מצביעים למצביעים (Pointer to Pointer)

מצביע למצביע הוא משתנה המאחסן כתובת של מצביע אחר. כלומר, מדובר ברמת הפנייה השנייה – במקום שמצביע יכיל כתובת של משתנה רגיל, הוא מכיל כתובת של מצביע אחר.

מצביע למצביע משמש בעיקר כשאנחנו רוצים לשנות מצביע בתוך פונקציה, לעבור על מערכים של מחרוזות, או כאשר עובדים עם הקצאות דינמיות.

הכרזה והגדרה של מצביע למצביע

כדי להגדיר מצביע למצביע, משתמשים בשני כוכביות (**), למשל:

```c
int main() {
    int x = 10;
    int *p = &x;  // p points to x
    int **pp = &p; // pp points to p

    printf("Value of x: %d\n", x);
    printf("Value of x using *p: %d\n", *p);
    printf("Value of x using **pp: %d\n", **pp);

    return 0;
}
```

הסבר:

1. x מכיל את הערך 10.
2. p הוא מצביע ל-x ומכיל את הכתובת של x (&x).
3. pp הוא מצביע למצביע ומכיל את הכתובת של p (&p).
4. *p מחלץ את הערך של x. **pp עושה זאת דרך pp.

שינוי מצביע בתוך פונקציה

מצביע למצביע מאפשר לשנות מצביע שהועבר לפונקציה, למשל:

```c
void allocateMemory(int **ptr) {
    *ptr = (int*)malloc(sizeof(int)); // Allocating memory for an integer
    if (*ptr != NULL) {
        **ptr = 42; // Assigning value
    }
}

int main() {
    int *p = NULL;
    allocateMemory(&p);

    if (p != NULL) {
        printf("Allocated value: %d\n", *p); // Prints 42
        free(p);
    }

    return 0;
}
```

הסבר:

1. allocateMemory(int **ptr) מקבל מצביע למצביע כדי לשנות את p שמחוץ לפונקציה.
2. *ptr = malloc(sizeof(int)) משנה את הכתובת של p ומקצה זיכרון.
3. **ptr = 42 מכניס ערך לזיכרון שהוקצה.

ללא שימוש במצביע למצביע p, לא היה משתנה כלום מחוץ לפונקציה.

בעייתיות של realloc בעת הקצאה מחדש של אותו מצביע

realloc היא פונקציה חשובה ב-C להרחבה או צמצום של אזור זיכרון מוקצה דינמית. עם זאת, שימוש לא נכון בה עלול לגרום **לזליגת זיכרון** (Memory Leak).

מה הבעיה?

כאשר אנו מבצעים realloc(ptr, new_size), קיימת אפשרות שההרחבה לא תצליח. במקרה כזה:

- **realloc מחזירNULL**, אך הכתובת המקורית (ptr) אינה נשמרת.
- אם השמה נעשית ישירות (ptr = realloc(ptr, new_size)), אז **מאבדים את הכתובת המקורית** ואין אפשרות לבצע free(ptr), מה שגורם **לזליגת זיכרון**.

הסבר:

1. malloc מקצה זיכרון ל-5 מספרים שלמים.
2. realloc מנסה להגדיל את הזיכרון ל-10 מספרים.
3. אם ההקצאה נכשלת, realloc מחזיר NULL, אבל arr כבר הוחלף, ולכן אין לנו גישה לזיכרון הישן ולא ניתן לשחרר אותו -> זליגת זיכרון.

```c
int main() {
    int *arr = (int *)malloc(5 * sizeof(int)); // Allocate memory for 5 integers
    if (!arr) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    // Incorrect usage of realloc - potential memory leak
    arr = (int *)realloc(arr, 10 * sizeof(int));
    if (!arr) {
        printf("Reallocation failed! Memory leak occurs here.\n");
        return 1; // The original memory is lost!
    }

    // Use arr safely...
    free(arr);
    return 0;
}
```

הפתרון: שימוש במצביע זמני

כדי למנוע אובדן של הכתובת המקורית, יש להשתמש במצביע עזר:

הסבר:

temp משמש כמצביע עזר:

- אם realloc נכשל (NULL מוחזר), arr נשאר תקף.
- אם realloc הצליח, arr מתעדכן בערך החדש.

במבחן כאשר מצויין שיש לטפל במקרים שבהם הקצאה לא הצליחה יש לבצע טיפול דומה כפי שתואר בקוד אך במקום להתקדם עם ההקצאה המקורית יש לבצע שחרור שלה ועצירה של המשך הקוד(כלומר ביצוע exit)

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

    // Correct usage of realloc
    int *temp = (int *)realloc(arr, 10 * sizeof(int));
    if (!temp) {
        printf("Reallocation failed! Keeping the original memory.\n");
        // The original memory is still valid, so we can safely use arr
    } else {
        arr = temp; // Update arr only if realloc succeeded
    }

    // Use arr safely...
    free(arr);
    return 0;
}
```

מבנים (struct)

מבנה (struct) הוא סוג נתונים מותאם אישית שמאפשר לאגד מספר משתנים (עם סוגים שונים) תחת ישות אחת. זה שימושי כאשר יש צורך בקיבוץ מידע שקשור זה לזה.

הגדרת מבנה בסיסי

הגדרת struct נעשית בעזרת מילת המפתח struct, ואחריה שם המבנה, ואז סוגי הנתונים והמשתנים שבתוכו.

דוגמה למבנה לייצוג נקודה דו-ממדית (Point):

```
struct Point {
    int x;
    int y;
};

int main() {
    struct Point p1 = {3, 4}; // יצירת משתנה מסוג המבנה
    printf("Point coordinates: (%d, %d)\n", p1.x, p1.y);
    return 0;
}
```

הסבר:

1. struct Point מכיל שני משתנים: x ו-y.
2. יצירת מופע (p1) של struct Point ואתחולו לערכים {3, 4}.
3. גישה לשדות המבנה מתבצעת באמצעות הסימון נקודה (dot operator).

הסבר על -> (Arrow Operator)

כאשר יש לנו משתנה שהוא מבנה (struct), ניתן לגשת לשדות שלו בעזרת נקודה (dot operator).

אבל כאשר יש לנו מצביע למבנה, לא ניתן להשתמש ישירות בנקודה מכיוון שהמצביע לא מאחסן את המבנה עצמו אלא את כתובתו בזיכרון.

במקום זאת, יש להשתמש ב--> (Arrow Operator) כדי לגשת לשדות דרך המצביע.

```
struct Point {
    int x, y;
};

int main() {
    struct Point p1 = {10, 20}; // Regular struct variable
    struct Point *pPtr = &p1;   // Pointer to struct

    // Accessing fields using a regular struct
    printf("x: %d, y: %d\n", p1.x, p1.y); // Using dot operator (.)

    // Attempting to use dot operator with a pointer (will cause an error)
    // printf("x: %d\n", pPtr.x); // ❌ Compilation error!

    // Correct way: Using arrow operator (->)
    printf("x: %d, y: %d\n", pPtr->x, pPtr->y); // ✅ Correct usage

    return 0;
}
```

הסבר:

1. במקרה הראשון p1 הוא משתנה רגיל (לא מצביע), ולכן ניתן לגשת לשדות באמצעות נקודה.
2. במקרה השני pPtr הוא מצביע למבנה, לכן אין לו שדה x, וקיימת שגיאת קומפילציה.
3. במקרה האחרון -> מאפשר גישה לשדות של p1 דרך המצביע pPtr.

איך -> בעצם עובד?

השימוש ב-ptr->field הוא קיצור לכתיבה (*ptr).field.

כלומר, שני השורות בקוד הבא זהות לחלוטין:

```
printf("x: %d\n", (*pPtr).x); // דרך ארוכה יותר
printf("x: %d\n", pPtr->x);   // דרך מקוצרת ונחמדה יותר
```

typedef עם struct

ברירת המחדל ב-C דורשת תמיד לכתוב struct לפני שם המבנה, למשל struct Point p1.

שימוש ב-typedef מאפשר שם מקוצר, מה שמקל על קריאות הקוד.

הסבר:

הגדרנו טיפוס חדש בשם Student על ידי שימוש ב-typedef. כעת, במקום לכתוב struct Student, ניתן פשוט להשתמש ב-Student.

```
typedef struct {
    int id;
    char name[50];
} Student;  // Create a new type 'Student' for the struct

int main() {
    Student s1 = {1, "Alice"};  // Create a variable of type 'Student'

    printf("ID: %d, Name: %s\n", s1.id, s1.name);

    return 0;
}
```

שימו לב שניתן לתת שם חדש(typedef) גם לפויינטרים, לדוגמא typedef Point* PointPtr יהווה בעצם פויינטר לטיפוס מסוג Point.

מבנים בתוך מבנים (Nested Structs)

במקרים מסוימים, אנחנו נרצה להגדיר מבנה בתוך מבנה אחר. ניתן לעשות זאת על ידי הצהרה של מבנה בתוך מבנה אחר. כל שדה במבנה יכול להיות מבנה נוסף, ובכך נוכל ליצור מבנים רב-ממדיים ומורכבים יותר.

הסבר:

במקרה הזה, המבנה Person מכיל שדה בשם birthDate שהוא סוג של מבנה נוסף מסוג Date. כך, ניתן לשמור נתונים של תאריך בתוך מבנה Person, והדבר מאפשר לארגן נתונים בצורה היררכית.

```
typedef struct {
    int day;
    int month;
    int year;
} Date;  // Define a new 'Date' structure

typedef struct {
    char name[50];
    Date birthDate;  // Nested struct: 'birthDate' is of type 'Date'
} Person;  // Define a new 'Person' structure

int main() {
    Person p1 = {"Alice", {15, 3, 1995}};  // Initialize Person with a nested struct for Date

    printf("Name: %s\n", p1.name);
    printf("Birth Date: %02d/%02d/%d\n", p1.birthDate.day, p1.birthDate.month, p1.birthDate.year);

    return 0;
}
```

נקודות נוספות שכדאי לשים לב אליהם

- ההמלצה הכללית היא לסדר את השדות במבנה מהגדול לקטן, כי זה בדרך כלל מונע מהוספת פדינג מיותר.
- שימו לב שכמו טיפוסים פשוטים, כאשר מעבירים מבנה לפונקציה הוא מועבר by value כלומר נוצר העתק שכל שינוי בו לא ישפיע על המקור.
כדי להימנע מהעתקים של מורכבים שעלולים לגרום לשימוש גבוה בזיכרון לרוב נעביר מבנים בין פונקציות בעזרת פויינטרים(שימו לב שבמקרה כזו יש להשתמש באופרטור חץ ולא באופרטור נקודה לגישה לתוך נתונים המבנה).

רשימות מקושרות (Linked Lists) בשפת C

1. מבוא לרשימות מקושרות

- מהי רשימה מקושרת?

רשימה מקושרת היא מבנה נתונים לינארי ודינאמי. בניגוד למערכים (Arrays), שגודלם נקבע מראש והם מאוחסנים ברצף בזיכרון, רשימה מקושרת מורכבת מאוסף של צמתים (Nodes). כל צומת מכיל שני חלקים עיקריים:

1. **נתונים (Data):** המידע שאנו רוצים לאחסן (יכול להיות מטיפוס פשוט כמו int או float, או מטיפוס מורכב כמו struct).
2. **מצביע (Pointer):** כתובת בזיכרון המצביעה על הצומת הבא ברשימה. בצומת האחרון, המצביע יהיה NULL, המסמן את סוף הרשימה.

הצמתים אינם חייבים להיות מאוחסנים ברצף בזיכרון; הקישור ביניהם נוצר באמצעות המצביעים.

- **למה להשתמש ברשימות מקושרות?**
  - **גודל דינאמי:** ניתן להוסיף או להסיר צמתים בקלות בזמן ריצה, מבלי צורך להקצות מחדש את כל המבנה כמו במערך דינאמי.
  - **הכנסה ומחיקה יעילות (באמצע הרשימה):** הכנסה או מחיקה של איבר באמצע הרשימה דורשת רק עדכון של מספר מצביעים (בדרך כלל פעולה בזמן קבוע, O(1), *אם* יש לנו מצביע למקום הרצוי או לצומת שלפניו), לעומת הזזה של איברים רבים במערך (פעולה בזמן לינארי, O(n)).
  - **חסרונות:** גישה לאיבר ספציפי (לפי אינדקס) דורשת מעבר סדרתי מתחילת הרשימה (פעולה בזמן לינארי, O(n)), בניגוד לגישה ישירה בזמן קבוע (O(1)) במערך. כמו כן, רשימות מקושרות דורשות זיכרון נוסף עבור המצביעים.
- מושג בסיסי: מבנה הצומת

אנו מתחילים בהגדרת מבנה (struct) שייצג צומת בודד.

2. רשימה מקושרת פשוטה (Singly Linked List)

ברשימה מקושרת פשוטה, כל צומת מצביע רק על הצומת הבא אחריו.

- הגדרת מבנה הצומת (Node Structure):

נתחיל עם צומת המכיל נתון מסוג int.

