## Reverse a String

Write a function void reverseString(char \*str) that takes a pointer to a string and reverses the string in place.

Sol: #include <stdio.h> #include <string.h>

void reverseString(char \*str) { int length = strlen(str);

int start = 0, end = length - 1; char temp;

while (start < end) {

temp = str[start]; str[start] = str[end]; str[end] = temp;

start++;

end--;

}

}

int main() {

char str[] = "Hello, World!"; printf("Original string: %s\n", str);

reverseString(str);

printf("Reversed string: %s\n", str); return 0;

}

O/p:

Original string: Hello, World! Reversed string: !dlroW ,olleH

## Concatenate Two Strings

Implement a function void concatenateStrings(char \*dest, const char \*src) that appends the source string to the destination string using pointers.

Sol: #include <stdio.h>

void concatenateStrings(char \*dest, const char \*src) {

// Move the destination pointer to the end of the existing string while (\*dest != '\0') {

dest++;

}

// Copy the source string to the destination while (\*src != '\0') {

\*dest = \*src; dest++; src++;

}

// Null-terminate the concatenated string

\*dest = '\0';

}

int main() {

char dest[50] = "Hello, "; // Ensure dest has enough space for concatenation const char src[] = "World!";

printf("Before concatenation: %s\n", dest);

concatenateStrings(dest, src);

printf("After concatenation: %s\n", dest);

return 0;

}

O/p: Before concatenation: Hello, After concatenation: Hello, World!

## String Length

Create a function int stringLength(const char \*str) that calculates and returns the length of a string using pointers.

Sol: #include <stdio.h>

int stringLength(const char \*str) { const char \*ptr = str;

int length = 0;

while (\*ptr != '\0') { length++;

ptr++;

}

return length;

}

int main() {

const char str[] = "Hello, World!";

printf("The length of the string \"%s\" is: %d\n", str, stringLength(str)); return 0;

}

O/p:

The length of the string "Hello, World!" is: 13

## Compare Two Strings

Write a function int compareStrings(const char \*str1, const char \*str2) that compares two strings lexicographically and returns 0 if they are equal, a positive number if str1 is greater, or a negative number if str2 is greater.

Sol: #include <stdio.h>

int compareStrings(const char \*str1, const char \*str2) { while (\*str1 && (\*str1 == \*str2)) {

str1++; str2++;

}

return \*(unsigned char \*)str1 - \*(unsigned char \*)str2;

}

int main() {

const char str1[] = "Hello"; const char str2[] = "World";

const char str3[] = "Hello";

printf("Comparing \"%s\" and \"%s\": %d\n", str1, str2, compareStrings(str1, str2));

printf("Comparing \"%s\" and \"%s\": %d\n", str1, str3, compareStrings(str1, str3));

printf("Comparing \"%s\" and \"%s\": %d\n", str2, str1, compareStrings(str2, str1));

return 0;

}

O/p: Comparing "Hello" and "World": -15 Comparing "Hello" and "Hello": 0 Comparing "World" and "Hello": 15

## Find Substring

Implement char\* findSubstring(const char \*str, const char \*sub) that returns a pointer to the first occurrence of the substring sub in the string str, or NULL if the substring is not found.

Sol: #include <stdio.h>

char\* findSubstring(const char \*str, const char \*sub) { if (\*sub == '\0') {

return (char\*)str; // Empty substring matches at the beginning

}

while (\*str != '\0') { const char \*s = str; const char \*p = sub;

position while (\*s != '\0' && \*p != '\0' && \*s == \*p) {

s++; p++;

}

found if (\*p == '\0') {

return (char\*)str;

}

str++;

}

return NULL; // Substring not found

}

int main() {

const char str[] = "Hello, World!"; const char sub1[] = "World"; const char sub2[] = "Moon";

char \*result1 = findSubstring(str, sub1); char \*result2 = findSubstring(str, sub2);

if (result1) {

printf("Substring \"%s\" found at position: %ld\n", sub1, result1 - str);

} else {

printf("Substring \"%s\" not found.\n", sub1);

}

if (result2) {

printf("Substring \"%s\" found at position: %ld\n", sub2, result2 - str);

} else {

printf("Substring \"%s\" not found.\n", sub2);

}

return 0;

}

O/p: Substring "World" found at position: 7 Substring "Moon" not found.

