Pointers & Strings(sfe)Dynamic Allocation

1. Reverse a String

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

```
#include<stdio.h>

#include<string.h>

#include<stdlib.h>

void reverseString(char *);

int main(){

    int size;

    printf("Enter the size of string:\n");

    scanf("%d",&size);

    char *input=(char *)malloc(size*sizeof(char));

    if(input==NULL){

        printf("Memory allocation failed\n");

        return 1;

    }

    printf("Enter the string:\n");

    scanf("%s",input);

// getchar();

  //  fgets(input, size + 1, stdin); To get a sentence and reverse it

    reverseString(input);

    printf("Reversed string: %s\n",input);

    free(input);

    return 0;

}
```

```
void reverseString(char *string){
   int start=0;
   int end=strlen(string)-1;
   char temp;
   while(start<end){
      temp=string[start];
      string[start]=string[end];
      string[end]=temp;
      start++;
      end--;
   }

}
```

2. Concatenate Two Strings

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

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
void concatenate(char *, char *);
int main(){
    int destSize,sourceSize;
    printf("Size of destination:\n");
    scanf("%d",&destSize);
    printf("Size of source:\n");
    scanf("%d",&sourceSize);
```

```
    // Allocate memory for the destination and source strings
    char *destination=(char *)malloc((destSize+sourceSize+1)*sizeof(char));
    char *source=(char *)malloc((sourceSize+1)*sizeof(char));

    printf("Enter destination string:\n");
    scanf("%s",destination);
    printf("Enter source string:\n");
    scanf("%s",source);

    concatenate(destination,source);
    printf("Concatenated string:%s\n",destination);
    free(destination);
    free(source);

}
void concatenate(char *destination, char *source){
    while(*destination!='\0'){
        destination++;
    }
    while(*source!='\0')
    {
        *destination=*source;
        destination++;
        source++;
```

```
    }
    *destination='\0';
}
```

3. String Length

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

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

int stringLength(const char *);
int main(){
    // char *string=malloc(100*sizeof(char));
    // if(string==NULL){
    //     printf("Memory allocation failed:\n");
    //     return 1;
    // }
    char string[100];
    printf("Enter the string:\n");
    scanf("%s",string);

   int count= stringLength(string);
    printf("string length:%d\n",count);
    //free(string);
    return 0;
}
```

```
int stringLength(const char *string){
    int count=0;
    while(*string!='\0'){
        count++;
        string++;
    }
    return count;
}
```

4. 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.

```
#include <stdio.h>

int compareStrings(const char *str1, const char *str2);

int main() {
    char string1[100], string2[100];

    printf("Enter the first string:\n");
    scanf("%s", string1);

    printf("Enter the second string:\n");
    scanf("%s", string2);

    int result = compareStrings(string1, string2);
```

```
    if (result == 0) {
        printf("The strings are equal.\n");
    } else if (result > 0) {
        printf("The first string is greater.\n");
    } else {
        printf("The second string is greater.\n");
    }

    return 0;
}

int compareStrings(const char *str1, const char *str2) {
    while (*str1 != '\0' && *str2 != '\0') {
        if (*str1 != *str2) {
            return *str1 - *str2;
        }
        str1++;
        str2++;
    }

    return *str1 - *str2;
}
```

5. 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.

```c
#include <stdio.h>

char* findSubstring(const char *str, const char *sub);

int main() {
    char string[100], substring[100];

    printf("Enter the main string:\n");
    scanf("%s", string);

    printf("Enter the substring to find:\n");
    scanf("%s", substring);

    char *result = findSubstring(string, substring);

    if (result != NULL) {
        printf("Substring found at position: %ld\n", result - string);
    } else {
        printf("Substring not found.\n");
    }

    return 0;
}
char* findSubstring(const char *str, const char *sub) {
    const char *p1, *p2, *p1_advance;
```

```
    if (*sub == '\0') {
        return (char*)str; // If substring is empty, return the main string
    }

    while (*str != '\0') {
        p1 = str;
        p2 = sub;

        // Compare characters of str and sub
        while (*p1 != '\0' && *p2 != '\0' && *p1 == *p2) {
            p1++;
            p2++;
        }
        // If we reached the end of the substring, it means we found a match
        if (*p2 == '\0') {
            return (char*)str;
        }

        str++;
    }

    return NULL; // Substring not found
}
```

6. 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.

```
// . 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.

#include<stdio.h>

void replaceChar(char *str,char oldChar, char newChar);



int main(){

    char string[]={"Athira"};

    char oldChar='h';

    char newChar='m';

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

    replaceChar(string,oldChar,newChar);

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

    return 0;



}

void replaceChar(char *str,char oldChar, char newChar){

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

        if(*str==oldChar){

            *str=newChar;

        }

        str++;

    }
```

}

7. 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.