```
#include <stdio.h>
#include <stdlib.h> // For malloc and free

// Define the structure for a node
typedef struct Node {
    int data;           // Data stored in the node
    struct Node *next;  // Pointer to the next node in the list
} Node;

// It's common practice to use typedef to avoid writing 'struct Node' repeatedly.

```

- יצירת רשימה (ריקה):

רשימה מיוצגת בדרך כלל על ידי מצביע לצומת הראשון שלה, הנקרא head. רשימה ריקה מיוצגת על ידי head המצביע ל-NULL.

```
Node *head = NULL; // Initialize an empty list

```

- יצירת צומת חדש:

פונקציה ליצירת צומת חדש תקצה זיכרון דינאמי (malloc) ותאתחל את הנתונים והמצביע.

```
// Function to create a new node
Node* createNode(int data) {
    // Allocate memory for the new node
    Node* newNode = (Node*)malloc(sizeof(Node));

    // Check if memory allocation was successful
    if (newNode == NULL) {
        perror("Memory allocation failed");
        exit(EXIT_FAILURE); // Exit if allocation fails
    }

    // Initialize the node's data and next pointer
    newNode->data = data;
    newNode->next = NULL; // The new node initially points to nothing

    return newNode;
}

```

חשוב: תמיד לבדוק אם malloc הצליח להקצות זיכרון.

- **פעולות הכנסה (Insertion):**
  - הכנסה לתחילת הרשימה (Insert at Beginning):

זו הפעולה היעילה ביותר (O(1)). יוצרים צומת חדש, גורמים לו להצביע על ה-head הנוכחי, ומעדכנים את head כך שיצביע על הצומת החדש.

```
// Function to insert a node at the beginning of the list
void insertAtBeginning(Node** headRef, int data) {
    // Create the new node
    Node* newNode = createNode(data);

    // Make the new node point to the current head
    newNode->next = *headRef;

    // Update the head to point to the new node
    *headRef = newNode;
}
// Note: We pass Node** (pointer to a pointer) so we can modify the original head pointer.

```

- הכנסה לסוף הרשימה (Insert at End):

צריך לעבור על כל הרשימה עד שמגיעים לצומת האחרון (זה שהמצביע next שלו הוא NULL), ואז לגרום לו להצביע על הצומת החדש. אם הרשימה ריקה, הצומת החדש הופך להיות ה-head. פעולה זו היא בזמן לינארי (O(n)).

```
// Function to insert a node at the end of the list
void insertAtEnd(Node** headRef, int data) {
    // Create the new node
    Node* newNode = createNode(data);

    // If the list is empty, the new node becomes the head
    if (*headRef == NULL) {
        *headRef = newNode;
        return;
    }

    // Traverse to the last node
    Node* current = *headRef;
    while (current->next != NULL) {
        current = current->next;
    }

    // Make the last node point to the new node
    current->next = newNode;
}

```

- הכנסה במיקום ספציפי (Insert at Specific Position):

דורש מעבר עד לצומת שלפני המיקום הרצוי, ואז עדכון המצביעים כך שהצומת החדש ישתלב בין הצומת הקודם לצומת הבא.

- פעולות מחיקה (Deletion):

מחיקה דורשת שני שלבים עיקריים: עדכון המצביעים כדי "לעקוף" את הצומת הנמחק, ושחרור הזיכרון של הצומת הנמחק באמצעות free. חשוב מאוד לשחרר זיכרון כדי למנוע דליפות זיכרון (Memory Leaks).

- מחיקת הצומת הראשון (Delete First Node):

שומרים מצביע זמני לצומת השני, משחררים את ה-head המקורי, ומעדכנים את head כך שיצביע על הצומת השני. פעולה זו היא O(1).

```
// Function to delete the first node of the list
void deleteFirstNode(Node** headRef) {
    // Check if the list is empty
    if (*headRef == NULL) {
        printf("List is empty, nothing to delete.\n");
        return;
    }

    // Store the node to be deleted (the current head)
    Node* temp = *headRef;

    // Update the head to point to the next node
    *headRef = (*headRef)->next;

    // Free the memory of the original head node
    printf("Deleting node with data: %d\n", temp->data);
    free(temp);
}

```

- מחיקת הצומת האחרון (Delete Last Node):

צריך לעבור על הרשימה תוך שמירת מצביע לצומת שלפני הצומת האחרון. לאחר מכן, מעדכנים את המצביע next של הצומת הלפני אחרון ל-NULL, ומשחררים את הצומת האחרון. פעולה זו היא O(n). יש לטפל במקרה קצה של רשימה עם צומת אחד בלבד.

```
// Function to delete the last node of the list
void deleteLastNode(Node** headRef) {
    // Check for empty list
    if (*headRef == NULL) {
         printf("List is empty, nothing to delete.\n");
        return;
    }

    // Handle list with only one node
    if ((*headRef)->next == NULL) {
        printf("Deleting node with data: %d\n", (*headRef)->data);
        free(*headRef);
        *headRef = NULL; // List is now empty
        return;
    }

    // Traverse to the second-to-last node
    Node* current = *headRef;
    while (current->next->next != NULL) {
        current = current->next;
    }

    // Now 'current' points to the second-to-last node
    Node* nodeToDelete = current->next;
    printf("Deleting node with data: %d\n", nodeToDelete->data);
    free(nodeToDelete); // Free the last node
    current->next = NULL; // Update the new last node's next pointer
}

```

- מחיקת צומת עם ערך מסוים (Delete Node by Value):

עוברים על הרשימה. אם מוצאים את הצומת עם הערך הרצוי, צריך לעדכן את המצביע next של הצומת הקודם לו כך שיצביע על הצומת שאחרי הצומת הנמחק. לאחר מכן, משחררים את הצומת. יש לטפל במקרה קצה שהצומת למחיקה הוא ה-head. פעולה זו היא O(n).

```
// Function to delete a node with a specific value
void deleteNodeByValue(Node** headRef, int valueToDelete) {
    Node* current = *headRef;
    Node* previous = NULL;

    // Traverse the list to find the node
    while (current != NULL && current->data != valueToDelete) {
        previous = current;
        current = current->next;
    }

    // If the node was not found
    if (current == NULL) {
        printf("Node with value %d not found.\n", valueToDelete);
        return;
    }

    // If the node to delete is the head node
    if (previous == NULL) { // Same as: if (current == *headRef)
        *headRef = current->next; // Update head
    } else {
        // Link the previous node to the node after the current one
        previous->next = current->next;
    }

    // Free the memory of the found node
    printf("Deleting node with data: %d\n", current->data);
    free(current);
}

```

- מעבר על הרשימה / הדפסה (Traversal / Printing):

מתחילים מה-head ועוקבים אחרי מצביעי next עד שמגיעים ל-NULL.

```
// Function to print the elements of the list
void printList(Node* head) {
    Node* current = head;
    printf("List: ");
    while (current != NULL) {
        printf("%d -> ", current->data);
        current = current->next; // Move to the next node
    }
    printf("NULL\n");
}

```

- חיפוש (Searching):

דומה למעבר, עוברים על הרשימה עד שמוצאים צומת עם הנתון המבוקש או מגיעים לסוף הרשימה.

```
// Function to search for a value in the list
// Returns a pointer to the node if found, NULL otherwise
Node* search(Node* head, int valueToFind) {
    Node* current = head;
    while (current != NULL) {
        if (current->data == valueToFind) {
            return current; // Value found
        }
        current = current->next;
    }
    return NULL; // Value not found
}

```

- שחרור כל הזיכרון של הרשימה (Freeing the Entire List):

חשוב מאוד לשחרר את הזיכרון של כל הצמתים כאשר הרשימה אינה נחוצה יותר, כדי למנוע דליפות זיכרון.
עוברים על הרשימה, שומרים מצביע לצומת הבא לפני שמשחררים את הנוכחי.

```
// Function to free all nodes in the list
void freeList(Node** headRef) {
    Node* current = *headRef;
    Node* nextNode;

    while (current != NULL) {
        nextNode = current->next; // Store pointer to the next node
        printf("Freeing node with data: %d\n", current->data);
        free(current);           // Free the current node
        current = nextNode;        // Move to the next node
    }

    *headRef = NULL; // Ensure the head pointer is NULL after freeing
}

```

3. רשימה מקושרת עם טיפוסי נתונים מורכבים

הרעיון זהה, אך במקום int, שדה הנתונים (data) בצומת יכיל struct או מצביע ל-struct.

- הגדרת מבנה הצומת:

לדוגמה, רשימה של סטודנטים:

```
// Define a structure for student data
typedef struct Student {
    int id;
    char name[50];
    // Add other fields as needed
} Student;

// Define the structure for a node containing student data
typedef struct StudentNode {
    Student data;               // Store the Student struct directly
    // OR: Student *data;      // Store a pointer to a Student struct (requires separate allocation for Student)
    struct StudentNode *next;   // Pointer to the next node
} StudentNode;

// Functions would now operate on StudentNode* and Student data
// e.g., StudentNode* createStudentNode(Student studentData);
// void insertStudentAtBeginning(StudentNode** headRef, Student studentData);

```

- **שינויים בפעולות:**
  - **יצירת צומת:** צריך לאתחל את כל שדות ה-struct של הנתונים. אם שדה הנתונים הוא מצביע ל-struct, צריך להקצות זיכרון גם עבור ה-struct עצמו.
  - **השוואה/חיפוש:** ההשוואה תתבצע לפי שדה ספציפי ב-struct (למשל, id של הסטודנט).
  - **הדפסה:** יש להדפיס את השדות הרלוונטיים מה-struct.
  - **שחרור זיכרון:** אם שדה הנתונים הוא מצביע ל-struct שהוקצה דינמית, יש לשחרר גם את הזיכרון של ה-struct לפני שמשחררים את הצומת.

4. רשימה מקושרת כפולה / דו כיוונית (Doubly Linked List)

ברשימה מקושרת כפולה, כל צומת מכיל *שני* מצביעים:

1. next: מצביע לצומת הבא.
2. prev: מצביע לצומת הקודם.

הצומת הראשון (head) יכיל prev = NULL, והצומת האחרון יכיל next = NULL.

- **הגדרת מבנה הצומת:**

```
// Define the structure for a doubly linked list node
typedef struct DNode {
    int data;
    struct DNode *next; // Pointer to the next node
    struct DNode *prev; // Pointer to the previous node
} DNode;

```

- **יתרונות:**
  - מאפשרת מעבר על הרשימה בשני הכיוונים (קדימה ואחורה).
  - מחיקה של צומת (אם יש לנו מצביע אליו) היא יעילה יותר (O(1)) מכיוון שניתן לגשת ישירות לצומת הקודם דרך מצביע ה-prev, ללא צורך במעבר מההתחלה או שמירת מצביע קודם במהלך המעבר.
- **חסרונות:**
  - דורשת יותר זיכרון לכל צומת (עבור המצביע הנוסף).
  - פעולות ההכנסה והמחיקה מעט מורכבות יותר, מכיוון שצריך לעדכן שני מצביעים (next ו-prev) בכל שינוי קישוריות.
- פעולות (דוגמה - הכנסה להתחלה):

יש לעדכן גם את מצביע prev של ה-head הישן (אם קיים) וגם את מצביעי next ו-prev של הצומת החדש.

```
// Function to insert a node at the beginning of a doubly linked list
void insertAtBeginningDLL(DNode** headRef, int data) {
    // Allocate memory for the new node
    DNode* newNode = (DNode*)malloc(sizeof(DNode));
    if (newNode == NULL) {
        perror("Memory allocation failed");
        exit(EXIT_FAILURE);
    }
    newNode->data = data;
    newNode->prev = NULL; // The new node's prev is always NULL at the beginning

    // Link the new node to the current head
    newNode->next = *headRef;

    // If the list was not empty, update the previous head's prev pointer
    if (*headRef != NULL) {
        (*headRef)->prev = newNode;
    }

    // Update the head to point to the new node
    *headRef = newNode;
}

```

באופן דומה, פעולות הכנסה אחרות ומחיקה ידרשו עדכון קפדני של מצביעי next ו-prev של הצמתים המעורבים.

5. רשימה מקושרת מעגלית (Circular Linked List)

ברשימה מעגלית, המצביע next של הצומת האחרון אינו NULL, אלא מצביע בחזרה לצומת הראשון (head).

- **סוגים:** יכולה להיות רשימה מעגלית פשוטה (עם מצביע next בלבד) או רשימה מעגלית כפולה (עם next ו-prev, כאשר prev<-head מצביע על הצומת האחרון, ו-next<-last מצביע על ה-head).
- **הגדרת מבנה הצומת:** זהה לרשימה פשוטה או כפולה. ההבדל הוא בלוגיקה של הפעולות ובניהול המצביעים.
- מעבר על הרשימה:

התנאי לעצירת המעבר משתנה. במקום לבדוק current != NULL, מתחילים מה-head ועוצרים כשחוזרים אליו שוב. יש לטפל במקרה של רשימה ריקה.

