Catalog

STRUCTURE

Structure is a user defined data type that allows you to hold ***different data type of elements***. Each element in the structure is called as member. It is used to store student information, employee information, product information, book information etc.

Defining structure

A structure is declared by using the ***keyword 'struct'*** followed by a structure name followed by the body of the structure. The members of the structure are defined within the body of the structure.

Syntax:

```
 struct  structurename
{
            Datatype    member1;
            Datatype     member2;
               . .  .
            Datatype     member n;
} structurevariable1, structurevariable2, . . . structurevariablen;
```

Example:

```
struct student
{
            int RegNo;
            char name[20], dept[10];
}S1, S2;
```

Here, student" is a structure name, S1 & S2 are structure variables and RegNo, name and dept are structure members.

Declaring structure

We can declare variable for the structure, so that we can access the member of structure easily. There are two ways to declare structure variable:

1. By using struct keyword within main() function

2. By declaring variable at the time of defining structure.

Method1:

It should be declared within the main function.

```
struct employee
{
        int id;
        char name[50];
        float salary;
};
void main
{
        struct employee e1, e2;
}
```

Method2:

Declare the variable of the structure at the time of defining the structure.

```
struct employee
{
        int id;
        char name[50];
        float salary;
}e1,e2;
```

Which approach is good

- If no. of variable are not fixed, use method1. It provides you flexibility to declare the structure variable many times.
- If no. of variables are fixed, use 2nd approach. It saves your code to declare variable in main() fuction.

Accessing members of structure

There are two ways to access structure members:

1. By using . (member or dot operator)
2. By using -> (structure pointer operator)

Syntax:

structurevariable**.**membername

structurevariable→membername

Example:

S1.name

S1→name

Initialization of Structure

Like normal variables, the structure variables can also be initialized. A structure is initialized by assigning some constants to the members of the structure. By default, the members of type int and float are initialized to zero and the members of type char and string are initialized to „\0"

Method 1

```
struct student
{
        int Reg_No;
        char name[20];
}S1 = {4001, "Joshna"};
```

Method 2

```
struct student
{
        int Reg_No;
        char name[20];
};
void main()
{
        struct student S1 = {4001, "Joshna"};
}
```

Program: Store student Information and Display it Using Structure

```
#include <stdio.h>
#include<conio.h>
struct student
```

```
{
        char name[50];
        int roll;
        float marks;
} s;

void  main()
{
        printf("Enter student information:\n");
        printf("Enter student name: ");
        scanf("%s", s.name);
         printf("Enter roll number: ");
        scanf("%d", &s.roll);
        printf("Enter marks: ");
        scanf("%f", &s.marks);
         printf("Displaying Information:\n");
        printf("Name: %s\n”,s.name);
        printf("Roll number: %d\n",s.roll);
        printf("Marks: %f\n", s.marks);
        getch();
}
```

Output

```
Enter information:
Enter name: Jack
Enter roll number: 23
Enter marks: 34.5
Displaying Information:
Name: Jack
Roll number: 23
Marks: 34.5
```

NESTED STRUCTURES

A structure can be placed within another structure is called nesting of structures. Structures can contain other structures as members. Nested structure is also called as ***structure within structure.***

Declaration of nested structure

There are two ways to declare nested structure in c language:

a) By separate structure
b) By Embedded structure

Separate structure

We can create 2 structures, but dependent structure should be used inside the main structure as a member.

Syntax

```
struct structure1
 {
       - - - - - - - - - -
       - - - - - - - - - -
 };
 struct structure2
 {
       - - - - - - - - - -
       - - - - - - - - - -
       struct structure1 v1;
 }v2;
```

Example

```
struct Date
{
        int  dd;
        int mm;
        int yyyy;
};
struct Employee
{
        int id;
        char name[20];
        struct  Date doj;
}emp1;
```

a) Embedded structure

We can define structure within the structure also. It requires less code than previous way. But it can't be used in many structures.

