**Variable, Static, Const, and Switch Case**  **20-01-2025**

1. Write a C program that declares a static variable and a const variable within a function. The program should increment the static variable each time the function is called and use a switch case to check the value of the const variable. The function should handle at least three different cases for the const variable and demonstrate the persistence of the static variable across multiple calls.
2. // program to demonstrate static function
3. #include <stdio.h>
4. void displayStatic() {
5. static int staticVar = 0;
6. const int constVar = 2;
7. // Increment the static variable
8. staticVar++;
9. printf("Static variable value: %d\n", staticVar);
10. switch (constVar) {
11. case 1:
12. printf("Const var 1\n");
13. break;
14. case 2:
15. printf("Const var 2\n");
16. break;
17. case 3:
18. printf("Const var is 3\n");
19. break;
20. default:
21. printf("Const not handled\n");
22. break;
23. }
24. }
25. int main() {
26. printf("First call:\n");
27. displayStatic();
28. printf("\nSecond call:\n");
29. displayStatic();
30. printf("\nThird call:\n");
31. displayStatic();
32. return 0;
33. }
34. Create a C program where a static variable is used to keep track of the number of times a function has been called. Implement a switch case to print a different message based on the number of times the function has been invoked (e.g., first call, second call, more than two calls). Ensure that a const variable is used to define a maximum call limit and terminate further calls once the limit is reached.
35. //program to demonstrate maximum call reach and terminate.
36. #include <stdio.h>
37. void constCall() {
38. static int callCount = 0;
39. const int maxCalls = 3;
40. // Increment call count
41. callCount++;
42. if (callCount > maxCalls) {
43. printf("Maximum call limit of %d reached.\n", maxCalls);
44. return;
45. }
46. switch (callCount) {
47. case 1:
48. printf(" first call .\n");
49. break;
50. case 2:
51. printf("second call .\n");
52. break;
53. default:
54. printf("This is call number %d.\n", callCount);
55. break;
56. }
57. }
58. int main() {
59. printf("Calling function:\n");
60. constCall();
61. constCall();
62. constCall();
63. constCall();
64. constCall();
65. return 0;
66. }
67. Develop a C program that utilizes a static array inside a function to store values across multiple calls. Use a const variable to define the size of the array. Implement a switch case to perform different operations on the array elements (e.g., add, subtract, multiply) based on user input. Ensure the array values persist between function calls.
68. // program to demonstrate implements a static array
69. #include <stdio.h>
70. void arithmeticArr(int operation, int value) {
71. static int arr[5] = {0};
72. const int size = 5;
73. int i;
74. if (operation == 1) {
75. for (i = 0; i < size; i++) {
76. arr[i] = arr[i] + value;
77. }
78. printf("Add %d to each element.\n", value);
79. }
80. else if (operation == 2) {
81. for (i = 0; i < size; i++) {
82. arr[i] = arr[i] - value;
83. }
84. printf("Subtract %d each element.\n", value);
85. }
86. else if (operation == 3) {
87. for (i = 0; i < size; i++) {
88. arr[i] = arr[i] \* value;
89. }
90. printf("Multiplied by %d.\n", value);
91. }
92. else {
93. printf("Not valid Operation.\n");
94. return;
95. }
96. // Print the array
97. printf("Array: ");
98. for (i = 0; i < size; i++) {
99. printf("%d ", arr[i]);
100. }
101. printf("\n");
102. }
103. int main() {
104. arithmeticArr(1, 10);
105. arithmeticArr(2, 5);
106. arithmeticArr(3, 2);
107. return 0;
108. }
109. Write a program that demonstrates the difference between const and static variables. Use a static variable to count the number of times a specific switch case is executed, and a const variable to define a threshold value for triggering a specific case. The program should execute different actions based on the value of the static counter compared to the const threshold.
110. //program to demonstrate difference between const and static
111. #include <stdio.h>
112. void ConstandSatic() {
113. static int counter = 0;
114. const int threshold = 3;
115. counter++;
116. switch (counter) {
117. case 1:
118. printf("First Counter is 1.\n");
119. break;
120. case 2:
121. printf("Second Counter is 2.\n");
122. break;
123. case 3:
124. printf("Threshold reached %d,\n", counter);
125. break;
126. default:
127. printf("Counter is %d, beyond threshold. .\n", counter);
128. break;
129. }
130. printf("counter value: %d\n", counter);
131. }
132. int main() {
133. for (int i = 0; i < 5; i++) {
134. ConstandSatic();
135. }
136. return 0;
137. }
138. Create a C program with a static counter and a const limit. The program should include a switch case to print different messages based on the value of the counter. After every 5 calls, reset the counter using the const limit. The program should also demonstrate the immutability of the const variable by attempting to modify it and showing the compilation error.
139. //program to demonstrate the static and constant and resting counter.
140. #include <stdio.h>
141. void StaticAndConst() {
142. static int counter = 0;
143. const int limit = 5;
144. counter++;
145. switch (counter) {
146. case 1:
147. printf("Counter is at 1.\n");
148. break;
149. case 2:
150. printf("Counter is at 2.\n");
151. break;
152. case 3:
153. printf("Counter is at 3.\n");
154. break;
155. case 4:
156. printf("Counter is at 4.\n");
157. break;
158. case 5:
159. printf("Counter is at 5. Reset counter.\n");
160. counter = 0;
161. break;
162. default:
163. printf("Counter is reset.\n");
164. break;
165. }
166. printf("counter value: %d\n", counter);
167. }
168. int main() {
169. for (int i = 0; i < 7; i++) {
170. StaticAndConst();
171. }
172. return 0;
173. }

Looping Statements, Pointers, Const with Pointers, Functions

* Write a C program that demonstrates the use of both single and double pointers. Implement a function that uses a for loop to initialize an array and a second function that modifies the array elements using a double pointer. Use the const keyword to prevent modification of the array elements in one of the functions.
* // Program to demonstrate intialize array using single pointer and modify the array using the double pointer.
* #include <stdio.h>
* void initializeArray(const int \*arr, int size) {
* printf("Initial array values:\n");
* for (int i = 0; i < size; i++) {
* printf("%d ", \*(arr + i));
* }
* printf("\n");
* }
* void modifyArray(int \*\*arr, int size) {
* for (int i = 0; i < size; i++) {
* \*(\*arr + i) \*= 2;
* }
* }
* int main() {
* int array[5] = {1, 2, 3, 4, 5};
* int \*ptr = array;
* initializeArray(ptr, 5);
* modifyArray(&ptr, 5);
* printf("Modified array values:\n");
* for (int i = 0; i < 5; i++) {
* printf("%d ", array[i]);
* }
* printf("\n");
* return 0;
* }
* Develop a program that reads a matrix from the user and uses a function to transpose the matrix. The function should use a double pointer to manipulate the matrix. Demonstrate both call by value and call by reference in the program. Use a const pointer to ensure the original matrix is not modified during the transpose operation.
* #include <stdio.h>
* #define MAX\_SIZE 10
* // Transpose the matrix
* void transpose(int matrix[MAX\_SIZE][MAX\_SIZE], int rows, int cols) {
* int temp;
* for (int i = 0; i < rows; i++) {
* for (int j = i + 1; j < cols; j++) {
* temp = matrix[i][j];
* matrix[i][j] = matrix[j][i];
* matrix[j][i] = temp;
* }
* }
* }
* // Print the matrix
* void printMatrix(int matrix[MAX\_SIZE][MAX\_SIZE], int rows, int cols) {
* for (int i = 0; i < rows; i++) {
* for (int j = 0; j < cols; j++) {
* printf("%d ", matrix[i][j]);
* }
* printf("\n");
* }
* }
* int main() {
* int matrix[MAX\_SIZE][MAX\_SIZE], rows, cols;
* // Input matrix size and elements
* printf("Enter rows and columns: ");
* scanf("%d %d", &rows, &cols);
* printf("Enter matrix elements:\n");
* for (int i = 0; i < rows; i++) {
* for (int j = 0; j < cols; j++) {
* scanf("%d", &matrix[i][j]);
* }
* }
* // Print original matrix
* printf("Original Matrix:\n");
* printMatrix(matrix, rows, cols);
* // Transpose the matrix
* transpose(matrix, rows, cols);
* // Print transposed matrix
* printf("Transposed Matrix:\n");
* printMatrix(matrix, rows, cols);
* return 0;
* }
* Question 3: Create a C program that uses a single pointer to dynamically allocate memory for an array. Write a function to initialize the array using a while loop, and another function to print the array. Use a const pointer to ensure the printing function does not modify the array.
* //Print array and dynamic memory Allocation
* #include <stdio.h>
* #include <stdlib.h>
* void initializeArray(int \*arr, int size) {
* int i = 0;
* while (i < size) {
* arr[i] = i + 1;
* i++;
* }
* }
* void printArray(const int \*arr, int size) {
* int i = 0;
* while (i < size) {
* printf("%d ", arr[i]);
* i++;
* }
* printf("\n");
* }
* int main() {
* int \*arr;
* int size;
* printf("Enter the size of the array: ");
* scanf("%d", &size);
* // Dynamically allocate memory for the array
* arr = (int \*)malloc(size \* sizeof(int));
* if (arr == NULL) {
* printf("Memory allocation failed!\n");
* return 1;
* }
* initializeArray(arr, size);
* printf("Array elements: ");
* printArray(arr, size);
* free(arr);
* return 0;
* }
* Question 4: Write a program that demonstrates the use of double pointers to swap two arrays. Implement functions using both call by value and call by reference. Use a for loop to print the swapped arrays and apply the const keyword appropriately to ensure no modification occurs in certain operations.
* #include <stdio.h>
* // Swap arrays using double pointers (call by reference)
* void swapByReference(int \*arr1, int \*arr2, int size) {
* for (int i = 0; i < size; i++) {
* int temp = arr1[i];
* arr1[i] = arr2[i];
* arr2[i] = temp;
* }
* }
* // Swap arrays (call by value)
* void swapByValue(int arr1[], int arr2[], int size) {
* for (int i = 0; i < size; i++) {
* int temp = arr1[i];
* arr1[i] = arr2[i];
* arr2[i] = temp;
* }
* }
* void printArray(const int \*arr, int size) {
* for (int i = 0; i < size; i++) {
* printf("%d ", arr[i]);
* }
* printf("\n");
* }
* int main() {
* int arr1[] = {1, 2, 3}, arr2[] = {4, 5, 6};
* int size = 3;
* printArray(arr1, size);
* printArray(arr2, size);
* // Swap arrays by reference
* swapByReference(arr1, arr2, size);
* printArray(arr1, size);
* printArray(arr2, size);
* // Swap arrays back by value
* swapByValue(arr1, arr2, size);
* printArray(arr1, size);
* printArray(arr2, size);
* return 0;
* }
* Question 5: Develop a C program that demonstrates the application of const with pointers. Create a function to read a string from the user and another function to count the frequency of each character using a do-while loop. Use a const pointer to ensure the original string is not modified during character frequency calculation.
* #include <stdio.h>
* #define MAX\_LENGTH 100
* // Function to read a string from the user
* void readString(char \*str) {
* printf("Enter a string: ");
* scanf("%s", str);
* }
* // Function to count the frequency of each character
* void countFrequency(const char \*str) {
* int freq[256] = {0};
* int i = 0;
* do {
* freq[(unsigned char)str[i]]++;
* i++;
* } while (str[i] != '\0');
* // Print frequency of each character
* printf("\nCharacter Frequencies:\n");
* for (int i = 0; i < 256; i++) {
* if (freq[i] > 0) {
* printf("'%c' : %d\n", i, freq[i]);
* }
* }
* }
* int main() {
* char str[MAX\_LENGTH];
* readString(str);
* countFrequency(str);
* return 0;
* }