```
// Function to print a circular linked list (singly)
void printCircularList(Node* head) {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

    Node* current = head;
    printf("Circular List: ");
    do {
        printf("%d -> ", current->data);
        current = current->next;
    } while (current != head); // Stop when we loop back to the head
    printf("(back to head)\n");
}

```

- **יתרונות:**
  - שימושי למבנים הדורשים התנהגות מעגלית, כמו תזמון תהליכים בשיטת Round Robin.
  - כל צומת "נגיש" מכל צומת אחר.
  - ניתן לייצג את הרשימה באמצעות מצביע לצומת *האחרון* דווקא, מה שמאפשר גישה נוחה גם לראשון (next<-last) וגם לאחרון (last) בזמן O(1).
- **פעולות:** הכנסה ומחיקה דורשות תשומת לב מיוחדת לעדכון המצביע של הצומת האחרון כך שימשיך להצביע על ה-head (או לעדכון מצביעי prev ברשימה מעגלית כפולה).

6. נושאים נוספים ושיקולים

- **צומת דמה (Header/Dummy Node):** לפעמים מוסיפים צומת "דמה" קבוע בתחילת הרשימה (ולפעמים גם בסופה). צומת זה אינו מכיל נתונים רלוונטיים, אך הוא מפשט את הלוגיקה של פעולות הכנסה ומחיקה, מכיוון שהוא מבטל את הצורך לטפל במקרים מיוחדים של שינוי ה-head או פעולה על רשימה ריקה (הפעולות תמיד מתבצעות "אחרי" צומת הדמה).
- **סיבוכיות זמן ומקום:**
  - **מקום:** O(n) - הזיכרון גדל לינארית עם מספר האיברים (n).
  - **זמן (ממוצע/גרוע ביותר):**
    - גישה לאיבר i: O(n)
    - חיפוש: O(n)
    - הכנסה/מחיקה בתחילה (עם מצביע ל-head): O(1)
    - הכנסה בסוף (עם מצביע ל-head בלבד): O(n) לרשימה פשוטה, (אפשר O(1) אם שומרים מצביע לזנב).
    - הכנסה/מחיקה באמצע (עם מצביע לצומת הקודם/הנוכחי): O(1) (לא כולל זמן החיפוש).
- **השוואה למערכים:** בחירה בין מערך לרשימה מקושרת תלויה בשימוש הצפוי: גישה תכופה לפי אינדקס? מערך עדיף. הכנסות/מחיקות תכופות באמצע? רשימה מקושרת עדיפה. גודל לא ידוע מראש? רשימה מקושרת גמישה יותר.
- **מלכודות נפוצות:**
  - **דליפות זיכרון:** שכחה לשחרר זיכרון (free) שהוקצה עם malloc.
  - **Dangling Pointers:** שימוש במצביע לאחר שהזיכרון שאליו הוא הצביע שוחרר.
  - **טעויות Off-by-one:** טעויות בלולאות מעבר, במיוחד בעת מחיקה או הכנסה.
  - **טיפול שגוי ב-NULL:** אי בדיקה של מצביעי NULL לפני גישה לשדות (current->next כש-current הוא NULL).
  - **איבוד ה-head:** שינוי head מבלי לשמור מצביע לצומת הבא בעת מחיקת הצומת הראשון.

מערכים דו מימדיים דינאמיים

מערך דו־ממדי דינאמי מאפשר להקצות ולהשתחרר מזיכרון בזמן ריצה, בהתאם למספר השורות והעמודות שהמשתמש קובע. שימוש במטריצה דינאמית גמיש במיוחד כאשר הגודל אינו ידוע מראש או משתנה במהלך התכנית.

הקצאה באמצעות מצביעים לשורות

בקורס הזה נבצע הקצאה של מערכים דו מימדיים בעזרת הקצאת מערך פויינטרים שכל פויינטר במערך מצביע למערך איברים, סדר הפעולות הוא:

1. מקצים מערך של מצביעים בגודל rows.
2. עבור כל שורה מקצים בלוק בגודל cols.
3. ניגשים לכל תא בצורה טבעית בעזרת A[i][j].
4. בסיום השימוש משחררים בלולאה כל שורה במערך השורות.
5. לאחר שחרור השורות משחררים את מערך השורות.

יצירת מטריצה דינאמית (create_matrix)

```
int **create_matrix(int rows, int cols) {
    // Allocate array of row pointers
    int **M = (int**)malloc(rows * sizeof(int*));

    // Allocate each row
    for (int i = 0; i < rows; i++) {
        M[i] = (int*)malloc(cols * sizeof(int));
    }

    return M;
}
```

הסבר:

1. מקצים תחילה מערך של rows מצביעים מסוג int*.
2. לכל אינדקס i מקצים בלוק בגודל cols * sizeof(int).
3. מחזירים מצביע ל־int** שמייצג את המטריצה.

שחרור זיכרון (free_matrix)

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

הסבר:

1. עוברים על כל שורה ומשחררים את הבלוק שהוקצה לה.
2. בסיום משחררים את מערך המצביעים עצמו.

פונקציות לבדיקת תווים (ספריית ctype.h)

פונקציות אלו משמשות לבדיקת סוגים שונים של תווים. הן מקבלות תו (מסוג int, אך בדרך כלל מייצג תו ASCII) ומחזירות ערך שונה מאפס (אמת) אם התו עונה על התנאי, ואפס (שקר) אחרת.

1. int isdigit(int c);
   - **תיאור:** בודקת האם התו c הוא ספרה (0-9).
   - **דוגמה:**

```
1. #include <stdio.h>
2. #include <ctype.h>
3.
4. int main() {
5.     char ch1 = '7';
6.     char ch2 = 'a';
7.     if (isdigit(ch1)) {
8.         printf("Character '%c' is a digit.\n", ch1); // Output: Character '7' is a digit.
9.     }
10.     if (!isdigit(ch2)) {
11.         printf("Character '%c' is not a digit.\n", ch2); // Output: Character 'a' is not a
digit.
12.     }
13.     return 0;
14. }
15.
```

2. int isalpha(int c);
   - **תיאור:** בודקת האם התו c הוא אות (a-z או A-Z).
   - **דוגמה:**

```
1. #include <stdio.h>
2. #include <ctype.h>
3.
4. int main() {
5.     char ch1 = 'X';
6.     char ch2 = '3';
7.     if (isalpha(ch1)) {
8.         printf("Character '%c' is an alphabet.\n", ch1); // Output: Character 'X' is an
alphabet.
9.     }
10.     if (!isalpha(ch2)) {
11.         printf("Character '%c' is not an alphabet.\n", ch2); // Output: Character '3' is not
an alphabet.
12.     }
13.     return 0;
14. }
15.
```

3. int isalnum(int c);
   - **תיאור :**בודקת האם התו c הוא אות או ספרה.
   - **דוגמה :**

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

int main() {
    char ch1 = 'F';
    char ch2 = '9';
    char ch3 = '!';
    if (isalnum(ch1)) {
        printf("Character '%c' is alphanumeric.\n", ch1); // Output: Character 'F' is alphanumeric.
    }
    if (isalnum(ch2)) {
        printf("Character '%c' is alphanumeric.\n", ch2); // Output: Character '9' is alphanumeric.
    }
    if (!isalnum(ch3)) {
        printf("Character '%c' is not alphanumeric.\n", ch3); // Output: Character '!' is not alphanumeric.
    }
    return 0;
}

```

4. int isxdigit(int c);
   - **תיאור :**בודקת האם התו c הוא ספרה הקסדצימלית (0-9 או A-F).
   - **דוגמה :**

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

int main() {
    char ch1 = 'C';
    char ch2 = '5';
    char ch3 = 'g';
    if (isxdigit(ch1)) {
        printf("Character '%c' is a hexadecimal digit.\n", ch1); // Output: Character 'C' is a hexadecimal digit.
    }
    if (isxdigit(ch2)) {
        printf("Character '%c' is a hexadecimal digit.\n", ch2); // Output: Character '5' is a hexadecimal digit.
    }
    if (!isxdigit(ch3)) {
        printf("Character '%c' is not a hexadecimal digit.\n", ch3); // Output: Character 'g' is not a hexadecimal digit.
    }
    return 0;
}

```

5. int islower(int c);
   - **תיאור:** בודקת האם התו c הוא אות קטנה (a-z).
   - **דוגמה :**

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

int main() {
    char ch1 = 'd';
    char ch2 = 'D';
    if (islower(ch1)) {
        printf("Character '%c' is a lowercase letter.\n", ch1); // Output: Character 'd' is a lowercase letter.
    }
    if (!islower(ch2)) {
        printf("Character '%c' is not a lowercase letter.\n", ch2); // Output: Character 'D' is not a lowercase letter.
    }
    return 0;
}

```

6. int isupper(int c);
   - **תיאור:** בודקת האם התו c הוא אות גדולה (A-Z).
   - **דוגמה :**

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

int main() {
    char ch1 = 'Q';
    char ch2 = 'q';
    if (isupper(ch1)) {
        printf("Character '%c' is an uppercase letter.\n", ch1); // Output: Character 'Q' is an uppercase letter.
    }
    if (!isupper(ch2)) {
        printf("Character '%c' is not an uppercase letter.\n", ch2); // Output: Character 'q' is not an uppercase letter.
    }
    return 0;
}

```

7. int tolower(int c);
   - **תיאור:** אם התו c הוא אות גדולה, הפונקציה מחזירה את האות הקטנה המקבילה. אחרת, היא מחזירה את התו המקורי ללא שינוי.
   - **דוגמה :**

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

int main() {
    char ch1 = 'H';
    char ch2 = 'h';
    char ch3 = '5';
    printf("tolower('%c') -> %c\n", ch1, tolower(ch1)); // Output: tolower('H') -> h
    printf("tolower('%c') -> %c\n", ch2, tolower(ch2)); // Output: tolower('h') -> h
    printf("tolower('%c') -> %c\n", ch3, tolower(ch3)); // Output: tolower('5') -> 5
    return 0;
}

```

8. int toupper(int c);
   - **תיאור :**אם התו c הוא אות קטנה, הפונקציה מחזירה את האות הגדולה המקבילה. אחרת, היא מחזירה את התו המקורי ללא שינוי.
   - **דוגמה :**

```
1. #include <stdio.h>
2. #include <ctype.h>
3.
4. int main() {
5.     char ch1 = 'h';
6.     char ch2 = 'H';
7.     char ch3 = '?';
8.     printf("toupper('%c') -> %c\n", ch1, toupper(ch1)); // Output: toupper('h') -> H
9.     printf("toupper('%c') -> %c\n", ch2, toupper(ch2)); // Output: toupper('H') -> H
10.    printf("toupper('%c') -> %c\n", ch3, toupper(ch3)); // Output: toupper('?') -> ?
11.    return 0;
12. }
13.
```

9. int isspace(int c);
   - **תיאור :**בודקת האם התו c הוא תו רווח לבן (כמו רווח, טאב t\, ירידת שורה n\ וכו׳).
   - **דוגמה :**

```
1. #include <stdio.h>
2. #include <ctype.h>
3.
4. int main() {
5.     char ch1 = ' ';
6.     char ch2 = '\n';
7.     char ch3 = 'A';
8.     if (isspace(ch1)) {
9.         printf("Character ' ' is a whitespace character.\n"); // Output: Character ' ' is a
whitespace character.
10.    }
11.    if (isspace(ch2)) {
12.        printf("Character '\\n' is a whitespace character.\n"); // Output: Character '\n' is
a whitespace character.
13.    }
14.    if (!isspace(ch3)) {
15.        printf("Character '%c' is not a whitespace character.\n", ch3); // Output: Character
'A' is not a whitespace character.
16.    }
17.    return 0;
18. }
19.
```

10. int iscntrl(int c);
- **תיאור :**בודקת האם התו c הוא תו בקרה (תו שאינו מודפס כמו t\ n\ וכו').
- **דוגמה :**

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

int main() {
    char ch1 = '\t'; // Tab character
    char ch2 = 'A';
    if (iscntrl(ch1)) {
        printf("Character '\\t' is a control character.\n"); // Output: Character '\t' is a control character.
    }
    if (!iscntrl(ch2)) {
        printf("Character '%c' is not a control character.\n", ch2); // Output: Character 'A' is not a control character.
    }
    return 0;
}

```

11. int ispunct(int c);
- **תיאור :**בודקת האם התו c הוא תו פיסוק (תו הדפסה שאינו רווח, ספרה או אות).
- **דוגמה :**

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

int main() {
    char ch1 = ';';
    char ch2 = 'A';
    char ch3 = ' ';
    if (ispunct(ch1)) {
        printf("Character '%c' is a punctuation character.\n", ch1); // Output: Character ';' is a punctuation character.
    }
    if (!ispunct(ch2)) {
        printf("Character '%c' is not a punctuation character.\n", ch2); // Output: Character 'A' is not a punctuation character.
    }
    if (!ispunct(ch3)) {
        printf("Character ' ' is not a punctuation character.\n"); // Output: Character ' ' is not a punctuation character.
    }
    return 0;
}

```

12. int isprint(int c);
- **תיאור:** בודקת האם התו c הוא תו הניתן להדפסה (כולל רווח).
- **דוגמה:**

```
1. #include <stdio.h>
2. #include <ctype.h>
3.
4. int main() {
5.     char ch1 = 'k';
6.     char ch2 = ' ';
7.     char ch3 = '\n'; // Newline is not a printable character
8.     if (isprint(ch1)) {
9.         printf("Character '%c' is a printable character.\n", ch1); // Output: Character 'k'
is a printable character.
10.     }
11.     if (isprint(ch2)) {
12.         printf("Character ' ' is a printable character.\n"); // Output: Character ' ' is a
printable character.
13.     }
14.     if (!isprint(ch3)) {
15.         printf("Character '\\n' is not a printable character.\n"); // Output: Character '\n'
is not a printable character.
16.     }
17.     return 0;
18. }
19.
```

13. int isgraph(int c);
- **תיאור:** בודקת האם התו c הוא תו הניתן להדפסה (לא כולל רווח).
- **דוגמה:**

```
1. #include <stdio.h>
2. #include <ctype.h>
3.
4. int main() {
5.     char ch1 = 'k';
6.     char ch2 = '$';
7.     char ch3 = ' '; // Space is not a graphical character
8.     if (isgraph(ch1)) {
9.         printf("Character '%c' is a graphical character.\n", ch1); // Output: Character 'k'
is a graphical character.
10.     }
11.     if (isgraph(ch2)) {
12.         printf("Character '%c' is a graphical character.\n", ch2); // Output: Character '$'
is a graphical character.
13.     }
14.     if (!isgraph(ch3)) {
15.         printf("Character ' ' is not a graphical character.\n"); // Output: Character ' ' is
not a graphical character.
16.     }
17.     return 0;
18. }
19.
```

פונקציות להמרת מחרוזות למספרים (ספריית stdlib.h)

פונקציות אלו ממירות מחרוזת (המייצגת מספר) לטיפוס מספרי מתאים.