## Replace Character in String

Write a function void replaceChar(char \*str, char oldChar, char newChar) that replaces all occurrences of oldChar with newChar in the given string.

Sol: #include <stdio.h>

void replaceChar(char \*str, char oldChar, char newChar) { while (\*str != '\0') {

if (\*str == oldChar) {

\*str = newChar;

}

str++;

}

}

int main() {

char str[] = "Hello, World!"; char oldChar = 'o';

char newChar = 'a';

printf("Original string: %s\n", str);

replaceChar(str, oldChar, newChar);

printf("Modified string: %s\n", str); return 0;

}

O/p: Original string: Hello, World! Modified string: Hella, Warld!

## Copy String

Create a function void copyString(char \*dest, const char \*src) that copies the content of the source string src to the destination string dest.

Sol: #include <stdio.h>

void copyString(char \*dest, const char \*src) { while (\*src != '\0') {

\*dest = \*src; dest++; src++;

}

\*dest = '\0'; // Null-terminate the destination string

}

int main() {

const char src[] = "Hello, World!";

char dest[50]; // Ensure the destination array is large enough

copyString(dest, src);

printf("Source string: %s\n", src); printf("Copied string: %s\n", dest);

return 0;

}

O/p: Source string: Hello, World! Copied string: Hello, World!

## Count Vowels in a String

Implement int countVowels(const char \*str) that counts and returns the number of vowels in a given string.

Sol: #include <stdio.h>

int countVowels(const char \*str) { int count = 0;

while (\*str != '\0') {

// Check if the current character is a vowel (both uppercase and lowercase) if (\*str == 'a' || \*str == 'e' || \*str == 'i' || \*str == 'o' || \*str == 'u' ||

\*str == 'A' || \*str == 'E' || \*str == 'I' || \*str == 'O' || \*str == 'U') { count++;

}

str++;

}

return count;

}

int main() {

const char str[] = "Hello, World!";

printf("The number of vowels in \"%s\" is: %d\n", str, countVowels(str)); return 0;

}

O/p: The number of vowels in "Hello, World!" is: 3

## Check Palindrome

Write a function int isPalindrome(const char \*str) that checks if a given string is a palindrome and returns 1 if true, otherwise 0.

Sol: #include <stdio.h>

#include <string.h>

int isPalindrome(const char \*str) { int length = strlen(str);

const char \*start = str;

const char \*end = str + length - 1;

while (start < end) { if (\*start != \*end) {

return 0; // Not a palindrome

}

start++; end--;

}

return 1; // It is a palindrome

}

int main() {

const char str1[] = "madam"; const char str2[] = "hello";

printf("Is \"%s\" a palindrome? %s\n", str1, isPalindrome(str1) ? "Yes" : "No"); printf("Is \"%s\" a palindrome? %s\n", str2, isPalindrome(str2) ? "Yes" : "No");

return 0;

}

O/p: Is "madam" a palindrome? Yes Is "hello" a palindrome? No

## Tokenize String

Create a function void tokenizeString(char \*str, const char \*delim, void (\*processToken)(const char \*)) that tokenizes the string str using delimiters in delim, and for each token, calls processToken.

Sol: #include <stdio.h> #include <string.h>

void tokenizeString(char \*str, const char \*delim, void (\*processToken)(const char

\*)) {

char \*token = strtok(str, delim); while (token != NULL) {

processToken(token); // Call the function pointer with the current token token = strtok(NULL, delim);

}

}

void printToken(const char \*token) { printf("Token: %s\n", token);

}

int main() {

char str[] = "Hello, World! Welcome to C programming."; const char delim[] = " ,.!";

printf("Original string: \"%s\"\n", str); tokenizeString(str, delim, printToken);

return 0;

}

O/p: Original string: "Hello, World! Welcome to C programming." Token: Hello

Token: World Token: Welcome Token: to Token: C

Token: programming

## Allocate and Free Integer Array

Write a program that dynamically allocates memory for an array of integers, fills it with values from 1 to n, and then frees the allocated memory.

