26.Matrix Multiplication: Write a C program to multiply two matrices. If the program fails for certain dimensions, document the causes (e.g., dimension mismatch, uninitialized values).  
#include <stdio.h>

int main() {

int r1, c1, r2, c2;

int i, j, k;

// Input matrix dimensions

printf("Enter rows and columns for first matrix: ");

scanf("%d %d", &r1, &c1);

printf("Enter rows and columns for second matrix: ");

scanf("%d %d", &r2, &c2);

// Condition for multiplication

if (c1 != r2) {

printf("Matrix multiplication not possible! (Dimension mismatch)\n");

return 0;

}

int A[r1][c1], B[r2][c2], C[r1][c2];

// Input elements of first matrix

printf("Enter elements of first matrix:\n");

for (i = 0; i < r1; i++) {

for (j = 0; j < c1; j++) {

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

}

}

// Input elements of second matrix

printf("Enter elements of second matrix:\n");

for (i = 0; i < r2; i++) {

for (j = 0; j < c2; j++) {

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

}

}

// Initialize result matrix to 0

for (i = 0; i < r1; i++) {

for (j = 0; j < c2; j++) {

C[i][j] = 0;

}

}

// Matrix multiplication

for (i = 0; i < r1; i++) {

for (j = 0; j < c2; j++) {

for (k = 0; k < c1; k++) {

C[i][j] += A[i][k] \* B[k][j];

}

}

}

// Display result

printf("Resultant matrix:\n");

for (i = 0; i < r1; i++) {

for (j = 0; j < c2; j++) {

printf("%d ", C[i][j]);

}

printf("\n");

}

return 0;

}

27.Bubble Sort with Index Out of Bounds: Implement a bubble sort in C. Simulate an "index out of bounds" error, analyze the causes, and propose precautions to prevent such errors  
  
#include <stdio.h>

int main() {

int n, i, j, temp;

printf("Enter number of elements: ");

scanf("%d", &n);

int arr[n];

printf("Enter %d elements:\n", n);

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

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

}

// Bubble Sort

for (i = 0; i < n - 1; i++) {

for (j = 0; j < n - i - 1; j++) {

if (arr[j] > arr[j + 1]) {

// Swap

temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

printf("Sorted array:\n");

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

printf("%d ", arr[i]);

}

return 0;

}

28.File Operations: Write a C program to read from and write to a file. Test for failure scenarios such as missing files, permission issues, or incorrect paths. Document findings and suggest fixes  
  
#include <stdio.h>

#include <stdlib.h>

int main() {

FILE \*fptr;

char filename[50], ch;

// ---------- Writing to a file ----------

printf("Enter the filename to write: ");

scanf("%s", filename);

fptr = fopen(filename, "w"); // Open for writing

if (fptr == NULL) {

perror("Error opening file for writing");

return 1;

}

fprintf(fptr, "Hello, this is a file write test!\n");

fprintf(fptr, "File operations in C are powerful.\n");

fclose(fptr);

printf("Data written successfully to %s\n", filename);

// ---------- Reading from the file ----------

fptr = fopen(filename, "r"); // Open for reading

if (fptr == NULL) {

perror("Error opening file for reading");

return 1;

}

printf("\nContents of %s:\n", filename);

while ((ch = fgetc(fptr)) != EOF) {

putchar(ch);

}

fclose(fptr);

return 0;

}

29.Recursive Factorial Function: Implement a recursive factorial function in C. Analyze its behavior when recursion depth exceeds the stack size and propose solutions to handle this gracefully.  
  
#include <stdio.h>

// Recursive factorial function

long long factorial(int n) {

if (n == 0 || n == 1) {

return 1; // base case

} else {

return n \* factorial(n - 1); // recursive case

}

}

int main() {

int num;

printf("Enter a number: ");

scanf("%d", &num);

if (num < 0) {

printf("Factorial is not defined for negative numbers.\n");

} else {

printf("Factorial of %d = %lld\n", num, factorial(num));

}

return 0;

}

30.Dynamic Memory Allocation: Write a C program to dynamically allocate memory for an array. Use tools like valgrind to detect memory leaks. Document findings and propose solutions to prevent leaks.  
  
#include <stdio.h>

#include <stdlib.h>

int main() {

int n, i;

int \*arr;

printf("Enter number of elements: ");

scanf("%d", &n);

// Dynamically allocate memory

arr = (int \*)malloc(n \* sizeof(int));

if (arr == NULL) {

printf("Memory allocation failed!\n");

return 1;

}

printf("Enter %d integers:\n", n);

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

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

}

printf("You entered: ");

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

printf("%d ", arr[i]);

}

printf("\n");

// ❌ Forgot to free(arr); (intentional leak for testing)

return 0;

}

31.Power Calculation with Overflow Detection: Write a program to calculate the power of a number (e.g., base^exponent). Identify scenarios where integer overflow occurs, document the conditions leading to overflow, and suggest strategies to avoid it  
  
#include <stdio.h>

#include <limits.h> // for INT\_MAX and LONG\_LONG\_MAX

// Function to calculate power with overflow detection

long long power(int base, int exp) {

long long result = 1;

for (int i = 0; i < exp; i++) {

// Check for overflow before multiplying

if (base != 0 && result > LLONG\_MAX / base) {

printf("?? Overflow detected at step %d!\n", i + 1);

return -1; // Indicate overflow

}

result \*= base;

}

return result;

}

int main() {

int base, exp;

printf("Enter base and exponent: ");

scanf("%d %d", &base, &exp);

if (exp < 0) {

printf("Negative exponents not supported with integers.\n");

return 1;

}

long long result = power(base, exp);

if (result != -1) {

printf("%d^%d = %lld\n", base, exp, result);

} else {

printf("Calculation failed due to overflow.\n");

}

return 0;

}

32.Sum of Digits to Single Digit: Create a program to repeatedly calculate the sum of digits of a number until a single digit is obtained. Simulate an infinite loop condition, investigate its cause, and provide strategies to prevent and resolve such loops.  
  
