Day 11 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) {

    char *start = str;

    char *end = str + strlen(str) - 1;

    char temp;

    while (start < end) {

        temp = *start;

        *start = *end;

        *end = temp;

        start++;

        end--;

    }

}


int main() {

    char str[] = "hello";

    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 *dest, const char *src) {
    while (*dest) dest++;
    while ((*dest++ = *src++));
}

int main() {
    char dest[50] = "Hello, ";
    char src[] = "World!";
    concatenateStrings(dest, src);
    printf("Concatenated String: %s\n", dest);
    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;
    while (*ptr) ptr++;
    return ptr - str;
}
```

```
int main() {
    char str[] = "Hello";
    printf("String Length: %d\n", stringLength(str));
    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) {
    while (*str1 && (*str1 == *str2)) {
        str1++;
        str2++;
    }
    return *(unsigned char *)str1 - *(unsigned char *)str2;
}

int main() {
    char str1[] = "abc";
    char str2[] = "abd";
    printf("Comparison Result: %d\n", compareStrings(str1, str2));
    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>
#include <string.h>

char* findSubstring(const char *str, const char *sub) {
    const char *p1 = str, *p2 = sub;
    while (*str) {
        p1 = str;
        p2 = sub;
        while (*p1 && *p2 && (*p1 == *p2)) {
            p1++;
            p2++;
        }
        if (!*p2) return (char *)str;
        str++;
    }
    return NULL;
}

int main() {
    char str[] = "hello world";
    char sub[] = "world";
    char *result = findSubstring(str, sub);
    if (result)
        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) {
    while (*str) {
        if (*str == oldChar) *str = newChar;
        str++;
    }
}

int main() {
    char str[] = "hello";
    replaceChar(str, 'l', 'x');
    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) {
    while ((*dest++ = *src++));
}

int main() {
    char src[] = "Hello, World!";
    char dest[50];
    copyString(dest, src);
    printf("Copied String: %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;
    while (*str) {
        char c = *str | 32; // Convert to lowercase
        if (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u') count++;
        str++;
    }
    return count;
}

int main() {
    char str[] = "Hello World";
    printf("Number of Vowels: %d\n", countVowels(str));
    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) {
    const char *start = str, *end = str + strlen(str) - 1;
    while (start < end) {
        if (*start != *end) return 0;
        start++;
        end--;
    }
    return 1;
}

int main() {
    char str[] = "madam";
    if (isPalindrome(str))
        printf("The string is a palindrome.\n");
    else
        printf("The string is not a palindrome.\n");
    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 *)) {

   char *token = strtok(str, delim);

   while (token) {

      processToken(token);

      token = strtok(NULL, delim);

   }

}


void printToken(const char *token) {

   printf("Token: %s\n", token);

}


int main() {

   char str[] = "Hello,World,Example";

   tokenizeString(str, ",", printToken);

   return 0;