1. double atof(const char *nPtr);
   - **תיאור:** ממירה את המחרוזת nPtr למספר מסוג double.
   - **דוגמה:**

```
1. #include <stdio.h>
2. #include <stdlib.h>
3.
4. int main() {
5.     char str[] = "123.456";
6.     double num = atof(str);
7.     printf("String: \"%s\", Number: %f\n", str, num); // Output: String: "123.456", Number: 123.456000
8.     return 0;
9. }
10.
```

2. int atoi(const char *nPtr);
   - **תיאור:** ממירה את המחרוזת nPtr למספר מסוג int.
   - **דוגמה:**

```
1. #include <stdio.h>
2. #include <stdlib.h>
3.
4. int main() {
5.     char str[] = "789";
6.     int num = atoi(str);
7.     printf("String: \"%s\", Number: %d\n", str, num); // Output: String: "789", Number: 789
8.     return 0;
9. }
10.
```

3. long atol(const char *nPtr);
   - **תיאור:** ממירה את המחרוזת nPtr למספר מסוג long int.
   - **דוגמה:**

```
1. #include <stdio.h>
2. #include <stdlib.h>
3.
4. int main() {
5.     char str[] = "1234567890";
6.     long num = atol(str);
7.     printf("String: \"%s\", Number: %ld\n", str, num); // Output: String: "1234567890", Number: 1234567890
8.     return 0;
9. }
10.
```

פונקציות לטיפול במחרוזות (ספריית string.h)

פונקציות אלו משמשות לביצוע פעולות שונות על מחרוזות (מערכים של תווים המסתיימים בתו \0).

העתקה ושירשור מחרוזות

1. char *strcpy(char *s1, const char *s2);
   - **תיאור :**מעתיקה את המחרוזת s2 לתוך המחרוזת s1(כולל התו \0).
   - **חשוב :**יש לוודא שs1 גדולה מספיק כדי להכיל את s2.
   - **דוגמה :**

```
1. #include <stdio.h>
2. #include <string.h>
3.
4. int main() {
5.     char source[] = "Hello World";
6.     char destination[20] = “example”;
7.     strcpy(destination, source);
8.     printf("Source string: %s\n", source);
9.     printf("Destination string: %s\n", destination); // Output: Destination string: Hello
World
10.     return 0;
11. }
12.
```

2. char *strncpy(char *s1, const char *s2, size_t n);
   - **תיאור :**מעתיקה לכל היותר n תווים מהמחרוזת s2 לתוך המחרוזת s1 ומחזירה מצביע לs1.
   - **חשוב :**אם אורך s2 קטן מ-n יתווספו תווי \0 עד הגעה לn. אם אורך s2 גדול שווה מ-n התו המסיים \0 לא יועתק אוטומטית אם הוא לא נמצא בתוך n התווים הראשונים ולכן אם יש ספק כדאי להוסיף אותו ידנית.
   - **דוגמה :**

```
1. #include <stdio.h>
2. #include <string.h>
3.
4. int main() {
5.     char source[] = "Hello World";
6.     char destination[10];
7.     strncpy(destination, source, 5);
8.     destination[5] = '\0'; // Ensure null termination
9.     printf("Destination string (5 chars): %s\n", destination); // Output: Destination string
(5 chars): Hello
10.
11.     strncpy(destination, source, 9);
12.     destination[sizeof(destination) - 1] = '\0'; // Safer way to ensure null termination
13.     printf("Destination string (9 chars): %s\n", destination); // Output: Destination string
(9 chars): Hello Wor
14.     return 0;
15. }
16.
```

3. char *strcat(char *s1, const char *s2);
   - **תיאור :**משרשרת (מוסיפה בסוף) את המחרוזת s2 למחרוזת s1. התו הראשון של s2 דורס את התו \0 של המחרוזת s1. הערך המוחזר הוא מצביע לs1.
   - **חשוב :**יש לוודא ש-s1 גדולה מספיק כדי להכיל את שתי המחרוזות יחד ואת התו \0 המסיים.
   - **דוגמה :**

```
1. #include <stdio.h>
2. #include <string.h>
3.
4. int main() {
5.     char str1[20] = "Hello ";
6.     char str2[] = "World!";
7.     strcat(str1, str2);
8.     printf("Concatenated string: %s\n", str1); // Output: Concatenated string: Hello World!
9.     return 0;
10. }
11.
```

4. char *strncat(char *s1, const char *s2, size_t n);
   - **תיאור :**משרשרת לכל היותר n תווים מהמחרוזת s2 למחרוזת s1 ומוסיפה תו \0 בסוף. הערך המוחזר הוא מצביע לs1.
   - **חשוב :**יש לוודא ש-s1 גדול מספיק.
   - **דוגמה :**

```
1. #include <stdio.h>
2. #include <string.h>
3.
4. int main() {
5.     char str1[20] = "Hello ";
6.     char str2[] = "WorldWideWeb";
7.     strncat(str1, str2, 5); // Will append "World"
8.     printf("Concatenated string (5 chars): %s\n", str1); // Output: Concatenated string (5
chars): Hello World
9.     return 0;
10. }
11.
```

השוואת מחרוזות

השוואת מחרוזות מתבצעת על ידי השוואת ערכי ה ASCII-של התווים במחרוזות.

1. int strcmp(const char *s1, const char *s2);
   - **תיאור:** משווה את המחרוזת s1 למחרוזת s2.
   - **ערך חוזר :**
     - מספר שלילי אם s1 קטן מs2 בסדר לקסיקוגרפי
     - אפס אם s1 שווה לs2
     - מספר חיובי אם s1 גדול מs2 בסדר לקסיקוגרפי
   - **דוגמה :**

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

int main() {
    char str1[] = "apple";
    char str2[] = "apply";
    char str3[] = "apple";

    if (strcmp(str1, str2) < 0) {
        printf("\"%s\" is less than \"%s\"\n", str1, str2); // Output: "apple" is less than "apply"
    }
    if (strcmp(str1, str3) == 0) {
        printf("\"%s\" is equal to \"%s\"\n", str1, str3); // Output: "apple" is equal to "apple"
    }
    return 0;
}

```

2. int strncmp(const char *s1, const char *s2, size_t n);
   - **תיאור:** משווה לכל היותר n תווים ראשונים של המחרוזת s1 למחרוזת s2.
   - **ערך חוזר:** זהה לstrcmp.
   - **דוגמה :**

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

int main() {
    char str1[] = "banana";
    char str2[] = "bandana";

    if (strncmp(str1, str2, 3) == 0) {
        printf("First 3 characters of \"%s\" and \"%s\" are identical.\n", str1, str2); // Output
    }
    if (strncmp(str1, str2, 4) != 0) {
        printf("First 4 characters of \"%s\" and \"%s\" are different.\n", str1, str2); // Output
    }
    return 0;
}

```

חיפוש במחרוזות וטוקניזציה

1. char *strchr(const char *s, int c);
   - **תיאור:** מאתרת את המופע הראשון של התו c שהומר לchar במחרוזת s.
   - **ערך חוזר:** מצביע למופע הראשון של c במחרוזת s או NULL אם התו לא נמצא.
   - **דוגמה :**

```
1. #include <stdio.h>
2. #include <string.h>
3.
4. int main() {
5.     const char str[] = "This is a string";
6.     const char ch = 's';
7.     char *result;
8.
9.     result = strchr(str, ch);
10.     if (result != NULL) {
11.         printf("Character '%c' found at position: %ld\n", ch, result - str); // Output:
Character 's' found at position: 3
12.         printf("Substring from found character: %s\n", result); // Output: Substring from
found character: s is a string
13.     } else {
14.         printf("Character '%c' not found.\n", ch);
15.     }
16.     return 0;
17. }
18.
```

2. size_t strcspn(const char *s1, const char *s2);
   - **תיאור:** מחשבת את אורך החלק ההתחלתי של המחרוזת s1 המורכב כולו מתווים *שאינם* מופיעים במחרוזת s2.
   - **ערך חוזר:** אורך הרצף ההתחלתי של s1 שאין בו תווים מs2.
   - **דוגמה :**

```
1. #include <stdio.h>
2. #include <string.h>
3.
4. int main() {
5.     const char s1[] = "abcdefg123hijkl";
6.     const char s2[] = "1234567890"; // Characters to search for
7.     size_t len = strcspn(s1, s2);
8.     printf("Number of characters before the first digit in s1 is: %zu\n", len); // Output: 7
(abcdefg)
9.     return 0;
10. }
11.
```

3. size_t strspn(const char *s1, const char *s2);
   - **תיאור:** מחשבת את אורך החלק ההתחלתי של המחרוזת s1 המורכב כולו מתווים *המופיעים* במחרוזת s2.
   - **ערך חוזר:** אורך הרצף ההתחלתי של s1 שכל התווים בו נמצאים גם בs2.
   - **דוגמה :**

```
1. #include <stdio.h>
2. #include <string.h>
3.
4. int main() {
5.     const char s1[] = "123abcdefg";
6.     const char s2[] = "0123456789"; // Allowed characters
7.     size_t len = strspn(s1, s2);
8.     printf("Length of the initial segment of s1 containing only digits is: %zu\n", len); //
Output: 3 (123)
9.     return 0;
10. }
11.
```

4. char *strpbrk(const char *s1, const char *s2);
   - **תיאור:** מאתרת את המופע הראשון במחרוזת s1 של כל תו שנמצא במחרוזת s2.
   - **ערך חוזר:** מצביע למופע הראשון של תו מs2 בתוך s1 או NULL אם אין תו כזה.
   - **דוגמה:**

```
1. #include <stdio.h>
2. #include <string.h>
3.
4. int main() {
5.     const char s1[] = "Hello beautiful world";
6.     const char s2[] = "auw"; // Search for 'a', 'u', or 'w'
7.     char *found = strpbrk(s1, s2);
8.
9.     if (found != NULL) {
10.         printf("First character from s2 found in s1 is '%c' at position %ld.\n", *found,
found - s1);
11.         // Output: First character from s2 found in s1 is 'a' at position 8.
12.     } else {
13.         printf("No character from s2 was found in s1.\n");
14.     }
15.     return 0;
16. }
17.
```

5. char *strrchr(const char *s, int c);
   - **תיאור:** מאתרת את המופע האחרון של התו c שהומר לchar במחרוזת s.
   - **ערך חוזר:** מצביע למופע האחרון של c במחרוזת s או NULL אם התו לא נמצא.
   - **דוגמה:**

```
1. #include <stdio.h>
2. #include <string.h>
3.
4. int main() {
5.     const char str[] = "This is a string, this is a test";
6.     const char ch = 's';
7.     char *result;
8.
9.     result = strrchr(str, ch);
10.     if (result != NULL) {
11.         printf("Last occurrence of '%c' found at position: %ld\n", ch, result - str); //
Output: ... at position: 29
12.         printf("Substring from last found character: %s\n", result); // Output: Substring
from last found character: st
13.     } else {
14.         printf("Character '%c' not found.\n", ch);
15.     }
16.     return 0;
17. }
18.
```

