POINTERS AND ARRAYS
==========

```
#include <stdio.h>
void CopyString(char*,char*);
int main()
{
  char str1[]="noel";
  char str2[20];
  char *Pstr1,*Pstr2;
  Pstr1=str1;
  Pstr2=str2;
  CopyString(Pstr1,Pstr2);
  printf("str2 = %s \n",str2);
  return 0;
}
void CopyString(char*from,char*to){
    for(;*from!='\0';from++,to++){
        *to=*from;

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

MALLOC FUNCTION
==========

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
 int *pNumber= (int*)malloc(100);
 printf("pNumber = %p",pNumber);
 return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
 int *pNumber= (int*)malloc(100);
 printf("pNumber = %p",pNumber);
 free(pNumber);
 return 0;
}
```

DOUBLE AND TRIPPLE POINTERS

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
```

```
    int num =20;
    int *p=&num;
    int **dp=&p;
    int ***tp=&dp;
 printf("001num = %d \n",*p);
 printf("002num = %d \n",**dp);
 printf("003num = %d \n",***tp);


 return 0;
}
```

ADDRESS OF NUM AND *dp are same.

==========

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int num =20;
    int *p=&num;
    int **dp=&p;
    int ***tp=&dp;
 printf("001num = %d \n",*p);
 printf("Address of a=%p \n",&num);
 printf("002num = %p \n",*dp);


 return 0;
}
```

DOUBLE POINTER

==========

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int num =20;
    int *p=&num;
    int **dp=&p;
    int ***tp=&dp;
 printf("001num = %d \n",*p);
 printf("Address of a=%p \n",&num);
 printf("002num = %d \n",**dp);


 return 0;
}
```

DEREFERENCING

==========

```
#include <stdio.h>

int main()
{
  int i = 5,j=6,k=7;
```

```
  int *ip1 = &i;

  printf("Address of i = %p\n",&i);
   printf("Address of j = %p\n",&j);
    printf("Address of k = %p\n",&k);

  printf("value of ip1 = %d\n",*ip1);
  int *ip2 = &j;
   printf("value of ip2 = %d\n",*ip2);
  int **ipp = &ip1;

 // *ipp=ip2;
 *ipp = &k;


  printf("*ipp = %p\n",*ipp);
  printf("**ipp = %d\n", **ipp);


    return 0;
}
```

SET OF PROGRAMS
==========

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>

void reverseString(char *str) {
    // Get the length of the string
    int len = strlen(str);

    // Swap characters from both ends of the string
    int start = 0;
    int end = len - 1;

    while (start < end) {
        // Swap characters
        char temp = str[start];
        str[start] = str[end];
        str[end] = temp;

        // Move towards the center
        start++;
        end--;
    }
}

int main() {
    char str[] = "Hello, World!";

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

```
    reverseString(str);

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

    return 0;
}
```

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>

void concatenateStrings(char *ch, const char *src) {
    // Move the pointer ‘dest‘ to the end of the current string
    while (*ch != ’\0’) {
        ch++;  // Increment pointer until we find the null-terminator
    }

    // Now, append the characters from ‘src‘ to ‘ch‘
    while (*src != ’\0’) {
        *ch = *src;  // Copy the character from src to ch
        ch++;        // Move destination pointer forward
        src++;         // Move source pointer forward
    }

    // Null-terminate the destination string
    *ch = ’\0’;
}

int main() {
    char ch[100] = "Hello, ";
    const char *src = "World!";

    concatenateStrings(ch, src);

    printf("Concatenated string: %s\n", ch);  // Should print "Hello, World!"

    return 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>

int stringLength(const char *str) {
    const char *ptr = str;  // Pointer to the string

    // Iterate through the string until we hit the null terminator
    while (*ptr != ’\0’) {
        ptr++;  // Move the pointer to the next character
    }
```

```
    // Return the difference between the final pointer and the original pointer
    return ptr - str;
}

int main() {
    const char *myString = "Hello, World!";
    printf("Length of the string: %d\n", stringLength(myString));
    return 0;
}
```

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) {
    // Loop through both strings character by character
    while (*str1 != '\0' && *str2 != '\0') {
        // If characters are different, return the difference between them
        if (*str1 != *str2) {
            return (unsigned char)*str1 - (unsigned char)*str2;
        }
        // Move to the next character in both strings
        str1++;
        str2++;
    }

