MASTERS OF COMPUTER APPLICATIONS

SEMESTER 1

PROGRAMMING & PROBLEM-SOLVING USING C

Unit 13

Advanced Pointers

Table of Contents

1. INTRODUCTION

In the previous unit, you read about error handling. In this unit, you will be learning some advanced concepts about pointers. A pointer is a variable that points at or refers to another variable. Pointers are very useful when you want to refer to some other variable by pointing at it. Pointers not only point to single variables but can also point to the cells of an array. Pointers provide a convenient way to represent multidimensional arrays, allowing a single multidimensional array to be replaced by a lower-dimensional array of pointers. This feature permits a collection of strings to be represented within a single array, even though the individual strings may differ in length.

1.1. Objectives:

At studying this unit, you should be able to:

- *Define pointers to pointers and pointers to functions.*
- *Recall the syntax used for declaring and initialising pointers to pointers and pointers to functions in C/C++.*
- *Explain the concept of pointers to pointers and pointers to functions,*

2. POINTER TO POINTERS

As we know, a pointer is used to store the address of a variable in C. A pointer reduces the access time of a variable. However, In C, we can also define a pointer to store the address of another pointer. Such a pointer is known as a double pointer (pointer to pointer). The first pointer is used to store the address of a variable, whereas the second pointer is used to store the address of the first pointer. The pointer to a pointer in C is used when we want to store the address of another pointer. The first pointer is used to store the address of the variable. The second pointer is used to store the address of the first pointer. That is why they are also known as ***double-pointers***. We can use a pointer to a pointer to change the values of normal pointers or create a variable-sized 2-D array. A double pointer occupies the same amount of space in the memory stack as a normal pointer.

Declaration of Pointer to a Pointer in C

Declaring Pointer to Pointer is similar to declaring a pointer in C. The difference is we have to place an additional '*' before the name of the pointer.

```
data_type_of_pointer **name_of_variable = & normal_pointer_variable;
```

```
int val = 5;
int *ptr = &val;    // storing address of val to pointer ptr.
int **d_ptr = &ptr; // pointer to a pointer declared
                    // which is pointing to an integer.
```

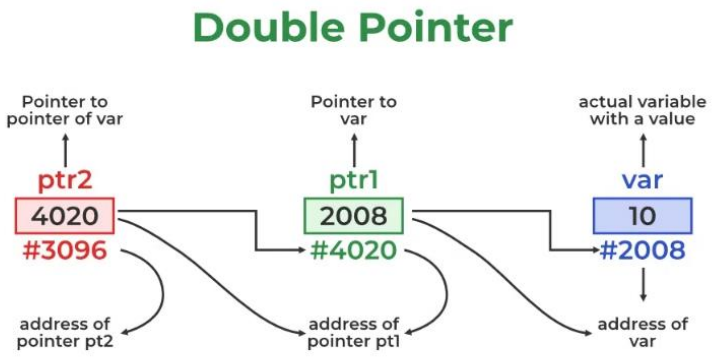

Figure 1: Pointer to pointer

The above diagram shows the memory representation of a pointer to a pointer. The first pointer, ptr1, stores the address of the variable, and the second pointer, ptr2, stores the address of the first pointer.

Example:

```
#include <stdio.h>

int main()

{       int var = 789;

        // pointer for var

        int* ptr2;

        // double pointer for ptr2

        int** ptr1;

        // storing address of var in ptr2

        ptr2 = &var;



        // Storing address of ptr2 in ptr1

        ptr1 = &ptr2;

        // Displaying value of var using

        // both single and double pointers

        printf("Value of var = %d\n", var);

        printf("Value of var using single pointer = %d\n", *ptr2);

        printf("Value of var using double pointer = %d\n", **ptr1);

        return 0;

}
```

Example:

```
#include <stdio.h>

int main() {

   int num = 10;

   int *ptr1 = &num;  // Pointer to int

   int **ptr2 = &ptr1;  // Pointer to pointer to int



   // Accessing the value of num using a pointer to pointer

   printf("Value of num: %d\n", **ptr2);



   // Modifying the value of num using a pointer to pointer

   **ptr2 = 20;

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



   return 0;

}
```

Explanation: In this example, we first declare a variable num of type int and initialise it to 10. Then, we declare a pointer ptr1 which points to the address of num. Next, we declare a pointer ptr2, which points to the address of ptr1. This makes ptr2 a pointer to pointer to int. We can access the value of num using ptr2 by dereferencing it twice (**ptr2). Similarly, we can modify the value of num using ptr2.