Sol: #include <stdio.h> #include <stdlib.h>

int main() { int n;

// Prompt the user to enter the size of the array printf("Enter the size of the array: "); scanf("%d", &n);

// Dynamically allocate memory for the array of integers int \*arr = (int \*)malloc(n \* sizeof(int));

// Check if memory allocation was successful if (arr == NULL) {

printf("Memory allocation failed!\n"); return 1; // Exit if memory allocation fails

}

// Fill the array with values from 1 to n for (int i = 0; i < n; i++) {

arr[i] = i + 1;

}

// Print the array values printf("Array elements:\n"); for (int i = 0; i < n; i++) {

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

}

printf("\n");

// Free the allocated memory free(arr);

return 0;

}

Sol:

Enter the size of the array: 5 Array elements:

1 2 3 4 5

## Dynamic String Input

Implement a function that dynamically allocates memory for a string, reads a string input from the user, and then prints the string. Free the memory after use.

Sol: #include <stdio.h> #include <stdlib.h> #include <string.h>

void readAndPrintString() { char \*str;

int size;

// Prompt the user to enter the maximum size of the string printf("Enter the maximum length of the string: "); scanf("%d", &size);

getchar(); // To consume the newline character after entering the size

// Dynamically allocate memory for the string

str = (char \*)malloc((size + 1) \* sizeof(char)); // +1 for null terminator

// Check if memory allocation was successful if (str == NULL) {

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

return; // Exit if memory allocation fails

}

// Prompt the user to enter the string printf("Enter a string: ");

fgets(str, size + 1, stdin); // Read the string including spaces

// Print the entered string printf("You entered: %s\n", str);

// Free the allocated memory free(str);

}

int main() {

readAndPrintString(); // Call the function to read and print a string return 0;

}

O/p: Enter the maximum length of the string: 50 Enter a string: likitha s

You entered: likitha s

## Resize an Array

Write a program that dynamically allocates memory for an array of n integers, fills it with values, resizes the array to 2n using realloc(), and fills the new elements with values.

Sol: #include <stdio.h> #include <stdlib.h>

int main() { int n;

// Prompt the user to enter the size of the array printf("Enter the size of the array: "); scanf("%d", &n);

// Dynamically allocate memory for an array of n integers int \*arr = (int \*)malloc(n \* sizeof(int));

// Check if memory allocation was successful if (arr == NULL) {

printf("Memory allocation failed!\n"); return 1; // Exit if memory allocation fails

}

// Fill the array with values from 1 to n for (int i = 0; i < n; i++) {

arr[i] = i + 1;

}

// Print the original array printf("Original array elements:\n"); for (int i = 0; i < n; i++) {

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

}

printf("\n");

// Resize the array to 2n using realloc()

arr = (int \*)realloc(arr, 2 \* n \* sizeof(int));

// Check if realloc was successful if (arr == NULL) {

printf("Memory reallocation failed!\n"); return 1; // Exit if realloc fails

}

// Fill the new elements in the resized array with values from n+1 to 2n for (int i = n; i < 2 \* n; i++) {

arr[i] = i + 1;

}

// Print the resized array printf("Resized array elements:\n"); for (int i = 0; i < 2 \* n; i++) {

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

}

printf("\n");

// Free the allocated memory free(arr);

return 0;

}

O/p: Enter the size of the array: 5 Original array elements:

1 2 3 4 5

Resized array elements:

1 2 3 4 5 6 7 8 9 10

## Matrix Allocation

Create a function that dynamically allocates memory for a 2D array (matrix) of size m x n, fills it with values, and then deallocates the memory.