Arrays, Structures, Nested Structures, Unions, Nested Unions, Strings, Typedef

* Question 1: Write a C program that uses an array of structures to store information about employees. Each structure should contain a nested structure for the address. Use typedef to simplify the structure definitions. The program should allow the user to enter and display employee information.
* #include <stdio.h>
* typedef struct {
* char street[50];
* char city[30];
* char state[30];
* int zip;
* } Address;
* typedef struct {
* char name[50];
* int age;
* float salary;
* Address addr;
* } Employee;
* // Function to read employee data
* void readData(Employee \*emp) {
* printf("Enter employee name: ");
* scanf("%s", emp->name);
* printf("Enter age: ");
* scanf("%d", &emp->age);
* printf("Enter salary: ");
* scanf("%f", &emp->salary);
* printf("Enter street address: ");
* scanf("%s", emp->addr.street);
* printf("Enter city: ");
* scanf("%s", emp->addr.city);
* printf("Enter state: ");
* scanf("%s", emp->addr.state);
* printf("Enter zip code: ");
* scanf("%d", &emp->addr.zip);
* }
* // Function to display employee data
* void displayData(const Employee \*emp) {
* printf("\nEmployee Information:\n");
* printf("Name: %s\n", emp->name);
* printf("Age: %d\n", emp->age);
* printf("Salary: %.2f\n", emp->salary);
* printf("Address:\n");
* printf("Street: %s\n", emp->addr.street);
* printf("City: %s\n", emp->addr.city);
* printf("State: %s\n", emp->addr.state);
* printf("Zip Code: %d\n", emp->addr.zip);
* }
* int main() {
* int n;
* printf("Enter the number of employees: ");
* scanf("%d", &n);
* Employee employees[n];
* // Read employee data for each employee
* for (int i = 0; i < n; i++) {
* printf("\nEnter details for employee %d:\n", i + 1);
* readData(&employees[i]);
* }
* // Display employee data for each employee
* for (int i = 0; i < n; i++) {
* displayData(&employees[i]);
* }
* return 0;
* }
* Question 2: Create a program that demonstrates the use of a union to store different types of data. Implement a nested union within a structure and use a typedef to define the structure. Use an array of this structure to store and display information about different data types (e.g., integer, float, string).
* #include <stdio.h>
* // defining union
* typedef union {
* int i;
* float f;
* char str[50];
* } Data;
* // Define a structure that contains a nested union
* typedef struct {
* int type;
* Data data;
* } DataItem;
* void displayData(DataItem item) {
* if (item.type == 1) {
* printf("Integer: %d\n", item.data.i);
* } else if (item.type == 2) {
* printf("Float: %.2f\n", item.data.f);
* } else if (item.type == 3) {
* printf("String: %s\n", item.data.str);
* }
* }
* int main() {
* DataItem items[3];
* printf("Enter an integer: ");
* items[0].type = 1;
* printf("Enter a float: ");
* items[1].type = 2;
* scanf("%f", &items[1].data.f);
* printf("Enter a string: ");
* items[2].type = 3;
* scanf("%s", items[2].data.str);
* for (int i = 0; i < 3; i++) {
* displayData(items[i]);
* }
* return 0;
* }
* Question 3: Write a C program that uses an array of strings to store names. Implement a structure containing a nested union to store either the length of the string or the reversed string. Use typedef to simplify the structure definition and display the stored information.
* #include <stdio.h>
* #include <string.h>
* // Defining the union to store either length or reversed string
* typedef union {
* int length;
* char reversed[50];
* } StringInfo;
* // Defining the structure containing the nested union
* typedef struct {
* char name[50];
* StringInfo info;
* int isLength;
* } NameItem;
* // Function to reverse a string
* void reverseString(char \*str, char \*revStr) {
* int len = strlen(str);
* for (int i = 0; i < len; i++) {
* revStr[i] = str[len - 1 - i];
* }
* revStr[len] = '\0';
* }
* void displayData(NameItem item) {
* if (item.isLength) {
* printf("Length of \"%s\": %d\n", item.name, item.info.length);
* } else {
* printf("Reversed string of \"%s\": %s\n", item.name, item.info.reversed);
* }
* }
* int main() {
* NameItem items[3]; // Array to store 3 names
* // Input 3 names and process them
* for (int i = 0; i < 3; i++) {
* printf("Enter name %d: ", i + 1);
* scanf("%s", items[i].name);
* // Store length of the string
* items[i].info.length = strlen(items[i].name);
* items[i].isLength = 1;
* // Store reversed string
* reverseString(items[i].name, items[i].info.reversed);
* items[i].isLength = 0;
* }
* // Display stored data
* for (int i = 0; i < 3; i++) {
* displayData(items[i]);
* }
* return 0;
* }
* Question 4: Develop a program that demonstrates the use of nested structures and unions. Create a structure that contains a union, and within the union, define another structure. Use an array to manage multiple instances of this complex structure and typedef to define the structure.
* #include <stdio.h>
* // Define structure
* typedef struct {
* char city[30];
* int zip;
* } Address;
* // Define a union
* typedef union {
* int number;
* Address addr;
* } Data;
* // Define the main structure that contains the union
* typedef struct {
* char name[50];
* Data info;
* int isAddress;
* } Item;
* void displayData(Item item) {
* printf("Name: %s\n", item.name);
* if (item.isAddress) {
* printf("Address: %s, Zip: %d\n", item.info.addr.city, item.info.addr.zip);
* } else {
* printf("Number: %d\n", item.info.number);
* }
* }
* int main() {
* Item items[3];
* // Input and manage data for 3 items
* for (int i = 0; i < 3; i++) {
* printf("Enter name %d: ", i + 1);
* scanf("%s", items[i].name);
* printf("Enter 1 for number, 2 for address: ");
* scanf("%d", &items[i].isAddress);
* if (items[i].isAddress == 1) {
* printf("Enter number: ");
* scanf("%d", &items[i].info.number);
* } else {
* printf("Enter city: ");
* scanf("%s", items[i].info.addr.city);
* printf("Enter zip: ");
* scanf("%d", &items[i].info.addr.zip);
* }
* }
* for (int i = 0; i < 3; i++) {
* displayData(items[i]);
* }
* return 0;
* }
* Question 5: Write a C program that defines a structure to store information about books. Use a nested structure to store the author's details and a union to store either the number of pages or the publication year. Use typedef to simplify the structure and implement functions to input and display the information.
* #include <stdio.h>
* #include <string.h>
* // Define the structure for author's details
* typedef struct {
* char name[50];
* char country[30];
* } Author;
* // Define a union to store either the number of pages or the publication year
* typedef union {
* int pages;
* int publicationYear;
* } BookDetails;
* // Define the main structure for a book
* typedef struct {
* char title[50];
* Author author;
* BookDetails details;
* int isPages;
* } Book;
* // Function to input book details
* void inputBook(Book \*b) {
* printf("Enter book title: ");
* scanf(" %[^\n]s", b->title);
* printf("Enter author's name: ");
* scanf(" %[^\n]s", b->author.name);
* printf("Enter author's country: ");
* scanf(" %[^\n]s", b->author.country);
* printf("Enter 1 for number of pages, 2 for publication year: ");
* scanf("%d", &b->isPages);
* if (b->isPages == 1) {
* printf("Enter the number of pages: ");
* scanf("%d", &b->details.pages);
* } else {
* printf("Enter the publication year: ");
* scanf("%d", &b->details.publicationYear);
* }
* }
* // Function to display book details
* void displayBook(Book b) {
* printf("\nBook Title: %s\n", b.title);
* printf("Author: %s, Country: %s\n", b.author.name, b.author.country);
* if (b.isPages == 1) {
* printf("Number of Pages: %d\n", b.details.pages);
* } else {
* printf("Publication Year: %d\n", b.details.publicationYear);
* }
* }
* int main() {
* Book books[3];
* for (int i = 0; i < 3; i++) {
* printf("\nEnter details for book %d:\n", i + 1);
* inputBook(&books[i]);
* displayBook(books[i]);
* }
* return 0;
* }