#include <stdio.h>

int sumOfDigits(int n) {

int sum = 0;

while (n > 0) {

sum += n % 10;

n /= 10;

}

return sum;

}

int main() {

int num;

printf("Enter a number: ");

scanf("%d", &num);

// Repeatedly reduce until single digit

while (num >= 10) {

num = sumOfDigits(num);

}

printf("Single digit result = %d\n", num);

return 0;

}

33.Producer-Consumer Problem with Deadlock: Write a multi-threaded C program to simulate the producer-consumer problem. Intentionally introduce a deadlock, analyze its causes, and suggest techniques like using mutexes or semaphores to resolve it.  
  
#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

#define BUFFER\_SIZE 5

int buffer[BUFFER\_SIZE];

int count = 0;

int in = 0, out = 0;

pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;

pthread\_cond\_t notFull = PTHREAD\_COND\_INITIALIZER;

pthread\_cond\_t notEmpty = PTHREAD\_COND\_INITIALIZER;

void \*producer(void \*arg) {

for (int i = 1; i <= 10; i++) {

pthread\_mutex\_lock(&mutex);

while (count == BUFFER\_SIZE) {

pthread\_cond\_wait(&notFull, &mutex);

}

buffer[in] = i;

printf("Producer produced %d\n", i);

in = (in + 1) % BUFFER\_SIZE;

count++;

pthread\_cond\_signal(&notEmpty); // ✅ notify consumer

pthread\_mutex\_unlock(&mutex);

sleep(1);

}

return NULL;

}

void \*consumer(void \*arg) {

for (int i = 1; i <= 10; i++) {

pthread\_mutex\_lock(&mutex);

while (count == 0) {

pthread\_cond\_wait(&notEmpty, &mutex);

}

int item = buffer[out];

printf("Consumer consumed %d\n", item);

out = (out + 1) % BUFFER\_SIZE;

count--;

pthread\_cond\_signal(&notFull); // ✅ notify producer

pthread\_mutex\_unlock(&mutex);

sleep(2);

}

return NULL;

}

int main() {

pthread\_t prod, cons;

pthread\_create(&prod, NULL, producer, NULL);

pthread\_create(&cons, NULL, consumer, NULL);

pthread\_join(prod, NULL);

pthread\_join(cons, NULL);

return 0;

}

34.Basic Calculator with Division by Zero Handling: Develop a calculator program in C that performs basic arithmetic operations. Test for division by zero scenarios, analyze failures, and implement error-handling mechanisms to address such cases.  
  
#include <stdio.h>

int main() {

char operator;

double num1, num2, result;

printf("Enter operator (+, -, \*, /): ");

scanf(" %c", &operator);

printf("Enter two numbers: ");

scanf("%lf %lf", &num1, &num2);

switch (operator) {

case '+':

result = num1 + num2;

printf("Result: %.2lf\n", result);

break;

case '-':

result = num1 - num2;

printf("Result: %.2lf\n", result);

break;

case '\*':

result = num1 \* num2;

printf("Result: %.2lf\n", result);

break;

case '/':

if (num2 == 0) {

printf("Error: Division by zero is not allowed.\n");

} else {

result = num1 / num2;

printf("Result: %.2lf\n", result);

}

break;

default:

printf("Error: Invalid operator.\n");

}

return 0;

}

35.Quicksort with Performance Analysis: Implement a quicksort algorithm in C. Simulate a case where the algorithm performs poorly (e.g., selecting a poor pivot). Investigate the causes of the performance drop and propose optimizations.  
  
#include <stdio.h>

// Global counter for comparisons

long long comparisons = 0;

// Function to swap two elements

void swap(int \*a, int \*b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

// Partition function (choosing first element as pivot -> can cause poor performance)

int partition(int arr[], int low, int high) {

int pivot = arr[low]; // always pick first element as pivot

int i = low + 1;

int j = high;

while (1) {

while (i <= high && arr[i] <= pivot) {

comparisons++;

i++;

}

while (j >= low && arr[j] > pivot) {

comparisons++;

j--;

}

if (i < j) {

swap(&arr[i], &arr[j]);

} else {

break;

}

}

swap(&arr[low], &arr[j]);

return j;

}

// Quicksort recursive function

void quicksort(int arr[], int low, int high) {

if (low < high) {

int pivotIndex = partition(arr, low, high);

quicksort(arr, low, pivotIndex - 1);

quicksort(arr, pivotIndex + 1, high);

}

}

int main() {

int n, i;

printf("Enter number of elements: ");

scanf("%d", &n);

int arr[n];

printf("Enter elements:\n");

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

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

}

comparisons = 0; // reset counter

quicksort(arr, 0, n - 1);

printf("\nSorted array: ");

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

printf("%d ", arr[i]);

}

printf("\nTotal comparisons: %lld\n", comparisons);

// Worst-case simulation: when input is already sorted

// QuickSort with first element as pivot -> O(n^2)

return 0;

}