Sol: #include <stdio.h> #include <stdlib.h>

void allocateAndFillMatrix(int m, int n) {

// Dynamically allocate memory for a 2D matrix of size m x n int \*\*matrix = (int \*\*)malloc(m \* sizeof(int \*));

// Check if memory allocation for rows was successful if (matrix == NULL) {

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

}

// Dynamically allocate memory for each column in each row for (int i = 0; i < m; i++) {

matrix[i] = (int \*)malloc(n \* sizeof(int));

// Check if memory allocation for columns in this row was successful if (matrix[i] == NULL) {

printf("Memory allocation for row %d failed!\n", i); return;

}

}

// Fill the matrix with values int value = 1;

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

matrix[i][j] = value++;

}

}

// Print the matrix printf("Matrix elements:\n"); for (int i = 0; i < m; i++) {

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

}

printf("\n");

}

// Deallocate the memory for (int i = 0; i < m; i++) {

free(matrix[i]);

}

free(matrix);

}

int main() { int m, n;

// Prompt the user to enter the dimensions of the matrix printf("Enter the number of rows (m): ");

scanf("%d", &m);

printf("Enter the number of columns (n): "); scanf("%d", &n);

// Call the function to allocate, fill, and deallocate the matrix

allocateAndFillMatrix(m, n);

return 0;

}

O/p:

Enter the number of rows (m): 4 Enter the number of columns (n): 3 Matrix elements:

1 2 3

4 5 6

7 8 9

10 11 12

## String Concatenation with Dynamic Memory

Implement a function that takes two strings, dynamically allocates memory to concatenate them, and returns the new concatenated string. Ensure to free the memory after use.

Sol: #include <stdio.h> #include <stdlib.h> #include <string.h>

// Function to concatenate two strings dynamically

char\* concatenateStrings(const char \*str1, const char \*str2) {

// Allocate memory for the new concatenated string

// The new string will have the length of str1 + str2 + 1 (for the null terminator) int len1 = strlen(str1);

int len2 = strlen(str2);

char \*result = (char \*)malloc((len1 + len2 + 1) \* sizeof(char));

// Check if memory allocation was successful if (result == NULL) {

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

}

// Copy the first string to result strcpy(result, str1);

// Concatenate the second string to result strcat(result, str2);

// Return the concatenated string return result;

}

int main() {

const char \*str1 = "Hello, "; const char \*str2 = "world!";

// Call the function to concatenate the strings

char \*concatenatedStr = concatenateStrings(str1, str2);

// Check if memory allocation was successful if (concatenatedStr != NULL) {

// Print the concatenated string

printf("Concatenated string: %s\n", concatenatedStr);

// Free the dynamically allocated memory free(concatenatedStr);

}

return 0;

}

O/p:

Concatenated string: Hello, world!

## Dynamic Memory for Structure

Define a struct for a student with fields like name, age, and grade. Write a program that dynamically allocates memory for a student, fills in the details, and then frees the memory.

Sol: #include <stdio.h> #include <stdlib.h> #include <string.h>

// Define a structure for a student struct Student {

char name[50]; int age;

float grade;

};

// Function to dynamically allocate memory for a student void allocateAndFillStudent() {

// Dynamically allocate memory for a Student

struct Student \*student = (struct Student \*)malloc(sizeof(struct Student));

// Check if memory allocation was successful if (student == NULL) {

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

}

// Fill in the details of the student printf("Enter student's name: ");

fgets(student->name, sizeof(student->name), stdin); // Read name with spaces

student->name[strcspn(student->name, "\n")] = '\0'; // Remove newline character at the end

printf("Enter student's age: "); scanf("%d", &student->age);

printf("Enter student's grade: "); scanf("%f", &student->grade);

// Print the student's details printf("\nStudent details:\n"); printf("Name: %s\n", student->name); printf("Age: %d\n", student->age); printf("Grade: %.2f\n", student->grade);

// Free the dynamically allocated memory free(student);

}

int main() {

// Call the function to allocate, fill, and display student details allocateAndFillStudent();

return 0;

}

Sol:

Enter student's name: LIKITHA Enter student's age: 23

Enter student's grade: 95

Student details:

Name: LIKITHA

Age: 23

Grade: 95.00

## Dynamic Array of Pointers

Write a program that dynamically allocates memory for an array of pointers to integers, fills each integer with values, and then frees all the allocated memory.