6. char *strstr(const char *s1, const char *s2);
   - **תיאור :**מאתרת את המופע הראשון של כל המחרוזת s2 למעט התו המסיים \0 בתוך המחרוזת s1.
   - **ערך חוזר :**מצביע למופע הראשון של s2 בתוך s1 או NULL אם s2 לא נמצאת ב s1.
   - **דוגמה :**

```
1. #include <stdio.h>
2. #include <string.h>
3.
4. int main() {
5.     const char haystack[] = "This is the full string";
6.     const char needle[] = "full";
7.     char *found = strstr(haystack, needle);
8.
9.     if (found != NULL) {
10.         printf("String '%s' found at position %ld.\n", needle, found - haystack);
11.         // Output: String 'full' found at position 12.
12.         printf("Substring from found string: %s\n", found); // Output: Substring from found
string: full string
13.     } else {
14.         printf("String '%s' not found.\n", needle);
15.     }
16.     return 0;
17. }
18.
```

7. char *strtok(char *s1, const char *s2);
   - **תיאור :**מפרקת את המחרוזת s1 ל-"אסימונים" (tokens) שהם בעצם רצפים לוגיים כמו מילים במשפט המופרדים על ידי תווים הנמצאים במחרוזת s2 הנקראים delimiters.
   - **אופן פעולה :**
     - בקריאה הראשונה, s1 צריך להיות המחרוזת שיש לפרק. strtok מאתרת את האסימון הראשון (רצף תווים עד לתו הפרדה הראשון), מחליפה את תו ההפרדה בתו \0, ושומרת מצביע פנימי למיקום שאחרי האסימון. היא מחזירה מצביע לאסימון הראשון.
     - בקריאות הבאות, יש להעביר NULL כארגומנט הראשון. strtok תמשיך מהמיקום השמור ותחזיר את האסימון הבא.
     - כאשר לא נותרו עוד אסימונים strtok מחזירה NULL.
   - **חשוב:** strtok משנה את המחרוזת המקורית (s1) על ידי הכנסת תווי \0. אם יש צורך לשמור על המחרוזת המקורית, יש להעתיק אותה לפני השימוש בstrtok.
   - **דוגמה :**

```
1. #include <stdio.h>
2. #include <string.h>
3.
4. int main() {
5.     char str[] = "Hello, world! This is a test.";
6.     const char delimiters[] = " ,!."; // Space, comma, period, exclamation mark
7.     char *token;
8.
9.     // Get the first token
10.    token = strtok(str, delimiters);
11.
12.    // Walk through other tokens
13.    while (token != NULL) {
14.        printf("%s\n", token);
15.        token = strtok(NULL, delimiters); // Get next token
16.    }
17.    /*
18.    Output:
19.    Hello
20.    world
21.    This
22.    is
23.    a
24.    test
25.    */
26.    return 0;
27. }
28.
```

עבודה עם קבצים בשפת C

עבודה עם קבצים היא חלק חיוני בתכנות, המאפשרת לתוכניות לשמור נתונים באופן קבוע ולקרוא נתונים מקלט חיצוני. שפת C מספקת סט פונקציות עשיר, המרוכז בעיקר בספרייה הסטנדרטית <stdio.h>, לטיפול בקבצים.

מושגי יסוד

- **מצביע לקובץ (FILE*)**: כאשר פותחים קובץ, מערכת ההפעלה מחזירה מצביע למבנה נתונים מסוג FILE. מבנה זה מכיל מידע על הקובץ, כמו מצב הקובץ, מיקום הסמן הנוכחי בקובץ, ומידע על החוצץ (buffer) המשמש לקריאה/כתיבה. כל הפעולות על קבצים מתבצעות באמצעות מצביע זה.
- **זרם (Stream)**: ב-C, קבצים מטופלים כ"זרמים" של בתים. זרם הוא רצף של נתונים שניתן לקרוא ממנו או לכתוב אליו.
- **מצבי פתיחה**: כאשר פותחים קובץ, יש לציין את מצב הפתיחה, הקובע אילו פעולות ניתן לבצע על הקובץ (קריאה, כתיבה, הוספה וכו') והאם הקובץ הוא טקסטואלי או בינארי.

עבודה עם קבצי טקסט

קבצי טקסט מכילים מידע טקסטואלי שניתן לקריאה על ידי אדם (למשל, קוד מקור, קובצי הגדרות, מסמכים פשוטים).

1. פתיחת קובץ טקסט: fopen() או fopen_s()

הפונקציה fopen() משמשת לפתיחת קובץ. היא מקבלת את שם הקובץ ואת מצב הפתיחה, ומחזירה מצביע לקובץ (FILE*) או NULL במקרה של כישלון.

- **חתימת הפונקציה:**
  FILE *fopen(const char *filename, const char *mode);

- **מצבי פתיחה נפוצים לקבצי טקסט:**
  - "r": פתיחה לקריאה. הקובץ חייב להיות קיים.
  - "w": פתיחה לכתיבה. אם הקובץ קיים, תוכנו נמחק. אם אינו קיים, הוא נוצר.
  - "a": פתיחה להוספה (append). הכתיבה מתבצעת בסוף הקובץ. אם הקובץ אינו קיים, הוא נוצר.
  - "r+": פתיחה לקריאה וכתיבה. הקובץ חייב להיות קיים.
  - "w+": פתיחה לקריאה וכתיבה. אם הקובץ קיים, תוכנו נמחק. אם אינו קיים, הוא נוצר.
  - "a+": פתיחה לקריאה והוספה. הכתיבה מתבצעת בסוף הקובץ. אם הקובץ אינו קיים, הוא נוצר.

דוגמה (fopen):

```c
#include <stdio.h>
#include <stdlib.h> // For exit()

int main() {
    FILE *filePointer;
    char *filename = "example.txt";

    // Open file for writing
    filePointer = fopen(filename, "w");
    if (filePointer == NULL) {
        perror("Error opening file for writing"); // Print system error message
        exit(EXIT_FAILURE);
    }
    printf("File '%s' opened successfully for writing.\n", filename);
    // ... operations on file ...
    fclose(filePointer); // Always close the file

    // Open file for reading
```

```
19.     filePointer = fopen(filename, "r");
20.     if (filePointer == NULL) {
21.         perror("Error opening file for reading");
22.         exit(EXIT_FAILURE);
23.     }
24.     printf("File '%s' opened successfully for reading.\n", filename);
25.     // ... operations on file ...
26.     fclose(filePointer);
27.
28.     return 0;
29. }
```

2. סגירת קובץ: fclose()

לאחר סיום העבודה עם הקובץ, חובה לסגור אותו באמצעות fclose(). פונקציה זו מרוקנת את חוצץ הפלט (אם קיים), משחררת משאבים שנקשרו לקובץ ומנתקת את הקשר אליו.

- **חתימת הפונקציה:**
  int fclose(FILE *stream);

  הפונקציה מחזירה 0 בהצלחה ו- EOF (End Of File) במקרה של שגיאה.

- **דוגמה:** (ראו בדוגמה של fopen)

3. כתיבה לקובץ טקסט

ישנן מספר פונקציות לכתיבת נתונים לקובץ טקסט:

א. כתיבת תו בודד: fputc()

כותבת תו בודד לזרם הנתונים.

- **חתימת הפונקציה:**
  int fputc(int character, FILE *stream);

  מחזירה את התו שנכתב בהצלחה, או EOF במקרה שגיאה.

- **דוגמה:**

```
1. #include <stdio.h>
2. #include <stdlib.h>
3.
4. int main() {
5.     FILE *fp = fopen("char_example.txt", "w");
6.     if (fp == NULL) {
7.         perror("Error opening file");
8.         return EXIT_FAILURE;
9.     }
10.
11.     char myChar = 'A';
12.     if (fputc(myChar, fp) == EOF) {
13.         fprintf(stderr, "Error writing character '%c'.\n", myChar);
14.     } else {
15.         printf("Character '%c' written successfully.\n", myChar);
16.     }
17.     fputc('\n', fp); // Write a newline character
18.
19.     fclose(fp);
20.     return 0;
21. }
```

ב. כתיבת מחרוזת: fputs()

כותבת מחרוזת (עד לתו \0, אך לא כולל אותו) לזרם הנתונים.

- **חתימת הפונקציה:**
  int fputs(const char *str, FILE *stream);

מחזירה ערך שאינו שלילי בהצלחה, או EOF במקרה שגיאה.

דוגמה:

```
1. #include <stdio.h>
2. #include <stdlib.h>
3.
4. int main() {
5.     FILE *fp = fopen("string_example.txt", "w");
6.     if (fp == NULL) {
7.         perror("Error opening file");
8.         return EXIT_FAILURE;
9.     }
10.
11.     char myString[] = "Hello, File!";
12.     if (fputs(myString, fp) == EOF) {
13.         fprintf(stderr, "Error writing string.\n");
14.     } else {
15.         printf("String written successfully: \"%s\"\n", myString);
16.     }
17.     fputs("\nAnother line.\n", fp);
18.
19.     fclose(fp);
20.     return 0;
21. }
```

ג. כתיבה מפורמטת: fprintf()

כותבת נתונים מפורמטים לזרם, בדומה ל-printf(), אך הפלט נכתב לקובץ במקום למסך.

- **חתימת הפונקציה:**
  int fprintf(FILE *stream, const char *format, ...);

מחזירה את מספר התווים שנכתבו בהצלחה, או ערך שלילי במקרה שגיאה.

דוגמה:

```
1. #include <stdio.h>
2. #include <stdlib.h>
3.
4. int main() {
5.     FILE *fp = fopen("formatted_example.txt", "w");
6.     if (fp == NULL) {
7.         perror("Error opening file");
8.         return EXIT_FAILURE;
9.     }
10.
11.     char name[] = "Alice";
12.     int age = 30;
13.     double score = 95.5;
14.
15.     int charsWritten = fprintf(fp, "Name: %s, Age: %d, Score: %.1f\n", name, age, score);
16.     if (charsWritten < 0) {
17.         fprintf(stderr, "Error writing formatted data.\n");
18.     } else {
19.         printf("%d characters written successfully.\n", charsWritten);
20.     }
```

```
21.
22.     fclose(fp);
23.     return 0;
24. }
```

4. קריאה מקובץ טקסט

א. קריאת תו בודד: fgetc()

קוראת את התו הבא מהזרם ומקדמת את סמן הקובץ.

- **חתימת הפונקציה:**
  int fgetc(FILE *stream);

מחזירה את התו שנקרא (כ-int), או EOF אם הגיעה לסוף הקובץ או אם אירעה שגיאה.

- **דוגמה (קריאת כל התווים בקובץ):**

```
1. #include <stdio.h>
2. #include <stdlib.h>
3.
4. int main() {
5.     FILE *fp = fopen("char_example.txt", "r"); // Assuming char_example.txt exists
6.     if (fp == NULL) {
7.         perror("Error opening file for reading");
8.         return EXIT_FAILURE;
9.     }
10.
11.     int ch;
12.     printf("Content of the file:\n");
13.     while ((ch = fgetc(fp)) != EOF) {
14.         putchar(ch); // Print character to console
15.     }
16.
17.     if (feof(fp)) {
18.         printf("\nEnd of file reached.\n");
19.     } else if (ferror(fp)) {
20.         perror("Error during reading");
21.     }
22.
23.     fclose(fp);
24.     return 0;
25. }
```

ב. קריאת מחרוזת (שורה): fgets()

קוראת שורה מהזרם לתוך מחרוזת (מערך תווים). הקריאה נפסקת לאחר קריאת n-1 תווים, לאחר קריאת תו ירידת שורה (\n), או בהגעה לסוף הקובץ. תו ירידת השורה, אם נקרא, נשמר במחרוזת. תו \0 מוסף אוטומטית בסוף המחרוזת.

- **חתימת הפונקציה:**
  char *fgets(char *str, int n, FILE *stream);

מחזירה את str בהצלחה, או NULL אם הגיעה לסוף הקובץ לפני שקראה תווים כלשהם, או אם אירעה שגיאה.

- **דוגמה:**

```
1. #include <stdio.h>
2. #include <stdlib.h>
3.
4. #define MAX_LINE_LENGTH 100
5.
```

```
6. int main() {
7.     FILE *fp = fopen("string_example.txt", "r"); // Assuming string_example.txt exists
8.     if (fp == NULL) {
9.         perror("Error opening file for reading");
10.         return EXIT_FAILURE;
11.     }
12.
13.     char lineBuffer[MAX_LINE_LENGTH];
14.     printf("Reading lines from file:\n");
15.     while (fgets(lineBuffer, MAX_LINE_LENGTH, fp) != NULL) {
16.         printf("%s", lineBuffer); // lineBuffer already contains newline if present in file
17.     }
18.
19.     if (feof(fp)) {
20.         printf("\nEnd of file reached.\n");
21.     } else if (ferror(fp)) {
22.         perror("Error during reading");
23.     }
24.
25.     fclose(fp);
26.     return 0;
27. }
```

ג. קריאה מפורמטת: fscanf()

קוראת נתונים מפורמטים מהזרם, בדומה ל-scanf(), אך הקלט נקרא מהקובץ.

- **חתימת הפונקציה:**
  int fscanf(FILE *stream, const char *format, ...);

מחזירה את מספר הפריטים שהוקצו בהצלחה, או EOF אם אירעה שגיאת קלט לפני ההקצאה הראשונה.