}
```

Dynamic Memory Allocation 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;

    printf("Enter the size of the array: ");

    scanf("%d", &n);


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


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

        arr[i] = i + 1;

    }


    printf("Array: ");

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

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

    }


    free(arr);

    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 *str;
    int size;

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

    str = (char *)malloc((size + 1) * sizeof(char));

    printf("Enter the string: ");
    scanf(" %[^\n]", str);

    printf("You entered: %s\n", str);

    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;
    printf("Enter the initial size of the array: ");
    scanf("%d", &n);
    int *arr = (int *)malloc(n * sizeof(int));

    for (int i = 0; i < n; i++) {
        arr[i] = i + 1;
    }

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

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

    for (int i = n; i < 2 * n; i++) {
        arr[i] = i + 1;
    }

    printf("Resized Array: ");
    for (int i = 0; i < 2 * n; i++) {
        printf("%d ", arr[i]);
    }

    free(arr);
    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 allocateAndFreeMatrix(int m, int n) {
    int **matrix = (int **)malloc(m * sizeof(int *));
    if (!matrix) {
        printf("Memory allocation failed.\n");
        return;
    }

    for (int i = 0; i < m; i++) {
        matrix[i] = (int *)malloc(n * sizeof(int));
        if (!matrix[i]) {
            printf("Memory allocation failed for row %d.\n", i);
            return;
        }
    }

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            matrix[i][j] = i * n + j + 1;
        }
    }

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

    for (int i = 0; i < m; i++) {
        free(matrix[i]);
    }
    free(matrix);
}

int main() {
    int m = 3, n = 4;
    allocateAndFreeMatrix(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>

char *concatenateStrings(const char *str1, const char *str2) {
    char *result = (char *)malloc((strlen(str1) + strlen(str2) + 1) * sizeof(char));
    if (!result) {
        printf("Memory allocation failed.\n");
        return NULL;
    }
    strcpy(result, str1);
    strcat(result, str2);
    return result;
}

int main() {
    char str1[] = "Hello, ";
    char str2[] = "World!";

    char *result = concatenateStrings(str1, str2);
    if (result) {
        printf("Concatenated String: %s\n", result);
        free(result);
    }
    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>

typedef struct {
    char name[50];
    int age;
    float grade;
} Student;

int main() {
    Student *student = (Student *)malloc(sizeof(Student));
    if (!student) {
        printf("Memory allocation failed.\n");
        return 1;
    }

    printf("Enter name: ");
    scanf(" %[^\n]", student->name);
    printf("Enter age: ");
    scanf("%d", &student->age);
    printf("Enter grade: ");
    scanf("%f", &student->grade);

    printf("Student Details:\nName: %s\nAge: %d\nGrade: %.2f\n", student->name, student->age, student->grade);

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

7. 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 = 5;
    int **arr = (int **)malloc(n * sizeof(int *));
    if (!arr) {
        printf("Memory allocation failed.\n");
        return 1;
    }

    for (int i = 0; i < n; i++) {
        arr[i] = (int *)malloc(sizeof(int));
        if (!arr[i]) {
            printf("Memory allocation failed for element %d.\n", i);
            return 1;
        }
        *arr[i] = i + 1;
    }

    printf("Array values: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", *arr[i]);
        free(arr[i]);
    }
    printf("\n");

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

8. 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 = 3, y = 3, z = 3;

    int ***arr = (int ***)malloc(x * sizeof(int **));

    if (!arr) {

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

        return 1;

    }


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

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

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

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

        }

    }


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

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

            for (int k = 0; k < z; k++) {

                arr[i][j][k] = i * y * z + j * z + k + 1;

            }

        }

    }
    printf("3D Array:\n");
    for (int i = 0; i < x; i++) {
        for (int j = 0; j < y; j++) {
            for (int k = 0; k < z; k++) {
                printf("%d ", arr[i][j][k]);
            }
            printf("\n");
        }
        printf("\n");
    }

    for (int i = 0; i < x; i++) {
        for (int j = 0; j < y; j++) {
            free(arr[i][j]);
        }
        free(arr[i]);
    }
    free(arr);

    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>
```

```
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 Swap: *ptr1 = %d, *ptr2 = %d\n", *ptr1, *ptr2);
    swap(&ptr1, &ptr2);
    printf("After Swap: *ptr1 = %d, *ptr2 = %d\n", *ptr1, *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>

void allocateArray(int **arr, int size) {
    *arr = (int *)malloc(size * sizeof(int));
    if (!*arr) {
        printf("Memory allocation failed.\n");
        return;
    }
    for (int i = 0; i < size; i++) {
        (*arr)[i] = i + 1;
    }
}

int main() {
    int *arr;
    int size = 5;

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

    free(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>
```

```
void modifyString(char **str) {
    *str = (char *)malloc(50 * sizeof(char));
    if (!*str) {
        printf("Memory allocation failed.\n");
        return;
    }
    strcpy(*str, "Modified String");
}

int main() {
    char *str = NULL;

    modifyString(&str);
    if (str) {
        printf("String: %s\n", str);
        free(str);
    }

    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 value = 42;
    int *ptr = &value;
    int **doublePtr = &ptr;

    printf("Value: %d\n", **doublePtr);
    **doublePtr = 100;
    printf("Modified Value: %d\n", value);

    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>

int** create2DArray(int rows, int cols) {
    int **arr = (int **)malloc(rows * sizeof(int *));
    for (int i = 0; i < rows; i++) {
        arr[i] = (int *)malloc(cols * sizeof(int));
    }

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            arr[i][j] = i * cols + j + 1;
        }
    }
    return arr;
}


int main() {
    int rows = 3, cols = 4;
    int **arr = create2DArray(rows, cols);


    printf("2D Array:\n");
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("%d ", arr[i][j]);
        }
        printf("\n");
    }


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


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

int main() {
    int rows = 3, cols = 4;
    int **arr = (int **)malloc(rows * sizeof(int *));
    for (int i = 0; i < rows; i++) {
        arr[i] = (int *)malloc(cols * sizeof(int));
    }

    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>

void setPointer(int **ptr) {
    *ptr = (int *)malloc(sizeof(int));
    if (!*ptr) {
        printf("Memory allocation failed.\n");
        return;
    }
    **ptr = 42;
}

int main() {
    int *ptr = NULL;

    setPointer(&ptr);
    if (ptr) {
        printf("Value: %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 *));
    for (int i = 0; i < n; i++) {
        (*arr)[i] = (char *)malloc(50 * sizeof(char));
        sprintf((*arr)[i], "String %d", i + 1);
    }
}

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

    allocateStringArray(&arr, n);
    for (int i = 0; i < n; i++) {
        printf("%s\n", arr[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>
#include <stdlib.h>
#include <string.h>

void modifyStringArray(char **arr, int n) {
    for (int i = 0; i < n; i++) {
        strcat(arr[i], " - Modified");
    }
}

int main() {
    int n = 3;
    char **arr = (char **)malloc(n * sizeof(char *));
    for (int i = 0; i < n; i++) {
        arr[i] = (char *)malloc(50 * sizeof(char));
        sprintf(arr[i], "String %d", i + 1);
    }

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

    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>
```

