Contents

Chapter 2

In this chapter, we begin by introducing pointers—how they work, how to manipulate them, and why they are essential for managing dynamic memory. Next, we explore the different types of linked lists, their internal representation using pointers, and how to perform basic operations such as insertion, deletion, and traversal.

2.1 Pointers

Pointers are essential in C for memory management, efficient data manipulation, and implementing data structures like linked lists.

2.1.1 What is a Pointer

A pointer is a variable that stores the memory address of another variable. Instead of holding a value directly, it refers to the location in memory where the value is stored.

2.1.2 Declaring and Initializing Pointers

To declare a pointer variable, we use the * symbol before the pointer's name.

```
1 int *ptr;  // pointer to an integer
```

To store the address of a variable in a pointer, we use the address-of operator &:

```
1 int x = 10;
2 ptr = &x;  // ptr now points to x
```

2.1.3 Dereferencing a Pointer

Using the dereference operator *, we can access or modify the value at the memory location pointed to:

```
1 printf("%d", *ptr);  // prints value at the address (10)
```

Symbol	Meaning
*	Used to declare or dereference a pointer
&	Address-of operator

Example

```c
#include <stdio.h>

int main() {
    int x = 5;
    int *p = &x;

    printf("Value of x: %d\n", x);
    printf("Address of x: %p\n", (void*)&x);
    printf("Value stored in p: %p\n", (void*)p);
    printf("Value pointed to by p: %d\n", *p);

    return 0;
}
```

Expected Output:

```
Value of x: 5
Address of x: 0x7ffee85d6a2c
Value stored in p: 0x7ffee85d6a2c
Value pointed to by p: 5
```

2.1.4 Pointers and Arrays

An array name acts as a pointer to its first element:

```c
int arr[] = {10, 20, 30};
int *p = arr;
printf("%d\n", *p);  // prints 10
```

Pointer Arithmetic

Pointers can be incremented or decremented to move through elements in an array:

```c
int arr[] = {10, 20, 30};
int *p = arr;
printf("%d\n", *(p + 1));  // prints 20 (second element)
```

2.1.5 Pointers and Functions

Pointers allow functions to modify values outside their scope:

```c
#include <stdio.h>

void changeValue(int *nbr){
    *nbr = 50;
}

int main() {
    int x = 10;
    changeValue(&x);
    printf("%d", x);  // prints 50
    return 0;
}
```

> **Note:** When passing an array to a function, the address of its first element is passed implicitly.

2.1.6 Dynamic Memory Allocation

Dynamic memory allocation assigns memory at run time using functions like `malloc()` from `stdlib.h`.

```c
#include <stdlib.h>

int *p = (int*) malloc(sizeof(int));  // allocate space for 1 int
*p = 42;
free(p);  // always free allocated memory
```

To create a dynamic array:

```c
int *arr = (int*) malloc(sizeof(int) * 5);
if (arr != NULL) {
    arr[0] = 1;
    arr[1] = 2;
    // ...
    free(arr);
}
```

2.1.7 Pointer to Pointer

A pointer can also store the address of another pointer:

```c
int x = 5;
int *p = &x;
int **pp = &p;

printf("%d", **pp);  // prints 5
```

2.1.8 NULL and void Pointers

NULL Pointer: A pointer that doesn't point to any memory location.

```
1 int *p = NULL;
```

Void Pointer: A generic pointer that can point to any data type:

```
1 void *vp;
2 int x = 10;
3 vp = &x;
4 printf("%d", *(int*)vp);  // typecast before dereferencing
```

2.1.9 Exercises

Consider the following declarations:

```
1 int x = 5;
2 int *p = &x;
3 float y = 10.0;
4 int arr[] = {1, 9, 7, 4, 5};
5 int *ptr = arr;
```

1. What is the size of memory occupied by each of the following variables:
   - `x` (Hint: size of an `int`)
   - `p` (Hint: size of a pointer to `int`)
   - `y` (Hint: size of a `float`)
   - `arr` (Hint: 5 integers)
   - `ptr` (Hint: pointer to int)

