Department of Computer Science and Engineering,
PES University, Bangalore, India

Lecture Notes
Problem Solving With C
UE24CS151B

Lecture #19
Linked List Implementation in C

By,

Prof. Sindhu R Pai,

Theory Anchor, Feb-May, 2025

Assistant Professor

Dept. of CSE, PESU

Many Thanks to

Dr. Shylaja S S (Director, CCBD and CDSAML Research Center, PES University)

Prof. Nitin V Pujari (Dean, Internal Quality Assurance Cell, PES University)

Unit #: 3

Unit Name: Text Processing and User-Defined Types

Topic: Linked List Implementation in C

Course objectives: The objective(s) of this course is to make students

- Acquire knowledge on how to solve relevant and logical problems using computing Machine.
- Map algorithmic solutions to relevant features of C programming language constructs.
- Gain knowledge about C constructs and its associated ecosystem.
- Appreciate and gain knowledge about the issues with C Standards and it's respective behaviours.

Course outcomes: At the end of the course, the student will be able to:

- Understand and Apply algorithmic solutions to counting problems using appropriate C Constructs.
- Understand, Analyze and Apply sorting and Searching techniques.
- Understand, Analyze and Apply text processing and string manipulation methods using Arrays, Pointers and functions.
- Understand user defined type creation and implement the same using C structures, unions and other ways by reading and storing the data in secondary systems which are portable.

Sindhu R Pai

Theory Anchor, Feb - May, 2025

Dept. of CSE,

PES University

Introduction

A linked list is a collection **of nodes that are stored at different locations in memory but are logically connected using links** (pointers). Each node usually contains some set of data and a pointer that refers to the next node in the sequence. The node and link has a special meaning which we will be discussing in this chapter. Before diving deeper, let's first address a few points.

Point #1: Is a pointer allowed as a data member in a structure? Absolutely.

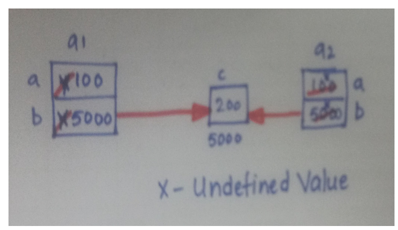
Consider the below structure. Let us see some of the versions of accessing the data members.

```
struct A
{       int a;    int *b;       };
```

Coding Example_1:

Version 1:

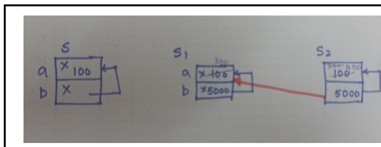
```
int main()
{
        struct A a1; struct A a2; a1.a = 100;
        int c = 200; a1.b = &c;
        printf("a1 values:%d and %d\n", a1.a, *(a1.b));
        a2 = a1;
        printf("a2 values:%d and %d\n", a2.a, *(a2.b));
        a1.a = 300;
        *(a1.b) = 400;
        printf("a2 values:%d and %d\n", a2.a, *(a2.b));
        return 0;
}
```




```
C:\Users\Dell>a
a1 values:100 and 200
a2 values:100 and 200
a2 values:100 and 400
```

Version 2:

```
#include<stdio.h>
int main()
{
        struct A s;       s.a = 100;       s.b = &(s.a);
        printf("%d %d",s.a,*(s.b));
        struct A s1;      s1.a = 100;      s1.b = &(s1.a);
        printf("%d %d\n",s1.a,*(s1.b));
        struct A s2 = s1;
        printf("%p %p\n",s1.b,s2.b);
        printf("%d %d\n",s2.a,*(s2.b));
```

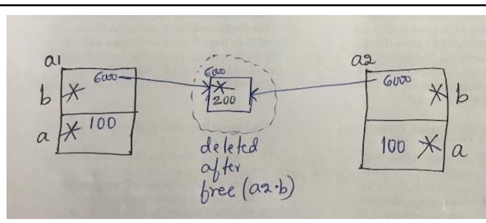


```
        s2.a = 200;     printf("%p %p\n",s1.b,s2.b);
        printf("%d %d\n",s1.a,*(s1.b));
        printf("%d %d\n",s2.a,*(s2.b)); // very imp
        *(s2.b) = 300; printf("%p %p\n",s1.b,s2.b);
        printf("%d %d\n",s1.a,*(s1.b));
        printf("%d %d\n",s2.a,*(s2.b));
        s2.b = &(s2.a);
        *(s2.b) = 400;
        printf("%p %p\n",s1.b,s2.b);  printf("%d %d\n",s1.a,*(s1.b));
        printf("%d %d\n",s2.a,*(s2.b));        return 0;
}
```

```
C:\Users\Dell>a
100 100100 100
0061FF10 0061FF10
100 100
0061FF10 0061FF10
100 100
        200 100
0061FF10 0061FF10
300 300
200 300
0061FF10 0061FF08
300 300
400 400
```