// . 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.

```
#include<stdio.h>

void copyString(char *dest, const char *src);



int main(){

    char destination[]={"Athira"};

    char source[]={"Harikumar"};

    printf("Destination string: %s\n",destination);

    printf("Source string: %s\n",source);

    copyString(destination,source);

    printf("Copied destination string: %s\n",destination);

    return 0;



}

void copyString(char *dest, const char *src){

    while(*src!='\0')

    {

        *dest=*src;

        dest++;

        src++;
```

```
    }

    *dest='\0';

}
```

8. Count Vowels in a String

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

```
// . 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.

#include<stdio.h>

int countVowels(const char *str);



int main(){

    const char source[]={"Harikumar"};

    printf("Source string: %s\n",source);

    int count=countVowels(source);

    printf("String count: %d\n",count);

    return 0;



}

int countVowels(const char *str){

    int count=0;

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

        if(*str=='a' || *str=='A' || *str=='e' || *str=='E' || *str=='i' || *str=='I' || *str=='O' || *str=='o' || *str=='u' || *str=='U')

       {

           count++;
```

```
        }
        str++;
    }
    return count;
}
```

9. 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.

// . 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.

```
#include<stdio.h>
#include<string.h>
int isPalindrome(const char *str);

int main(){
    char string[50];
    printf("Enter a string:\n");
    scanf("%s",string);
    int check=isPalindrome(string);
    if(check){
        printf("%s is palindrome\n",string);
    }
    else
    printf("%s is not palindrome\n",string);
    return 0;
```

```
}

int isPalindrome(const char *str){

        int length=strlen(str);

        int start=0,end=length-1;

        while(start<end){

           if(str[start]!=str[end])

           return 0;

           start++;

           end--;

        }

      return 1;

}
```

10. 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.

Dynamic Memory Allocations

1. 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.

```
#include<stdio.h>

#include<stdlib.h>

int main(){

   int n;

   printf("Enter the size of array\n");
```

```
    scanf("%d",&n);
    int *array=(int *)malloc(n * sizeof(int));
    if(array==NULL){
        printf("Memory allocation failed:\n");
        return 0;
    }
    for(int i=0;i<n;i++){
        array[i]=i+1;
        printf("%d,",*(array+i));
    }
    free(array);
    return 0;
}
```

2. 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.

```
#include<stdio.h>
#include<stdlib.h>
int main(){
    char *string=(char *)malloc(10*sizeof(char));
    if(string==NULL){
        printf("Memory allocation failed\n");
        return 0;
    }
    printf("Enter the string:\n");
    scanf("%s",string);
```

```
    printf("String: %s\n",string);
    free(string);
    return 0;
}
```

3. 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.

```
#include<stdio.h>
#include<stdlib.h>
int main(){
    int n;
    printf("Enter size:\n");
    scanf("%d",&n);
    int *array=(int *)malloc(n*sizeof(int));
    if(array==NULL){
        printf("Memory allocation failed\n");
        return 0;
    }
    printf("Enter array elements:\n");
    for(int i=0;i<n;i++){
        scanf("%d",&array[i]);
    }
     for(int i=0;i<n;i++){
        printf("%d,",array[i]);
    }
    //reallocating array to 2n
```

```
    array=(int *)realloc(array,2*n*sizeof(int));
    for(int i=n;i<2*n;i++){
        array[i]=i+1;
        printf("%d,",*(array+i));


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

4. 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.

5. 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.

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
char* concatenate(char *,char *);
int main(){
    char string1[]="Athira";
    char string2[]=" Harikumar";
    char *strConcat=concatenate(string1,string2);
    printf("%s",strConcat);
    free(strConcat);
```

```
    return 0;
}
char* concatenate(char *str1,char *str2){
    int length1=strlen(str1);
    int length2=strlen(str2);
    char *strConcat=(char *)malloc((length1+length2+1)*sizeof(char));

    if(strConcat==NULL){
        printf("Memory allocation failed\n");
        return 0;
    }
    char *startAddress=strConcat;
    while(*str1!='\0'){
        *strConcat=*str1;
        str1++;
        strConcat++;
    }
    while(*str2!='\0'){
        *strConcat=*str2;
        str2++;
        strConcat++;

    }
    *strConcat='\0';
    return startAddress;
```

}

6. 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.