Syntax:

```
struct structure1
 {
        - - -
        - - -
        struct structure2
        {

                } v1;
}v2;
```

Example

```
struct Employee
{
    int id;
    char name[20];
    struct Date
    {
        int dd;
        int mm;
        int yyyy;
    }doj;
}emp;
```

Program

```
#include <stdio.h>
#include<conio.h>
struct Employee
{
        char name[20];
        int    no;
        float  salary;
        struct    date
        {
                int    date;
                int     month;
                int    year;
        }doj;
}emp;

void  main()
{
        printf("\nEnter Employee Name : ");

        scanf("%s",&emp.name);

        printf("Employee number:");

        scanf("%d",& emp.no);

        printf("\nEmployee Salary : ");
```

```
            scanf(“%f",&emp.salary);
            printf("\nEmployee DOJ : “);
            sacnf(“%d/%d/%d", &emp.doj.date,&emp.doj.month,&emp.doj.year);
            printf(“*******Employee information”);
            printf("\nEmployee  Name  :  %s",emp.name);
            printf("\nEmployee  Number  :  %d",emp.no);
            printf("\nEmployee Salary : %f",emp.salary);
            printf("Employee DOJ: %d/%d/%d",emp.doj.date,emp.doj.month, emp.doj.year );
            getch();
        }
```

Output :

```
Employee Name : Jayden
Employee Number : 1000
Employee Salary : 1000.500000
Employee DOJ    : 22/6/1990
```

POINTER AND STRUCTURES

Structure is a user defined data type that allows you to hold ***different data type of elements***. Structures can be created and accessed using pointers. A pointer variable of a structure can be created as below:

Syntax:

```
struct  structurename
{
        Datatype    member1;
        Datatype    member2;
           . . .
        Datatype    member n;
} *ptr;
```

(or)

```
struct  structurename
{
        Datatype    member1;


         Datatype    member2;
            . . .
         Datatype    member n;
 };
 void main()
 {
         struct  structurename  *ptr;
 }
```

Accessing the members of a structure by pointer

We use the ***arrow operator*** **->** (member selection operator) to access the members of a structure via pointer variable.

Syntax

```
ptrName->member
```

Example

```
printf("First Name: %s", ptr->firstname);
```

Assigning structure variable to pointer

We use the following syntax to assign a structure variable address to a pointer.

```
ptrName = &structVarName;
```

In the following example we are assigning the address of the structure variable std to the structure pointer variable ptr. So, ptr is pointing at std.

Example

```
ptr = &std;
```

Program:

```
#include  <stdio.h>
#include <conio.h>
struct student
{
	char name[50];
	int roll;
	float marks;
}s;
```

```
void main()
{
        struct student *ptr;
        ptr = &s;

        printf("Enter student information:\n");
        printf("Enter student name: ");
        scanf("%s", &ptr→name);
        printf("Enter roll number: ");
        scanf("%d",  &ptr→roll);
        printf("Enter marks: ");
        scanf("%f", &ptr→marks);
        printf("Displaying Information:\n");
        printf("Name: %s\n”,ptr→name);
        printf("Roll number: %d\n",ptr→roll);
        printf("Marks: %f\n", ptr→marks);
        getch();
}
```

Output

```
Enter student information:
Enter student name: Jacks
Enter roll number: 23
Enter marks: 34.5
Displaying Information:
Name: Jacks
Roll number: 23
Marks: 34.5
```