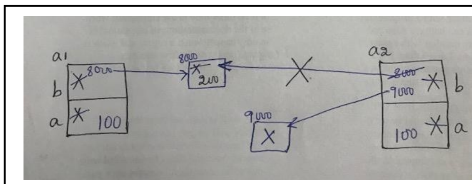
Version 3: Dynamic Memory Management and dereferencing the dangling pointer

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
        struct A a1; struct A a2; a1.a = 100;
        a1.b = (int*) malloc(sizeof(int));
        *(a1.b) = 200;
        printf("a1 values:%d and %d\n", a1.a, *(a1.b)); // 100 200
        a2 = a1;
        printf("a2 values:%d and %d\n", a2.a, *(a2.b)); // 100 200
        free(a2.b); // a1.b too becomes dangling pointer
        printf("a1 values:%d and %d\n", a1.a, *(a1.b)); // 100 undefined behaviour
        return 0;
}
```




```
C:\Users\Dell>a
a1 values:100 and 200
a2 values:100 and 200
a1 values:100 and 14619624
```

Version 4:




```
int main()
{
        struct A a1; struct A a2; a1.a = 100;
        a1.b = (int*) malloc(sizeof(int));
        *(a1.b) = 200;
```

```
    printf("a1 values:%d and %d\n", a1.a, *(a1.b)); // 100 200
    a2 = a1;
    a2.b = (int*) malloc(sizeof(int)); // changing this to a1.b creates garbage and output differs
    printf("a2 values:%d and %d\n", a2.a, *(a2.b));
    printf("a1 values:%d and %d\n", a1.a, *(a1.b));
    return 0;
}
```

```
C:\Users\Dell>a
a1 values:100 and 200
a2 values:100 and 12193256
a1 values:100 and 200
```

Point #2: Can a structure contain another structure as a member? Absolutely. This is one type of nesting the structure inside another – Nested structures

Coding Example_2:

```
struct A {    int a;     };
struct B{     int a;   struct A a1;
}; // structure variable a1 as a data member in B
int main()
{
    printf("sizeof A %d\n",sizeof(structA);
    printf("sizeof B %d\n",sizeof(struct B));  // more than A
    return 0;
}
```

```
C:\Users\Dell>a
sizeof A 4
sizeof B 8
```

Point #3: Can we have a structure variable inside the same structure? – No.

Coding Example_3: Results in Compiletime Error

```
struct A {    int a;   struct A a1;     };
```

```
C:\Users\Dell>gcc -c check.c -w
check.c:1:30: error: field 'a1' has incomplete type
 struct A {  int a;  struct A a1;       };
                              ^~
```

The solution to the earlier issue is to **include a pointer to the same structure type** within the structure itself. Since the **size of a pointer is fixed for any given system**, the **compiler can determine the total size of the structure at compile time.** This leads to the usage of **Self-Referential Structures.**

Self Referential Structures

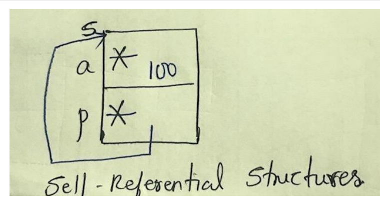
A structure which has a **pointer to itself as a data member** is called a self - referential structure. In this type of structure, the pointer inside the structure points to the same structure. Structure can have one or more pointers pointing to the same type of structure as their member. It is **widely used in building dynamic data models such as trees, linked lists, graphs** etc. where each element needs to connect to others of the same type. Usually, such a structure is called as a **Node.**

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

This allows creating a chain of nodes dynamically linked together, forming the basis for structures like linked lists.

Coding Example_4: Recursive Pointer Access in Self-Referential Struct

```
struct Sample  {       int a;      struct Sample *p;       };
int main()
{
        printf("%d\n",sizeof(struct Sample) );
        struct Sample s;
        s.a = 100;
        s.p = &s;
        printf("%d %d %d\n", s.a, s.p->a, s.p->p->a);
        return 0;
}
```




