**C program 26 -40**

**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;

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

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

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

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

if (c1 != r2) {

printf("Error: Matrix multiplication not possible\n");

return 0;

}

int A[10][10], B[10][10], C[10][10];

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

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

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

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

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

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

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

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

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

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

C[i][j] = 0;

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

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

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

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

printf("Resultant Matrix:\n");

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

for (int 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;

printf("Enter number of elements: ");

scanf("%d", &n);

int arr[10];

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

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

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

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

for (int j = 0; j <= n - i; j++) { // index out of bounds error here

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

int temp = arr[j];

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

arr[j + 1] = temp;

}

}

}

printf("Sorted array:\n");

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

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

printf("\n");

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>

int main() {

FILE \*fp;

char str[100];

fp = fopen("test.txt", "w");

if (fp == NULL) {

printf("Error: Cannot open file for writing\n");

return 1;

}

fprintf(fp, "Hello, File Operations!\n");

fclose(fp);

fp = fopen("test.txt", "r");

if (fp == NULL) {

printf("Error: Cannot open file for reading\n");

return 1;

}

while (fgets(str, 100, fp) != NULL)

printf("%s", str);

fclose(fp);

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>

long long factorial(int n) {

if (n <= 1) return 1;

return n \* factorial(n - 1);

}

int main() {

int n;

printf("Enter a number: ");

scanf("%d", &n);

printf( n, factorial(n));

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;

printf("Enter number of elements: ");

scanf("%d", &n);

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

if (arr == NULL) {

printf("Memory allocation failed\n");

return 1;

}

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

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

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

printf("You entered:\n");

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

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

free(arr);

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>

long long power(int base, int exp) {

long long result = 1;

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

if (result > LLONG\_MAX / base) {

printf("Overflow detected!\n");

return -1;

}

result \*= base;

}

return result;

}

int main() {

int base, exp;

printf("Enter base and exponent: ");

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

long long ans = power(base, exp);

if (ans != -1)

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

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 main() {

int n;

printf("Enter a number: ");

scanf("%d", &n);

while (n >= 10) {

int sum = 0, temp = n;

while (temp > 0) {

sum += temp % 10;

temp /= 10;

}

if (sum == n) { // simulate infinite loop case

printf("Infinite loop detected!\n");

break;

}

n = sum;

}

if (n < 10)

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

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 <pthread.h>

#include <semaphore.h>

#include <unistd.h>

#define SIZE 5

int buffer[SIZE], count = 0;

pthread\_mutex\_t mutex;

sem\_t full, empty;

void\* producer(void\* arg) {

for (int i = 0; i < 10; i++) {

sem\_wait(&empty);

pthread\_mutex\_lock(&mutex);

buffer[count++] = i;

printf("Produced %d\n", i);

if (i == 5) return NULL;

pthread\_mutex\_unlock(&mutex);

sem\_post(&full);

sleep(1);

}

return NULL;

}

void\* consumer(void\* arg) {

for (int i = 0; i < 10; i++) {

sem\_wait(&full);

pthread\_mutex\_lock(&mutex);

printf("Consumed %d\n", buffer[--count]);

pthread\_mutex\_unlock(&mutex);

sem\_post(&empty);

sleep(1);

}

return NULL;

}

int main() {

pthread\_t p, c;

pthread\_mutex\_init(&mutex, NULL);

sem\_init(&full, 0, 0);

sem\_init(&empty, 0, SIZE);

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

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

pthread\_join(p, NULL);

pthread\_join(c, NULL);

pthread\_mutex\_destroy(&mutex);

sem\_destroy(&full);

sem\_destroy(&empty);

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() {

int a, b;

char op;

printf("Enter expression (a op b): ");

scanf("%d %c %d", &a, &op, &b);

switch (op) {

case '+': printf("%d\n", a + b); break;

case '-': printf("%d\n", a - b); break;

case '\*': printf("%d\n", a \* b); break;

case '/':

if (b == 0) printf("Error: Division by zero\n");

else printf("%d\n", a / b);

break;

default: printf("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>

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

int t = \*a; \*a = \*b; \*b = t;

}

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

int pivot = arr[low];

int i = low, j = high;

while (i < j) {

while (arr[i] <= pivot) i++;

while (arr[j] > pivot) j--;

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

}

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

return j;

}

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

if (low < high) {

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

quicksort(arr, low, p - 1);

quicksort(arr, p + 1, high);

}

}

int main() {

int arr[] = {1,2,3,4,5,6,7,8,9};

int n = sizeof(arr)/sizeof(arr[0]);

quicksort(arr, 0, n-1);

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

return 0;

}

**36) Peer Review for Test Cases and Test Plans:**

**Implement and practice a peer review process for test cases and test**

**plans to improve quality and identify potential issues early.**

#include <stdio.h>

int main() {

printf("Peer Review Process Simulation\n");

printf("1. Author prepares test cases and test plan.\n");

printf("2. Reviewer checks coverage, correctness, clarity.\n");

printf("3. Missing edge cases and defects are identified.\n");

printf("4. Feedback is given, author updates documents.\n");

printf("5. Reviewed test cases improve quality.\n");

return 0;

}

**37) Defect Tracking with Tools:**

**Use tools like Jira or Bugzilla to report testing progress and track defect**

**resolution effectively.**

#include <stdio.h>

int main() {

printf("Defect Tracking Simulation\n");

printf("1. Tester finds a bug during execution.\n");

printf("2. Bug is reported in Jira/Bugzilla with severity & priority.\n");

printf("3. Developer fixes the bug.\n");

printf("4. Tester retests and closes the defect if resolved.\n");

printf("5. Tracking tools help monitor defect lifecycle.\n");

return 0;

}

**38) Implement version control and configuration management for test scripts**

**and environments.**

#include <stdio.h>

int main() {

printf("Version Control & Configuration Management Simulation\n");

printf("1. Test scripts are stored in Git/SVN repository.\n");

printf("2. Each change is committed with version history.\n");

printf("3. Configuration files for test environments are managed.\n");

printf("4. Rollback and branching ensure consistency.\n");

return 0;

}

**39) Use tools like Git, SVN, or Mercurial for managing test scripts, defect**

**logs, and test results.**

#include <stdio.h>

int main() {

printf("Tool Usage for Managing Test Artifacts\n");

printf("1. Git/SVN/Mercurial track changes in test scripts.\n");

printf("2. Defect logs and test results are version-controlled.\n");

printf("3. Collaboration among testers is simplified.\n");

printf("4. History and rollback are available when needed.\n");

return 0;

}

**40) Perform test closure activities, including documentation, analysis, and**

**final reporting for a project**.

#include <stdio.h>

int main() {

printf("Test Closure Activities\n");

printf("1. Finalize and archive test cases, plans, and results.\n");

printf("2. Document lessons learned during testing.\n");

printf("3. Summarize defects found and resolved.\n");

printf("4. Prepare final test report for project closure.\n");

return 0;

}