- **דוגמה (בהנחה שקובץ formatted_example.txt מכיל " Name: Alice, Age: 30, Score: 95.5"):**

```
1. #include <stdio.h>
2. #include <stdlib.h>
3.
4. int main() {
5.     FILE *fp = fopen("formatted_example.txt", "r");
6.     if (fp == NULL) {
7.         perror("Error opening file for reading");
8.         return EXIT_FAILURE;
9.     }
10.
11.     char name[50];
12.     int age;
13.     double score;
14.
15.     // Note: fscanf can be tricky with spaces and newlines in the format string.
16.     // This example assumes a simple, predictable format.
17.     int itemsRead = fscanf(fp, "Name: %49s Age: %d, Score: %lf", name, &age, &score);
18.                             // Read up to 49 chars for name to prevent buffer overflow
19.                             // Note the comma after %49s is consumed if present in file before " Age:"
20.
21.     if (itemsRead == 3) {
22.         printf("Read from file: Name: %s, Age: %d, Score: %.1f\n", name, age, score);
23.     } else if (itemsRead == EOF) {
24.         if (feof(fp)) {
25.             printf("End of file reached before all items were read.\n");
26.         } else {
27.             perror("Input error before any conversion");
28.         }
29.     } else {
30.         printf("Error: Only %d items were successfully read.\n", itemsRead);
```

```
    }

    fclose(fp);
    return 0;
}
```

עבודה עם קבצים בינאריים

קבצים בינאריים מאחסנים נתונים כרצף של בתים, כפי שהם מיוצגים בזיכרון המחשב. זה יעיל לאחסון מבני נתונים מורכבים, תמונות, קבצי קול וכו'.

1. פתיחת קובץ בינארי

הפתיחה דומה לקבצי טקסט, אך יש להוסיף b למצב הפתיחה.

- **מצבי פתיחה נפוצים לקבצים בינאריים:**
  - "rb": פתיחה לקריאה בינארית.
  - "wb": פתיחה לכתיבה בינארית.
  - "ab": פתיחה להוספה בינארית.
  - "rb+": פתיחה לקריאה וכתיבה בינארית (הקובץ חייב להיות קיים).
  - "wb+": פתיחה לקריאה וכתיבה בינארית (תוכן הקובץ נמחק אם קיים, או נוצר חדש).
  - "ab+": פתיחה לקריאה והוספה בינארית.

- **דוגמה:**

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

int main() {
    FILE *binaryFile;
    char *filename = "data.bin";

    binaryFile = fopen(filename, "wb"); // Open for binary writing
    if (binaryFile == NULL) {
        perror("Error opening binary file for writing");
        exit(EXIT_FAILURE);
    }
    printf("Binary file '%s' opened for writing.\n", filename);
    fclose(binaryFile);

    binaryFile = fopen(filename, "rb"); // Open for binary reading
    if (binaryFile == NULL) {
        perror("Error opening binary file for reading");
        exit(EXIT_FAILURE);
    }
    printf("Binary file '%s' opened for reading.\n", filename);
    fclose(binaryFile);

    return 0;
}
```

סגירת קובץ בינארי מתבצעת גם היא באמצעות fclose().

2. כתיבה לקובץ בינארי: fwrite()

כותבת בלוק של נתונים (רצף של בתים) מהזיכרון לקובץ.

- **חתימת הפונקציה:**
  size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);

  - ptr: מצביע לבלוק הנתונים בזיכרון שיש לכתוב.
  - size: גודל (בבתים) של כל פריט נתונים.

- nmemb: מספר הפריטים שיש לכתוב.
- stream: מצביע לקובץ. מחזירה את מספר הפריטים (nmemb) שנכתבו בהצלחה. אם ערך זה קטן מ-nmemb המבוקש, אירעה שגיאה.

- **דוגמה (כתיבת מערך של מספרים שלמים):**

```
1. #include <stdio.h>
2. #include <stdlib.h>
3.
4. typedef struct {
5.     int id;
6.     char name[50];
7.     double value;
8. } Record;
9.
10. int main() {
11.     FILE *fp = fopen("records.dat", "wb");
12.     if (fp == NULL) {
13.         perror("Error opening file for binary writing");
14.         return EXIT_FAILURE;
15.     }
16.
17.     Record recordsToWrite[] = {
18.         {1, "ItemA", 10.5},
19.         {2, "ItemB", 20.75},
20.         {3, "ItemC", 0.0}
21.     };
22.     size_t numRecords = sizeof(recordsToWrite) / sizeof(Record);
23.
24.     size_t itemsWritten = fwrite(recordsToWrite, sizeof(Record), numRecords, fp);
25.     if (itemsWritten < numRecords) {
26.         fprintf(stderr, "Error: Only %zu out of %zu records written.\n", itemsWritten, numRecords);
27.         if (ferror(fp)) {
28.             perror("fwrite error");
29.         }
30.     } else {
31.         printf("%zu records written successfully to binary file.\n", itemsWritten);
32.     }
33.
34.     fclose(fp);
35.     return 0;
36. }
```

3. קריאה מקובץ בינארי: fread()

קוראת בלוק של נתונים מהקובץ לתוך הזיכרון.

- **חתימת הפונקציה:**
  size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);

  - ptr: מצביע לאזור בזיכרון שאליו ייקראו הנתונים.
  - size: גודל (בבתים) של כל פריט נתונים.
  - nmemb: מספר הפריטים המקסימלי שיש לקרוא.
  - stream: מצביע לקובץ. מחזירה את מספר הפריטים (nmemb) שנקראו בהצלחה. אם ערך זה קטן מ-nmemb המבוקש, ייתכן שהגיע לסוף הקובץ או שאירעה שגיאה. יש להשתמש ב-feof() או ferror() כדי להבחין בין המצבים.

- **דוגמה (קריאת מערך של מספרים שלמים):**

```
1. #include <stdio.h>
2. #include <stdlib.h>
3.
4. typedef struct { // Ensure this struct matches the one used for writing
5.     int id;
```

```
    char name[50];
    double value;
} Record;

#define MAX_RECORDS 10

int main() {
    FILE *fp = fopen("records.dat", "rb"); // Assuming records.dat was written previously
    if (fp == NULL) {
        perror("Error opening file for binary reading");
        return EXIT_FAILURE;
    }

    Record recordsRead[MAX_RECORDS];
    size_t itemsRead = fread(recordsRead, sizeof(Record), MAX_RECORDS, fp);

    printf("Attempted to read up to %d records. Actually read: %zu\n", MAX_RECORDS, itemsRead);

    if (itemsRead > 0) {
        printf("Records read from binary file:\n");
        for (size_t i = 0; i < itemsRead; ++i) {
            printf("ID: %d, Name: %s, Value: %.2f\n",
                   recordsRead[i].id, recordsRead[i].name, recordsRead[i].value);
        }
    }

    if (feof(fp) && itemsRead < MAX_RECORDS) {
        printf("End of file reached.\n");
    } else if (ferror(fp)) {
        perror("fread error");
    }


    fclose(fp);
    return 0;
}
```

פונקציות לניווט בקבצים (מיקום הסמן)

פונקציות אלו מאפשרות לשנות או לקבל את מיקום סמן הקריאה/כתיבה הנוכחי בקובץ.

1. fseek()

משנה את מיקום הסמן בקובץ.

- **חתימת הפונקציה:**
  int fseek(FILE *stream, long int offset, int whence);

  - stream: מצביע לקובץ.
  - offset: ההיסט (offset) בבתים ביחס ל-whence.

  - whence: נקודת הייחוס לחישוב המיקום החדש:
    - SEEK_SET: תחילת הקובץ.
    - SEEK_CUR: המיקום הנוכחי של הסמן.
    - SEEK_END: סוף הקובץ. מחזירה 0 בהצלחה, וערך שאינו אפס במקרה שגיאה.

- **דוגמה:**

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

typedef struct { int id; double val; } Data;

```

```
int main() {
    FILE *fp = fopen("seek_example.dat", "wb+"); // Open for binary read/write, create if not exists
    if (fp == NULL) {
        perror("Error opening file");
        return EXIT_FAILURE;
    }

    Data d1 = {1, 10.0}, d2 = {2, 20.0}, d3 = {3, 30.0};
    fwrite(&d1, sizeof(Data), 1, fp);
    fwrite(&d2, sizeof(Data), 1, fp);
    fwrite(&d3, sizeof(Data), 1, fp);
    printf("Wrote 3 records.\n");

    // Move to the beginning of the second record (index 1)
    if (fseek(fp, 1 * sizeof(Data), SEEK_SET) != 0) {
        perror("fseek error");
        fclose(fp);
        return EXIT_FAILURE;
    }
    printf("Seeked to the start of the second record.\n");

    Data readData;
    if (fread(&readData, sizeof(Data), 1, fp) == 1) {
        printf("Read record at current position: ID=%d, Val=%.1f\n", readData.id, readData.val); // Should be d2
    }

    // Move 1 record back from current position (which is after reading d2)
    if (fseek(fp, -1 * (long)sizeof(Data), SEEK_CUR) != 0) { // Cast to long for safety with negative offset
        perror("fseek error (SEEK_CUR)");
        fclose(fp);
        return EXIT_FAILURE;
    }
    if (fread(&readData, sizeof(Data), 1, fp) == 1) {
        printf("Read record after seeking back: ID=%d, Val=%.1f\n", readData.id, readData.val); // Should be d2 again
    }


    // Move to a position relative to the end of the file
    if (fseek(fp, -1 * (long)sizeof(Data), SEEK_END) != 0) {
        perror("fseek error (SEEK_END)");
        fclose(fp);
        return EXIT_FAILURE;
    }
    printf("Seeked to the start of the last record from end.\n");
    if (fread(&readData, sizeof(Data), 1, fp) == 1) {
        printf("Read last record: ID=%d, Val=%.1f\n", readData.id, readData.val); // Should be d3
    }
    fclose(fp);
    return 0;
}
```

2. ftell()

מחזירה את המיקום הנוכחי של סמן הקובץ (בבתים מתחילת הקובץ).

- **חתימת הפונקציה:**
  long int ftell(FILE *stream);

מחזירה את המיקום הנוכחי בהצלחה, או -L1 במקרה שגיאה.

- **דוגמה:**

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

int main() {
    FILE *fp = fopen("tell_example.txt", "w+");
    if (fp == NULL) {
```

```
        perror("Error opening file");
        return EXIT_FAILURE;
    }
    fputs("Hello", fp);
    long pos = ftell(fp);
    if (pos == -1L) {
        perror("ftell error");
    } else {
        printf("Current file position after writing 'Hello': %ld bytes from start.\n", pos); // Output: 5
    }

    fseek(fp, 0, SEEK_END); // Go to end of file
    pos = ftell(fp);
    printf("File size (position at end): %ld bytes.\n", pos); // Output: 5

    fclose(fp);
    return 0;
}
```

3. rewind()

מחזירה את סמן הקובץ לתחילת הקובץ. שקול ל-fseek(stream, 0L, SEEK_SET).

- **חתימת הפונקציה:**
  void rewind(FILE *stream);

- **דוגמה:**

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

int main() {
    FILE *fp = fopen("rewind_example.txt", "w+");
    if (fp == NULL) {
        perror("Error opening file");
        return EXIT_FAILURE;
    }
    fputs("Line 1\n", fp);
    fputs("Line 2\n", fp);
    printf("Wrote two lines. Current position: %ld\n", ftell(fp));

    rewind(fp); // Move to the beginning of the file
    printf("After rewind, current position: %ld\n", ftell(fp)); // Output: 0

    char buffer[50];
    if (fgets(buffer, sizeof(buffer), fp) != NULL) {
        printf("Read after rewind: %s", buffer); // Output: Line 1
    }

    fclose(fp);
    return 0;
}
```

4. feof()

בודקת האם הושג סוף הקובץ (End Of File) עבור הזרם הנתון.

חתימת הפונקציה:
int feof(FILE *stream);

מחזירה ערך שאינו אפס אם סוף הקובץ הושג, ואחרת 0.

enum (מנייה)

מנייה (enum) היא סוג נתונים מוגדר על ידי המשתמש, המורכב מקבוצה של קבועים בעלי שם. זהו דרך להפוך קוד ליותר קריא וקל לתחזוקה על ידי מתן שמות משמעותיים לערכים מספריים.