ARRAY OF STRUCTURES

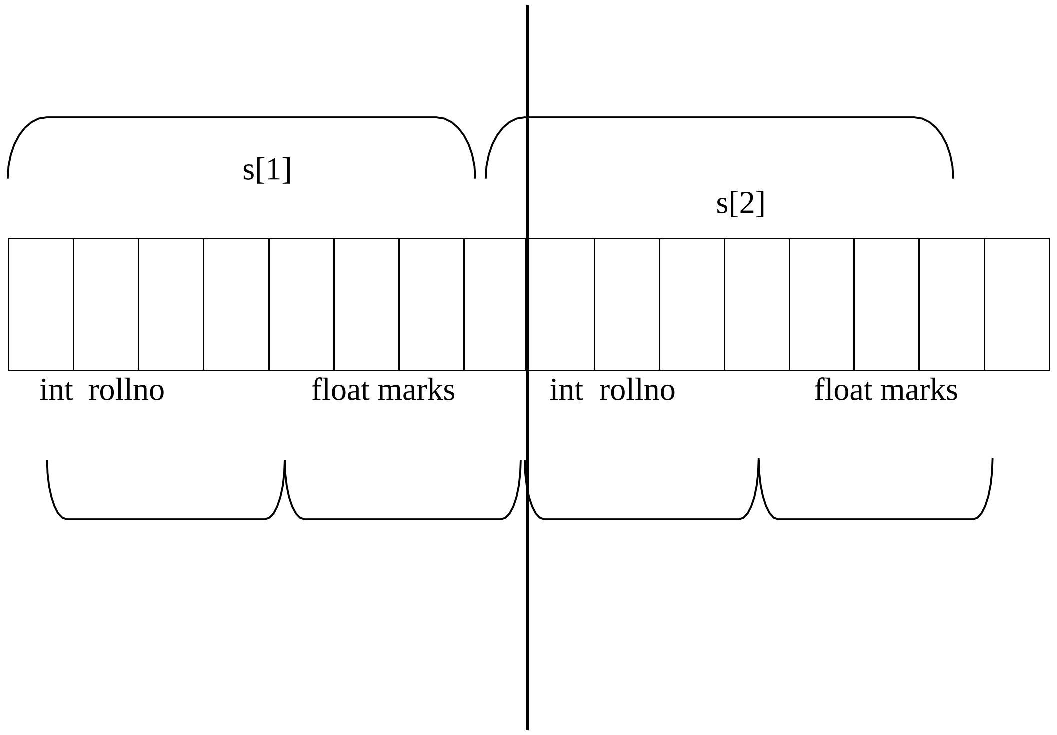
The array of structures is used to store information about multiple entities of different data types. The array of structures is also known as the collection of structures. An array of structures in C can be defined as the collection of multiple structures variables where each variable contains information about different entities.

Syntax

```
struct structname
{       Datatype    member1;
        Datatype     member2;
           . . .
        Datatype     member n;
}arrayname[ size];
struct structurename
{       Datatype    member1;
        Datatype     member2;
           . . .
        Datatype     member n;
};
void main()
{
        struct  structurename  arrayname[size];
}
```

Example

```
struct student
{       int rollno;
        float marks;
}s[2];
```



Array of Structures for student structure

Program

```
#include<stdio.h>
#include <conio.h>
struct student
{
        int rollno;
        char name[10];
};
void main()
{
         int i;
        struct student st[5];
        printf("Enter Records of 5 students");
        for(i=0;i<5;i++)
        {
                printf("\nEnter Rollno:");
                scanf("%d",&st[i].rollno);
                printf("\nEnter Name:");
                scanf("%s",st[i].name);
        }
        printf("\nStudent Information List:");
        for(i=0;i<5;i++)
        {
                printf("\nRollno:%d, Name:%s",st[i].rollno,st[i].name);
        }
        getch();
}
```

Output:

```
Enter Records of 5 students
Enter Rollno:1
Enter Name:Sonu
Enter Rollno:2
Enter Name:Ratan
Enter Rollno:3
Enter Name:Vimal
Enter Rollno:4
Enter Name:James
Enter Rollno:5
Enter Name:Raja
Student Information List:
Rollno:1, Name:Sonu
Rollno:2, Name:Ratan
Rollno:3, Name:Vimal
Rollno:4, Name:James
Rollno:5, Name:Raja
```

SELF REFERENTIAL STRUCTURES

Self Referential structures are those structures that have one or more pointers which point to the same type of structure, as their member. In other words, structures pointing to the same type of structures are self-referential in nature.