    // If we reached the end of both strings at the same time, they are equal
    // If str1 is longer than str2, return a positive value
    // If str2 is longer than str1, return a negative value
    return (unsigned char)*str1 - (unsigned char)*str2;
}

int main() {
    const char *str1 = "apple";
    const char *str2 = "apple";

    int result = compareStrings(str1, str2);
    if (result == 0) {
        printf("The strings are equal.\n");
    } else if (result > 0) {
        printf("str1 is greater.\n");
    } else {
        printf("str2 is greater.\n");
    }

    return 0;
}
```

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.

```
#include <stdio.h>

char* findSubstring(const char *str, const char *sub) {
    // If sub is an empty string, return the beginning of str
    if (*sub == '\0') {
        return (char*)str;
    }

    // Loop through str
    for (const char *s = str; *s != '\0'; s++) {
        const char *s1 = s;
        const char *s2 = sub;

        // Check if the substring starting at s matches sub
        while (*s1 == *s2 && *s2 != '\0') {
            s1++;
            s2++;
        }

        // If the full substring matched, return the pointer to the start of the match
        if (*s2 == '\0') {
            return (char*)s;
        }
    }

    // Return NULL if substring not found
    return NULL;
}

int main() {
    const char *str = "Hello, world!";
    const char *sub = "world";

    char *result = findSubstring(str, sub);

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

    return 0;
}
```

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.

```
#include <stdio.h>

void replaceChar(char *str, char oldChar, char newChar) {
    // Iterate over each character in the string
    while (*str != '\0') {
        // If the current character matches oldChar, replace it
```

```
        if (*str == oldChar) {
            *str = newChar;
        }
        // Move to the next character
        str++;
    }
}

int main() {
    char str[] = "Hello, World!";

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

    // Replace 'o' with '0'
    replaceChar(str, 'o', '0');

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

    return 0;
}
```

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.

```
#include <stdio.h>

void copyString(char *dest, const char *src) {
    // Loop through the source string and copy each character to the destination
    while (*src != '\0') {
        *dest = *src;  // Copy character
        dest++;        // Move destination pointer to next position
        src++;         // Move source pointer to next character
    }
    *dest = '\0';  // Null-terminate the destination string
}

int main() {
    char src[] = "Hello, World!";
    char dest[50];  // Ensure the destination has enough space

    copyString(dest, src);  // Call the function to copy the string

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

    return 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.

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

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

    // Iterate through each character of the string
    while (*str != '\0') {
        char ch = *str;

        // Check if the character is a vowel (both lowercase and uppercase)
        if (ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o' || ch == 'u' ||
            ch == 'A' || ch == 'E' || ch == 'I' || ch == 'O' || ch == 'U') {
            count++;  // Increment the count if it's a vowel
        }

        // Move to the next character
        str++;
    }

    return count;
}

int main() {
    const char *inputString = "Hello World!";
    int vowelCount = countVowels(inputString);
    printf("Number of vowels: %d\n", vowelCount);

    return 0;
}
```

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.

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

int isPalindrome(const char *str) {
    int left = 0;
    int right = strlen(str) - 1;

    // Check for palindrome by comparing characters from both ends
    while (left < right) {
        if (str[left] != str[right]) {
            return 0; // Not a palindrome
        }
        left++;
        right--;
    }
    return 1; // It is a palindrome
}

int main() {
    const char *str = "madam";  // Example input
    if (isPalindrome(str)) {
        printf("\"%s\" is a palindrome.\n", str);
    } else {
```

```
        printf("\"%s\" is not a palindrome.\n", str);
    }
    return 0;
}
```

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.