Sol: #include <stdio.h> #include <stdlib.h>

int main() { int n;

// Prompt the user to enter the size of the array printf("Enter the number of elements: "); scanf("%d", &n);

// Dynamically allocate memory for an array of n pointers to integers int \*\*arr = (int \*\*)malloc(n \* sizeof(int \*));

// Check if memory allocation was successful if (arr == NULL) {

printf("Memory allocation failed!\n"); return 1; // Exit if memory allocation fails

}

// Dynamically allocate memory for each integer and assign values for (int i = 0; i < n; i++) {

arr[i] = (int \*)malloc(sizeof(int)); // Allocate memory for a single integer if (arr[i] == NULL) {

printf("Memory allocation for arr[%d] failed!\n", i);

return 1; // Exit if memory allocation fails for any element

}

// Assign value to the integer

\*(arr[i]) = i + 1; // Filling with values from 1 to n

}

// Print the array of integers printf("Array elements:\n"); for (int i = 0; i < n; i++) {

printf("%d ", \*(arr[i])); // Dereference pointer to print the value

}

printf("\n");

// Free the dynamically allocated memory for each integer for (int i = 0; i < n; i++) {

free(arr[i]); // Free the memory allocated for each integer

}

// Free the array of pointers free(arr);

return 0;

}

O/p: Enter the number of elements: 5 Array elements:

1 2 3 4 5

## Dynamic Memory for Multidimensional Arrays

Create a program that dynamically allocates memory for a 3D array of integers, fills it with values, and deallocates the memory.

Sol: #include <stdio.h> #include <stdlib.h>

int main() {

int x = 2, y = 3, z = 4; // Dimensions of the 3D array

// Dynamically allocate memory for a 3D array (x \* y \* z integers) int \*\*\*array = (int \*\*\*)malloc(x \* sizeof(int \*\*));

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

array[i] = (int \*\*)malloc(y \* sizeof(int \*)); for (int j = 0; j < y; j++) {

array[i][j] = (int \*)malloc(z \* sizeof(int));

}

}

// Fill the 3D array with values int value = 1;

for (int i = 0; i < x; i++) { for (int j = 0; j < y; j++) {

for (int k = 0; k < z; k++) { array[i][j][k] = value++;

}

}

}

// Print the 3D array

printf("3D Array elements:\n"); for (int i = 0; i < x; i++) {

printf("Layer %d:\n", i + 1); for (int j = 0; j < y; j++) {

for (int k = 0; k < z; k++) { printf("%d ", array[i][j][k]);

}

printf("\n");

}

printf("\n");

}

// Free the dynamically allocated memory for (int i = 0; i < x; i++) {

for (int j = 0; j < y; j++) { free(array[i][j]); // Free each row

}

free(array[i]); // Free each 2D layer

}

free(array); // Free the 3D array

return 0;

}

O/p: 3D Array elements: Layer 1:

1 2 3 4

5 6 7 8

9 10 11 12

Layer 2:

13 14 15 16

17 18 19 20

21 22 23 24

# Double Pointers

## Swap Two Numbers Using Double Pointers

Write a function void swap(int \*\*a, int \*\*b) that swaps the values of two integer pointers using double pointers.

Sol: #include <stdio.h>

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

\*a = \*b;

\*b = temp;

}

int main() {

int x = 5, y = 10;

int \*px = &x, \*py = &y;

printf("Before swap: x = %d, y = %d\n", x, y);

swap(&px, &py);

printf("After swap: x = %d, y = %d\n", x, y); return 0;

}

O/p:

Before swap: x = 5, y = 10 After swap: x = 5, y = 10

## Dynamic Memory Allocation Using Double Pointer

Implement a function void allocateArray(int \*\*arr, int size) that dynamically allocates memory for an array of integers using a double pointer.

Sol: #include <stdio.h> #include <stdlib.h>

void allocateArray(int \*\*arr, int size) {

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

}

int main() { int \*arr;

int size = 5; allocateArray(&arr, size); for (int i = 0; i < size; i++) {

arr[i] = i \* 2; printf("%d ", arr[i]);

}

free(arr); return 0;

}

Sol:

0 2 4 6 8

## Modify a String Using Double Pointer

Write a function void modifyString(char \*\*str) that takes a double pointer to a string, dynamically allocates a new string, assigns it to the pointer, and modifies the original string.