Dynamic Allocation with Pointer to Pointer:

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

int main () {
   int **ptr;
   int rows = 3, cols = 2;

   // Allocate memory for 2D array using pointer to pointer
   ptr = (int **) malloc(rows * sizeof(int *));
   for (int i = 0; i < rows; i++) {
      ptr[i] = (int *) malloc(cols * sizeof(int));
   }

   // Initialize and access elements of the 2D array
   for (int i = 0; i < rows; i++) {
      for (int j = 0; j < cols; j++) {
         ptr[i][j] = i + j;
         printf("%d ", ptr[i][j]);
      }
      printf("\n");
   }

   // Free memory
   for (int i = 0; i < rows; i++) {
      free(ptr[i]);
```

```
    free(ptr[i]);
  }
  free(ptr);

  return 0;
}
```

Explanation: In this example, we dynamically allocate memory for a 2D array using a pointer to pointer (ptr). We first allocate memory for an array of int* of size rows, and then for each row, we allocate memory for an array of int of size cols. This creates a 2D array where ptr[i][j] represents the element at row i and column j. Finally, we free the allocated memory to avoid memory leaks. This demonstrates how pointer-to-pointer can be used for dynamic memory allocation.

Key characteristics:

- Storage: Like regular pointers, pointers to pointers also occupy memory space to store addresses. However, instead of pointing to a value directly, they point to the memory address of another pointer variable.
- Indirection: Pointers to pointers provide an additional level of indirection. That is, they allow accessing the memory location of another pointer, which in turn can point to the actual data. This extra level of indirection enables more flexible memory manipulation and data access.
- Dynamic Memory Allocation: Pointers to pointers are commonly used in scenarios where dynamic memory allocation is required. They allow for the creation of dynamic arrays of pointers, which can then be used to dynamically allocate memory for individual elements.

- Multidimensional Arrays: Pointers to pointers are often employed in the creation and manipulation of multidimensional arrays in C. By using arrays of pointers, it becomes possible to create arrays with varying sizes for each dimension, providing flexibility in memory management.
- Function Arguments: Pointers to pointers are sometimes used to modify variables passed as function arguments. Since modifications made to a pointer to a pointer affect the original pointer, they can be used to implement functions that need to return multiple values or to modify pointers themselves.
- Error Handling: Pointers to pointers are sometimes used for error handling purposes. Functions that return pointers can use pointers to pointers as parameters to indicate whether they were successful in allocating memory. By passing a pointer to a pointer, the function can update the original pointer to indicate success or failure.
- Complex Data Structures: Pointers to pointers can be used to build complex data structures such as trees, graphs, or linked lists of pointers. This allows for the creation of flexible and efficient data structures that can adapt to changing requirements.
- Pointer Arithmetic: Like regular pointers, pointer arithmetic can be performed on pointers to pointers. However, it's important to handle pointer arithmetic with caution to avoid unintended memory access violations or undefined behaviour.

Advantages:

- Dynamic Memory Allocation: Pointers to pointers allow for more flexible and dynamic memory allocation. They can be used to manage multi-dimensional arrays dynamically, providing efficient memory usage and reducing memory fragmentation.
- Linked Data Structures: Pointers to pointers are essential for implementing linked data structures like linked lists, trees, and graphs. They enable the creation of nodes that contain pointers to other nodes, facilitating efficient traversal and manipulation of the data structure.
- Function Parameter Passing: In certain scenarios, passing pointers to pointers as function parameters allows for modifying pointer values within the function, leading

to changes reflected outside the function scope. This can be useful for functions that need to modify pointers directly, such as functions for dynamic memory allocation or linked list operations.

- Indirect Access: Pointers to pointers provide an additional level of indirection, allowing for indirect access to memory locations. This can be advantageous in scenarios where multiple levels of abstraction are required, such as when dealing with arrays of pointers or implementing data structures like trees with nodes pointing to other nodes.
- Error Handling: Pointers to pointers can be used to implement error-handling mechanisms, particularly in functions that allocate memory dynamically. By passing pointers to pointers as function parameters, functions can update the pointer value to NULL upon encountering errors, providing a way to indicate failure and avoid memory leaks.

SELF-ASSESSMENT QUESTIONS – 1

1. What is the purpose of using pointers to pointers in C programming?
   (a) To store addresses of variables
   (b) To manipulate memory addresses directly
   (c) To store addresses of other pointers
   (d) To perform arithmetic operations on pointers

2. Which of the following correctly declares a pointer to a pointer in C?
   (a) int **ptr_ptr;
   (b) int *ptr_ptr;
   (c) int *&ptr_ptr;
   (d) int &*ptr_ptr;

3. Pointers to functions can be beneficial in C for:

(a) Implementing callback mechanisms

(b) Performing arithmetic operations

(c) Defining complex data structures

(d) Allocating memory dynamically

4. How are double pointers commonly used in dynamic memory allocation?

(a) To access array elements

(b) To pass arrays to functions

(c) To create linked lists and trees

(d) To manage memory blocks efficiently

3. POINTERS TO FUNCTIONS

In C, functions are first-class citizens, meaning they can be treated just like any other variable. Pointers to functions enable us to store the addresses of functions and call them indirectly. In C, like normal data pointers (int *, char *, etc), we can have pointers to functions. Following is a simple example that shows declaration and function call using function pointer.

Example:

```
#include <stdio.h>

// A normal function with an int parameter

// and void return type

void fun(int a)

{
```

```
    printf("Value of a is %d\n", a);
}
int main()
{
    // fun_ptr is a pointer to function fun()
    void (*fun_ptr)(int) = &fun;

    /* The above line is equivalent to the following two
    void (*fun_ptr)(int);
    fun_ptr = &fun;
    */
    // Invoking fun() using fun_ptr
    (*fun_ptr)(100);
    return 0;
}
```

Output:

Value of a is 100

Why do we need an extra bracket around function pointers like fun_ptr in the above example? If we remove bracket, then the expression “void (*fun_ptr) (int)” becomes “void *fun_ptr(int)” which is declaration of a function that returns void pointer. Following are some interesting facts about function pointers.

1) Unlike normal pointers, a function pointer points to code, not data. Typically, a function pointer stores the start of executable code.
2) Unlike normal pointers, we do not allocate or de-allocate memory using function pointers.
3) A function's name can also be used to get the function's address. For example, in the below program, we have removed the address operator '&' in the assignment. We have also changed function call by removing *, the program still works.

Example:

```
#include <stdio.h>
// A normal function with an int parameter
// and void return type
void fun(int a)
{
    printf("Value of a is %d\n", a);
}

int main()
{
    void (*fun_ptr)(int) = fun; // & removed

    fun_ptr(10); // * removed

    return 0;
}
```

Output:

Value of a is 10

4) Like normal pointers, we can have an array of function pointers. The below example in point 5 shows syntax for an array of pointers.
5) Function pointer can be used in place of switch case. For example, in the program below, the user is asked to choose between 0 and 2 to do different tasks.
6) Like normal data pointers, a function pointer can be passed as an argument and can also be returned from a function. For example, consider the following C program where wrapper() receives a void fun() as a parameter and calls the passed function.

This point, in particular, is very useful in C. In C, we can use function pointers to avoid code redundancy. For example a simple qsort() function can be used to sort arrays in ascending order or descending or by any other order in case of array of structures. Not only this but with function pointers and void pointers, it is possible to use qsort for any data type.

Key characteristics of pointers to functions:

Data Type: Pointers to functions are a data type in C, just like pointers to variables. They store memory addresses of functions rather than data.

Declaration Syntax: Pointers to functions are declared using the function's return type, followed by an asterisk (*), the name of the pointer variable, and finally, the function parameters in parentheses.

Example:

```
int (*func_ptr)(int, int);
```

Initialisation: Pointers to functions can be initialised with the address of a function that has a compatible signature (matching return type and parameter types). The address of a function can be obtained by simply using the function name without parentheses.

Example:

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

int (*func_ptr)(int, int) = add;
```

Dereferencing: To call a function through a function pointer, you must dereference the pointer using the asterisk (*) operator and provide arguments in parentheses as you would with a regular function call.