הגדרה:
כדי להגדיר מנייה, משתמשים במילת המפתח enum, ואחריה שם המנייה, וסוגריים מסולסלים המכילים רשימה של חברים (הקבועים בעלי השם) מופרדים בפסיקים.

```
1. enum Boolean { // Defining an enumeration named 'Boolean'
2.     FALSE,      // First member, by default assigned the value 0
3.     TRUE        // Second member, by default assigned the value 1
4. };
```

בברירת מחדל, החבר הראשון במנייה מקבל את הערך 0, השני מקבל את הערך 1, וכן הלאה. עם זאת, אפשר להקצות ערכים ספציפיים לחברים במהלך ההגדרה:

```
1. enum DaysOfWeek { // Defining an enumeration named 'DaysOfWeek'
2.     SUNDAY = 1,  // Assigning the value 1 to SUNDAY
3.     MONDAY,      // MONDAY will automatically get the value 2
4.     TUESDAY,     // TUESDAY will automatically get the value 3
5.     WEDNESDAY,   // and so on...
6.     THURSDAY,
7.     FRIDAY,
8.     SATURDAY
9. };
```

שימוש:
אפשר להכריז על משתנים מסוג מנייה בדיוק כמו סוגי נתונים אחרים. המשתנים האלה יכולים להכיל רק את אחד הערכים המוגדרים במנייה.

```
1. enum Boolean isTrue; // Declaring a variable 'isTrue' of type 'enum Boolean'
2. isTrue = TRUE;       // Assigning the value TRUE to the variable
3.
4. enum DaysOfWeek today; // Declaring a variable 'today' of type 'enum DaysOfWeek'
5. today = MONDAY;       // Assigning the value MONDAY to the variable
6.
7. if (today == SATURDAY || today == SUNDAY) {
8.     // Do something on the weekend
9. }
```

יתרונות:

- **קריאות:** הופך את הקוד לקריא יותר מכיוון שמשתמשים בשמות משמעותיים במקום במספרים קסם (magic numbers).
- **תחזוקה:** קל יותר לשנות את הערכים המשויכים לשמות במקום לעבור על כל הקוד ולעדכן מספרים.
- **בטיחות טיפוסים:** מהדרים יכולים לעתים קרובות לספק בדיקה מסוימת של בטיחות טיפוסים עבור משתני מנייה.

static (סטטי)

1. משתנים סטטיים מקומיים (Static Local Variables):

- מוכרזים בתוך פונקציה.
- אורך החיים שלהם הוא משך כל חיי התוכנית, לא רק משך זמן ביצוע הפונקציה.
- הם מאותחלים רק פעם אחת, בפעם הראשונה שהפונקציה נקראת.
- הם גלויים רק בתוך הפונקציה בה הם הוכרזו (scope).

```
1. #include <stdio.h>
2.
3. void incrementCounter() {
4.     static int counter = 0; // Static local variable, initialized only once
5.     counter++;
6.     printf("Counter: %d\n", counter); // Counter: %d\n
7. }
8.
9. int main() {
10.     incrementCounter(); // Output: Counter: 1
11.     incrementCounter(); // Output: Counter: 2
12.     incrementCounter(); // Output: Counter: 3
13.     return 0;
14. }
15.
```

2. משתנים גלובליים סטטיים (Static Global Variables):

- מוכרזים מחוץ לכל פונקציה.
- אורך החיים שלהם הוא משך כל חיי התוכנית.
- הם מאותחלים ל-0 אם לא אותחלו במפורש.
- הם גלויים רק בתוך קובץ המקור בו הם הוכרזו (internal linkage). זה אומר שלא ניתן לגשת אליהם ישירות מקבצי מקור אחרים.

```
1. // file1.c
2. static int globalVar = 10; // Static global variable, only accessible within file1.c
3.
4. void printGlobalVar() {
5.     printf("Global variable in file1: %d\n", globalVar); // Global variable in file1: 10
6. }
7.
8. // file2.c
9. // extern int globalVar; // This would cause an error because globalVar in file1.c has
internal linkage
10.
11. // void tryAccessGlobalVar() {
12. //     printf("Trying to access global variable: %d\n", globalVar);
13. // }
14.
```

3. פונקציות סטטיות (Static Functions):

- מוכרזות מחוץ לכל פונקציה.
- הן גלויות רק בתוך קובץ המקור בו הן הוכרזו (internal linkage). זה אומר שלא ניתן לקרוא להן ישירות מקבצי מקור אחרים. זה שימושי כדי ליצור פונקציות עוזרות פנימיות לקובץ מסוים.

```
 1. // utils.c
 2. static void internalHelperFunction() { // Static function, only callable within utils.c
 3.     printf("This is an internal helper function.\n"); // This is an internal helper
function.
 4. }
 5.
 6. void publicFunction() {
 7.     printf("This is a public function.\n"); // This is a public function.
 8.     internalHelperFunction();           // Calling the static helper function
 9. }
10.
11. // main.c
12. // void internalHelperFunction(); // This would cause a linker error
13. // publicFunction();
14.
```

const (קבוע)

המילה השמורה const משמשת כדי להצהיר על משתנה כקבוע. משמעות הדבר היא שמרגע שהמשתנה אותחל, לא ניתן לשנות את ערכו.

הגדרה ושימוש:

```
 1. const int MAX_VALUE = 100; // Declaring a constant integer
 2. // MAX_VALUE = 200;        // This would cause a compiler error because MAX_VALUE is const
 3.
 4. const float PI = 3.14159; // Declaring a constant float
 5.
 6. void printValue(const int value) { // Declaring a function parameter as const
 7.     printf("Value is: %d\n", value); // Value is: ...
 8.     // value++;                     // This would cause a compiler error because 'value' is
const in this scope
 9. }
10.
11. int main() {
12.     printValue(MAX_VALUE);
13.     return 0;
14. }
15.
```

const ופויינטרים:

1. **Pointer to a constant value (מצביע לערך קבוע):** הערך שאליו המצביע מצביע אינו יכול להשתנות.

```
1. const int x = 10;
2. const int *ptr = &x; // Pointer to a constant integer
3.
4. // *ptr = 20;         // Error: Cannot modify the value through ptr
5. ptr = &MAX_VALUE;   // OK: The pointer itself can point to a different constant value
6.
```

2. **Constant pointer to a value (מצביע קבוע לערך):** המצביע עצמו (כתובת הזיכרון שהוא מכיל) אינו יכול להשתנות, אך הערך שאליו הוא מצביע יכול להשתנות.

```
1. int y = 30;
2. int *const ptr2 = &y; // Constant pointer to an integer
3.
4. *ptr2 = 40;          // OK: The value y can be modified through ptr2
5. // ptr2 = &x;        // Error: Cannot change the address ptr2 points to
6.
```

3. **Constant pointer to a constant value (מצביע קבוע לערך קבוע):** גם המצביע וגם הערך שאליו הוא מצביע אינם יכולים להשתנות.

```
const int z = 50;1. const int *const ptr3 = &z; // Constant pointer to a constant integer
2.
3. // *ptr3 = 60;         // Error: Cannot modify the value through ptr3
4. // ptr3 = &x;        // Error: Cannot change the address ptr3 points to
5.
```

יתרונות השימוש ב-const:

- **מניעת טעויות:** מסייע במניעת שינוי ערכים שלא אמורים להשתנות, מה שיכול להוביל לבאגים.
- **קריאות:** מציין בבירור אילו משתנים נועדו להיות קבועים, מה שמקל על הבנת הקוד.
- **אופטימיזציה:** מהדרים עשויים להיות מסוגלים לבצע אופטימיזציות מסוימות עבור משתנים שהוכרזו כ-const.

Type	Declaration	Pointer Value Change (*ptr = 100)	Pointing Value Change (ptr = &a)
Pointer to Variable	int * ptr	Yes	Yes
Pointer to Constant	const int * ptr int const * ptr	No	Yes
Constant Pointer to Variable	int * const ptr	Yes	No
Constant Pointer to Constant	const int * const ptr	No	No

Bitwise Operations (פעולות ביטים)

פעולות ביטים מאפשרות לבצע מניפולציות ברמת הביטים הבודדים של נתונים שלמים (integers). הן שימושיות במיוחד בתכנות מערכות, תכנות משובץ, ובמצבים בהם יש צורך בשליטה מדויקת על ייצוג הנתונים בזיכרון.

הפעולות הבינאריות העיקריות:

- **AND (&):** מבצע פעולת AND בין כל זוג ביטים מתאים משני האופרנדים. הביט המתקבל יהיה 1 רק אם שני הביטים המתאימים הם 1.

```
1. unsigned char a = 5;  // Binary: 00000101
2. unsigned char b = 3;  // Binary: 00000011
3. unsigned char result = a & b; // Binary: 00000001 (Decimal: 1)
4. //   0101
5. // & 0011
6. // ------
7. //   0001
8.
```

- **OR (|):** מבצע פעולת OR בין כל זוג ביטים מתאים משני האופרנדים. הביט המתקבל יהיה 1 אם לפחות אחד מהביטים המתאימים הוא 1.

```
1. unsigned char a = 5;  // Binary: 00000101
2. unsigned char b = 3;  // Binary: 00000011
3. unsigned char result = a | b; // Binary: 00000111 (Decimal: 7)
4. //   0101
5. // | 0011
6. // ------
7. //   0111
8.
```

- **XOR (^):** מבצע פעולת XOR (exclusive OR) בין כל זוג ביטים מתאים משני האופרנדים. הביט המתקבל יהיה 1 אם הביטים המתאימים שונים זה מזה.

```
1. unsigned char a = 5;  // Binary: 00000101
2. unsigned char b = 3;  // Binary: 00000011
3. unsigned char result = a ^ b; // Binary: 00000110 (Decimal: 6)
4. //   0101
5. // ^ 0011
6. // ------
7. //   0110
8.
```

- **NOT (~):** מבצע פעולת NOT (complement) על אופרנד יחיד. היא הופכת כל ביט: 0 הופך ל-1 ו-1 הופך ל-0.

```
1. unsigned char a = 5;      // Binary: 00000101
2. unsigned char result = ~a; // Binary: 11111010 (Decimal: 250, assuming 8-bit unsigned char)
3. // ~ 0101
4. // ------
5. //   1010 (for the last 4 bits)
6.
```

פעולות הזזה (Shift Operations):

- **Left Shift (<<):** מזיז את הביטים של האופרנד השמאלי שמאלה במספר הביטים שצוין על ידי האופרנד הימני. ביטים חדשים מימין מתמלאים ב-0. פעולה זו שקולה בדרך כלל לכפל בחזקה של 2.

```
1. unsigned char a = 5;       // Binary: 00000101 (Decimal: 5)
2. unsigned char result = a << 2; // Binary: 00010100 (Decimal: 20) (5 * 2^2 = 20)
3.
```

- **Right Shift (>>):** מזיז את הביטים של האופרנד השמאלי ימינה במספר הביטים שצוין על ידי האופרנד הימני. האופן בו הביטים החדשים משמאל מתמלאים תלוי אם האופרנד הוא מסוג signed או unsigned. עבור unsigned, הם מתמלאים ב-0 (logical right shift). עבור signed, זה תלוי במימוש המהדר (arithmetic right shift שומר על הסימן או logical right shift). פעולה זו שקולה בדרך כלל לחילוק בשלם בחזקה של 2.

```
1. unsigned char a = 20;      // Binary: 00010100 (Decimal: 20)
2. unsigned char result = a >> 2; // Binary: 00000101 (Decimal: 5) (20 / 2^2 = 5)
3.
4. signed char b = -20;        // Assuming 8-bit signed char (e.g., 11101100 in two's
complement)
5. signed char result2 = b >> 2; // Result depends on the compiler (likely 11111011 or 00111011)
6.
```

שימושים נפוצים בפעולות ביטים:

- **ניהול דגלים (Flags):** שימוש בביטים בודדים כדי לייצג מצבים בוליאניים (אמת/שקר) מרובים בתוך משתנה שלם אחד.
- **מסכות ביטים (Bit Masks):** שימוש בתבנית ביטים ספציפית כדי לבחור, להגדיר או לבדוק ביטים מסוימים בתוך ערך.
- **תקשורת עם חומרה:** פרוטוקולי תקשורת רבים פועלים ברמת הביטים.
- **אלגוריתמים יעילים:** פעולות ביטים יכולות להיות יעילות יותר מבחינה חישובית מפעולות כפל וחילוק.
- **ייצוג קבוצות:** אפשר להשתמש בביטים של משתנה כדי לייצג נוכחות או היעדר של אלמנטים בקבוצה.

משתנים גלובליים (Global Variables)

משתנים גלובליים הם משתנים המוצהרים מחוץ לכל פונקציה. המשמעות שלהם היא שהם נגישים (גלויים) מכל חלק של התוכנית - מכל פונקציה בתוך קובץ המקור שבו הם הוגדרו, ואם נעשה בהם שימוש נכון (באמצעות extern), גם מקבצי מקור אחרים.