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

void tokenizeString(char *str, const char *delim, void (*processToken)(const char *)) {
    // Use strtok to tokenize the string
    char *token = strtok(str, delim);

    // While there are tokens left
    while (token != NULL) {
        // Call the processToken function for each token
        processToken(token);

        // Get the next token
        token = strtok(NULL, delim);
    }
}

// Sample processToken function that simply prints the token
void printToken(const char *token) {
    printf("Token: %s\n", token);
}

int main() {
    // Example string to tokenize
    char str[] = "Hello, world! This is C programming.";

    // Tokenize the string with space, comma, and exclamation mark as delimiters
    tokenizeString(str, " ,.!?", printToken);

    return 0;
}
```

SET OF PROBLEMS
==========
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;

    // Ask the user for the size of the array
    printf("Enter the size of the array (n): ");
```

```
    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;  // Return an error code if malloc fails
    }

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

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

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

    // Confirm that memory has been freed
    printf("Memory has been freed.\n");

    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() {
    // Declare a pointer to store the dynamically allocated string
    char *str;
    int size = 10;  // Initial size of the string (can be adjusted based on input length)

    // Dynamically allocate memory for the string
    str = (char *)malloc(size * sizeof(char));
    if (str == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }

    printf("Enter a string: ");

    // Read characters one by one and store them in the dynamically allocated memory
    int i = 0;
```

```
    char ch;
    while ((ch = getchar()) != '\n' && ch != EOF) {
        // Check if there is enough space, reallocate if necessary
        if (i >= size - 1) {
            size *= 2;  // Double the size of the memory block
            str = (char *)realloc(str, size * sizeof(char));
            if (str == NULL) {
                printf("Memory reallocation failed.\n");
                return 1;
            }
        }
        str[i] = ch;
        i++;
    }

    // Null-terminate the string
    str[i] = '\0';

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

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

    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;

    // Ask for the number of elements
    printf("Enter the number of elements: ");
    scanf("%d", &n);

    // Dynamically allocate memory for an array of n integers
    int *arr = (int *)malloc(n * sizeof(int));
    if (arr == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

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

    // Resize the array to 2n using realloc
```

```
    int *newArr = (int *)realloc(arr, 2 * n * sizeof(int));
    if (newArr == NULL) {
        printf("Memory reallocation failed!\n");
        free(arr);  // Free the originally allocated memory
        return 1;
    }

    // Fill the newly allocated elements
    printf("Enter %d more integers to fill the new part of the array:\n", n);
    for (int i = n; i < 2 * n; i++) {
        scanf("%d", &newArr[i]);
    }

    // Print the resized array
    printf("The resized array is:\n");
    for (int i = 0; i < 2 * n; i++) {
        printf("%d ", newArr[i]);
    }
    printf("\n");

    // Free the allocated memory
    free(newArr);

    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.

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

void allocate_and_fill_matrix(int m, int n) {
    // Dynamically allocate memory for the matrix (array of pointers to rows)
    int **matrix = (int **)malloc(m * sizeof(int *));

    if (matrix == NULL) {
        printf("Memory allocation failed for rows!\n");
        return;
    }

    // Dynamically allocate memory for each row
    for (int i = 0; i < m; i++) {
        matrix[i] = (int *)malloc(n * sizeof(int));
        if (matrix[i] == NULL) {
            printf("Memory allocation failed for columns in row %d!\n", i);
            return;
        }
    }

    // Fill the matrix with values
    printf("Enter the values for the matrix (%d x %d):\n", m, n);
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
```

```
            printf("matrix[%d][%d]: ", i, j);
            // Use scanf to input the values into the matrix
            scanf("%d", &matrix[i][j]);
        }
    }

    // Print the matrix to verify the values entered
    printf("\nThe matrix is:\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 the matrix
    for (int i = 0; i < m; i++) {
        free(matrix[i]);  // Free each row
    }
    free(matrix);  // Free the array of row pointers

    printf("\nMemory deallocated successfully.\n");
}

int main() {
    int m, n;

    // Take the dimensions of the matrix from the user
    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
    allocate_and_fill_matrix(m, n);

    return 0;
}
```

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>

// Function to concatenate two strings dynamically
char* concatenate_strings(const char *str1, const char *str2) {
    // Calculate the length of the new string
    size_t len1 = strlen(str1);
    size_t len2 = strlen(str2);

    // Allocate memory for the new string (including space for null terminator)
    char *result = (char*) malloc((len1 + len2 + 1) * sizeof(char));
```

```
    // Check if memory allocation was successful
    if (result == NULL) {
        printf("Memory allocation failed.\n");
        exit(1); // Terminate program if memory allocation fails
    }