Example:

```
int result = (*func_ptr)(2, 3);
```

Function Pointer Arithmetic: It's possible to perform arithmetic operations on function pointers, but it's generally not recommended as the behaviour is implementation-defined and can lead to undefined behaviour.

Callback Functions: Function pointers are commonly used for implementing callback mechanisms, where a function is passed as an argument to another function to be called back later during execution.

Polymorphism: Pointers to functions enable polymorphic behaviour in C, allowing different functions to be called based on the context or runtime conditions.

Function Pointers in Data Structures: Function pointers can be stored in data structures like arrays or structs, allowing for dynamic selection and invocation of functions at runtime.

Function Pointer Typedefs: Typedefs can be used to create aliases for complex function pointer types, improving code readability and maintainability.

Passing as Parameters: Function pointers can be passed as parameters to other functions, enabling dynamic dispatch and allowing functions to operate on different types of data or perform different operations based on the function pointer passed.

Example: Function Pointer as a Callback

```
#include <stdio.h>


// Function to add two numbers
int add(int a, int b) {
   return a + b;
}


// Function to subtract two numbers
int subtract(int a, int b) {
   return a - b;
}


// Function to perform an operation using a function pointer as a callback
int operate(int a, int b, int (*operation)(int, int)) {
   return operation(a, b);
}


int main() {
   int result;
   // Declare a function pointer
   int (*operation)(int, int);


   // Assign the add function to the function pointer
   operation = add;
   result = operate(5, 3, operation);
```

```
    result = operate(5, 3, operation);
    printf("Result of addition: %d\n", result);

    // Assign the subtract function to the function pointer
    operation = subtract;
    result = operate(5, 3, operation);
    printf("Result of subtraction: %d\n", result);

    return 0;
}
```

Explanation: In this example, we define two functions, add and subtract, which perform addition and subtraction operations, respectively. Then, we define a function operate that takes two integers and a function pointer as arguments. This function calls the function pointed to by the function pointer, passing the two integers as arguments. In the main function, we demonstrate using the operate function with both the add and subtract functions by assigning their addresses to the function pointer operation.

Sorting Array Using Function Pointer

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

// Comparison function for ascending order
int compare_asc(const void *a, const void *b) {
    return (*(int *)a - *(int *)b);
}
```

```
// Comparison function for descending order
int compare_desc(const void *a, const void *b) {
    return (*(int *)b - *(int *)a);
}

int main() {
    int arr[] = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3};
    int n = sizeof(arr) / sizeof(arr[0]);

    // Sort array in ascending order
    qsort(arr, n, sizeof(int), compare_asc);
    printf("Sorted array in ascending order: ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");

    // Sort array in descending order
    qsort(arr, n, sizeof(int), compare_desc);
    printf("Sorted array in descending order: ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");