Stacks Using Arrays and Linked List

* Question 1: Write a C program to implement a stack using arrays. The program should include functions for all stack operations: push, pop, peek, isEmpty, and isFull. Demonstrate the working of the stack with sample data.
* #include <stdio.h>
* #include <stdlib.h>
* struct Stack {
* int size;
* int top;
* int \*s;
* };
* // Function prototypes
* void create(struct Stack \*);
* void display(struct Stack);
* void push(struct Stack \*, int);
* int pop(struct Stack \*);
* int peek(struct Stack, int);
* int isEmpty(struct Stack);
* int isFull(struct Stack);
* int main() {
* struct Stack st;
* create(&st);
* // Push elements into the stack
* push(&st, 5);
* push(&st, 10);
* push(&st, 15);
* push(&st, 20);
* display(st);
* // Pop an element
* int popedValue = pop(&st);
* printf("Popped value = %d\n", popedValue);
* display(st);
* // Peek at a specific position
* printf("Element at index position 1: %d\n", peek(st, 1));
* printf("Is stack empty? %d\n", isEmpty(st));
* printf("Is stack full? %d\n", isFull(st));
* free(st.s);
* return 0;
* }
* void create(struct Stack \*st) {
* printf("Enter the size of the stack: ");
* scanf("%d", &st->size);
* st->top = -1;
* st->s = (int \*)malloc(st->size \* sizeof(int));
* if (st->s == NULL) {
* printf("Memory allocation failed\n");
* exit(1);
* }
* }
* void push(struct Stack \*st, int x) {
* if (isFull(\*st)) {
* printf("Stack is full!\n");
* } else {
* st->top++;
* st->s[st->top] = x;
* }
* }
* int pop(struct Stack \*st) {
* if (isEmpty(\*st)) {
* printf("Stack is empty!\n");
* return -1;
* } else {
* int poppedValue = st->s[st->top];
* st->top--;
* return poppedValue;
* }
* }
* int peek(struct Stack st, int index) {
* int position = st.top - index + 1;
* if (position < 0 || position > st.top) {
* printf("Invalid position!\n");
* return -1;
* }
* return st.s[position];
* }
* int isEmpty(struct Stack st) {
* return st.top == -1;
* }
* int isFull(struct Stack st) {
* return st.top == st.size - 1;
* }
* void display(struct Stack st) {
* if (isEmpty(st)) {
* printf("Stack is empty\n");
* } else {
* printf("Stack elements: ");
* for (int i = st.top; i >= 0; i--) {
* printf("%d ", st.s[i]);
* }
* printf("\n");
* }
* }
* Question 2: Develop a program to implement a stack using a linked list. Include functions for all stack operations: push, pop, peek, isEmpty, and isFull. Ensure proper memory management by handling dynamic allocation and deallocation.
* #include <stdio.h>
* #include <stdlib.h>
* struct Node {
* int data;
* struct Node \*next;
* } \*top = NULL;
* void push(int);
* int pop();
* void display();
* int peek();
* int isEmpty();
* int isFull();
* int main() {
* push(20);
* push(30);
* push(40);
* display();
* int poppedValue = pop();
* printf("Popped Value: %d\n", poppedValue);
* display();
* printf("Peek top value: %d\n", peek());
* // Check if the stack is empty or full
* printf("Is stack empty? %d\n", isEmpty());
* printf("Is stack full? %d\n", isFull());
* return 0;
* }
* // Push operation
* void push(int x) {
* struct Node \*t;
* t = (struct Node \*)malloc(sizeof(struct Node));
* if (t == NULL) {
* printf("Stack is Full\n");
* } else {
* t->data = x;
* t->next = top;
* top = t;
* }
* }
* // Pop operation
* int pop() {
* struct Node \*t;
* int x = -1;
* if (top == NULL) {
* printf("Stack is Empty\n");
* } else {
* t = top;
* top = top->next;
* x = t->data;
* free(t);
* }
* return x;
* }
* // Display the stack elements
* void display() {
* struct Node \*p;
* p = top;
* if (p == NULL) {
* printf("Stack is Empty\n");
* } else {
* printf("Stack elements: ");
* while (p != NULL) {
* printf("%d ", p->data);
* p = p->next;
* }
* printf("\n");
* }
* }
* // Peek operation: Returns the top value without removing it
* int peek() {
* if (top == NULL) {
* printf("Stack is Empty\n");
* return -1;
* }
* return top->data;
* }
* // Check if the stack is empty
* int isEmpty() {
* if (top == NULL)
* return 1;
* return 0;
* }
* // Check if the stack is full
* int isFull() {
* struct Node \*t = (struct Node \*)malloc(sizeof(struct Node));
* if (t == NULL)
* return 1;
* free(t);
* return 0;
* }
* Question 3: Create a C program to implement a stack using arrays. Include an additional operation to reverse the contents of the stack. Demonstrate the reversal operation with sample data.
* #include <stdio.h>
* #include <stdlib.h>
* struct Stack {
* int size;
* int top;
* int \*s;
* };
* void create(struct Stack \*);
* void display(struct Stack);
* void push(struct Stack \*, int);
* int pop(struct Stack \*);
* int isEmpty(struct Stack);
* int isFull(struct Stack);
* void reverse(struct Stack \*);
* int main() {
* struct Stack st;
* create(&st);
* push(&st, 10);
* push(&st, 20);
* push(&st, 30);
* push(&st, 40);
* printf("Original Stack:\n");
* display(st);
* reverse(&st);
* printf("Reversed Stack:\n");
* display(st);
* return 0;
* }
* // Create a stack and initialize it
* void create(struct Stack \*st) {
* printf("Enter the size of the stack: ");
* scanf("%d", &st->size);
* st->top = -1;
* st->s = (int \*)malloc(st->size \* sizeof(int));
* }
* // Push operation
* void push(struct Stack \*st, int x) {
* if (st->top == st->size - 1) {
* printf("Stack is Full\n");
* } else {
* st->top++;
* st->s[st->top] = x;
* }
* }
* // Pop operation
* int pop(struct Stack \*st) {
* int x = -1;
* if (st->top == -1) {
* printf("Stack is Empty\n");
* } else {
* x = st->s[st->top];
* st->top--;
* }
* return x;
* }
* // Display the stack elements
* void display(struct Stack st) {
* if (st.top == -1) {
* printf("Stack is Empty\n");
* } else {
* for (int i = st.top; i >= 0; i--) {
* printf("%d\n", st.s[i]);
* }
* }
* }
* int isEmpty(struct Stack st) {
* return st.top == -1;
* }
* int isFull(struct Stack st) {
* return st.top == st.size - 1;
* }
* // Reverse the stack contents
* void reverse(struct Stack \*st) {
* int start = 0;
* int end = st->top;
* while (start < end) {
* int temp = st->s[start];
* st->s[start] = st->s[end];
* st->s[end] = temp;
* start++;
* end--;
* }
* }
* Question 4: Write a program to implement a stack using a linked list. Extend the program to include an operation to merge two stacks. Demonstrate the merging operation by combining two stacks and displaying the resulting stack.
* #include <stdio.h>
* #include <stdlib.h>
* // Define the node structure for the stack
* struct Node {
* int data;
* struct Node \*next;
* };
* // Function declarations
* void push(struct Node \*\*top, int x);
* int pop(struct Node \*\*top);
* void display(struct Node \*top);
* int isEmpty(struct Node \*top);
* void mergeStacks(struct Node \*\*stack1, struct Node \*\*stack2, struct Node \*\*mergedStack);
* int main() {
* struct Node \*stack1 = NULL;
* struct Node \*stack2 = NULL;
* struct Node \*mergedStack = NULL;
* // Pushing data to stack1
* push(&stack1, 10);
* push(&stack1, 20);
* push(&stack1, 30);
* push(&stack2, 40);
* push(&stack2, 50);
* printf("Stack 1 before merge:\n");
* display(stack1);
* printf("Stack 2 before merge:\n");
* display(stack2);
* // Merging stack1 and stack2 into mergedStack
* mergeStacks(&stack1, &stack2, &mergedStack);
* printf("Merged Stack:\n");
* display(mergedStack);
* return 0;
* }
* // Push operation to add an element to the stack
* void push(struct Node \*\*top, int x) {
* struct Node \*t = (struct Node \*)malloc(sizeof(struct Node));
* if (t == NULL) {
* printf("Stack Overflow\n");
* } else {
* t->data = x;
* t->next = \*top;
* \*top = t;
* }
* }
* // Pop operation to remove an element from the stack
* int pop(struct Node \*\*top) {
* int x = -1;
* if (\*top == NULL) {
* printf("Stack Underflow\n");
* } else {
* struct Node \*t = \*top;
* x = t->data;
* \*top = t->next;
* free(t);
* }
* return x;
* }
* // Display the stack elements
* void display(struct Node \*top) {
* if (top == NULL) {
* printf("Stack is Empty\n");
* } else {
* while (top != NULL) {
* printf("%d\n", top->data);
* top = top->next;
* }
* }
* }
* // Check if the stack is empty
* int isEmpty(struct Node \*top) {
* return top == NULL;
* }
* // Merge two stacks into one stack
* void mergeStacks(struct Node \*\*stack1, struct Node \*\*stack2, struct Node \*\*mergedStack) {
* while (\*stack1 != NULL) {
* push(mergedStack, pop(stack1));
* }
* while (\*stack2 != NULL) {
* push(mergedStack, pop(stack2));
* }
* }
* Question 5: Develop a program that implements a stack using arrays. Add functionality to check for balanced parentheses in an expression using the stack. Demonstrate this with sample expressions.
* #include <stdio.h>
* #include <stdlib.h>
* // Stack structure
* struct Stack {
* int size;
* int top;
* char \*arr;
* };
* // Function declarations
* void create(struct Stack \*, int);
* void push(struct Stack \*, char);
* char pop(struct Stack \*);
* int isEmpty(struct Stack \*);
* int checkBalancedParentheses(char \*, struct Stack \*);
* int main() {
* struct Stack st;
* char expr[100];
* create(&st, 100);  // Create stack with size 100
* printf("Enter an expression: ");
* scanf("%s", expr);
* if (checkBalancedParentheses(expr, &st))
* printf("Balanced parentheses.\n");
* else
* printf("Unbalanced parentheses.\n");
* return 0;
* }
* void create(struct Stack \*st, int size) {
* st->size = size;
* st->top = -1;
* st->arr = (char \*)malloc(st->size \* sizeof(char));
* }
* void push(struct Stack \*st, char ch) {
* if (st->top < st->size - 1)
* st->arr[++st->top] = ch;
* }
* char pop(struct Stack \*st) {
* if (!isEmpty(st))
* return st->arr[st->top--];
* return -1;
* }
* int isEmpty(struct Stack \*st) {
* return st->top == -1;
* }
* int checkBalancedParentheses(char \*expr, struct Stack \*st) {
* for (int i = 0; expr[i]; i++) {
* char ch = expr[i];
* if (ch == '(' || ch == '{' || ch == '[')
* push(st, ch);
* else if ((ch == ')' && pop(st) != '(') ||
* (ch == '}' && pop(st) != '{') ||
* (ch == ']' && pop(st) != '['))
* return 0;
* }
* return isEmpty(st);
* }
* Question 6: Create a C program to implement a stack using a linked list. Extend the program to implement a stack-based evaluation of postfix expressions. Include all necessary stack operations and demonstrate the evaluation with sample expressions
* #include <stdio.h>
* #include <stdlib.h>
* #include <ctype.h>
* // Define the structure for a stack node
* struct Node {
* int data;
* struct Node \*next;
* } \*top = NULL;
* // Function prototypes
* void push(int x);
* int pop();
* int peek();
* int isEmpty();
* int evaluatePostfix(char \*expr);
* int main() {
* char expr[100];
* printf("Enter a postfix expression: ");
* scanf("%s", expr);  // Read the postfix expression
* int result = evaluatePostfix(expr);
* printf("The result of the postfix evaluation is: %d\n", result);
* return 0;
* }
* // Push function to insert data into the stack
* void push(int x) {
* struct Node \*t = (struct Node \*)malloc(sizeof(struct Node));
* if (t == NULL) {
* printf("Stack Overflow\n");
* } else {
* t->data = x;
* t->next = top;
* top = t;
* }
* }
* // Pop function to remove data from the stack
* int pop() {
* int x = -1;
* if (top == NULL) {
* printf("Stack Underflow\n");
* } else {
* struct Node \*t = top;
* x = t->data;
* top = top->next;
* free(t);
* }
* return x;
* }
* // Peek function to return the top element of the stack
* int peek() {
* if (top == NULL) {
* printf("Stack is Empty\n");
* return -1;
* } else {
* return top->data;
* }
* }
* // Function to check if the stack is empty
* int isEmpty() {
* return top == NULL;
* }
* // Function to evaluate a postfix expression using a stack
* int evaluatePostfix(char \*expr) {
* int i = 0;
* int op1, op2, result;
* while (expr[i] != '\0') {
* if (isdigit(expr[i])) {
* // If the character is a digit, push it onto the stack
* push(expr[i] - '0');
* } else {
* // If the character is an operator, pop two operands, apply the operator, and push the result
* op2 = pop();
* op1 = pop();
* switch (expr[i]) {
* case '+':
* result = op1 + op2;
* break;
* case '-':
* result = op1 - op2;
* break;
* case '\*':
* result = op1 \* op2;
* break;
* case '/':
* result = op1 / op2;
* break;
* default:
* printf("Invalid operator\n");
* return -1;
* }
* // Push the result back onto the stack
* push(result);
* }
* i++;
* }
* // The result will be the only element left in the stack
* return pop();
* }

**Queue**

1. **Student Admission Queue: Write a program to simulate a student admission process. Implement a queue using arrays to manage students waiting for admission. Include operations to enqueue (add a student), dequeue (admit a student), and display the current queue of students.**
2. #include <stdio.h>
3. #include <stdlib.h>
4. #include <string.h>
5. struct Queue {
6. int size;
7. int front;
8. int rear;
9. char \*\*Q; // Array of strings to store student names
10. };
11. void create(struct Queue \*, int);
12. void enqueue(struct Queue \*, char \*);
13. void display(struct Queue);
14. char \*dequeue(struct Queue \*);
15. int main() {
16. struct Queue q;
17. create(&q, 5);
18. enqueue(&q, "Ravi");
19. enqueue(&q, "Priya");
20. enqueue(&q, "Anjali");
21. display(q);
22. printf("Admitted Student: %s\n", dequeue(&q));
23. printf("\nQueue after admission:\n");
24. display(q);
25. return 0;
26. }
27. void create(struct Queue \*q, int size) {
28. q->size = size;
29. q->front = q->rear = -1;
30. q->Q = (char \*\*)malloc(q->size \* sizeof(char \*));
31. for (int i = 0; i < q->size; i++) {
32. q->Q[i] = (char \*)malloc(50 \* sizeof(char)); // Allocate space for each student's name
33. }
34. }
35. void enqueue(struct Queue \*q, char \*name) {
36. if (q->rear == q->size - 1) {
37. printf("Queue is full\n");
38. } else {
39. q->rear++;
40. strcpy(q->Q[q->rear], name); // Copy the student's name to the queue
41. }
42. }
43. void display(struct Queue q) {
44. if (q.front == q.rear) {
45. printf("Queue is empty\n");
46. } else {
47. for (int i = q.front + 1; i <= q.rear; i++) {
48. printf("%s -> ", q.Q[i]);
49. }
50. printf("NULL\n");
51. }
52. }
53. char \*dequeue(struct Queue \*q) {
54. if (q->front == q->rear) {
55. printf("Queue is Empty\n");
56. return NULL;
57. } else {
58. q->front++;
59. return q->Q[q->front]; // Return the dequeued student's name
60. }
61. }