Syntax:

```
struct structname
{
        Datatype    member1;
        Datatype     member2;
                   . . .
        Datatype     member n;
 };
```

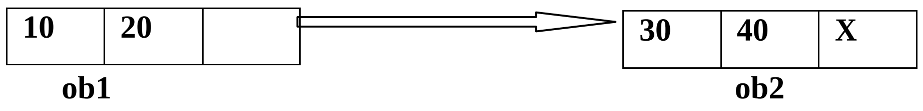
Example:

```
 struct node
 {
        int data1;
        char data2;
        struct node* link;
 };
 int main()
 {
        struct node ob;
        return 0;
 }
```

Types of Self Referential Structures

- Self Referential Structure with Single Link
- Self Referential Structure with Multiple Links

Self Referential Structure with Single Link: These structures can have only one self-pointer as their member. The following example will show us how to connect the objects of a self- referential structure with the single link and access the corresponding data members. The connection formed is shown in the following figure.



Self Referential Structure with Single Link

Program

```
#include <stdio.h>
#include<conio.h>
struct node
{
        int data1;
        char data2;
        struct node* link;
};
int main()
{
        struct node ob1; // Node1

         // Intialization
ob1.link = NULL;
ob1.data1 = 10;
     ob1.data2 = 20;
struct node ob2; // Node2
```

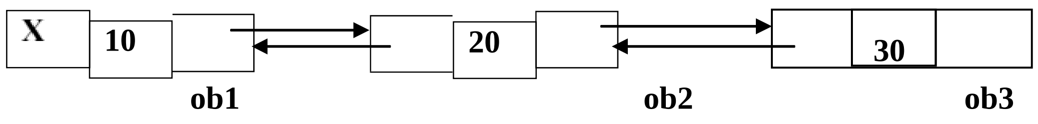
```
                // Initialization
                ob2.link = NULL;
                ob2.data1 = 30;
                ob2.data2 = 40;
                // Linking ob1 and ob2
                ob1.link = &ob2;
                // Accessing data members of ob2 using ob1
                printf("%d", ob1.link->data1);
                printf("\n%d", ob1.link->data2);
                getch();
        }
```

Output:

```
        30
        40
```

Self Referential Structure with Multiple Links:

Self referential structures with multiple links can have more than one self-pointers. Many complicated data structures can be easily constructed using these structures. Such structures can easily connect to more than one nodes at a time. The following example shows one such structure with more than one links.



Self Referential Structure with Multiple Links

Applications:

Self referential structures are very useful in creation of other complex data structures like:

- Linked Lists
- Stacks
- Queues
- Trees
- Graphs etc

DYNAMIC MEMORY ALLOCATION

The process of allocating memory at the time of execution or at the runtime is called dynamic memory allocation. It means dynamically allocate only the amount of memory needed. The program must include ***<stdlib.h>*** to support the dynamic memory allocation.

The comparison between Static memory allocation and Dynamic memory allocation is given below:

Static memory allocation	**Dynamic memory allocation**
Memory is allocated at compile time.	Memory is allocated at run time.
Memory can't be increased while executing program.	Memory can be increased while executing program.
Used in array.	Used in linked list.

Static memory allocation Vs Dynamic memory allocation

Need of Dynamic Memory Allocation

- Sometimes amount of data cannot be predicted beforehand
- Number of data items keeps changing during program execution

Memory Allocation Functions

1. **malloc( )**

   Allocates requested number of bytes ***(block of memory)*** and returns a pointer to the first byte of the allocated space

2. **calloc( )**

   Allocates space for an array of elements, initializes them to zero and then returns a pointer to the memory.

3. **free( )**

   Frees previously allocated space.