Sol: #include <stdio.h> #include <stdlib.h> #include <string.h>

void modifyString(char \*\*str) {

\*str = (char \*)malloc(20 \* sizeof(char)); strcpy(\*str, "New Modified String");

}

int main() {

char \*str = "Original String"; modifyString(&str); printf("%s\n", str);

free(str); return 0;

}

O/p: New Modified String

## Pointer to Pointer Example

Create a simple program that demonstrates how to use a pointer to a pointer to access and modify the value of an integer.

Sol: #include <stdio.h>

int main() { int x = 10;

int \*px = &x;

int \*\*ppx = &px;

printf("Value of x: %d\n", x);

printf("Value using pointer to pointer: %d\n", \*\*ppx);

\*\*ppx = 20;

printf("Modified value of x: %d\n", x);

return 0;

}

O/p: Value of x: 10

Value using pointer to pointer: 10 Modified value of x: 20

## 2D Array Using Double Pointer

Write a function int\*\* create2DArray(int rows, int cols) that dynamically allocates memory for a 2D array of integers using a double pointer and returns the pointer to the array.

Sol: #include <stdio.h> #include <stdlib.h>

int\*\* create2DArray(int rows, int cols) {

int \*\*arr = (int \*\*)malloc(rows \* sizeof(int \*)); for (int i = 0; i < rows; i++) {

arr[i] = (int \*)malloc(cols \* sizeof(int));

}

return arr;

}

int main() {

int rows = 2, cols = 3;

int \*\*arr = create2DArray(rows, cols);

for (int i = 0; i < rows; i++) { for (int j = 0; j < cols; j++) {

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

}

printf("\n");

}

for (int i = 0; i < rows; i++) { free(arr[i]);

}

free(arr); return 0;

}

O/p: 0 1 2

1 2 3

## Freeing 2D Array Using Double Pointer

Implement a function void free2DArray(int \*\*arr, int rows) that deallocates the memory allocated for a 2D array using a double pointer.

Sol: #include <stdio.h>

#include <stdlib.h>

void free2DArray(int \*\*arr, int rows) { for (int i = 0; i < rows; i++) {

free(arr[i]);

}

free(arr);

}

int main() {

int rows = 2, cols = 3;

int \*\*arr = (int \*\*)malloc(rows \* sizeof(int \*)); for (int i = 0; i < rows; i++) {

arr[i] = (int \*)malloc(cols \* sizeof(int));

}

// Fill the array and print

for (int i = 0; i < rows; i++) { for (int j = 0; j < cols; j++) {

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

}

printf("\n");

}

free2DArray(arr, rows); return 0;

}

O/p: 0 1 2

1 2 3

## Pass a Double Pointer to a Function

Write a function void setPointer(int \*\*ptr) that sets the pointer passed to it to point to a dynamically allocated integer.

Sol: #include <stdio.h> #include <stdlib.h>

void setPointer(int \*\*ptr) {

\*ptr = (int \*)malloc(sizeof(int));

\*\*ptr = 10;

}

int main() {

int \*ptr = NULL;

setPointer(&ptr); printf("Value: %d\n", \*ptr); free(ptr);

return 0;

}

Sol: Value: 10

## Dynamic Array of Strings

Create a function void allocateStringArray(char \*\*\*arr, int n) that dynamically allocates memory for an array of n strings using a double pointer.

Sol: #include <stdio.h> #include <stdlib.h>

void allocateStringArray(char \*\*\*arr, int n) {

\*arr = (char \*\*)malloc(n \* sizeof(char \*)); for (int i = 0; i < n; i++) {

(\*arr)[i] = (char \*)malloc(20 \* sizeof(char));

}

}

int main() { char \*\*arr; int n = 3;

allocateStringArray(&arr, n);

for (int i = 0; i < n; i++) { sprintf(arr[i], "String %d", i + 1); printf("%s\n", arr[i]);

}

for (int i = 0; i < n; i++) { free(arr[i]);

}

free(arr); return 0;

}

O/p: String 1

String 2

String 3

## String Array Manipulation Using Double Pointer

Implement a function void modifyStringArray(char \*\*arr, int n) that modifies each string in an array of strings using a double pointer.