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

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

    return result;  // Return the concatenated string
}

int main() {
    // Define two strings to concatenate
    const char *str1 = "Hello, ";
    const char *str2 = "World!";

    // Concatenate strings
    char *concatenated_string = concatenate_strings(str1, str2);

    // Print the concatenated string
    printf("Concatenated String: %s\n", concatenated_string);

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

    return 0;
}
```

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.

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

struct Student {
    char name[100];
    int age;
    float grade;
};

int main() {
    // Dynamically allocate memory for one 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 1;
    }
```

```
    // Input student details
    printf("Enter student's name: ");
    scanf("%s", student->name);  // Reading a string (name)

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

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

    // Output the student 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);

    return 0;
}
```

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, i;

    // Ask the user how many integers they want to store
    printf("Enter the number of integers: ");
    scanf("%d", &n);

    // Dynamically allocate memory for an array of n pointers
    int **arr = (int **)malloc(n * sizeof(int *));
    if (arr == NULL) {
        printf("Memory allocation failed!\n");
        return 1; // Exit if memory allocation fails
    }

    // Dynamically allocate memory for each integer in the array
    for (i = 0; i < n; i++) {
        arr[i] = (int *)malloc(sizeof(int));
        if (arr[i] == NULL) {
            printf("Memory allocation failed for arr[%d]!\n", i);
            return 1; // Exit if memory allocation fails
        }
    }

    // Fill the array with values (you can modify this as needed)
    for (i = 0; i < n; i++) {
```

```
        printf("Enter value for arr[%d]: ", i);
        scanf("%d", arr[i]);
    }

    // Display the values stored in the dynamically allocated memory
    printf("\nValues stored in the dynamic array:\n");
    for (i = 0; i < n; i++) {
        printf("arr[%d] = %d\n", i, *arr[i]);
    }

    // Free the dynamically allocated memory
    for (i = 0; i < n; i++) {
        free(arr[i]);  // Free memory for each integer
    }
    free(arr);  // Free the memory for the array of pointers

    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;

    // Get dimensions of the 3D array
    printf("Enter dimensions for the 3D array (x, y, z): ");
    scanf("%d %d %d", &x, &y, &z);

    // Dynamically allocate memory for a 3D array
    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 array with values (for example, i+j+k)
    printf("Filling the 3D array with values (i + j + k)...\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 filling logic
            }
        }
    }

    // Print the 3D array
    printf("The 3D array contents are:\n");
```

```
    for (int i = 0; i < x; i++) {
        for (int j = 0; j < y; j++) {
            for (int k = 0; k < z; k++) {
                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(array[i]);
    }
    free(array);

    printf("Memory deallocated successfully.\n");

    return 0;
}
```

FUNCTION POINTER

==========

```
#include <stdio.h>
#include <string.h>
int add(int,int);
int main() {
    int (*fptr)(int,int);//declaration of function pointer
    //intialize the function pointer
    fptr=&add;
    //call the function using pointer
    printf("%d",fptr(10,5));
    return 0;
}
 int add(int a,int b){
     return a+b;
 }
```

USING VOID

==========

```
#include <stdio.h>
#include <string.h>
void add(int,int);
int main() {
    void (*fptr)(int,int);//declaration of function pointer
    //intialize the function pointer
    fptr=&add;
    //call the function using pointer
   fptr(10,5);
    return 0;
}
  void add(int a,int b){
```

```
    int sum= a+b;
    printf("%d \n",sum);
 }
```

MORE FUNCTION
==========

```
#include <stdio.h>
#include <string.h>
void add(int,int);
void sub(int, int);
int main() {
 void (*fptr)(int,int);//declaration of function pointer
 //intialize the function pointer
 fptr=&add;
 printf("001 fptr=%p \n",fptr);
 //call the function using pointer
 fptr(10,5);

 fptr=&sub;
  printf("002 fptr=%p \n",fptr);

 fptr(6,5);