   Note: The actual size depends on the system architecture. On most 64-bit systems:
   - `sizeof(int *)` = 8 bytes
2. What does each of the following expressions return?
   - `*p` (Value pointed to by `p`, i.e., `5`)
   - `p` (Address of `x`)
   - `&p` (Address of pointer variable `p`)
   - `ptr` (Address of first element in `arr`)
   - `*ptr` (Value of first element in `arr`, i.e., `1`)
   - `arr` (Address of first element in `arr`)
   - `*(arr)` (Same as `arr[0]`, i.e., `1`)
   - `*(arr + 0)` (Same as `arr[0]`, i.e., `1`)
   - `*(arr + 4)` (Same as `arr[4]`, i.e., `5`)

2.2 Linked Lists

Linked lists are fundamental data structures that play a crucial role in computer science and programming. Unlike arrays, which store elements in contiguous memory locations, linked lists organize data in a sequence of nodes, where each node points to the next one in the chain.

2.2.1 Definition

A linked list is a data structure that consists of a sequence of nodes. Each node contains two components: the data itself and a pointer to the next node. This pointer-based design enables the linked list to grow or shrink without the need for memory reallocation, offering flexibility and performance benefits in many scenarios.

2.2.2 Implementing a Linked List

To implement a linked list in C, we define a node structure that contains two fields: one for data and another for the pointer to the next node. The following code demonstrates how to define a node, create a simple linked list with three nodes, and print its contents.

Node Structure

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

struct Node {
    int data;
    List* next;
};
typedef struct Node List;
```

Creating Nodes and Linking Them

We dynamically allocate memory for each node and link them together to form a list.

```c
int main() {
    List* head = NULL;
    List* second = NULL;
    List* third = NULL;

    // Allocate nodes in the heap
    head = (List*)malloc(sizeof(List));
    second = (List*)malloc(sizeof(List));
    third = (List*)malloc(sizeof(List));

    // Assign data and link nodes
```

```
12      head->data = 1;
13      head->next = second;
14
15      second->data = 2;
16      second->next = third;
17
18      third->data = 3;
19      third->next = NULL;
20
21      return 0;
22  }
```

2.2.3 Traversal of a Linked List

Traversal of a linked list refers to the process of visiting and accessing each node in the list, usually starting from the head and moving sequentially to the next node until the end (or until a specific condition is met).

Iterative Traversal

- Using a loop to move through the linked list.

```
1  void traverse(List* head) {
2      List* p = head;
3      while (p != NULL) {
4          printf("%d␣->␣", p->data);
5          p = p->next;
6      }
7      printf("NULL\n");
8  }
```

Recursive Traversal

- Uses recursion to process each node before moving to the next one.

```
1  void traverseRecursive(List* head) {
2      if (head != NULL) {
3      printf("%d␣->␣", head->data);
4      traverseRecursive(head->next);
5      }
6  }
```

Traversal is useful for various operations like searching, modifying data, or printing elements of the linked list.

2.2.4 Insertion Operations

Insertion operations allow us to add new nodes at various positions in the list.

Insertion at the Beginning

To insert a node at the beginning of the list:

- Create a new node.
- Point its `next` to the current head.
- Update the head to the new node.

```
1  void insertAtBeginning(List** head, int value) {
2      List* newNode = (List*)malloc(sizeof(List));
3      newNode->data = value;
4      newNode->next = *head;
5      *head = newNode;
6  }
```

Insertion at the End

To insert a node at the end of the list:

- Create a new node.
- Traverse the list to the last node.
- Set the last node's `next` to the new node.

```
1  void insertAtEnd(List** head, int value) {
2      List* newNode = (List*)malloc(sizeof(List));
3      List* P = *head;
4      newNode->data = value;
5      newNode->next = NULL;
6
7      if (*head == NULL) {
8          *head = newNode;
9          return;
10     }
11
12     while (P->next != NULL) {
13         P = P->next;
14     }
15     P->next = newNode;
16 }
```

Insertion at a Specific Position

To insert a node at a given position (starting from 1):

- Traverse to the node just before the desired position.
- Adjust the `next` pointers to insert the new node.

```
void insertAtPosition(List** head, int value, int position) {
    List* newNode = (List*)malloc(sizeof(List));
    newNode->data = value;

    if (position == 1) {
        newNode->next = *head;
        *head = newNode;
    }
    else {
    List* P = *head;
    for (int i = 0;  i < position - 1; i++) {
        if (P == NULL) break;
        P = P->next;
    }
    if (P != NULL) {
    newNode->next = P->next;
    P->next = newNode;
    }

    }
}
```

2.2.5 Deletion Operations in a Linked List

Deletion in a linked list involves removing a node and properly updating the pointers to maintain the structure of the list.

Deletion at the Beginning

To delete a node from the beginning of the list:

- Store the current head in a temporary pointer.
- Move the head pointer to the next node.
- Free the memory of the old head.