* **2. Library Book Borrowing Queue: Develop a program that simulates a library's book borrowing system. Use a queue to manage students waiting to borrow books. Include functions to add a student to the queue, remove a student after borrowing a book, and display the queue status.**
* #include <stdio.h>
* #include <stdlib.h>
* #include <string.h>
* struct Queue {
* int size;
* int front;
* int rear;
* char \*\*Q; // Array of strings to store student names
* };
* void create(struct Queue \*, int);
* void enqueue(struct Queue \*, char \*);
* void display(struct Queue);
* char \*dequeue(struct Queue \*);
* int main() {
* struct Queue q;
* create(&q, 5);
* printf("Adding students to the queue:\n");
* enqueue(&q, "Rohit");
* enqueue(&q, "Simran");
* enqueue(&q, "Meera");
* display(q);
* printf("\nStudent borrowing a book: %s\n", dequeue(&q));
* printf("\nQueue after borrowing:\n");
* display(q);
* return 0;
* }
* void create(struct Queue \*q, int size) {
* q->size = size;
* q->front = q->rear = -1;
* q->Q = (char \*\*)malloc(q->size \* sizeof(char \*));
* for (int i = 0; i < q->size; i++) {
* q->Q[i] = (char \*)malloc(50 \* sizeof(char));
* }
* }
* void enqueue(struct Queue \*q, char \*name) {
* if (q->rear == q->size - 1) {
* printf("Queue is full. No more students can be added.\n");
* } else {
* q->rear++;
* strcpy(q->Q[q->rear], name);
* printf("%s added to the queue.\n", name);
* }
* }
* void display(struct Queue q) {
* if (q.front == q.rear) {
* printf("Queue is empty. No students waiting.\n");
* } else {
* printf("Students waiting to borrow books:\n");
* for (int i = q.front + 1; i <= q.rear; i++) {
* printf("%s -> ", q.Q[i]);
* }
* printf("NULL\n");
* }
* }
* char \*dequeue(struct Queue \*q) {
* if (q->front == q->rear) {
* printf("Queue is empty. No students to borrow books.\n");
* return NULL;
* } else {
* q->front++;
* return q->Q[q->front];
* }
* }
* **3. Cafeteria Token System: Create a program that simulates a cafeteria token system for students. Implement a queue using arrays to manage students waiting for their turn. Provide operations to issue tokens (enqueue), serve students (dequeue), and display the queue of students.**
* #include <stdio.h>
* #include <stdlib.h>
* struct Queue {
* int size;
* int front;
* int rear;
* char \*\*Q; // Array of string pointers for student names
* };
* void create(struct Queue \*q, int size) {
* q->size = size;
* q->front = q->rear = -1;
* q->Q = (char \*\*)malloc(size \* sizeof(char \*));
* }
* void enqueue(struct Queue \*q, char \*name) {
* if (q->rear == q->size - 1) {
* printf("Queue is full\n");
* } else {
* q->rear++;
* q->Q[q->rear] = name; // Assign student name directly
* }
* }
* char \*dequeue(struct Queue \*q) {
* if (q->front == q->rear) {
* printf("Queue is empty\n");
* return NULL;
* } else {
* q->front++;
* return q->Q[q->front]; // Serve the student (dequeue)
* }
* }
* void display(struct Queue q) {
* if (q.front == q.rear) {
* printf("Queue is empty\n");
* } else {
* for (int i = q.front + 1; i <= q.rear; i++) {
* printf("%s -> ", q.Q[i]);
* }
* printf("NULL\n");
* }
* }
* int main() {
* struct Queue q;
* create(&q, 5);
* enqueue(&q, "Ravi");
* enqueue(&q, "Simran");
* enqueue(&q, "Kiran");
* display(q);
* printf("Serving student: %s\n", dequeue(&q));
* display(q);
* return 0;
* }
* **4. Classroom Help Desk Queue: Write a program to manage a help desk queue in a classroom. Use a queue to track students waiting for assistance. Include functions to add students to the queue, remove them once helped, and view the current queue.**
* #include <stdio.h>
* #include <stdlib.h>
* struct Queue {
* int size;
* int front;
* int rear;
* char \*\*Q; // Array of string pointers for student names
* };
* void create(struct Queue \*q, int size) {
* q->size = size;
* q->front = q->rear = -1;
* q->Q = (char \*\*)malloc(size \* sizeof(char \*));
* }
* void enqueue(struct Queue \*q, char \*name) {
* if (q->rear == q->size - 1) {
* printf("Queue is full\n");
* } else {
* q->rear++;
* q->Q[q->rear] = name; // Add student to the queue
* }
* }
* char \*dequeue(struct Queue \*q) {
* if (q->front == q->rear) {
* printf("Queue is empty\n");
* return NULL;
* } else {
* q->front++;
* return q->Q[q->front]; // Remove student once helped
* }
* }
* void display(struct Queue q) {
* if (q.front == q.rear) {
* printf("Queue is empty\n");
* } else {
* for (int i = q.front + 1; i <= q.rear; i++) {
* printf("%s -> ", q.Q[i]);
* }
* printf("NULL\n");
* }
* }
* int main() {
* struct Queue q;
* create(&q, 5);
* enqueue(&q, "Ravi");
* enqueue(&q, "Simran");
* enqueue(&q, "Kiran");
* display(q);
* printf("Helped student: %s\n", dequeue(&q));
* display(q);
* return 0;
* }
* **5. Exam Registration Queue: Develop a program to simulate the exam registration process. Use a queue to manage the order of student registrations. Implement operations to add students to the queue, process their registration, and display the queue status.**
* #include <stdio.h>
* #include <stdlib.h>
* struct Queue {
* int size;
* int front;
* int rear;
* char \*\*Q; // Array of string pointers for student names
* };
* void create(struct Queue \*q, int size) {
* q->size = size;
* q->front = q->rear = -1;
* q->Q = (char \*\*)malloc(size \* sizeof(char \*));
* }
* void enqueue(struct Queue \*q, char \*name) {
* if (q->rear == q->size - 1) {
* printf("Queue is full\n");
* } else {
* q->rear++;
* q->Q[q->rear] = name; // Add student to the queue for exam registration
* }
* }
* char \*dequeue(struct Queue \*q) {
* if (q->front == q->rear) {
* printf("Queue is empty\n");
* return NULL;
* } else {
* q->front++;
* return q->Q[q->front]; // Process the registration for the student
* }
* }
* void display(struct Queue q) {
* if (q.front == q.rear) {
* printf("Queue is empty\n");
* } else {
* for (int i = q.front + 1; i <= q.rear; i++) {
* printf("%s -> ", q.Q[i]);
* }
* printf("NULL\n");
* }
* }
* int main() {
* struct Queue q;
* create(&q, 5);
* enqueue(&q, "Ravi");
* enqueue(&q, "Simran");
* enqueue(&q, "Kiran");
* display(q);
* printf("Processing registration for: %s\n", dequeue(&q));
* display(q);
* return 0;
* }
* **6. School Bus Boarding Queue: Create a program that simulates the boarding process of a school bus. Implement a queue to manage the order in which students board the bus. Include functions to enqueue students as they arrive and dequeue them as they board.**
* #include <stdio.h>
* #include <stdlib.h>
* struct Queue {
* int size;
* int front;
* int rear;
* char \*\*Q; // Array of string pointers for student names
* };
* void create(struct Queue \*q, int size) {
* q->size = size;
* q->front = q->rear = -1;
* q->Q = (char \*\*)malloc(size \* sizeof(char \*));
* }
* void enqueue(struct Queue \*q, char \*name) {
* if (q->rear == q->size - 1) {
* printf("Queue is full. Cannot add more students.\n");
* } else {
* q->rear++;
* q->Q[q->rear] = name; // Add student to the queue as they arrive
* }
* }
* char \*dequeue(struct Queue \*q) {
* if (q->front == q->rear) {
* printf("Queue is empty. No students to board.\n");
* return NULL;
* } else {
* q->front++;
* return q->Q[q->front]; // Remove the student as they board the bus
* }
* }
* void display(struct Queue q) {
* if (q.front == q.rear) {
* printf("Queue is empty. No students waiting to board.\n");
* } else {
* printf("Students waiting to board the bus: ");
* for (int i = q.front + 1; i <= q.rear; i++) {
* printf("%s -> ", q.Q[i]);
* }
* printf("NULL\n");
* }
* }
* int main() {
* struct Queue q;
* create(&q, 5);
* // Adding students to the queue as they arrive
* enqueue(&q, "Ravi");
* enqueue(&q, "Simran");
* enqueue(&q, "Kiran");
* // Displaying the queue
* display(q);
* // Boarding students
* printf("Boarding student: %s\n", dequeue(&q));
* display(q);
* return 0;
* }
* **7. Counseling Session Queue: Write a program to manage a queue for students waiting for a counseling session. Use an array-based queue to keep track of the students, with operations to add (enqueue) and serve (dequeue) students, and display the queue.**
* #include <stdio.h>
* #include <stdlib.h>
* struct Queue {
* int size;
* int front;
* int rear;
* char \*\*Q; // Array of string pointers for student names
* };
* void create(struct Queue \*q, int size) {
* q->size = size;
* q->front = q->rear = -1;
* q->Q = (char \*\*)malloc(size \* sizeof(char \*));
* }
* void enqueue(struct Queue \*q, char \*name) {
* if (q->rear == q->size - 1) {
* printf("Queue is full. No more students can be added.\n");
* } else {
* q->rear++;
* q->Q[q->rear] = name; // Add student to the queue for counseling
* }
* }
* char \*dequeue(struct Queue \*q) {
* if (q->front == q->rear) {
* printf("Queue is empty. No students left for counseling.\n");
* return NULL;
* } else {
* q->front++;
* return q->Q[q->front]; // Serve the student and remove from the queue
* }
* }
* void display(struct Queue q) {
* if (q.front == q.rear) {
* printf("Queue is empty. No students waiting.\n");
* } else {
* printf("Students waiting for counseling: ");
* for (int i = q.front + 1; i <= q.rear; i++) {
* printf("%s -> ", q.Q[i]);
* }
* printf("NULL\n");
* }
* }
* int main() {
* struct Queue q;
* create(&q, 5);
* // Adding students to the queue
* enqueue(&q, "Ravi");
* enqueue(&q, "Simran");
* enqueue(&q, "Kiran");
* // Display the queue
* display(q);
* // Serving students for counseling
* printf("Serving student: %s\n", dequeue(&q));
* display(q);
* return 0;
* }
* **8. Sports Event Registration Queue: Develop a program that manages the registration queue for a school sports event. Use a queue to handle the order of student registrations, with functions to add, process, and display the queue of registered students.**
* #include <stdio.h>
* #include <stdlib.h>
* struct Queue {
* int size;
* int front;
* int rear;
* char \*\*Q; // Array of string pointers for student names
* };
* void create(struct Queue \*q, int size) {
* q->size = size;
* q->front = q->rear = -1;
* q->Q = (char \*\*)malloc(size \* sizeof(char \*));
* }
* void enqueue(struct Queue \*q, char \*name) {
* if (q->rear == q->size - 1) {
* printf("Queue is full. Cannot register more students.\n");
* } else {
* q->rear++;
* q->Q[q->rear] = name; // Add student to the sports event registration queue
* }
* }
* char \*dequeue(struct Queue \*q) {
* if (q->front == q->rear) {
* printf("Queue is empty. No students to process.\n");
* return NULL;
* } else {
* q->front++;
* return q->Q[q->front]; // Process the student's registration
* }
* }
* void display(struct Queue q) {
* if (q.front == q.rear) {
* printf("Queue is empty. No students registered.\n");
* } else {
* printf("Registered students for sports event: ");
* for (int i = q.front + 1; i <= q.rear; i++) {
* printf("%s -> ", q.Q[i]);
* }
* printf("NULL\n");
* }
* }
* int main() {
* struct Queue q;
* create(&q, 5);
* // Adding students to the registration queue
* enqueue(&q, "Ravi");
* enqueue(&q, "Simran");
* enqueue(&q, "Kiran");
* // Display the queue of registered students
* display(q);
* // Processing registration of students
* printf("Processing registration for: %s\n", dequeue(&q));
* display(q);
* return 0;
* }
* **9. Laboratory Equipment Checkout Queue: Create a program to simulate a queue for students waiting to check out laboratory equipment. Implement operations to add students to the queue, remove them once they receive equipment, and view the current queue.**
* #include <stdio.h>
* #include <stdlib.h>
* struct Queue {
* int size;
* int front;
* int rear;
* char \*\*Q; // Array of string pointers for student names
* };
* void create(struct Queue \*q, int size) {
* q->size = size;
* q->front = q->rear = -1;
* q->Q = (char \*\*)malloc(size \* sizeof(char \*));
* }
* void enqueue(struct Queue \*q, char \*name) {
* if (q->rear == q->size - 1) {
* printf("Queue is full. Cannot add more students.\n");
* } else {
* q->rear++;
* q->Q[q->rear] = name; // Add student to the equipment checkout queue
* }
* }
* char \*dequeue(struct Queue \*q) {
* if (q->front == q->rear) {
* printf("Queue is empty. No students to serve.\n");
* return NULL;
* } else {
* q->front++;
* return q->Q[q->front]; // Serve the student (give equipment) and remove from the queue
* }
* }
* void display(struct Queue q) {
* if (q.front == q.rear) {
* printf("Queue is empty. No students waiting to checkout equipment.\n");
* } else {
* printf("Students waiting to checkout equipment: ");
* for (int i = q.front + 1; i <= q.rear; i++) {
* printf("%s -> ", q.Q[i]);
* }
* printf("NULL\n");
* }
* }
* int main() {
* struct Queue q;
* create(&q, 5);
* // Adding students to the checkout queue
* enqueue(&q, "Ravi");
* enqueue(&q, "Simran");
* enqueue(&q, "Kiran");
* // Display the queue of students
* display(q);
* // Serving students (checking out equipment)
* printf("Serving student: %s\n", dequeue(&q));
* display(q);
* return 0;
* }
* **10. Parent-Teacher Meeting Queue: Write a program to manage a queue for a parent-teacher meeting. Use a queue to organize the order in which parents meet the teacher. Include functions to enqueue parents, dequeue them after the meeting, and display the queue status.**
* #include <stdio.h>
* #include <stdlib.h>
* struct Queue {
* int size;
* int front;
* int rear;
* char \*\*Q; // Array of string pointers for parent names
* };
* void create(struct Queue \*q, int size) {
* q->size = size;
* q->front = q->rear = -1;
* q->Q = (char \*\*)malloc(size \* sizeof(char \*));
* }
* void enqueue(struct Queue \*q, char \*name) {
* if (q->rear == q->size - 1) {
* printf("Queue is full. No more parents can be added.\n");
* } else {
* q->rear++;
* q->Q[q->rear] = name; // Add parent to the queue for meeting
* }
* }
* char \*dequeue(struct Queue \*q) {
* if (q->front == q->rear) {
* printf("Queue is empty. No parents left to meet.\n");
* return NULL;
* } else {
* q->front++;
* return q->Q[q->front]; // Remove parent from the queue after meeting
* }
* }
* void display(struct Queue q) {
* if (q.front == q.rear) {
* printf("Queue is empty. No parents waiting to meet.\n");
* } else {
* printf("Parents waiting for the meeting: ");
* for (int i = q.front + 1; i <= q.rear; i++) {
* printf("%s -> ", q.Q[i]);
* }
* printf("NULL\n");
* }
* }
* int main() {
* struct Queue q;
* create(&q, 5);
* // Adding parents to the queue
* enqueue(&q, "Mr. Sharma");
* enqueue(&q, "Mrs. Gupta");
* enqueue(&q, "Mr. Verma");
* // Display the queue of parents
* display(q);
* // Serving parents for the meeting
* printf("Meeting with parent: %s\n", dequeue(&q));
* display(q);
* return 0;
* }