 return 0;
}
void add(int a,int b) {
 int sum= a+b;
 printf("%d \n",sum);
}

void sub(int a,int b){
    int subt= a-b;
    printf("%d \n",subt);
 }
```

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 the values of two integers using double pointers
void swap(int **a, int **b) {
    int *temp = *a;  // Temporary pointer to hold the value of *a
    *a = *b;         // Set *a to point to *b (value of b)
    *b = temp;       // Set *b to point to temp (original value of a)
}

int main() {
    int x = 5, y = 10;
    int *px = &x, *py = &y;
```

```
    // Print initial values
    printf("Before swapping: x = %d, y = %d\n", x, y);

    // Call the swap function
    swap(&px, &py);

    // Print the swapped values
    printf("After swapping: x = %d, y = %d\n", x, y);

    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 dynamically allocate memory for an array of integers
void allocateArray(int **arr, int size) {
    // Allocate memory for the array of integers
    *arr = (int *)malloc(size * sizeof(int));

    // Check if the memory allocation was successful
    if (*arr == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);  // Exit if allocation fails
    }

    // Optionally initialize the array
    for (int i = 0; i < size; i++) {
        (*arr)[i] = 0;  // Initialize all elements to 0
    }
}

// Function to free the dynamically allocated memory
void freeArray(int *arr) {
    free(arr);  // Free the allocated memory
}

int main() {
    int *arr = NULL;
    int size;

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

    // Allocate memory for the array
    allocateArray(&arr, size);

    // Display the allocated array (initially filled with 0)
    printf("Array elements after allocation and initialization:\n");
```

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

    // Free the allocated memory
    freeArray(arr);

    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>

// Function to modify the string using a double pointer
void modifyString(char **str) {
    // Dynamically allocate memory for a new string
    *str = (char *)malloc(50 * sizeof(char));  // Allocating memory for 50 characters

    // Check if memory allocation was successful
    if (*str == NULL) {
        printf("Memory allocation failed\n");
        return;
    }

    // Modify the string
    strcpy(*str, "This is the modified string!");

    // Print the modified string
    printf("Modified string: %s\n", *str);
}

int main() {
    // Declare a string pointer
    char *myString = NULL;

    // Call modifyString to change the original string
    modifyString(&myString);

    // Print the original string (it now points to the new memory location)
    printf("Original string after modification: %s\n", myString);

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

    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;          // Declare an integer variable
    int *ptr;              // Declare a pointer to an integer
    int **ptr2;            // Declare a pointer to a pointer to an integer

    ptr = &num;            // Point 'ptr' to the address of 'num'
    ptr2 = &ptr;           // Point 'ptr2' to the address of 'ptr'

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

    // Using ptr2 (pointer to pointer) to modify the value of num
    **ptr2 = 20;

    printf("Modified value of num using pointer to pointer: %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.

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

// Function to create a 2D array dynamically using double pointer
int** create2DArray(int rows, int cols) {
    // Step 1: Allocate memory for an array of row pointers
    int **array = (int **)malloc(rows * sizeof(int *));

    // Check if memory allocation was successful
    if (array == NULL) {
        printf("Memory allocation failed for rows.\n");
        return NULL;
    }

    // Step 2: Allocate memory for each row (array of integers)
    for (int i = 0; i < rows; i++) {
        array[i] = (int *)malloc(cols * sizeof(int));

        // Check if memory allocation was successful
        if (array[i] == NULL) {
            printf("Memory allocation failed for columns in row %d.\n", i);
            return NULL;
        }
    }

    // Return the pointer to the 2D array
    return array;
}

// Function to free the dynamically allocated memory
```

```
void free2DArray(int **array, int rows) {
    for (int i = 0; i < rows; i++) {
        free(array[i]); // Free each row
    }
    free(array); // Free the array of row pointers
}

int main() {
    int rows = 3;
    int cols = 4;

    // Create a 2D array
    int **array = create2DArray(rows, cols);

    // Check if memory allocation succeeded
    if (array == NULL) {
        return -1; // Exit if memory allocation failed
    }

    // Example of using the 2D array
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            array[i][j] = (i + 1) * (j + 1); // Assign some values
        }
    }