```
void deleteAtBeginning(List** head) {
    if (*head != NULL) {
    List* p = *head;
    *head = (*head)->next;
```

```
5       free(p);
6       }
7   }
```

Deletion at the End

To delete a node from the end of the list:

- If the list is empty, do nothing.
- If there is only one node, delete it and update head to NULL.
- Otherwise, traverse to the second last node.
- Free the last node and set the second last node's `next` to NULL.

```
1   void deleteAtEnd(List** head) {
2       if (*head != NULL) {
3       if ((*head)->next == NULL) {
4           free(*head);
5           *head = NULL;
6       }
7       else {
8            List* prec = prec;
9            List* p = (*head)->next;
10           while (p->next != NULL) {
11              prec = p;
12              p = p->next;
13           }
14      }
15      prec->next = NULL;
16      free(p);
17      }
18  }
```

2.2.6 Types of Linked Lists

The lists we have used so far are **single-linked lists**, where each node contains data and a pointer to the next node. However, there are several other types of linked lists in C, each designed for specific use cases. The primary types include:

- Circular Singly Linked List
- Doubly Linked List
- Circular Doubly Linked List

2.2.7 Circular Singly Linked List

In a circular singly linked list, the last node does not point to `NULL`; instead, it points back to the head node, forming a closed loop.

```
List* head = malloc(sizeof(List));
List* second = malloc(sizeof(List));
List* third = malloc(sizeof(List));

head->data = 1;
head->next = second;

second->data = 2;
second->next = third;

third->data = 3;
third->next = head; // Points back to head
```

2.2.8 Insertion in a Circular Singly Linked List

To maintain the circular structure, special care must be taken when inserting elements.

Insertion at the Beginning

To insert a new node at the beginning of a circular singly linked list:

- Allocate memory for the new node.
- Assign its `next` to the current head.
- Traverse the list to find the last node.
- Update the last node's `next` to point to the new node.
- Update the head to point to the new node.

```
void insertAtBeginning(List** head, int data) {
    List* newNode = (List*)malloc(sizeof(List));
    newNode->data = data;

    if (*head == NULL) {
        newNode->next = newNode;
        *head = newNode;
    } else {
        List* p = *head;
        while (p->next != *head) {
            p = p->next;
        }
        newNode->next = *head;
```

```
14          p->next = newNode;
15          *head = newNode;
16      }
17  }
```

Insertion at the End

To insert a node at the end of a circular singly linked list:

- Allocate memory for the new node.
- Assign its `next` to the head.
- Traverse the list to find the last node.
- Update the last node's `next` to point to the new node.

```
1  void insertAtEnd(List** head, int data) {
2      List* newNode = (List*)malloc(sizeof(List));
3      newNode->data = data;
4
5      if (*head == NULL) {
6          newNode->next = newNode;
7          *head = newNode;
8      } else {
9          List* p = *head;
10         while (p->next != *head) {
11             p = p->next;
12         }
13         p->next = newNode;
14         newNode->next = *head;
15     }
16 }
```

These operations ensure that the circular structure is preserved after each insertion. Deletion operations follow similar principles and must also preserve the loop by carefully updating links.

2.2.9 Doubly Linked List

A doubly linked list is a type of linked list where each node contains two pointers: one pointing to the next node and another pointing to the previous node. This bidirectional structure allows traversal in both forward and backward directions, making operations like insertion and deletion more flexible compared to singly linked lists.

```
1  typedef struct Node {
2      int data;
3      struct Node* next;
4      struct Node* prev;
5  } List;
```

Here is an example of manually creating a doubly linked list with three nodes:

```
List* head = malloc(sizeof(List));
List* second = malloc(sizeof(List));
List* third = malloc(sizeof(List));

head->data = 1;
head->prev = NULL;
head->next = second;

second->data = 2;
second->prev = head;
second->next = third;

third->data = 3;
third->prev = second;
third->next = NULL;
```

2.2.10 Insertion in a Doubly Linked List

In a doubly linked list, insertion operations must update both the `next` and `prev` pointers of the affected nodes to preserve the two-way links.

Insertion at the Beginning

To insert a new node at the beginning:

- Allocate memory for the new node and assign its data.
- Set its `next` to the current head.
- Set its `prev` to `NULL`.
- If the list is not empty, update the current head's `prev` to point to the new node.
- Update the head pointer to the new node.

