Print a single integer representing the sum of the array elements

Sample Input

5

1 2 3 4 5

Sample Output

```
15
```

Explanation
The array contains five elements: 1, 2, 3, 4, and 5. The program safely stores each value in a valid memory location and computes the sum without accessing any uninitialized or invalid memory.

Hidden Test Cases

- N = 1
- N = maximum allowed value
- All elements are zero
- Mix of positive and negative numbers
- Large input values that expose uninitialized variable bugs

Constraints

1 ≤ N ≤ 1000

Each element fits in 32-bit signed integer

Program

```c
#include <stdio.h>


int main() {

   int n;

   int arr[1000];

   int sum = 0;

   int i;


   scanf("%d", &n);


   if (n < 1 || n > 1000) {

      return 0;

   }
```

```
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    for (i = 0; i < n; i++) {
        sum += arr[i];
    }

    printf("%d", sum);
    return 0;
}
```