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

    // Free the memory used by the 2D array
    free2DArray(array, rows);

    return 0;
}
```

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.

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

void free2DArray(int **arr, int rows) {
    // Step 1: Free each row
    for (int i = 0; i < rows; i++) {
        free(arr[i]);  // Free the memory allocated for each row
    }

    // Step 2: Free the array of pointers
    free(arr);  // Free the memory allocated for the array of pointers
```

```
}

int main() {
    int rows = 3;
    int cols = 4;

    // Dynamically allocate memory for a 2D array
    int **arr = (int **)malloc(rows * sizeof(int *));
    for (int i = 0; i < rows; i++) {
        arr[i] = (int *)malloc(cols * sizeof(int));
    }

    // Fill the array with some values
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            arr[i][j] = i * cols + j;
        }
    }

    // Print the array
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("%d ", arr[i][j]);
        }
        printf("\n");
    }

    // Free the 2D array memory
    free2DArray(arr, rows);

    return 0;
}
```

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>

// Function to set the pointer to dynamically allocated memory
void setPointer(int **ptr) {
    // Dynamically allocate memory for an integer
    *ptr = (int*) malloc(sizeof(int));  // Allocate memory for one integer
    if (*ptr == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);  // Exit if memory allocation fails
    }

    // Set the allocated integer to a specific value (optional)
    **ptr = 42;  // Assign a value to the dynamically allocated integer
}

int main() {
    int *ptr = NULL;  // Initialize pointer to NULL
```

```
    // Pass the address of ptr to the function
    setPointer(&ptr);

    // Print the value pointed by ptr
    printf("Value of the dynamically allocated integer: %d\n", *ptr);

    // Free the allocated memory
    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>

void allocateStringArray(char ***arr, int n) {
    // Allocate memory for n pointers to char (for the strings)
    *arr = (char **)malloc(n * sizeof(char *));
    if (*arr == NULL) {
        printf("Memory allocation failed for the array of strings.\n");
        exit(1); // Exit if memory allocation fails
    }

    // Allocate memory for each string (you can specify a max length for each string, here we use 100)
    for (int i = 0; i < n; i++) {
        (*arr)[i] = (char *)malloc(100 * sizeof(char)); // assuming each string can be 100 chars long
        if ((*arr)[i] == NULL) {
            printf("Memory allocation failed for string %d.\n", i);
            exit(1); // Exit if memory allocation fails
        }
    }

    printf("Memory allocation successful for the array of %d strings.\n", n);
}

int main() {
    char **arr;
    int n = 5;  // Number of strings

    // Allocate the array of strings
    allocateStringArray(&arr, n);

    // Example of setting strings
    for (int i = 0; i < n; i++) {
        snprintf(arr[i], 100, "String %d", i + 1);
        printf("arr[%d]: %s\n", i, arr[i]);
    }

    // Free allocated memory
    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.

```
#include <stdio.h>

void modifyStringArray(char **arr, int n) {
    // Loop through each string in the array
    for (int i = 0; i < n; i++) {
        char *str = arr[i];

        // Loop through each character of the string and modify it
        for (int j = 0; str[j] != '\0'; j++) {
            // If the character is a lowercase letter, convert it to uppercase
            if (str[j] >= 'a' && str[j] <= 'z') {
                str[j] = str[j] - ('a' - 'A');
            }
        }
    }
}

int main() {
    // Sample array of strings
    char *arr[] = {"hello", "world", "example", "string"};
    int n = sizeof(arr) / sizeof(arr[0]);

    // Print original array
    printf("Original array:\n");
    for (int i = 0; i < n; i++) {
        printf("%s\n", arr[i]);
    }

    // Modify the strings using the function
    modifyStringArray(arr, n);

    // Print modified array
    printf("\nModified array:\n");
    for (int i = 0; i < n; i++) {
        printf("%s\n", arr[i]);
    }

    return 0;
}
```

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>

// Function definition for add
int add(int a, int b) {
    return a + b;
}

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

    // Assign the address of the add function to the function pointer
    functionPointer = add;

    // Call the add function using the function pointer
    int result = functionPointer(5, 3);