1. Real-Time Sensor Data Processing:
   * Implement a queue using a linked list to store real-time data from various sensors (e.g., temperature, pressure). The system should enqueue sensor readings, process and dequeue the oldest data when a new reading arrives, and search for specific readings based on timestamps.
2. #include <stdio.h>
3. #include <stdlib.h>
4. #include <string.h>
5. struct SensorData {
6. char timestamp[20];
7. float temperature;
8. float pressure;
9. struct SensorData \*next;
10. };
11. struct Queue {
12. struct SensorData \*front;
13. struct SensorData \*rear;
14. };
15. void initializeQueue(struct Queue \*q) {
16. q->front = q->rear = NULL;
17. }
18. void enqueue(struct Queue \*q, const char \*timestamp, float temperature, float pressure) {
19. struct SensorData \*newNode = (struct SensorData \*)malloc(sizeof(struct SensorData));
20. strcpy(newNode->timestamp, timestamp);
21. newNode->temperature = temperature;
22. newNode->pressure = pressure;
23. newNode->next = NULL;
24. if (q->rear == NULL) {
25. q->front = q->rear = newNode;
26. } else {
27. q->rear->next = newNode;
28. q->rear = newNode;
29. }
30. }
31. struct SensorData \*dequeue(struct Queue \*q) {
32. if (q->front == NULL) {
33. printf("Queue is empty. No data to process.\n");
34. return NULL;
35. }
36. struct SensorData \*temp = q->front;
37. q->front = q->front->next;
38. if (q->front == NULL) {
39. q->rear = NULL;
40. }
41. return temp;
42. }
43. void displayQueue(struct Queue q) {
44. if (q.front == NULL) {
45. printf("Queue is empty.\n");
46. return;
47. }
48. printf("Sensor Data Queue:\n");
49. struct SensorData \*temp = q.front;
50. while (temp != NULL) {
51. printf("Timestamp: %s, Temperature: %.2f, Pressure: %.2f\n",
52. temp->timestamp, temp->temperature, temp->pressure);
53. temp = temp->next;
54. }
55. }
56. void searchReading(struct Queue q, const char \*timestamp) {
57. struct SensorData \*temp = q.front;
58. while (temp != NULL) {
59. if (strcmp(temp->timestamp, timestamp) == 0) {
60. printf("Found Reading - Timestamp: %s, Temperature: %.2f, Pressure: %.2f\n",
61. temp->timestamp, temp->temperature, temp->pressure);
62. return;
63. }
64. temp = temp->next;
65. }
66. printf("Reading with timestamp %s not found.\n", timestamp);
67. }
68. int main() {
69. struct Queue sensorQueue;
70. initializeQueue(&sensorQueue);
71. // Enqueueing sensor data
72. enqueue(&sensorQueue, "2025-01-20 10:00", 25.5, 1013.2);
73. enqueue(&sensorQueue, "2025-01-20 10:05", 26.0, 1012.8);
74. enqueue(&sensorQueue, "2025-01-20 10:10", 25.8, 1013.0);
75. // Display the queue
76. displayQueue(sensorQueue);
77. // Searching for a specific reading
78. searchReading(sensorQueue, "2025-01-20 10:05");
79. // Dequeueing the oldest data
80. struct SensorData \*processedData = dequeue(&sensorQueue);
81. if (processedData != NULL) {
82. printf("Processed Reading - Timestamp: %s, Temperature: %.2f, Pressure: %.2f\n",
83. processedData->timestamp, processedData->temperature, processedData->pressure);
84. free(processedData);
85. }
86. // Display the queue after processing
87. displayQueue(sensorQueue);
88. return 0;
89. }

2. Task Scheduling in a Real-Time Operating System (RTOS):

* + Design a queue using a linked list to manage task scheduling in an RTOS. Each task should have a unique identifier, priority level, and execution time. Implement enqueue to add tasks, dequeue to remove the next task for execution, and search to find tasks by priority.

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <string.h>
4. // Define the structure for a Task
5. struct Task {
6. int id;
7. int priority;
8. int executionTime;
9. struct Task \*next;
10. };
11. // Define the structure for the Queue
12. struct Queue {
13. struct Task \*front;
14. struct Task \*rear;
15. };
16. // Initialize the queue
17. void initializeQueue(struct Queue \*q) {
18. q->front = q->rear = NULL;
19. }
20. // Enqueue a task into the queue
21. void enqueue(struct Queue \*q, int id, int priority, int executionTime) {
22. struct Task \*newTask = (struct Task \*)malloc(sizeof(struct Task));
23. newTask->id = id;
24. newTask->priority = priority;
25. newTask->executionTime = executionTime;
26. newTask->next = NULL;
27. if (q->rear == NULL) {
28. q->front = q->rear = newTask;
29. } else {
30. q->rear->next = newTask;
31. q->rear = newTask;
32. }
33. }
34. // Dequeue a task for execution (removes the front task)
35. struct Task \*dequeue(struct Queue \*q) {
36. if (q->front == NULL) {
37. printf("Queue is empty. No tasks to execute.\n");
38. return NULL;
39. }
40. struct Task \*temp = q->front;
41. q->front = q->front->next;
42. if (q->front == NULL) {
43. q->rear = NULL;
44. }
45. return temp;
46. }
47. // Display the current queue of tasks
48. void displayQueue(struct Queue q) {
49. if (q.front == NULL) {
50. printf("Queue is empty. No tasks scheduled.\n");
51. return;
52. }
53. printf("Task Queue:\n");
54. struct Task \*temp = q.front;
55. while (temp != NULL) {
56. printf("Task ID: %d, Priority: %d, Execution Time: %d ms\n",
57. temp->id, temp->priority, temp->executionTime);
58. temp = temp->next;
59. }
60. }
61. // Search for tasks by priority
62. void searchByPriority(struct Queue q, int priority) {
63. struct Task \*temp = q.front;
64. int found = 0;
65. printf("Tasks with Priority %d:\n", priority);
66. while (temp != NULL) {
67. if (temp->priority == priority) {
68. printf("Task ID: %d, Execution Time: %d ms\n", temp->id, temp->executionTime);
69. found = 1;
70. }
71. temp = temp->next;
72. }
73. if (!found) {
74. printf("No tasks found with Priority %d.\n", priority);
75. }
76. }
77. int main() {
78. struct Queue taskQueue;
79. initializeQueue(&taskQueue);
80. // Adding tasks to the queue
81. enqueue(&taskQueue, 1, 3, 200);
82. enqueue(&taskQueue, 2, 1, 500);
83. enqueue(&taskQueue, 3, 2, 300);
84. // Display the current task queue
85. displayQueue(taskQueue);
86. // Search for tasks with priority 2
87. searchByPriority(taskQueue, 2);
88. // Dequeue a task for execution
89. struct Task \*executedTask = dequeue(&taskQueue);
90. if (executedTask != NULL) {
91. printf("Executing Task - ID: %d, Priority: %d, Execution Time: %d ms\n",
92. executedTask->id, executedTask->priority, executedTask->executionTime);
93. free(executedTask);
94. }
95. // Display the queue after execution
96. displayQueue(taskQueue);
97. return 0;
98. }

3. Interrupt Handling Mechanism:

* + Create a queue using a linked list to manage interrupt requests (IRQs) in an embedded system. Each interrupt should have a priority level and a handler function. Implement operations to enqueue new interrupts, dequeue the highest-priority interrupt, and search for interrupts by their source.

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <string.h>
4. // Structure for an Interrupt
5. struct Interrupt {
6. int priority;
7. char source[20];
8. void (\*handler)(void);
9. struct Interrupt \*next;
10. };
11. // Queue structure
12. struct Queue {
13. struct Interrupt \*front;
14. };
15. // Initialize queue
16. void initializeQueue(struct Queue \*q) {
17. q->front = NULL;
18. }
19. // Example handlers
20. void handleTimer() { printf("Handling Timer Interrupt\n"); }
21. void handleKeyboard() { printf("Handling Keyboard Interrupt\n"); }
22. // Enqueue interrupt based on priority
23. void enqueue(struct Queue \*q, int priority, const char \*source, void (\*handler)(void)) {
24. struct Interrupt \*newInterrupt = (struct Interrupt \*)malloc(sizeof(struct Interrupt));
25. newInterrupt->priority = priority;
26. strcpy(newInterrupt->source, source);
27. newInterrupt->handler = handler;
28. newInterrupt->next = NULL;
29. if (!q->front || q->front->priority > priority) {
30. newInterrupt->next = q->front;
31. q->front = newInterrupt;
32. } else {
33. struct Interrupt \*temp = q->front;
34. while (temp->next && temp->next->priority <= priority) temp = temp->next;
35. newInterrupt->next = temp->next;
36. temp->next = newInterrupt;
37. }
38. }
39. // Dequeue and handle highest-priority interrupt
40. void dequeue(struct Queue \*q) {
41. if (!q->front) {
42. printf("Queue is empty\n");
43. return;
44. }
45. struct Interrupt \*temp = q->front;
46. q->front = q->front->next;
47. printf("Handling Interrupt: %s\n", temp->source);
48. temp->handler();
49. free(temp);
50. }
51. // Display the interrupt queue
52. void display(struct Queue q) {
53. struct Interrupt \*temp = q.front;
54. if (!temp) {
55. printf("Queue is empty\n");
56. return;
57. }
58. while (temp) {
59. printf("Priority: %d, Source: %s\n", temp->priority, temp->source);
60. temp = temp->next;
61. }
62. }
63. int main() {
64. struct Queue irqQueue;
65. initializeQueue(&irqQueue);
66. enqueue(&irqQueue, 2, "Timer", handleTimer);
67. enqueue(&irqQueue, 1, "Keyboard", handleKeyboard);
68. display(irqQueue);
69. dequeue(&irqQueue);
70. display(irqQueue);
71. return 0;
72. }

4. Message Passing in Embedded Communication Systems:

* + Implement a message queue using a linked list to handle inter-process communication in embedded systems. Each message should include a sender ID, receiver ID, and payload. Enqueue messages as they arrive, dequeue messages for processing, and search for messages from a specific sender.