```
void insertAtBeginning(List** head, int data) {
    List* newNode = (List*)malloc(sizeof(List));
    newNode->data = data;
    newNode->next = *head;
    newNode->prev = NULL;

    if (*head != NULL) {
        (*head)->prev = newNode;
    }

    *head = newNode;
}
```

Insertion at the End

To insert a new node at the end of a doubly linked list:

- Allocate memory for the new node and assign its data.
- If the list is empty, set its `prev` to `NULL` and make it the head.
- Otherwise, traverse to the last node.
- Update the last node's `next` to the new node.
- Set the new node's `prev` to the last node and `next` to `NULL`.

```
1  void insertAtEnd(List** head, int data) {
2      List* newNode = (List*)malloc(sizeof(List));
3      newNode->data = data;
4      newNode->next = NULL;
5
6      if (*head == NULL) {
7          newNode->prev = NULL;
8          *head = newNode;
9          return;
10     }
11
12     List* p = *head;
13     while (p->next != NULL) {
14         p = p->next;
15     }
16
17     p->next = newNode;
18     newNode->prev = p;
19 }
```

These operations ensure the list remains properly linked in both directions, allowing efficient traversal and updates from either end.

2.2.11 Circular Doubly Linked List

A circular doubly linked list is a variation of the doubly linked list where the last node's `next` pointer points back to the first node, and the first node's `prev` pointer points to the last node. This creates a continuous loop in the list.
Here's an example of creating a circular doubly linked list with three nodes:

```
1  List* head = malloc(sizeof(List));
2  List* second = malloc(sizeof(List));
3  List* third = malloc(sizeof(List));
4
5  head->data = 1;
6  head->prev = third;
```

```
head->next = second;

second->data = 2;
second->prev = head;
second->next = third;

third->data = 3;
third->prev = second;
third->next = head;  // Points back to the head, completing the circle
```

2.2.12 Insertion in a Circular Doubly Linked List

Special care must be taken to preserve the circular and bidirectional links during insertions.

Insertion at the Beginning

To insert a new node at the beginning:

- Allocate memory for the new node and assign its data.
- If the list is empty:
  - Set the new node's `next` and `prev` to point to itself.
  - Set the head to the new node.
- Otherwise:
  - Traverse to the last node by using `head->prev`.
  - Set the new node's `next` to the current head.
  - Set the new node's `prev` to the last node.
  - Update the current head's `prev` and the last node's `next` to point to the new node.
  - Update the head pointer to point to the new node.

```
void insertAtBeginning(List** head, int data) {
    List* newNode = (List*)malloc(sizeof(List));
    newNode->data = data;

    if (*head == NULL) {
        newNode->next = newNode;
        newNode->prev = newNode;
        *head = newNode;
    } else {
        List* last = (*head)->prev;

        newNode->next = *head;
        newNode->prev = last;
```

```
        last->next = newNode;
        (*head)->prev = newNode;
        *head = newNode;
    }
}
```

Insertion at the End

To insert a new node at the end:

- Allocate memory for the new node and assign its data.
- If the list is empty:
  - Set the new node's `next` and `prev` to point to itself.
  - Set the head to the new node.
- Otherwise:
  - Let the last node be `head->prev`.
  - Set the new node's `next` to point to the head.
  - Set the new node's `prev` to point to the last node.
  - Update the last node's `next` and the head's `prev` to point to the new node.

```
void insertAtEnd(List** head, int data) {
    List* newNode = (List*)malloc(sizeof(List));
    newNode->data = data;

    if (*head == NULL) {
        newNode->next = newNode;
        newNode->prev = newNode;
        *head = newNode;
    } else {
        List* last = (*head)->prev;

        newNode->next = *head;
        newNode->prev = last;
        last->next = newNode;
        (*head)->prev = newNode;
    }
}
```

These operations ensure that the circular and doubly-linked properties are preserved during insertions at both ends, maintaining the circular structure of the list.