Sol: #include <stdio.h> #include <string.h> #include <stdlib.h>

void modifyStringArray(char \*\*arr, int n) { for (int i = 0; i < n; i++) {

// Allocate memory for the modified string

arr[i] = (char \*)realloc(arr[i], strlen(arr[i]) + 9); // " Modified" is 9 characters strcat(arr[i], " Modified"); // Append " Modified" to each string

}

}

int main() {

// Dynamically allocate memory for the strings char \*arr[3];

arr[0] = (char \*)malloc(6 \* sizeof(char)); // "Hello" + '\0' arr[1] = (char \*)malloc(6 \* sizeof(char)); // "World" + '\0' arr[2] = (char \*)malloc(2 \* sizeof(char)); // "C" + '\0'

strcpy(arr[0], "Hello");

strcpy(arr[1], "World");

strcpy(arr[2], "C");

int n = 3;

modifyStringArray(arr, n);

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

free(arr[i]); // Don't forget to free the memory

}

return 0;

}

O/p:

Hello Modified World Modified C Modified

# Function Pointers

## Basic Function Pointer Declaration

Write a program that declares a function pointer for a function int add(int, int) and uses it to call the function and print the result.

Sol: #include <stdio.h>

int add(int a, int b) { return a + b;

}

int main() {

int (\*func\_ptr)(int, int) = add; int result = func\_ptr(5, 3); printf("Result: %d\n", result); return 0;

}

O/p:

Result: 8

## Function Pointer as Argument

Implement a function void performOperation(int (\*operation)(int, int), int a, int b) that takes a function pointer as an argument and applies it to two integers, printing the result.

Sol: #include <stdio.h>

void performOperation(int (\*operation)(int, int), int a, int b) { int result = operation(a, b);

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

}

int add(int a, int b) {

return a + b;

}

int main() { performOperation(add, 3, 3);

return 0;

}

O/p: Result: 6

## Function Pointer Returning Pointer

Write a program with a function int\* max(int \*a, int \*b) that returns a pointer to the larger of two integers, and use a function pointer to call this function.

Sol: #include <stdio.h>

int\* max(int \*a, int \*b) { return (\*a > \*b) ? a : b;

}

int main() {

int x = 5, y = 3;

int\* (\*func\_ptr)(int\*, int\*) = max; int \*result = func\_ptr(&x, &y); printf("Max: %d\n", \*result);

return 0;

}

O/p: Max: 5

## Function Pointer with Different Functions

Create a program that defines two functions int add(int, int) and int multiply(int, int) and uses a function pointer to dynamically switch between these functions based on user input.

Sol: #include <stdio.h>

int add(int a, int b) { return a + b;

}

int multiply(int a, int b) { return a \* b;

}

int main() {

int (\*func\_ptr)(int, int); char operation;

printf("Enter operation (+ or \*): ");

scanf(" %c", &operation);

if (operation == '+') { func\_ptr = add;

} else if (operation == '\*') { func\_ptr = multiply;

}

int result = func\_ptr(5, 3); printf("Result: %d\n", result); return 0;

}

O/p:

Enter operation (+ or \*): = + Result: 8

Enter operation (+ or \*): \* Result: 15

## Array of Function Pointers

Implement a program that creates an array of function pointers for basic arithmetic operations (addition, subtraction, multiplication, division) and allows the user to select and execute one operation.

Sol: #include <stdio.h>

int add(int a, int b) { return a + b;

}

int subtract(int a, int b) { return a - b;

}

int multiply(int a, int b) { return a \* b;

}

int divide(int a, int b) { return a / b;

}

int main() {

int (\*operations[])(int, int) = {add, subtract, multiply, divide}; int choice, a = 10, b = 2;

printf("Choose operation: 0-Add, 1-Subtract, 2-Multiply, 3-Divide: "); scanf("%d", &choice);

if (choice >= 0 && choice <= 3) {

int result = operations[choice](a, b); printf("Result: %d\n", result);

}

return 0;

}

O/p:

Choose operation: 0-Add, 1-Subtract, 2-Multiply, 3-Divide: 0 Result: 12

Choose operation: 0-Add, 1-Subtract, 2-Multiply, 3-Divide: 1 Result: 8

Choose operation: 0-Add, 1-Subtract, 2-Multiply, 3-Divide: 2 Result: 20

Choose operation: 0-Add, 1-Subtract, 2-Multiply, 3-Divide: 3 Result: 5

## Using Function Pointers for Sorting

Write a function void sort(int \*arr, int size, int (\*compare)(int, int)) that uses a function pointer to compare elements, allowing for both ascending and descending order sorting.