    // Print the result
    printf("The result of adding 5 and 3 is: %d\n", result);

    return 0;
}
```

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>

// Function to perform an operation on two integers and print the result
void performOperation(int (*operation)(int, int), int a, int b) {
    // Apply the operation to the two integers and print the result
    int result = operation(a, b);
    printf("Result: %d\n", result);
}

// Example operation functions
int add(int x, int y) {
    return x + y;
}

int multiply(int x, int y) {
    return x * y;
}

int subtract(int x, int y) {
    return x - y;
}

int main() {
    int a = 5, b = 3;
```

```
    // Pass different operations to performOperation
    performOperation(add, a, b);       // Output will be 8
    performOperation(multiply, a, b);  // Output will be 15
    performOperation(subtract, a, b);  // Output will be 2

    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>

// Function that returns a pointer to the larger of two integers
int* max(int *a, int *b) {
    if (*a > *b) {
        return a;  // Return pointer to the larger number
    } else {
        return b;  // Return pointer to the larger number
    }
}

int main() {
    int x = 10, y = 20;

    // Define a function pointer and assign it to the max function
    int* (*max_ptr)(int*, int*) = max;

    // Call the max function through the function pointer
    int* result = max_ptr(&x, &y);

    // Print the larger number
    printf("The larger number 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 prototypes
int add(int a, int b);
int multiply(int a, int b);

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

    int num1, num2, choice;

    // Get user input for numbers
```

```
    printf("Enter two integers: ");
    scanf("%d %d", &num1, &num2);

    // Ask the user which operation to perform
    printf("Choose operation:\n");
    printf("1. Add\n");
    printf("2. Multiply\n");
    printf("Enter your choice (1 or 2): ");
    scanf("%d", &choice);

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

    // Use the function pointer to call the selected function
    int result = operation(num1, num2);

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

    return 0;
}

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

// Function to multiply two integers
int multiply(int a, int b) {
    return a * b;
}
```

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 prototypes for arithmetic operations
int add(int a, int b);
int subtract(int a, int b);
int multiply(int a, int b);
float divide(int a, int b);

int main() {
    // Array of function pointers
    // Note that the divide function returns a float, so we declare the pointer as a function returning float.
    void* (*operations[])(int, int) = {add, subtract, multiply, divide};
```

```
    int choice, a, b;

    // Display menu for user input
    printf("Select operation:\n");
    printf("1. Addition\n");
    printf("2. Subtraction\n");
    printf("3. Multiplication\n");
    printf("4. Division\n");
    printf("Enter choice (1-4): ");
    scanf("%d", &choice);

    if (choice < 1 || choice > 4) {
        printf("Invalid choice!\n");
        return 1;
    }

    // Get the operands
    printf("Enter two integers: ");
    scanf("%d %d", &a, &b);

    // Execute the chosen operation using the function pointer array
    if (choice == 4 && b == 0) {
        printf("Error: Division by zero is not allowed.\n");
    } else {
        if (choice == 1) {
            printf("Result: %d\n", add(a, b));
        } else if (choice == 2) {
            printf("Result: %d\n", subtract(a, b));
        } else if (choice == 3) {
            printf("Result: %d\n", multiply(a, b));
        } else if (choice == 4) {
            printf("Result: %.2f\n", divide(a, b));
        }
    }

    return 0;
}

// Function definitions for 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;
}

float divide(int a, int b) {
    return (float) a / b;
}
```

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>

// Function prototype for the comparison function
int compare_ascending(int a, int b);
int compare_descending(int a, int b);

// Function to perform sorting
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) {
                // Swap if elements are in the wrong order
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
    }
}

// Ascending comparison function
int compare_ascending(int a, int b) {
    return a - b;
}

// Descending comparison function
int compare_descending(int a, int b) {
    return b - a;
}

// Function to print the array
void print_array(int *arr, int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {34, 23, 12, 45, 9, 18};
    int size = sizeof(arr) / sizeof(arr[0]);

    printf("Original array:\n");
    print_array(arr, size);

    // Sorting in ascending order
    sort(arr, size, compare_ascending);
    printf("Sorted in ascending order:\n");
    print_array(arr, size);
```

```
    // Sorting in descending order
    sort(arr, size, compare_descending);
    printf("Sorted in descending order:\n");
    print_array(arr, size);

    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).