    return 0;
}
```

Explanation: In this example, we use the qsort function from the C standard library to sort an array of integers. The qsort function requires a comparison function to determine the order of elements. We define two comparison functions, compare_asc and compare_desc, which compare two integers in ascending and descending order, respectively. We then pass the appropriate comparison function as a function pointer to qsort to sort the array in ascending and descending order.

Advantages:

- Dynamic Dispatch: Pointers to functions enable dynamic dispatch, allowing the selection of which function to call at runtime. This is particularly useful in scenarios where the appropriate function to execute may change based on user input, configuration settings, or other runtime conditions.
- Callback Mechanism: Pointers to functions are commonly used to implement callback mechanisms. Callback functions provide a way to extend the functionality of a program by allowing users to register custom functions to be called at specific points in the program's execution. This enhances modularity and extensibility.
- Function Pointers as Arguments: Passing function pointers as arguments to other functions enables the creation of generic and flexible code. This allows functions to operate on different types of data or to perform different operations based on the function pointer passed as an argument.
- Function Composition: Function pointers can be used to compose higher-order functions, enabling the creation of complex behaviour by combining simpler functions. This promotes code reuse and maintainability by separating concerns and promoting modular design.
- Dynamic Algorithm Selection: In certain applications, such as optimisation algorithms or event-driven systems, pointers to functions can be used to dynamically select algorithms or event handlers based on runtime conditions or input data. This provides adaptability and flexibility in algorithmic decision-making.

- Reduced Code Duplication: By using function pointers, it's possible to reduce code duplication by centralising common functionality in separate functions and invoking them through pointers. This promotes cleaner code and easier maintenance.

SELF-ASSESSMENT QUESTIONS – 2

5. Pointers to functions in C _____.
6. How is a pointer to a function declared in C?
   (a) int *func_ptr();
   (b) void func_ptr();
   (c) int (*func_ptr)();
   (d) void (*func_ptr)();
7. Which of the following demonstrates the correct syntax for calling a function indirectly using a pointer to a function?
   (a) (*func_ptr)();
   (b) *func_ptr();
   (c) func_ptr();
   (d) func_ptr(*);

4. SUMMARY

Pointers to pointers and pointers to functions are advanced features of C programming that provide flexibility and efficiency in memory management and function handling. While they may seem daunting initially, mastering these concepts can significantly enhance your ability to write efficient and flexible C programs. With practice and understanding, you can leverage these features to write cleaner, more modular, and powerful code.

5. TERMINAL QUESTIONS

1. What is the purpose of using pointers to pointers in C programming? Provide an example scenario where double pointers are useful.
2. Explain the syntax for declaring and dereferencing a pointer to a pointer in C with a code example.
3. How are pointers to pointers commonly used in dynamic memory allocation? Describe a situation where double pointers are essential for memory management.
4. What are pointers to functions in C? How do they differ from regular function pointers?
5. Compare and contrast the usage of pointers to pointers and pointers to functions in C programming. When would you prefer one over the other?
6. List the advantages of Pointers to Functions.
7. List the advantages of Pointers to Pointers.
8. Elaborate on the characteristics of Pointers to Functions.
9. List the characteristics of Pointers to Functions.

6. ANSWERS TO SELF ASSESSMENT QUESTIONS

1. (c) To store addresses of other pointers
2. (a) int **ptr_ptr;
3. (a) Implementing callback mechanisms
4. (d) To manage memory blocks efficiently
5. Passing functions as arguments to other functions
6. (c) int (*func_ptr)();
7. (a) (*func_ptr)();

7. ANSWERS TO TERMINAL QUESTIONS

1. The purpose of using pointers to pointers in C programming is to have indirect access to a pointer. This allows for dynamic manipulation of memory locations and data structures. One common scenario where double pointers are useful is in dynamically allocating memory for multi-dimensional arrays or linked data structures like linked lists, trees, or graphs.
2. Declaration of a pointer to a pointer in C follows the syntax: type **pointer_name;
3. Pointers to pointers are commonly used in dynamic memory allocation when dealing with multi-dimensional arrays or linked data structures. They are essential for memory management in scenarios where the number of elements or nodes is not known at compile time.
4. Pointers to functions in C are variables that store addresses of functions. They allow functions to be passed as arguments to other functions, returned from functions, and stored in data structures.

   The syntax for declaring a pointer to a function is: return_type (*pointer_name)(parameter_type1, parameter_type2, ...);

   Regular function pointers and pointers to functions are the same thing. The term "regular function pointers" may refer to function pointers in general, while "pointers to functions" specifically emphasises the capability of pointers to store function addresses.)
5. Dynamic Memory Allocation: Pointers to pointers allow for more flexible and dynamic memory allocation. They can be used to manage multi-dimensional arrays dynamically, providing efficient memory usage and reducing memory fragmentation.

   Linked Data Structures: Pointers to pointers are essential for implementing linked data structures like linked lists, trees, and graphs. They enable the creation of nodes that contain pointers to other nodes, facilitating efficient traversal and manipulation of the data structure. Refer to section 3.
6. Dynamic Dispatch: Pointers to functions enable dynamic dispatch, allowing the selection of which function to call at runtime. This is particularly useful in scenarios where the appropriate execution function may change based on user input, configuration settings, or other runtime conditions.

Callback Mechanism: Pointers to functions are commonly used to implement callback mechanisms. Callback functions provide a way to extend the functionality of a program by allowing users to register custom functions to be called at specific points in the program's execution. This enhances modularity and extensibility. Refer to section 4.

7. Storage: Like regular pointers, pointers to pointers also occupy memory space to store addresses. However, instead of pointing to a value directly, they point to the memory address of another pointer variable.

   Indirection: Pointers to pointers provide an additional level of indirection. That is, they allow accessing the memory location of another pointer, which in turn can point to the actual data. This extra level of indirection enables more flexible memory manipulation and data access. Refer to section 3.

8. Data Type: Pointers to functions are a data type in C, just like pointers to variables. They store memory addresses of functions rather than data.

   Declaration Syntax: Pointers to functions are declared using the function's return type, followed by an asterisk (*), the name of the pointer variable, and finally, the function parameters in parentheses. Refer to section 4.

8. EXERCISES

Exercise 1: Pointers to Pointers

1. Write a C program to swap two integer values using pointers to pointers.
2. Implement a function to find the maximum element in a 2D array using pointers to pointers.
3. Write a program to dynamically allocate memory for a 2D array using pointers to pointers.

Solutions:

1.