Sol: #include <stdio.h> #include <stdlib.h>

int compare\_ascending(int a, int b) { return a - b;

}

int compare\_descending(int a, int b) { return b - a;

}

void sort(int \*arr, int size, int (\*compare)(int, int)) { for (int i = 0; i < size - 1; i++) {

for (int j = i + 1; j < size; j++) {

if (compare(arr[i], arr[j]) > 0) { int temp = arr[i];

arr[i] = arr[j]; arr[j] = temp;

}

}

}

}

int main() {

int arr[] = {5, 2, 9, 1, 5, 6};

int size = sizeof(arr) / sizeof(arr[0]); sort(arr, size, compare\_ascending); for (int i = 0; i < size; i++) {

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

}

printf("\n"); return 0;

}

O/p: 1 2 5 5 6 9

## Callback Function

Create a program with a function void execute(int x, int (\*callback)(int)) that applies a callback function to an integer and prints the result. Demonstrate with multiple callback functions (e.g., square, cube).

Sol: #include <stdio.h>

int square(int x) {

return x \* x;

}

int cube(int x) { return x \* x \* x;

}

void execute(int x, int (\*callback)(int)) { int result = callback(x); printf("Result: %d\n", result);

}

int main() { execute(3, square); execute(3, cube); return 0;

}

O/p: Result: 9

Result: 27

## Menu System Using Function Pointers

Implement a simple menu system where each menu option corresponds to a different function, and a function pointer array is used to call the selected function based on user input.

Sol: #include <stdio.h>

void option1() {

printf("Option 1 selected\n");

}

void option2() {

printf("Option 2 selected\n");

}

void option3() {

printf("Option 3 selected\n");

}

int main() {

void (\*menu[])(void) = {option1, option2, option3}; int choice;

printf("Select an option (0-2): ");

scanf("%d", &choice);

if (choice >= 0 && choice <= 2) { menu[choice]();

} else {

printf("Invalid option!\n");

}

return 0;

}

O.p:

Select an option (0-2): 2

Option 3 selected

## Dynamic Function Selection

Write a program where the user inputs an operation symbol (+, -, \*, /) and the program uses a function pointer to call the corresponding function.

Sol: #include <stdio.h>

int add(int a, int b) { return a + b;

}

int subtract(int a, int b) { return a - b;

}

int multiply(int a, int b) { return a \* b;

}

int divide(int a, int b) { return a / b;

}

int main() {

int a = 6, b = 2;

int (\*func\_ptr)(int, int); char operator;

printf("Enter operation (+, -, \*, /): "); scanf(" %c", &operator);

switch (operator) {

case '+': func\_ptr = add; break; case '-': func\_ptr = subtract; break; case '\*': func\_ptr = multiply; break; case '/': func\_ptr = divide; break;

default: printf("Invalid operator\n"); return 1;

}

int result = func\_ptr(a, b); printf("Result: %d\n", result);

return 0;

}

O/p:

Enter operation (+, -, \*, /): + Result: 8

## State Machine with Function Pointers

Design a simple state machine where each state is represented by a function, and transitions are handled using function pointers. For example, implement a traffic light system with states like Red, Green, and Yellow.

Sol: #include <stdio.h>

void red() { printf("Red: Stop\n");

}

void yellow() {

printf("Yellow: Get Ready\n");

}

void green() { printf("Green: Go\n");

}

int main() {

void (\*trafficLightState[])(void) = {red, yellow, green}; int state = 0; // Start with Red

while (1) { trafficLightState[state]();

state = (state + 1) % 3; // Cycle through states: Red -> Yellow -> Green -> Red getchar(); // Wait for user input to proceed to next state

}

return 0;

} O/P:

Red: Stop yellow

Yellow: Get Ready Green: Go

Red: Stop

Yellow: Get Ready Green: Go

Red: Stop

Yellow: Get Ready