```
C:\Users\Dell>a
8
100 100 100
```

Linked List in C

When self referential structures are linked together, they form a **Linked list — a dynamic and flexible way of storing and managing sequential data.** A linked list is a collection **of nodes that are stored at different locations in memory but are logically connected using links** (pointers). Each node usually contains some set of data and a pointer that refers to the next node in the sequence.

Characteristics of Linked List

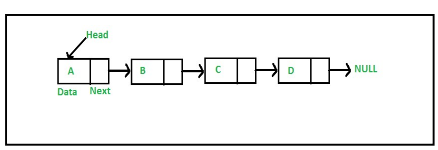
1. A data structure that consists of **zero or more nodes.** Every node is composed of two fields: **a data/component field and a pointer field**. The pointer field of every node point to the next node in the sequence.
2. We can access the nodes one after the other. There is **no way to access the node directly as random access is not possible in a linked list. Lists have sequential access**.
3. **Insertion and deletion in a list at a given position requires no shifting of elements.**

Operations on Linked List

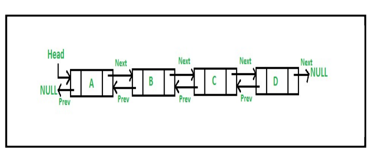
- **Insertion Operation on the list** includes Insertion at the beginning of the list, Insertion at the end of the list and Insertion at a Specific Node in the List
- **Deletion Operation on the list** includes Deletion at the beginning of the list, Deletion at the end of the list and Deletion of a Specific Node in the List**.**
- **Other Operations on the list** such as Traversing the list, Searching the list and Sorting the List, Merging two lists, Finding the Union, Set operations on nodes of the list,FindMin, FindMx, Find Repeated etc

Types of Linked List

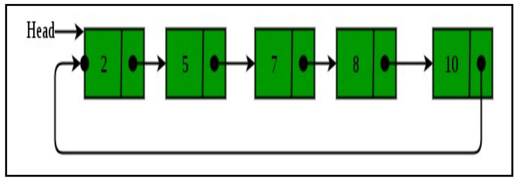
i. Singly Linked List



ii. Doubly Linked List



iii. Circular Linked List

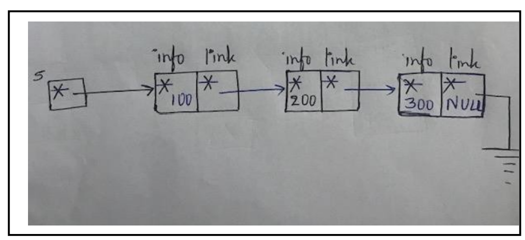


Singly Linked List

.A singly linked list consists of a **series of nodes, where each node holds data and a pointer to the next node** in the list. To build and work with a singly linked list efficiently, we'll focus on three fundamental operations: **insertion**, **traversal**, and **deletion**. These operations enable us **to add nodes, navigate through the list, and remove nodes**. As we implement these operations, it will become clear how **self-referential structures** enable flexible and dynamic handling of data in memory.

Implementation:

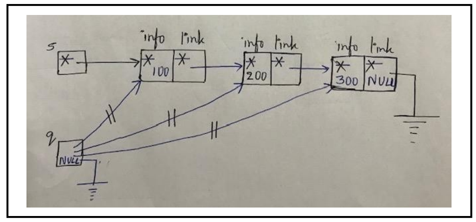
```
// Self referential structure template/entity
struct node
{
        int info; // component field
        struct node *link; // pointer field
};
typedef struct node NODE_T;    // Creation of alias/type

// Client code
#include<stdio.h>
int main()
{
        NODE_T *s;
        s = (NODE_T*) malloc(sizeof(NODE_T));
        s->info = 100;
        s->link = (NODE_T*) malloc(sizeof(NODE_T));
        s->link->info = 200;
        s->link->link = (NODE_T*) malloc(sizeof(NODE_T));
        s->link->link-> info = 300;
        s->link->link -> link = NULL;
        display(s); // Function call to display all nodes
        freelist(s);     // Function call to free all nodes
}
```




```
C:\Users\Dell>a
100     200     300
100 deleted
200 deleted
300 deleted
```

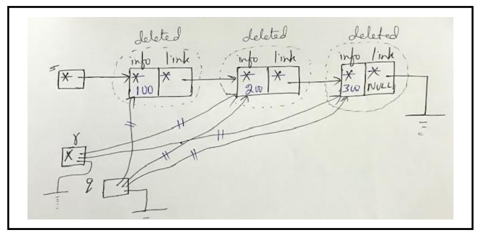
// Function Implementations
// To display all nodes in the list

```
void display(NODE_T* q)
{
        while(q != NULL)
        {
                printf("%d\t", q->info);
                q = q->link;
        }
}
```



// To delete all nodes in the list

```
void freelist(NODE_T* q)
{
        NODE_T* r;
        while(q != NULL)
        {
                printf("\n%d deleted",q->info);
                r = q->link;
                free(q);
                q = r;
        }
 }
```



Points to Think!

- Why can't we say just free(s) rather than writing freelist function and calling it in the client?
- Is the client code creating any dangling pointer at the end of execution? If yes, what guideline to be followed to avoid this?
- Can we write a create_node() function which can the data from the user and return the node?
- Can we include a definition of insert_to_list() function which can insert the node only if user wants to add a node to the list.
- To include these operations, menu driven application might help you. Try it yourself!!!

Enjoy Exploring Linked Lists!!