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <string.h>
4. // Node structure for a Message
5. struct Node {
6. int senderID;
7. int receiverID;
8. char payload[50];
9. struct Node \*next;
10. } \*front = NULL, \*rear = NULL;
11. // Function prototypes
12. void enqueue(int senderID, int receiverID, const char \*payload);
13. int dequeue();
14. void display();
15. void searchBySender(int senderID);
16. int main() {
17. // Adding messages to the queue
18. enqueue(101, 201, "Temperature Data: 25C");
19. enqueue(102, 202, "Pressure Data: 1.2 Bar");
20. enqueue(101, 203, "Sensor Update");
21. // Displaying the message queue
22. display();
23. // Searching for messages from a specific sender
24. printf("\nMessages from Sender 101:\n");
25. searchBySender(101);
26. // Dequeueing and processing the first message
27. printf("\nProcessing and removing the first message:\n");
28. dequeue();
29. // Displaying the queue after dequeue
30. display();
31. return 0;
32. }
33. // Enqueue a message
34. void enqueue(int senderID, int receiverID, const char \*payload) {
35. struct Node \*t = (struct Node \*)malloc(sizeof(struct Node));
36. if (t == NULL) {
37. printf("Queue is full\n");
38. } else {
39. t->senderID = senderID;
40. t->receiverID = receiverID;
41. strncpy(t->payload, payload, sizeof(t->payload) - 1);
42. t->payload[sizeof(t->payload) - 1] = '\0';  // Ensure null termination
43. t->next = NULL;
44. if (front == NULL) {
45. front = rear = t;
46. } else {
47. rear->next = t;
48. rear = t;
49. }
50. }
51. }
52. // Dequeue and process a message
53. int dequeue() {
54. if (front == NULL) {
55. printf("Queue is already empty\n");
56. return -1;
57. }
58. struct Node \*t = front;
59. printf("Sender: %d, Receiver: %d, Payload: %s\n", t->senderID, t->receiverID, t->payload);
60. front = front->next;
61. free(t);
62. if (front == NULL) {
63. rear = NULL;
64. }
65. return 0;
66. }
67. // Display all messages in the queue
68. void display() {
69. struct Node \*p = front;
70. if (!p) {
71. printf("Queue is empty\n");
72. return;
73. }
74. printf("Message Queue:\n");
75. while (p) {
76. printf("Sender: %d, Receiver: %d, Payload: %s\n", p->senderID, p->receiverID, p->payload);
77. p = p->next;
78. }
79. }
80. // Search for messages from a specific sender
81. void searchBySender(int senderID) {
82. struct Node \*p = front;
83. int found = 0;
84. while (p) {
85. if (p->senderID == senderID) {
86. printf("Receiver: %d, Payload: %s\n", p->receiverID, p->payload);
87. found = 1;
88. }
89. p = p->next;
90. }
91. if (!found) {
92. printf("No messages from Sender %d\n", senderID);
93. }
94. }

5. Data Logging System for Embedded Devices:

* + Design a queue using a linked list to log data in an embedded system. Each log entry should contain a timestamp, event type, and description. Implement enqueue to add new logs, dequeue old logs when memory is low, and search for logs by event type.

1. #include <stdio.h>
2. #include <stdlib.h>
3. // Structure for Log Entry
4. struct Log {
5. char timestamp[20];
6. char eventType[20];
7. char description[50];
8. struct Log \*next;
9. } \*front = NULL, \*rear = NULL;
10. // Function prototypes
11. void enqueue(const char \*timestamp, const char \*eventType, const char \*description);
12. void dequeue();
13. void display();
14. void searchByEventType(const char \*eventType);
15. int main() {
16. // Add log entries
17. enqueue("2025-01-20 12:00", "Error", "Device failure");
18. enqueue("2025-01-20 12:05", "Warning", "High temperature");
19. enqueue("2025-01-20 12:10", "Info", "System rebooted");
20. // Display logs
21. display();
22. // Search for logs with specific event type
23. printf("\nLogs for 'Error' event:\n");
24. searchByEventType("Error");
25. // Dequeue and process the oldest log
26. dequeue();
27. // Display logs after dequeue
28. display();
29. return 0;
30. }
31. // Enqueue a new log entry
32. void enqueue(const char \*timestamp, const char \*eventType, const char \*description) {
33. struct Log \*newLog = (struct Log \*)malloc(sizeof(struct Log));
34. if (newLog == NULL) {
35. printf("Memory is full\n");
36. return;
37. }
38. // Directly copying values without using strcpy
39. for (int i = 0; timestamp[i] != '\0'; i++) newLog->timestamp[i] = timestamp[i];
40. newLog->timestamp[19] = '\0';  // Ensure null termination
41. for (int i = 0; eventType[i] != '\0'; i++) newLog->eventType[i] = eventType[i];
42. newLog->eventType[19] = '\0';  // Ensure null termination
43. for (int i = 0; description[i] != '\0'; i++) newLog->description[i] = description[i];
44. newLog->description[49] = '\0';  // Ensure null termination
45. newLog->next = NULL;
46. if (front == NULL) {
47. front = rear = newLog;
48. } else {
49. rear->next = newLog;
50. rear = newLog;
51. }
52. }
53. // Dequeue the oldest log entry
54. void dequeue() {
55. if (front == NULL) {
56. printf("Queue is empty\n");
57. return;
58. }
59. struct Log \*temp = front;
60. front = front->next;
61. free(temp);
62. }
63. // Display all log entries
64. void display() {
65. if (front == NULL) {
66. printf("No logs available\n");
67. return;
68. }
69. struct Log \*temp = front;
70. while (temp) {
71. printf("Timestamp: %s, Event: %s, Description: %s\n", temp->timestamp, temp->eventType, temp->description);
72. temp = temp->next;
73. }
74. }
75. // Search for logs by event type
76. void searchByEventType(const char \*eventType) {
77. struct Log \*temp = front;
78. int found = 0;
79. while (temp) {
80. int match = 1;
81. for (int i = 0; eventType[i] != '\0'; i++) {
82. if (temp->eventType[i] != eventType[i]) {
83. match = 0;
84. break;
85. }
86. }
87. if (match) {
88. printf("Timestamp: %s, Description: %s\n", temp->timestamp, temp->description);
89. found = 1;
90. }
91. temp = temp->next;
92. }
93. if (!found) {
94. printf("No logs found for event type: %s\n", eventType);
95. }
96. }

6. Network Packet Management:

* + Create a queue using a linked list to manage network packets in an embedded router. Each packet should have a source IP, destination IP, and payload. Implement enqueue for incoming packets, dequeue for packets ready for transmission, and search for packets by IP address.

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <string.h>
4. // Structure for Network Packet
5. struct Packet {
6. char sourceIP[16];
7. char destIP[16];
8. char payload[100];
9. struct Packet \*next;
10. } \*front = NULL, \*rear = NULL;
11. // Function prototypes
12. void enqueue(const char \*sourceIP, const char \*destIP, const char \*payload);
13. void dequeue();
14. void display();
15. void searchByIP(const char \*ip);
16. int main() {
17. // Enqueue packets
18. enqueue("192.168.1.1", "192.168.1.2", "Packet 1 Data");
19. enqueue("192.168.1.3", "192.168.1.4", "Packet 2 Data");
20. enqueue("192.168.1.5", "192.168.1.6", "Packet 3 Data");
21. // Display all packets
22. display();
23. // Search for packets by source IP
24. printf("\nSearching for packets from IP '192.168.1.3':\n");
25. searchByIP("192.168.1.3");
26. dequeue();
27. display();
28. return 0;
29. }
30. // Enqueue a new packet into the queue
31. void enqueue(const char \*sourceIP, const char \*destIP, const char \*payload) {
32. struct Packet \*newPacket = (struct Packet \*)malloc(sizeof(struct Packet));
33. if (newPacket == NULL) {
34. printf("Memory is full\n");
35. return;
36. }
37. // Copy source IP, destination IP, and payload to the new packet
38. for (int i = 0; sourceIP[i] != '\0'; i++) newPacket->sourceIP[i] = sourceIP[i];
39. newPacket->sourceIP[15] = '\0';
40. for (int i = 0; destIP[i] != '\0'; i++) newPacket->destIP[i] = destIP[i];
41. newPacket->destIP[15] = '\0';
42. for (int i = 0; payload[i] != '\0'; i++) newPacket->payload[i] = payload[i];
43. newPacket->payload[99] = '\0';
44. newPacket->next = NULL;
45. if (front == NULL) {
46. front = rear = newPacket;
47. } else {
48. rear->next = newPacket;
49. rear = newPacket;
50. }
51. }
52. // Dequeue the oldest packet ready for transmission
53. void dequeue() {
54. if (front == NULL) {
55. printf("No packets to transmit\n");
56. return;
57. }
58. struct Packet \*temp = front;
59. front = front->next;
60. free(temp);
61. }
62. // Display all the packets in the queue
63. void display() {
64. if (front == NULL) {
65. printf("No packets in the queue\n");
66. return;
67. }
68. struct Packet \*temp = front;
69. while (temp) {
70. printf("Source IP: %s, Dest IP: %s, Payload: %s\n", temp->sourceIP, temp->destIP, temp->payload);
71. temp = temp->next;
72. }
73. }
74. // Search for packets by IP address (either source or destination)
75. void searchByIP(const char \*ip) {
76. struct Packet \*temp = front;
77. int found = 0;
78. while (temp) {
79. if (strcmp(temp->sourceIP, ip) == 0 || strcmp(temp->destIP, ip) == 0) {
80. printf("Found Packet: Source IP: %s, Dest IP: %s, Payload: %s\n", temp->sourceIP, temp->destIP, temp->payload);
81. found = 1;
82. }
83. temp = temp->next;
84. }
85. if (!found) {
86. printf("No packets found for IP: %s\n", ip);
87. }
88. }

7. Firmware Update Queue:

* Implement a queue using a linked list to manage firmware updates in an embedded system. Each update should include a version number, release notes, and file path. Enqueue updates as they become available, dequeue them for installation, and search for updates by version number.
* #include <stdio.h>
* #include <stdlib.h>
* // Structure for Firmware Update
* struct Update {
* int version;
* char releaseNotes[100];
* char filePath[100];
* struct Update \*next;
* } \*front = NULL, \*rear = NULL;
* // Function prototypes
* void enqueue(int version, const char \*releaseNotes, const char \*filePath);
* void dequeue();
* void display();
* void searchByVersion(int version);
* int main() {
* // Enqueue firmware updates
* enqueue(1, "Bug fixes and performance improvements", "/path/to/update1");
* enqueue(2, "Security patch applied", "/path/to/update2");
* enqueue(3, "New features added", "/path/to/update3");
* display();
* // Search for updates by version number
* printf("\nSearching for version 2:\n");
* searchByVersion(2);
* dequeue();
* display();
* return 0;
* }
* // Enqueue a new firmware update
* void enqueue(int version, const char \*releaseNotes, const char \*filePath) {
* struct Update \*newUpdate = (struct Update \*)malloc(sizeof(struct Update));
* if (newUpdate == NULL) {
* printf("Memory is full\n");
* return;
* }
* newUpdate->version = version;
* // Copy release notes and file path
* for (int i = 0; releaseNotes[i] != '\0'; i++) newUpdate->releaseNotes[i] = releaseNotes[i];
* newUpdate->releaseNotes[99] = '\0';  // Ensure null termination
* for (int i = 0; filePath[i] != '\0'; i++) newUpdate->filePath[i] = filePath[i];
* newUpdate->filePath[99] = '\0';  // Ensure null termination
* newUpdate->next = NULL;
* if (front == NULL) {
* front = rear = newUpdate;
* } else {
* rear->next = newUpdate;
* rear = newUpdate;
* }
* }
* // Dequeue the oldest firmware update for installation
* void dequeue() {
* if (front == NULL) {
* printf("No updates available for installation\n");
* return;
* }
* struct Update \*temp = front;
* front = front->next;
* free(temp);
* }
* // Display all firmware updates in the queue
* void display() {
* if (front == NULL) {
* printf("No updates in the queue\n");
* return;
* }
* struct Update \*temp = front;
* while (temp) {
* printf("Version: %d, Release Notes: %s, File Path: %s\n", temp->version, temp->releaseNotes, temp->filePath);
* temp = temp->next;
* }
* }
* // Search for firmware updates by version number
* void searchByVersion(int version) {
* struct Update \*temp = front;
* int found = 0;
* while (temp) {
* if (temp->version == version) {
* printf("Found Update - Version: %d, Release Notes: %s, File Path: %s\n", temp->version, temp->releaseNotes, temp->filePath);
* found = 1;
* }
* temp = temp->next;
* }
* if (!found) {
* printf("No updates found for version: %d\n", version);
* }
* }