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

int main() {
    int (*funcPtr)(int, int) = add;
    printf("Result: %d\n", funcPtr(5, 3));
    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>

void performOperation(int (*operation)(int, int), int a, int b) {
    printf("Result: %d\n", operation(a, b));
}

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

int main() {
    performOperation(add, 7, 4);
    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;
    int* (*funcPtr)(int*, int*) = max;


    int *result = funcPtr(&x, &y);
    printf("Max: %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>


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

```
int multiply(int a, int b) {
    return a * b;
}

int main() {
    int (*operation)(int, int);
    char choice;

    printf("Enter operation (a for add, m for multiply): ");
    scanf(" %c", &choice);

    if (choice == 'a') {
        operation = add;
    } else if (choice == 'm') {
        operation = multiply;
    } else {
        printf("Invalid choice.\n");
        return 1;
    }

    printf("Result: %d\n", operation(4, 5));
    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>

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 b != 0 ? a / b : 0; }

int main() {
    int (*operations[4])(int, int) = {add, subtract, multiply, divide};
    int choice, a, b;

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

    if (choice < 0 || choice > 3) {
        printf("Invalid choice.\n");
        return 1;
    }

    printf("Enter two numbers: ");
    scanf("%d %d", &a, &b);

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


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])) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}


int ascending(int a, int b) { return a > b; }
int descending(int a, int b) { return a < b; }


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


    printf("Ascending Order:\n");
    sort(arr, size, ascending);
    for (int i = 0; i < size; i++) printf("%d ", arr[i]);
    printf("\n");


    printf("Descending Order:\n");
    sort(arr, size, descending);
    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).

```
#include <stdio.h>

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

int square(int x) { return x * x; }
int cube(int x) { return x * x * x; }

int main() {
    execute(3, square);
    execute(3, 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>


void option1() { printf("Option 1 selected.\n"); }
void option2() { printf("Option 2 selected.\n"); }
void option3() { printf("Option 3 selected.\n"); }


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


   printf("Menu:\n1. Option 1\n2. Option 2\n3. Option 3\n");
   printf("Enter your choice: ");
   scanf("%d", &choice);


   if (choice < 1 || choice > 3) {
      printf("Invalid choice.\n");
      return 1;
   }


   menu[choice - 1]();
   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>

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 b != 0 ? a / b : 0; }

int main() {
    int (*operation)(int, int);
    char op;
    int a, b;

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

    if (op == '+') operation = add;
    else if (op == '-') operation = subtract;
    else if (op == '*') operation = multiply;
    else if (op == '/') operation = divide;
    else {
        printf("Invalid operation.\n");
        return 1;
    }

    printf("Enter two numbers: ");
    scanf("%d %d", &a, &b);

    printf("Result: %d\n", operation(a, b));
    return 0;
}
```

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>

void red() { printf("State: Red Light\n"); }
void yellow() { printf("State: Yellow Light\n"); }
void green() { printf("State: Green Light\n"); }

int main() {
    void (*states[])(void) = {red, yellow, green};
    int state = 0;

    while (1) {
        states[state]();
        printf("Press 1 for next state, 0 to exit: ");
        int choice;
        scanf("%d", &choice);

        if (choice == 0) break;
        state = (state + 1) % 3;
    }

    return 0;
}
```