```
#include <stdio.h>


// Function to swap two integer values using pointers to pointers
void swap(int **x, int **y) {
   int *temp = *x;
   *x = *y;
   *y = temp;
}


int main() {
   int a = 5, b = 10;
   int *ptr_a = &a, *ptr_b = &b;


   printf("Before swapping: a = %d, b = %d\n", a, b);


   swap(&ptr_a, &ptr_b);


   printf("After swapping: a = %d, b = %d\n", a, b);


   return 0;
}
```

2.

```
#include <stdio.h>


// Function to find the maximum element in a 2D array using pointers to pointers
int findMax(int **arr, int rows, int cols) {
   int max = **arr;
   for (int i = 0; i < rows; i++) {
      for (int j = 0; j < cols; j++) {
         if (*(*(arr + i) + j) > max) {
            max = *(*(arr + i) + j);
         }
      }
   }
   return max;
}


int main() {
   int rows = 3, cols = 3;
   int arr[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
   int *ptr_arr[3];


   // Initialize pointer array with base addresses of rows
   for (int i = 0; i < rows; i++) {
      ptr_arr[i] = arr[i];
   }
```

3.

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

// Function to dynamically allocate memory for a 2D array using pointers to pointers
int** allocate2DArray(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 = 3, cols = 3;
   int **ptr_arr = allocate2DArray(rows, cols);

   // Assign values to the dynamically allocated 2D array
   for (int i = 0; i < rows; i++) {
      for (int j = 0; j < cols; j++) {
         ptr_arr[i][j] = i * cols + j;
      }
   }

   // Print the dynamically allocated 2D array
   printf("Dynamically allocated 2D array:\n");
```

```
    printf("Dynamically allocated 2D array:\n");
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("%d ", ptr_arr[i][j]);
        }
        printf("\n");
    }

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

    return 0;
}
```

Exercise 2: Pointers to Functions

1. Implement a function 'applyfunction' that takes a function pointer as an argument and applies the function to an array of integers.
2. Write a program that sorts an array of integers using a function pointer to a sorting algorithm.
3. Create a calculator program that allows users to perform arithmetic operations using function pointers to corresponding operation functions.

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

// Function pointer typedef for comparison function
typedef int (*CompareFunc)(const void *, const void *);

// Function to sort an array of integers using a function pointer to a sorting algorithm
void sort(int *arr, int size, CompareFunc compare) {
   qsort(arr, size, sizeof(int), compare);
}

// Comparison function for ascending order
int compare_asc(const void *a, const void *b) {
   return (*(int *)a - *(int *)b);
}

// Comparison function for descending order
int compare_desc(const void *a, const void *b) {
   return (*(int *)b - *(int *)a);
}

int main() {
   int arr[] = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3};
   int n = sizeof(arr) / sizeof(arr[0]);
```

```
    // Sort array in ascending order
    sort(arr, n, compare_asc);
    printf("Sorted array in ascending order: ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");

    // Sort array in descending order
    sort(arr, n, compare_desc);
    printf("Sorted array in descending order: ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");

    return 0;
}
```

9. REFERENCES

1. C Programming: A Modern Approach, K. N. King, W. W. Norton & Company, 2008, ISBN-10: 0393979504, ISBN-13: 978-0393979503, Chapter(s): "Pointers and Arrays" covers pointers to pointers, while "Functions and Program Structure" covers pointers to functions.
2. The C Programming Language, Brian W. Kernighan, Dennis M. Ritchie, Prentice Hall, ISBN-10: 0131103628, ISBN-13: 978-0131103627, Chapter(s): Throughout the book, pointers to pointers and pointers to functions are discussed, particularly in chapters dealing with pointers and functions.