8. Power Management Events:

* Design a queue using a linked list to handle power management events in an embedded device. Each event should have a type (e.g., power on, sleep), timestamp, and associated action. Implement operations to enqueue events, dequeue events as they are handled, and search for events by type.
* #include <stdio.h>
* #include <stdlib.h>
* // Structure for Power Management Event
* struct Event {
* char type[20];    // Type of event (e.g., power on, sleep)
* int timestamp;    // Timestamp of the event
* char action[100]; // Associated action
* struct Event \*next;
* } \*front = NULL, \*rear = NULL;
* // Function prototypes
* void enqueue(const char \*type, int timestamp, const char \*action);
* void dequeue();
* void display();
* void searchByType(const char \*type);
* int main() {
* // Enqueue power management events
* enqueue("Power On", 101, "Device powered on");
* enqueue("Sleep", 102, "Device going to sleep");
* enqueue("Power On", 103, "Device powered on again");
* display();
* // Search for events by type
* printf("\nSearching for 'Power On' events:\n");
* searchByType("Power On");
* dequeue();
* display();
* return 0;
* }
* // Enqueue a new power management event
* void enqueue(const char \*type, int timestamp, const char \*action) {
* struct Event \*newEvent = (struct Event \*)malloc(sizeof(struct Event));
* if (newEvent == NULL) {
* printf("Memory is full\n");
* return;
* }
* // Copy event type and action
* for (int i = 0; type[i] != '\0'; i++) newEvent->type[i] = type[i];
* newEvent->type[19] = '\0';  // Ensure null termination
* newEvent->timestamp = timestamp;
* for (int i = 0; action[i] != '\0'; i++) newEvent->action[i] = action[i];
* newEvent->action[99] = '\0';  // Ensure null termination
* newEvent->next = NULL;
* if (front == NULL) {
* front = rear = newEvent;
* } else {
* rear->next = newEvent;
* rear = newEvent;
* }
* }
* // Dequeue the oldest power management event for processing
* void dequeue() {
* if (front == NULL) {
* printf("No events to handle\n");
* return;
* }
* struct Event \*temp = front;
* front = front->next;
* free(temp);
* }
* // Display all power management events in the queue
* void display() {
* if (front == NULL) {
* printf("No events in the queue\n");
* return;
* }
* struct Event \*temp = front;
* while (temp) {
* printf("Type: %s, Timestamp: %d, Action: %s\n", temp->type, temp->timestamp, temp->action);
* temp = temp->next;
* }
* }
* // Search for power management events by type
* void searchByType(const char \*type) {
* struct Event \*temp = front;
* int found = 0;
* while (temp) {
* if (strcmp(temp->type, type) == 0) {
* printf("Found Event - Type: %s, Timestamp: %d, Action: %s\n", temp->type, temp->timestamp, temp->action);
* found = 1;
* }
* temp = temp->next;
* }
* if (!found) {
* printf("No events found for type: %s\n", type);
* }
* }

9. Command Queue for Embedded Systems:

* Create a command queue using a linked list to handle user or system commands. Each command should have an ID, type, and parameters. Implement enqueue for new commands, dequeue for commands ready for execution, and search for commands by type.
* #include <stdio.h>
* #include <stdlib.h>
* // Structure for Command
* struct Command {
* int id;            // Command ID
* char type[20];     // Command type (e.g., "START", "STOP")
* char params[100];  // Command parameters (e.g., device ID or settings)
* struct Command \*next;
* } \*front = NULL, \*rear = NULL;
* // Function prototypes
* void enqueue(int id, const char \*type, const char \*params);
* void dequeue();
* void display();
* void searchByType(const char \*type);
* int main() {
* // Enqueue commands
* enqueue(1, "START", "Device 1");
* enqueue(2, "STOP", "Device 2");
* enqueue(3, "START", "Device 3");
* display();
* // Search for commands by type
* printf("\nSearching for 'START' commands:\n");
* searchByType("START");
* dequeue();
* display();
* return 0;
* }
* // Enqueue a new command
* void enqueue(int id, const char \*type, const char \*params) {
* struct Command \*newCmd = (struct Command \*)malloc(sizeof(struct Command));
* if (newCmd == NULL) {
* printf("Memory is full\n");
* return;
* }
* // Set command ID, type, and parameters
* newCmd->id = id;
* for (int i = 0; type[i] != '\0'; i++) newCmd->type[i] = type[i];
* newCmd->type[19] = '\0';  // Ensure null termination
* for (int i = 0; params[i] != '\0'; i++) newCmd->params[i] = params[i];
* newCmd->params[99] = '\0';  // Ensure null termination
* newCmd->next = NULL;
* if (front == NULL) {
* front = rear = newCmd;
* } else {
* rear->next = newCmd;
* rear = newCmd;
* }
* }
* // Dequeue the oldest command for execution
* void dequeue() {
* if (front == NULL) {
* printf("No commands to execute\n");
* return;
* }
* struct Command \*temp = front;
* front = front->next;
* free(temp);
* }
* // Display all commands in the queue
* void display() {
* if (front == NULL) {
* printf("No commands in the queue\n");
* return;
* }
* struct Command \*temp = front;
* while (temp) {
* printf("ID: %d, Type: %s, Params: %s\n", temp->id, temp->type, temp->params);
* temp = temp->next;
* }
* }
* // Search for commands by type
* void searchByType(const char \*type) {
* struct Command \*temp = front;
* int found = 0;
* while (temp) {
* if (strcmp(temp->type, type) == 0) {
* printf("Found Command - ID: %d, Type: %s, Params: %s\n", temp->id, temp->type, temp->params);
* found = 1;
* }
* temp = temp->next;
* }
* if (!found) {
* printf("No commands found for type: %s\n", type);
* }
* }

10. Audio Buffering in Embedded Audio Systems:

* Implement a queue using a linked list to buffer audio samples in an embedded audio system. Each buffer entry should include a timestamp and audio data. Enqueue new audio samples, dequeue samples for playback, and search for samples by timestamp.
* #include <stdio.h>
* #include <stdlib.h>
* // Structure for Audio Sample
* struct AudioSample {
* int timestamp;     // Timestamp of the audio sample
* char data[100];    // Audio data (can be a small chunk of audio data for simplicity)
* struct AudioSample \*next;
* } \*front = NULL, \*rear = NULL;
* // Function prototypes
* void enqueue(int timestamp, const char \*data);
* void dequeue();
* void display();
* void searchByTimestamp(int timestamp);
* int main() {
* // Enqueue audio samples
* enqueue(101, "AudioData1");
* enqueue(102, "AudioData2");
* enqueue(103, "AudioData3");
* display();
* // Search for an audio sample by timestamp
* printf("\nSearching for timestamp 102:\n");
* searchByTimestamp(102);
* dequeue();
* display();
* return 0;
* }
* // Enqueue a new audio sample
* void enqueue(int timestamp, const char \*data) {
* struct AudioSample \*newSample = (struct AudioSample \*)malloc(sizeof(struct AudioSample));
* if (newSample == NULL) {
* printf("Memory is full\n");
* return;
* }
* // Set timestamp and audio data
* newSample->timestamp = timestamp;
* for (int i = 0; data[i] != '\0'; i++) newSample->data[i] = data[i];
* newSample->data[99] = '\0';  // Ensure null termination
* newSample->next = NULL;
* if (front == NULL) {
* front = rear = newSample;
* } else {
* rear->next = newSample;
* rear = newSample;
* }
* }
* // Dequeue the oldest audio sample for playback
* void dequeue() {
* if (front == NULL) {
* printf("No samples in the buffer\n");
* return;
* }
* struct AudioSample \*temp = front;
* front = front->next;
* free(temp);
* }
* // Display all audio samples in the buffer
* void display() {
* if (front == NULL) {
* printf("No audio samples in the buffer\n");
* return;
* }
* struct AudioSample \*temp = front;
* while (temp) {
* printf("Timestamp: %d, Audio Data: %s\n", temp->timestamp, temp->data);
* temp = temp->next;
* }
* }
* // Search for audio samples by timestamp
* void searchByTimestamp(int timestamp) {
* struct AudioSample \*temp = front;
* int found = 0;
* while (temp) {
* if (temp->timestamp == timestamp) {
* printf("Found Sample - Timestamp: %d, Audio Data: %s\n", temp->timestamp, temp->data);
* found = 1;
* }
* temp = temp->next;
* }
* if (!found) {
* printf("No samples found with timestamp: %d\n", timestamp);
* }
* }

11. Event-Driven Programming in Embedded Systems:

* Design a queue using a linked list to manage events in an event-driven embedded system. Each event should have an ID, type, and associated data. Implement enqueue for new events, dequeue for event handling, and search for events by type or ID.
* #include <stdio.h>
* #include <stdlib.h>
* // Structure for Event
* struct Event {
* int id;             // Event ID
* char type[20];      // Event type (e.g., "ButtonPress", "SensorTrigger")
* char data[100];     // Associated event data (e.g., event-specific data)
* struct Event \*next;
* } \*front = NULL, \*rear = NULL;
* // Function prototypes
* void enqueue(int id, const char \*type, const char \*data);
* void dequeue();
* void display();
* void searchByID(int id);
* void searchByType(const char \*type);
* int main() {
* // Enqueue events
* enqueue(1, "ButtonPress", "Button A pressed");
* enqueue(2, "SensorTrigger", "Sensor B activated");
* enqueue(3, "ButtonPress", "Button C pressed");
* display();
* // Search for an event by ID
* printf("\nSearching for event with ID 2:\n");
* searchByID(2);
* // Search for events by type
* printf("\nSearching for 'ButtonPress' events:\n");
* searchByType("ButtonPress");
* dequeue();
* display();
* return 0;
* }
* // Enqueue a new event
* void enqueue(int id, const char \*type, const char \*data) {
* struct Event \*newEvent = (struct Event \*)malloc(sizeof(struct Event));
* if (newEvent == NULL) {
* printf("Memory is full\n");
* return;
* }
* // Set event ID, type, and associated data
* newEvent->id = id;
* for (int i = 0; type[i] != '\0'; i++) newEvent->type[i] = type[i];
* newEvent->type[19] = '\0';  // Ensure null termination
* for (int i = 0; data[i] != '\0'; i++) newEvent->data[i] = data[i];
* newEvent->data[99] = '\0';  // Ensure null termination
* newEvent->next = NULL;
* if (front == NULL) {
* front = rear = newEvent;
* } else {
* rear->next = newEvent;
* rear = newEvent;
* }
* }
* // Dequeue the oldest event for handling
* void dequeue() {
* if (front == NULL) {
* printf("No events to handle\n");
* return;
* }
* struct Event \*temp = front;
* front = front->next;
* free(temp);
* }
* // Display all events in the queue
* void display() {
* if (front == NULL) {
* printf("No events in the queue\n");
* return;
* }
* struct Event \*temp = front;
* while (temp) {
* printf("ID: %d, Type: %s, Data: %s\n", temp->id, temp->type, temp->data);
* temp = temp->next;
* }
* }
* // Search for an event by ID
* void searchByID(int id) {
* struct Event \*temp = front;
* int found = 0;
* while (temp) {
* if (temp->id == id) {
* printf("Found Event - ID: %d, Type: %s, Data: %s\n", temp->id, temp->type, temp->data);
* found = 1;
* }
* temp = temp->next;
* }
* if (!found) {
* printf("No events found with ID: %d\n", id);
* }
* }
* // Search for events by type
* void searchByType(const char \*type) {
* struct Event \*temp = front;
* int found = 0;
* while (temp) {
* if (strcmp(temp->type, type) == 0) {
* printf("Found Event - ID: %d, Type: %s, Data: %s\n", temp->id, temp->type, temp->data);
* found = 1;
* }
* temp = temp->next;
* }
* if (!found) {
* printf("No events found with type: %s\n", type);
* }
* }