הגדרה ושימוש:

```
1. #include <stdio.h>
2.
3. int globalCounter = 0; // Declaring a global variable
4.
5. void incrementGlobalCounter() {
6.     globalCounter++;
7.     printf("Global counter (inside function): %d\n", globalCounter); // Global counter
(inside function): ...
8. }
9.
10. int main() {
11.     printf("Global counter (before increment): %d\n", globalCounter); // Global counter
(before increment): 0
12.     incrementGlobalCounter();
13.     printf("Global counter (after increment): %d\n", globalCounter);  // Global counter
(after increment): 1
14.     return 0;
15. }
```

שימוש בין קבצים (באמצעות extern):

אם רוצים להשתמש במשתנה גלובלי שהוגדר בקובץ מקור אחד בקובץ מקור אחר, יש להצהיר עליו באמצעות מילת המפתח extern בקובץ השני. ההגדרה (היכן שהמשתנה מוקצה בזיכרון ואותחל) צריכה להופיע רק בקובץ אחד.

```
1. file1.c:
2. int globalData = 42; // Definition of the global variable
3. file2.c:
4. #include <stdio.h>
5.
6. extern int globalData; // Declaration of the global variable defined in file1.c
7.
8. void printGlobalData() {
9.     printf("Global data from file1: %d\n", globalData); // Global data from file1: 42
10. }
11.
12. int main() {
13.     printGlobalData();
14.     return 0;
15. }
```

יתרונות וחסרונות:

- **נוחות:** מאפשר גישה לנתונים ממקומות רבים בתוכנית ללא צורך להעביר אותם כארגומנטים לפונקציות.
- **קלות שימוש:** פשוטים להגדרה ולשימוש.
- **פוטנציאל לבעיות:** שימוש יתר במשתנים גלובליים יכול להוביל לקוד שקשה יותר להבנה, לתחזוקה ולניפוי שגיאות, מכיוון ששינוי ערכם יכול להתרחש בכל מקום בתוכנית, מה שמקשה על מעקב אחר מקור השינוי.
- **הצמדה הדוקה:** יוצר הצמדה הדוקה בין חלקים שונים של הקוד, מה שמקשה על שימוש חוזר ברכיבי קוד בתוכניות אחרות.

בדרך כלל מומלץ להשתמש במשתנים גלובליים בזהירות ובמקרים בהם הם באמת נחוצים לכל התוכנית.

מצביעים לפונקציות (Function Pointers)

מצביע לפונקציה הוא משתנה שמכיל את כתובת הזיכרון של פונקציה. זה מאפשר להתייחס לפונקציה כאל נתון, להעביר אותה כארגומנט לפונקציות אחרות, ולאחסן אותה במבני נתונים.

הגדרה:
כדי להכריז על מצביע לפונקציה, משתמשים בתחביר הבא:

return_type (*pointer_name)(parameter_types);

- return_type: סוג הערך שהפונקציה מצביעה עליו מחזירה.
- pointer_name: שם המצביע.
- parameter_types: רשימה של סוגי הפרמטרים שהפונקציה מצביעה עליה מקבלת.

דוגמה:

```
 1. #include <stdio.h>
 2.
 3. int add(int a, int b) {
 4.     return a + b;
 5. }
 6.
 7. int subtract(int a, int b) {
 8.     return a - b;
 9. }
10.
11. int main() {
12.     int (*operation)(int, int); // Declaring a function pointer
13.
14.     operation = add; // Assigning the address of the 'add' function to the pointer
15.     printf("Addition: %d\n", operation(5, 3)); // Calling the 'add' function through the
pointer
16.
17.     operation = subtract; // Assigning the address of the 'subtract' function to the pointer
18.     printf("Subtraction: %d\n", operation(5, 3)); // Calling the 'subtract' function through
the pointer
19.
20.     return 0;
21. }
22.
```

שימוש כארגומנט לפונקציה (Callback Functions):
אחד השימושים העיקריים במצביעים לפונקציות הוא העברת פונקציות כארגומנטים לפונקציות אחרות. זה מאפשר ליצור קוד גנרי וגמיש יותר.

```
 1. #include <stdio.h>
 2.
 3. void processArray(int arr[], int size, void (*func)(int)) {
 4.     for (int i = 0; i < size; i++) {
 5.         func(arr[i]); // Call the function pointed to by 'func' for each element
 6.     }
 7. }
 8.
 9. void printNumber(int num) {
10.     printf("%d ", num); // Print a number
11. }
12.
13. void printSquaredNumber(int num) {
14.     printf("%d ", num * num); // Print a squared number
15. }
16.
17. int main() {
18.     int numbers[] = {1, 2, 3, 4, 5};
19.     int size = sizeof(numbers) / sizeof(numbers[0]);
20.
```

```
21.     printf("Printing numbers: "); // Printing numbers: 1 2 3 4 5
22.     processArray(numbers, size, printNumber);
23.     printf("\n");
24.
25.     printf("Printing squared numbers: "); // Printing squared numbers: 1 4 9 16 25
26.     processArray(numbers, size, printSquaredNumber);
27.     printf("\n");
28.
29.     return 0;
30. }
31.
```

איגודים (Unions)

איגוד (union) הוא סוג נתונים מיוחד שיכול להכיל טיפוסים שונים של נתונים (אינט, פלוט, צ'אר וכו'), אך רק אחד מהם יכול להיות מאוחסן בזיכרון בכל זמן נתון. האיגוד מקצה מספיק זיכרון כדי להכיל את החבר הגדול ביותר בו, וכל החברים חולקים את אותה כתובת זיכרון.

הגדרה:
הגדרה של איגוד דומה להגדרה של מבנה (struct), אך משתמשים במילת המפתח union.

```
1. union Data { // Defining a union named 'Data'
2.     int i;
3.     float f;
4.     char str[20];
5. };
6.
```

באיגוד Data לעיל, מספיק זיכרון יוקצה כדי להכיל או int, או float, או מערך של 20 תווים ([char[20]), בהתאם לחבר הגדול ביותר.
שימוש:
כאשר מקצים ערך לאחד מחברי האיגוד, כל חבר אחר מאבד את ערכו (או יותר נכון, הפירוש של הנתונים בזיכרון משתנה).

```
 1. #include <stdio.h>
 2. #include <string.h>
 3.
 4. union Data data;
 5.
 6. int main() {
 7.     data.i = 10;
 8.     printf("Data.i: %d\n", data.i);   // Data.i: 10
 9.     printf("Data.f: %f\n", data.f);   // Data.f: 0.000000 (Garbage or reinterpreted bits)
10.     printf("Data.str: %s\n", data.str); // Data.str: (Garbage or reinterpreted bits)
11.
12.     data.f = 3.14;
13.     printf("Data.i: %d\n", data.i);   // Data.i: 1078523331 (Reinterpreted bits)
14.     printf("Data.f: %f\n", data.f);   // Data.f: 3.140000
15.     printf("Data.str: %s\n", data.str); // Data.str: (Garbage or reinterpreted bits)
16.
17.     strcpy(data.str, "Hello");
18.     printf("Data.i: %d\n", data.i);   // Data.i: 1819043144 (Reinterpreted bits)
19.     printf("Data.f: %f\n", data.f);   // Data.f: 2.732483 (Reinterpreted bits)
20.     printf("Data.str: %s\n", data.str); // Data.str: Hello
21.
22.     return 0;
23. }
24.
```

מתי להשתמש באיגודים:

- **חיסכון בזיכרון:** כאשר צריך לאחסן סוג נתונים אחד מתוך כמה אפשריים, אך לא את כולם בו-זמנית.
- **טיפול בנתונים שונים באותה פיסת זיכרון:** כאשר יש צורך לפרש את אותה סדרת ביטים בזיכרון כסוגי נתונים שונים.
- **תכנות ברמה נמוכה:** לעבודה עם ייצוג בינארי של נתונים.

חשוב לעקוב אחר איזה סוג נתונים מאוחסן באיגוד בכל רגע נתון כדי להימנע מפירוש שגוי של הנתונים. לעיתים קרובות משתמשים באיגודים בשילוב עם משתנה מנייה (enum) כדי לציין את סוג הנתונים הנוכחי באיגוד.

מאפייני אחסון משתנים (Storage Class Specifiers)

מאפייני אחסון (storage class specifiers) בשפת C הם מילות מפתח המשמשות כדי להגדיר את אורך החיים (lifetime), הטווח (scope), והקישוריות (linkage) של משתנים ופונקציות.

מאפייני האחסון העיקריים הם:

- auto: ברירת המחדל עבור משתנים מקומיים בתוך פונקציה. מציין שהמשתנה נוצר כאשר הבלוק בו הוא מוגדר נכנס לפעולה, ונהרס כאשר הבלוק מסתיים. הטווח שלו הוא הבלוק בו הוא מוגדר, ואין לו קישוריות חיצונית. כמעט תמיד משמיטים את auto כיוון שזו ברירת המחדל.
- static: כפי שהוסבר קודם, יש לו משמעויות שונות בהתאם להקשר:
  - **משתנה מקומי סטטי**: אורך חיים של כל התוכנית, טווח מקומי לפונקציה, אין קישוריות חיצונית.
  - **משתנה גלובלי סטטי**: אורך חיים של כל התוכנית, טווח קובץ, קישוריות פנימית (internal linkage).
  - **פונקציה סטטית**: טווח קובץ, קישוריות פנימית (internal linkage).
- extern: משמש כדי להצהיר על משתנה או פונקציה שהוגדרו במקום אחר (בדרך כלל בקובץ מקור אחר). מציין שהמשתנה או הפונקציה קיימים במקום אחר ויש להם קישוריות חיצונית (external linkage).
- register: מציע למהדר לאחסן את המשתנה באוגר המעבד (register) במקום בזיכרון הראשי. זה יכול להאיץ את הגישה למשתנה. עם זאת, זו רק המלצה למהדר, והוא רשאי להתעלם ממנה. לא ניתן לקחת את הכתובת של משתנה register באמצעות האופרטור &.
- typedef: טכנית אינו מאפיין אחסון במובן המסורתי, אלא הוא מילת מפתח המשמשת ליצירת שמות חלופיים (כינויים) עבור סוגי נתונים קיימים. זה משפר את קריאות הקוד.
- const: כפי שהוסבר קודם, מציין שהערך של משתנה לא ניתן לשינוי לאחר האתחול. הוא יכול לשמש בשילוב עם מאפייני אחסון אחרים (למשל, static const int).
- volatile: מציין שהערך של משתנה יכול להשתנות על ידי גורמים חיצוניים לתכנית (למשל, חומרה, פונקציות שירות פסיקה). זה מונע מהמהדר לבצע אופטימיזציות מסוימות על משתנים אלה, כיוון שהוא חייב לקרוא את ערכם מהזיכרון בכל פעם שהוא נדרש.

הנחיות קדם-מעבד (Preprocessor Directives / מאקרו)

הנחיות קדם-מעבד הן פקודות שמעובדות על ידי קדם-המעבד (preprocessor) לפני שהמהדר עצמו מתחיל את תהליך הקומפילציה האמיתי. הן מתחילות בסימן סולמית (#). אחד השימושים הנפוצים ביותר בהנחיות קדם-מעבד הוא הגדרת מאקרו.

מאקרו (#define):

מאקרו הוא קטע קוד או שם שמוחלף על ידי ערך או ביטוי מסוים על ידי קדם-המעבד לפני הקומפילציה. קיימים שני סוגים עיקריים של מאקרו:

1. **מאקרו דמוי-קבוע (Object-like Macro):** מגדיר שם שמייצג ערך קבוע.

```
1. #define PI 3.14159
2. #define MAX_SIZE 100
3.
```

בכל מקום בקוד שבו יופיע PI, הוא יוחלף ב-3.14159 לפני הקומפילציה.

2. **מאקרו דמוי-פונקציה (Function-like Macro):** מגדיר שם שמייצג קטע קוד שיכול לקבל ארגומנטים.

```
1. #define SQUARE(x) ((x) * (x))
2. #define MAX(a, b) ((a) > (b) ? (a) : (b))
3.
```

כאשר משתמשים במאקרו דמוי-פונקציה בקוד, קדם-המעבד יחליף את הקריאה למאקרו בקטע הקוד המוגדר, תוך החלפת הארגומנטים.

```
 1. #include <stdio.h>
 2.
 3. #define SQUARE(x) ((x) * (x))
 4. #define MAX(a, b) ((a) > (b) ? (a) : (b))
 5.
 6. int main() {
 7.     int num = 5;
 8.     int squared = SQUARE(num); // Will be replaced by: int squared = ((num) * (num));
 9.     printf("Square of %d is %d\n", num, squared); // Square of 5 is 25
10.
11.     int a = 10, b = 20;
12.     int maximum = MAX(a, b); // Will be replaced by: int maximum = ((a) > (b) ? (a) : (b));
13.     printf("Maximum of %d and %d is %d\n", a, b, maximum); // Maximum of 10 and 20 is 20
14.
15.     return 0;
16. }
17.
```

הנחיות קדם-מעבד נוספות נפוצות:

- #include: כולל את התוכן של קובץ אחר (בדרך כלל קובץ כותרת .h) בקובץ המקור הנוכחי.
- #ifdef, #ifndef, #else, #endif: משמשות לקומפילציה מותנית של קטעי קוד, בהתאם אם מאקרו מסוים מוגדר או לא.
- #if, #elif: מאפשרות קומפילציה מותנית על סמך ערך של ביטוי קבוע.
- #undef: מבטל את ההגדרה של מאקרו.
- #error: גורם לקדם-המעבד להפיק הודעת שגיאה ולעצור את הקומפילציה.
- #pragma: הנחיה ספציפית למהדר שיכולה לשלוט בהתנהגות המהדר (למשל, השבתת אזהרות).

זהירות עם מאקרו דמוי-פונקציה:

חשוב להשתמש בסוגריים סביב כל ארגומנט בתוך הגדרת המאקרו כדי למנוע בעיות בלתי צפויות עקב סדר פעולות. לדוגמה, בלי הסוגריים ב-SQUARE(x), הביטוי SQUARE(a + b) היה מתורגם ל- a + b * a +, מה שלא סביר שזו הכוונה.