```
#include <stdio.h>

// Callback function to calculate the square of a number
int square(int x) {
    return x * x;
}

// Callback function to calculate the cube of a number
int cube(int x) {
    return x * x * x;
}

// Function that takes an integer and a callback function as parameters
void execute(int x, int (*callback)(int)) {
    // Apply the callback function to x and print the result
    printf("Result: %d\n", callback(x));
}

int main() {
    int number = 5;

    // Demonstrate with the square callback function
    printf("Applying square callback to %d:\n", number);
    execute(number, square);

    // Demonstrate with the cube callback function
    printf("Applying cube callback to %d:\n", number);
    execute(number, cube);

    return 0;
}
```

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.

```
#include <stdio.h>

// Declare the function prototypes
void option1();
void option2();
void option3();
```

```
void exitProgram();

// Define an array of function pointers
void (*menuFunctions[])() = {option1, option2, option3, exitProgram};

// Function definitions
void option1() {
    printf("You selected Option 1\n");
}

void option2() {
    printf("You selected Option 2\n");
}

void option3() {
    printf("You selected Option 3\n");
}

void exitProgram() {
    printf("Exiting the program.\n");
}

int main() {
    int choice;

    // Menu system
    while(1) {
        printf("\nMenu:\n");
        printf("1. Option 1\n");
        printf("2. Option 2\n");
        printf("3. Option 3\n");
        printf("4. Exit\n");

        printf("Enter your choice: ");
        scanf("%d", &choice);

        // Check if the choice is within the valid range
        if (choice >= 1 && choice <= 4) {
            // Call the corresponding function based on the user input
            menuFunctions[choice - 1]();

            // If exit option is selected, break the loop
            if (choice == 4) {
                break;
            }
        } else {
            printf("Invalid choice, please try again.\n");
        }
    }

    return 0;
}
```

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.

```
#include <stdio.h>

// Function declarations
int add(int a, int b);
int subtract(int a, int b);
int multiply(int a, int b);
int divide(int a, int b);

int main() {
    char operator;
    int num1, num2;

    // Array of function pointers
    int (*operation)(int, int);

    // Input: user chooses an operation and inputs two numbers
    printf("Enter an operator (+, -, *, /): ");
    scanf(" %c", &operator);  // Note: Space before %c to capture any extra newline characters

    printf("Enter two integers: ");
    scanf("%d %d", &num1, &num2);

    // Select the function based on the operator using the function pointer
    switch (operator) {
        case '+':
            operation = add;
            break;
        case '-':
            operation = subtract;
            break;
        case '*':
            operation = multiply;
            break;
        case '/':
            if (num2 == 0) {
                printf("Error: Division by zero is not allowed.\n");
                return 1;
            }
            operation = divide;
            break;
        default:
            printf("Invalid operator.\n");
            return 1;
    }

    // Call the function using the pointer and display the result
    int result = operation(num1, num2);
    printf("Result: %d\n", result);

    return 0;
}

// Function definitions
```

```
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;
}
```

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.

```
#include <stdio.h>

// Declare the state functions
void redState();
void greenState();
void yellowState();

// Function pointers to handle state transitions
void (*currentState)();

// State functions
void redState() {
    printf("Traffic Light: RED\n");
    // Simulate a transition delay by printing and using a loop
    for (int i = 0; i < 2; i++) {
        printf(".");
        for (volatile int j = 0; j < 100000000; j++); // Busy-wait to simulate time delay
    }
    printf("\nTransitioning to GREEN...\n");
    currentState = greenState;  // Transition to green state
}

void greenState() {
    printf("Traffic Light: GREEN\n");
    // Simulate a transition delay by printing and using a loop
    for (int i = 0; i < 2; i++) {
        printf(".");
        for (volatile int j = 0; j < 100000000; j++); // Busy-wait to simulate time delay
    }
    printf("\nTransitioning to YELLOW...\n");
    currentState = yellowState;  // Transition to yellow state
}

void yellowState() {
```

```
    printf("Traffic Light: YELLOW\n");
    // Simulate a transition delay by printing and using a loop
    for (int i = 0; i < 1; i++) {
        printf(".");
        for (volatile int j = 0; j < 100000000; j++); // Busy-wait to simulate time delay
    }
    printf("\nTransitioning to RED...\n");
    currentState = redState;  // Transition to red state
}

int main() {
    // Initialize the current state as red
    currentState = redState;

    // Run the state machine in an infinite loop
    while (1) {
        currentState();  // Call the current state function
    }

    return 0;
}
```