12. Embedded GUI Event Queue:

* Create a queue using a linked list to manage GUI events (e.g., button clicks, screen touches) in an embedded system. Each event should have an event type, coordinates, and timestamp. Implement enqueue for new GUI events, dequeue for event handling, and search for events by type.
* #include <stdio.h>
* #include <stdlib.h>
* // Structure for GUI Event
* struct GUIEvent {
* char type[20];      // Event type (e.g., "ButtonClick", "ScreenTouch")
* int x, y;           // Coordinates of the event (e.g., button position, screen touch position)
* int timestamp;      // Timestamp of the event
* struct GUIEvent \*next;
* } \*front = NULL, \*rear = NULL;
* // Function prototypes
* void enqueue(const char \*type, int x, int y, int timestamp);
* void dequeue();
* void display();
* void searchByType(const char \*type);
* int main() {
* // Enqueue GUI events
* enqueue("ButtonClick", 50, 100, 101);
* enqueue("ScreenTouch", 150, 200, 102);
* enqueue("ButtonClick", 200, 250, 103);
* display();
* // Search for events by type
* printf("\nSearching for 'ButtonClick' events:\n");
* searchByType("ButtonClick");
* dequeue();
* display();
* return 0;
* }
* // Enqueue a new GUI event
* void enqueue(const char \*type, int x, int y, int timestamp) {
* struct GUIEvent \*newEvent = (struct GUIEvent \*)malloc(sizeof(struct GUIEvent));
* if (newEvent == NULL) {
* printf("Memory is full\n");
* return;
* }
* // Set event type, coordinates, and timestamp
* for (int i = 0; type[i] != '\0'; i++) newEvent->type[i] = type[i];
* newEvent->type[19] = '\0';  // Ensure null termination
* newEvent->x = x;
* newEvent->y = y;
* newEvent->timestamp = timestamp;
* newEvent->next = NULL;
* if (front == NULL) {
* front = rear = newEvent;
* } else {
* rear->next = newEvent;
* rear = newEvent;
* }
* }
* // Dequeue the oldest GUI event for handling
* void dequeue() {
* if (front == NULL) {
* printf("No GUI events to handle\n");
* return;
* }
* struct GUIEvent \*temp = front;
* front = front->next;
* free(temp);
* }
* // Display all GUI events in the queue
* void display() {
* if (front == NULL) {
* printf("No GUI events in the queue\n");
* return;
* }
* struct GUIEvent \*temp = front;
* while (temp) {
* printf("Type: %s, Coordinates: (%d, %d), Timestamp: %d\n", temp->type, temp->x, temp->y, temp->timestamp);
* temp = temp->next;
* }
* }
* // Search for GUI events by type
* void searchByType(const char \*type) {
* struct GUIEvent \*temp = front;
* int found = 0;
* while (temp) {
* if (strcmp(temp->type, type) == 0) {
* printf("Found Event - Type: %s, Coordinates: (%d, %d), Timestamp: %d\n", temp->type, temp->x, temp->y, temp->timestamp);
* found = 1;
* }
* temp = temp->next;
* }
* if (!found) {
* printf("No events found with type: %s\n", type);
* }
* }

13. Serial Communication Buffer:

* Implement a queue using a linked list to buffer data in a serial communication system. Each buffer entry should include data and its length. Enqueue new data chunks, dequeue them for transmission, and search for specific data patterns.
* #include <stdio.h>
* #include <stdlib.h>
* #include <string.h>
* // Structure for serial data chunk
* struct DataChunk {
* char \*data;        // Data in the chunk
* int length;        // Length of the data
* struct DataChunk \*next;
* } \*front = NULL, \*rear = NULL;
* // Function prototypes
* void enqueue(const char \*data, int length);
* void dequeue();
* void display();
* void searchData(const char \*pattern);
* int main() {
* // Enqueue data chunks
* enqueue("Hello", 5);
* enqueue("World", 5);
* enqueue("Serial", 6);
* display();
* // Search for a specific data pattern
* printf("\nSearching for 'World':\n");
* searchData("World");
* dequeue();
* display();
* return 0;
* }
* // Enqueue a new data chunk to the buffer
* void enqueue(const char \*data, int length) {
* struct DataChunk \*newChunk = (struct DataChunk \*)malloc(sizeof(struct DataChunk));
* if (newChunk == NULL) {
* printf("Memory is full\n");
* return;
* }
* newChunk->data = (char \*)malloc((length + 1) \* sizeof(char));  // Allocate memory for data
* if (newChunk->data == NULL) {
* printf("Memory allocation for data failed\n");
* free(newChunk);
* return;
* }
* strncpy(newChunk->data, data, length);  // Copy the data into the chunk
* newChunk->data[length] = '\0';          // Null terminate the string
* newChunk->length = length;
* newChunk->next = NULL;
* if (front == NULL) {
* front = rear = newChunk;
* } else {
* rear->next = newChunk;
* rear = newChunk;
* }
* }
* // Dequeue the oldest data chunk for transmission
* void dequeue() {
* if (front == NULL) {
* printf("Buffer is empty\n");
* return;
* }
* struct DataChunk \*temp = front;
* printf("Transmitting: %s\n", front->data);
* front = front->next;
* free(temp->data);  // Free the memory allocated for the data
* free(temp);
* }
* // Display all data chunks in the buffer
* void display() {
* if (front == NULL) {
* printf("Buffer is empty\n");
* return;
* }
* struct DataChunk \*temp = front;
* while (temp) {
* printf("Data: %s, Length: %d\n", temp->data, temp->length);
* temp = temp->next;
* }
* }
* // Search for a specific data pattern in the buffer
* void searchData(const char \*pattern) {
* struct DataChunk \*temp = front;
* int found = 0;
* while (temp) {
* if (strstr(temp->data, pattern) != NULL) {
* printf("Found pattern '%s' in data: %s\n", pattern, temp->data);
* found = 1;
* }
* temp = temp->next;
* }
* if (!found) {
* printf("Pattern '%s' not found in the buffer\n", pattern);
* }
* }

14. CAN Bus Message Queue:

* Design a queue using a linked list to manage CAN bus messages in an embedded automotive system. Each message should have an ID, data length, and payload. Implement enqueue for incoming messages, dequeue for processing, and search for messages by ID.
* #include <stdio.h>
* #include <stdlib.h>
* // Structure for CAN message
* struct CANMessage {
* int id;
* int dataLength;
* char \*payload;
* struct CANMessage \*next;
* } \*front = NULL, \*rear = NULL;
* // Function prototypes
* void enqueue(int id, int dataLength, const char \*payload);
* void dequeue();
* void display();
* void searchMessageByID(int id);
* int main() {
* // Enqueue CAN bus messages
* enqueue(101, 5, "Data1");
* enqueue(102, 6, "Message");
* enqueue(103, 4, "Test");
* display();
* // Search for a message by ID
* printf("\nSearching for message with ID 102:\n");
* searchMessageByID(102);
* dequeue();
* display();
* return 0;
* }
* // Enqueue a new CAN bus message
* void enqueue(int id, int dataLength, const char \*payload) {
* struct CANMessage \*newMessage = (struct CANMessage \*)malloc(sizeof(struct CANMessage));
* if (newMessage == NULL) {
* printf("Memory allocation failed\n");
* return;
* }
* newMessage->id = id;
* newMessage->dataLength = dataLength;
* newMessage->payload = (char \*)malloc((dataLength + 1) \* sizeof(char));
* if (newMessage->payload == NULL) {
* printf("Memory allocation for payload failed\n");
* free(newMessage);
* return;
* }
* strncpy(newMessage->payload, payload, dataLength);
* newMessage->payload[dataLength] = '\0';
* if (front == NULL) {
* front = rear = newMessage;
* } else {
* rear->next = newMessage;
* rear = newMessage;
* }
* }
* // Dequeue the oldest CAN message for processing
* void dequeue() {
* if (front == NULL) {
* printf("Queue is empty\n");
* return;
* }
* struct CANMessage \*temp = front;
* printf("Processing message with ID: %d, Payload: %s\n", front->id, front->payload);
* front = front->next;
* free(temp->payload);
* free(temp);
* }
* // Display all CAN messages in the queue
* void display() {
* if (front == NULL) {
* printf("Queue is empty\n");
* return;
* }
* struct CANMessage \*temp = front;
* while (temp) {
* printf("ID: %d, Data Length: %d, Payload: %s\n", temp->id, temp->dataLength, temp->payload);
* temp = temp->next;
* }
* }
* // Search for a CAN message by its ID
* void searchMessageByID(int id) {
* struct CANMessage \*temp = front;
* int found = 0;
* while (temp) {
* if (temp->id == id) {
* printf("Found message with ID %d, Payload: %s\n", temp->id, temp->payload);
* found = 1;
* }
* temp = temp->next;
* }
* if (!found) {
* printf("Message with ID %d not found\n", id);
* }
* }

15. Queue Management for Machine Learning Inference:

* Create a queue using a linked list to manage input data for machine learning inference in an embedded system. Each entry should contain input features and metadata. Enqueue new data, dequeue it for inference, and search for specific input data by metadata.
* Each problem requires creating a queue with the following operations using a linked list:
* enqueue: Add new elements to the queue.
* dequeue: Remove and process elements from the queue.
* search: Find elements based on specific criteria.
* display: Show all elements in the queue.
* #include <stdio.h>
* #include <stdlib.h>
* #include <string.h>
* // Structure for ML data
* struct MLData {
* int \*features;
* int numFeatures;
* char \*metadata;
* struct MLData \*next;
* } \*front = NULL, \*rear = NULL;
* // Enqueue data
* void enqueue(int \*features, int numFeatures, const char \*metadata) {
* struct MLData \*newData = (struct MLData \*)malloc(sizeof(struct MLData));
* newData->features = (int \*)malloc(numFeatures \* sizeof(int));
* memcpy(newData->features, features, numFeatures \* sizeof(int));
* // Allocate memory for metadata and copy it
* newData->metadata = (char \*)malloc(strlen(metadata) + 1);
* strcpy(newData->metadata, metadata);
* newData->numFeatures = numFeatures;
* newData->next = NULL;
* if (rear) rear->next = newData;
* rear = newData;
* if (!front) front = newData;
* }
* // Dequeue and process data
* void dequeue() {
* if (!front) {
* printf("Queue is empty.\n");
* return;
* }
* struct MLData \*temp = front;
* printf("Processing: %s\n", front->metadata);
* front = front->next;
* free(temp->features);
* free(temp->metadata);  // Free metadata memory
* free(temp);
* }
* // Search data by metadata
* void search(const char \*metadata) {
* struct MLData \*temp = front;
* while (temp) {
* if (strcmp(temp->metadata, metadata) == 0) {
* printf("Found: %s, Features: ", temp->metadata);
* for (int i = 0; i < temp->numFeatures; i++) {
* printf("%d ", temp->features[i]);
* }
* printf("\n");
* return;
* }
* temp = temp->next;
* }
* printf("Not found: %s\n", metadata);
* }
* // Display the queue
* void display() {
* struct MLData \*temp = front;
* if (!temp) {
* printf("Queue is empty.\n");
* return;
* }
* while (temp) {
* printf("Metadata: %s, Features: ", temp->metadata);
* for (int i = 0; i < temp->numFeatures; i++) {
* printf("%d ", temp->features[i]);
* }
* printf("\n");
* temp = temp->next;
* }
* }
* // Menu-driven interface
* int main() {
* int choice;
* int features[3];
* char metadata[50];
* while (1) {
* printf("\nMenu:\n");
* printf("1. Enqueue data\n");
* printf("2. Dequeue data\n");
* printf("3. Search data by metadata\n");
* printf("4. Display queue\n");
* printf("5. Exit\n");
* printf("Enter your choice: ");
* scanf("%d", &choice);
* switch (choice) {
* case 1:
* printf("Enter metadata: ");
* scanf("%s", metadata);
* printf("Enter number of features: ");
* int numFeatures;
* scanf("%d", &numFeatures);
* printf("Enter features: ");
* for (int i = 0; i < numFeatures; i++) {
* scanf("%d", &features[i]);
* }
* enqueue(features, numFeatures, metadata);
* break;
* case 2:
* dequeue();
* break;
* case 3:
* printf("Enter metadata to search: ");
* scanf("%s", metadata);
* search(metadata);
* break;
* case 4:
* display();
* break;
* case 5:
* exit(0);
* default:
* printf("Invalid choice! Please try again.\n");
* }
* }
* return 0;
* }