8. 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.

```
#include <stdio.h>

#include <stdlib.h>

int main() {

    int n;

    printf("Enter number of elements: ");

    scanf("%d", &n);

    int **array = (int **)malloc(n * sizeof(int *));

    if (array == NULL) {

        printf("Memory allocation failed\n");

        return 0;

    }

    // Dynamically allocate memory for each integer in the array of pointers

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

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


        if (array[i] == NULL) {

            printf("Memory allocation failed for element %d\n", i);

            return 0;
```

```
        }

        // Fill each integer with a value
        array[i][0] = i + 2;
        printf("%d, ", array[i][0]);
    }

    // Free the dynamically allocated memory
    for (int i = 0; i < n; i++) {
        free(array[i]);
    }
    free(array);

    return 0;
}
```

9. 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.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int x, y, z;
    // Input the dimensions of the 3D array
    printf("Enter the dimensions of the 3D array (x, y, z):\n");
    scanf("%d %d %d", &x, &y, &z);
```

```
// Dynamically allocate memory for the 3D array
int ***array = (int ***)malloc(x * sizeof(int **));
if (array == NULL) {
    printf("Memory allocation failed for the first dimension\n");
    return 1;
}
for (int i = 0; i < x; i++) {
    array[i] = (int **)malloc(y * sizeof(int *));
    if (array[i] == NULL) {
        printf("Memory allocation failed for the second dimension\n");
        return 1;
    }
    for (int j = 0; j < y; j++) {
        array[i][j] = (int *)malloc(z * sizeof(int));
        if (array[i][j] == NULL) {
            printf("Memory allocation failed for the third dimension\n");
            return 1;
        }
    }
}
// Fill the 3D array with values (e.g., i + j + k)
printf("\nFilling the 3D array with values:\n");
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] = i + j + k;  // Example value filling
            printf("array[%d][%d][%d] = %d\n", i, j, k, array[i][j][k]);
        }
    }
}
// Deallocate the memory
for (int i = 0; i < x; i++) {
    for (int j = 0; j < y; j++) {
        free(array[i][j]);  // Free each 1D array
    }
    free(array[i]);  // Free each 2D array
}
free(array);  // Free the 3D array

return 0;
}
```

Double Pointers

1. 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.

```
#include <stdio.h>

// Function to swap two integer pointers using double pointers
void swap(int **a, int **b) {
```

```
    int *temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int x = 10, y = 20;
    int *ptr1 = &x, *ptr2 = &y;

    printf("Before swapping:\n");
    printf("Value pointed by ptr1: %d\n", *ptr1);
    printf("Value pointed by ptr2: %d\n", *ptr2);

    swap(&ptr1, &ptr2);

    printf("After swapping:\n");
    printf("Value pointed by ptr1: %d\n", *ptr1);
    printf("Value pointed by ptr2: %d\n", *ptr2);

    return 0;
}
```

2. 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.

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

```
// Function to allocate memory for an array using a double pointer
void allocateArray(int **arr, int size) {
    *arr = (int *)malloc(size * sizeof(int));
    if (*arr == NULL) {
        printf("Memory allocation failed\n");
        exit(1); // Exit if memory allocation fails
    }
}

int main() {
    int *array;
    int size;

    printf("Enter the size of the array: ");
    scanf("%d", &size);

    allocateArray(&array, size);

    // Fill the array with values
    printf("Enter %d elements:\n", size);
    for (int i = 0; i < size; i++) {
        scanf("%d", &array[i]);
    }
```

```
    // Print the array
    printf("The array elements are:\n");
    for (int i = 0; i < size; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");
    free(array);
    return 0;
}
```

3. 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.

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

void modifyString(char **str) {

    char *newStr = (char *)malloc(50 * sizeof(char));
    if (newStr == NULL) {
        printf("Memory allocation failed\n");
        exit(1);
    }

    strcpy(newStr, "Hello!");
```

```
    *str = newStr;
}

int main() {
    char *originalStr = NULL;

    modifyString(&originalStr);

    printf("Modified String: %s\n", originalStr);

    free(originalStr);

    return 0;
}
```

4. 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.

```
#include <stdio.h>

int main() {
    int num = 10;
    int *ptr = &num;
    int **ptrToPtr = &ptr;

    printf("Original value of num: %d\n", num);
    **ptrToPtr = 20;
```

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


    return 0;
}
```

5. 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.

6. 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.

7. 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.

```
#include <stdio.h>

#include <stdlib.h>


void setPointer(int **ptr) {
    *ptr = (int *)malloc(sizeof(int));


    if (*ptr == NULL) {
        printf("Memory allocation failed\n");
        exit(1);
    }


    **ptr = 100;
}
```

```
int main() {
    int *ptr = NULL;

    setPointer(&ptr);

    printf("Value of dynamically allocated integer: %d\n", *ptr);

    free(ptr);

    return 0;
}
```

8. 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.

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

void allocateStringArray(char ***arr, int n) {
    *arr = (char **)malloc(n * sizeof(char *));

    if (*arr == NULL) {
        printf("Memory allocation failed\n");
        exit(1);
    }
```

```
    for (int i = 0; i < n; i++) {
        (*arr)[i] = (char *)malloc(100 * sizeof(char));

        if ((*arr)[i] == NULL) {
            printf("Memory allocation for string %d failed\n", i);
            exit(1);
        }

        sprintf((*arr)[i], "String %d", i + 1);
    }
}

int main() {
    int n;

    printf("Enter the number of strings: ");
    scanf("%d", &n);
    char **arr = NULL;

    allocateStringArray(&arr, n);

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

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

    free(arr);

    return 0;
}
```

9. 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.

Function Pointers

1. 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.

```
#include<stdio.h>

int add(int,int);

int main(){
     int (*fptr)(int,int);
     fptr=&add;
     int result = fptr(12,3);
     printf("Sum=%d",result);
}

int add(int x,int y){
```

```
    return x+y;
}
```

2. 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.

```
#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;
}

// Function that accepts a function pointer and applies the operation
void performOperation(int (*operation)(int, int), int a, int b) {
    int result = operation(a, b);  // Call the function using the function pointer
    printf("Result: %d\n", result);
}

int main() {
```

```
    int x = 10, y = 5;

    performOperation(add, x, y);


    performOperation(subtract, x, y);


    performOperation(multiply, x, y);


    return 0;

}
```

3. 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.

```
#include <stdio.h>


int* max(int *a, int *b) {

    return (*a > *b) ? a : b;

}


int main() {

    int x = 10, y = 20;


    // Declare a function pointer

    int* (*fptr)(int*, int*);

    fptr = max;


    // Use the function pointer to call the `max` function
```

```
    int *result = fptr(&x, &y);

    printf("The larger value is: %d\n", *result);



    return 0;

}
```

4. 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.

```
#include <stdio.h>

// Function definitions

int add(int a, int b) {

    return a + b;

}

int multiply(int a, int b) {

    return a * b;

}

int main() {

    int choice, x, y;

    int (*operation)(int, int);  // Function pointer declaration



    // Get user input for the operation

    printf("Choose an operation:\n");

    printf("1. Add\n");

    printf("2. Multiply\n");

    printf("Enter your choice: ");

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

```
    // Get input for the two numbers
    printf("Enter two integers:\n");
    scanf("%d %d", &x, &y);

    // Assign the function pointer based on user choice
    if (choice == 1) {
        operation = add;
    } else if (choice == 2) {
        operation = multiply;
    } else {
        printf("Invalid choice!\n");
        return 1;
    }

    // Call the selected function using the function pointer
    int result = operation(x, y);
    printf("Result: %d\n", result);
    return 0;
}
```

5. 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.

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

```
// Function definitions for basic arithmetic operations
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) {
    if (b != 0)
        return a / b;
    else {
        printf("Error: Division by zero!\n");
        return 0;
    }
}

int main() {
    // Array of function pointers
    int (*operations[4])(int, int) = {add, subtract, multiply, divide};
```

```
int choice, x, y;

// Menu for the user
printf("Choose an operation:\n");
printf("0. Add\n");
printf("1. Subtract\n");
printf("2. Multiply\n");
printf("3. Divide\n");
printf("Enter your choice: ");
scanf("%d", &choice);

// Validate the user's choice
if (choice < 0 || choice > 3) {
    printf("Invalid choice! Please select between 0 and 3.\n");
    return 1;
}

// Get input for the two numbers
printf("Enter two integers:\n");
scanf("%d %d", &x, &y);

// Call the selected operation using the array of function pointers
int result = operations[choice](x, y);
```

```
    // Print the result
    if (choice != 3 || y != 0) { // Avoid printing result if division by zero occurred
        printf("Result: %d\n", result);
    }

    return 0;
}
```

6. 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.

```
#include <stdio.h>

// Comparison functions for sorting
int ascending(int a, int b) {
    return a > b; // Returns true if a > b
}

int descending(int a, int b) {
    return a < b; // Returns true if a < b
}

// Sorting function using a function pointer
void sort(int *arr, int size, int (*compare)(int, int)) {
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - i - 1; j++) {
            if (compare(arr[j], arr[j + 1])) {
```

```
            // Swap elements if the comparison function returns true
            int temp = arr[j];
            arr[j] = arr[j + 1];
            arr[j + 1] = temp;
        }
    }
  }
}

int main() {
  int arr[] = {5, 2, 9, 1, 5, 6};
  int size = sizeof(arr) / sizeof(arr[0]);
  int choice;

  // Get user choice for sorting order
  printf("Choose sorting order:\n");
  printf("1. Ascending\n");
  printf("2. Descending\n");
  printf("Enter your choice: ");
  scanf("%d", &choice);

  // Select comparison function based on user choice
  if (choice == 1) {
    sort(arr, size, ascending);
  } else if (choice == 2) {
```

```
        sort(arr, size, descending);
    } else {
        printf("Invalid choice!\n");
        return 1;
    }

    // Print the sorted array
    printf("Sorted array: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

7. 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).

8. 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.

9. 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.

10. 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.