4. **realloc( )**

   Modifies the size of previously allocated space.

malloc( )

A block of memory can be allocated using the function malloc.

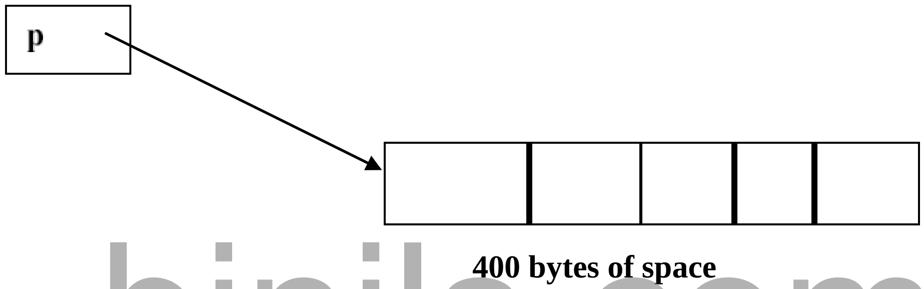
Syntax

```
type *p;
 p =  (type *) malloc (byte_size);
```

Example

```
p = (int *) malloc(100 * sizeof(int));
```

A memory space equivalent to 100 times the size of an int bytes is reserved. The address of the first byte of the allocated memory is assigned to the pointer p of type int.



memory allocation using malloc( )

- cptr = (char *) malloc (20);

  Allocates 20 bytes of space for the pointer cptr of type char
- sptr = (struct stud *) malloc(10*sizeof(struct stud));

  Allocates space for a structure array of 10 elements. sptr points to a structure element of type struct stud.

Program:

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void main()
{
        int n,i,*ptr,sum=0;
        printf("Enter number of elements: ");
        scanf("%d",&n);
        ptr=(int*)malloc(n*sizeof(int));  //memory allocated using malloc
```

```
                if(ptr==NULL)
                {
                        printf("Sorry! unable to allocate memory");
                        exit(0);
                }
                printf("Enter elements of array: ");
                for(i=0;i<n;++i)
                {
                        scanf("%d",ptr+i);
                        sum+=*(ptr+i);
                  }
                printf("Sum=%d",sum);
                free(ptr);
                getch();
        }
```

Output:

```
        Enter elements of array: 3
        Enter elements of array: 10
        10
        10
        Sum=30
```

calloc( )

Allocates space for an array of elements, initializes them to zero and then returns a pointer to the memory.

Synatx:

```
ptr=(cast-type*)calloc(number, byte-size)
```

Example:

```
int *ptr = calloc(10, sizeof (int));
```

Program:

```
#include<stdio.h>

 #include<stdlib.
 h>
 #include<conio.
 h>void main()
 {
        int n,i,*ptr,sum=0;
        printf("Enter number of elements:
         ");scanf("%d",&n);
        ptr=(int*)calloc(n,sizeof(int)); //memory allocated using calloc
         if(ptr==NULL)
        {
               printf("Sorry! unable to allocate
               memory");exit(0);
        }
        printf("Enter elements of array:
         ");for(i=0;i<n;++i)
        {
               scanf("%d",ptr+
               i);
               sum+=*(ptr+i);
        }
    printf("Sum=%d",sum);

              free(ptr);
              getch();
      }
```

Output:

```
Enter elements of array: 3
Enter elements of array: 10
10
10
Sum=30
```

realloc()

It modifies the size of previously allocated space. If memory is not sufficient for malloc()or calloc(), we can reallocate the memory by realloc() function. It changes the memory size.

Synatx:

```
ptr=realloc(ptr, new-size)
```

free()

Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on their own. We must explicitly use free() to release the space.

Syntax

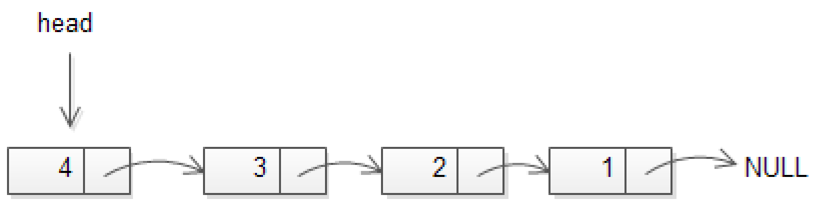
```
free(ptr);
```

This statement frees the space allocated in the memory pointed by ptr.

SINGLY LINKED LIST

A singly linked list is a data structure consisting of a sequence of nodes ***where each data points to the next data.*** Each node stores,

- Data (element)
- pointer to the next node



Singly Linked List in C

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

The node structure has two members:

- data stores the information
- next pointer holds the address of the next node.

Basic operation of singly list list.

- Create Linked List
- Traverse a linked list
- Insert elements to linked list
- Delete from a linked list

Creating a Node

To create the node, **sizeof()** is used to determine size in bytes of an element in C and **malloc()** is used to dynamically allocate a single block of memory in C.

```
struct LinkedList *node; //Define node as pointer of data type struct LinkedList
node    createNode()
{
         node temp; // declare a node
        temp = (node)malloc(sizeof(struct LinkedList));
```

```
                temp->next = NULL;// make next point to NULL
                return temp;//return the new node
        }
```

Traverse a linked list

Displaying the contents of a linked list is very simple. We keep moving the temp node to the next one and display its contents.

```
        struct node *temp = head;
        printf("\n\nList elements are - \n");
        while(temp != NULL)
        {
                 printf("%d --->",temp->data);
                temp = temp->next;
        }
```

Insert elements to linked list

We can insert elements to either beginning, middle or end of linked list.

Insert at beginning

- Allocate memory for new node
- Store data
- Change next of new node to point to head
- Change head to point to recently created node

```
                struct node *newNode;
                newNode = malloc(sizeof(struct node));

                newNode->data = 4;
                newNode->next =
                head;
                head = newNode;
```

Insert at end

- Allocate memory for new node
- Store data
- Traverse to last node
- Change next of last node to recently created node

```
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = 4;
newNode->next = NULL;
struct node *temp = head;
while(temp->next != NULL)
{
        temp = temp->next;
}
temp->next = newNode;
```

Insert to middle

- Allocate memory and store data for new node
- Traverse to node just before the required position of new node
- Change next pointers to include new node in between

```
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = 4;
struct node *temp = head;
for(int i=2; i < position; i++)
{
        if(temp->next != NULL)
        {
                temp = temp->next;
```

```
            }
        }
        newNode->next = temp->next;
        temp->next = newNode;
```

Delete from a linked list

We can delete either from beginning, end or from a particular position.

Delete from beginning

- Point head to the second node

```
            head = head->next;
```

Delete from end

- Traverse to second last element
- Change its next pointer to null

```
        struct node* temp = head;
        while(temp->next->next!=NULL){
        temp = temp->next;
        }
        temp->next = NULL;
```

Delete from middle

- Traverse to element before the element to be deleted
- Change next pointers to exclude the node from the chain

```
         for(int i=2; i< position; i++)
         {
             if(temp->next!=NULL)
             {
                 temp = temp->next;
             }
         }
```

Program for Singly Linked List:

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

```
struct node
{
            int data;
           struct node *next;
};
void display(struct node* head)
{
           struct node *temp = head;
           printf("\n\nList elements are - \n");
           while(temp != NULL)
           {
                      printf("%d --->",temp->data);
                      temp = temp->next;
           }
}
void insertAtMiddle(struct node *head, int position, int value)
{
           struct node *temp = head;
           struct node *newNode;
           newNode = malloc(sizeof(struct node));
           newNode->data = value;
            int i;
           for(i=2; inext != NULL)
           {
                       temp = temp->next;
           }
           newNode->next = temp->next;
           temp->next = newNode;
}
void insertAtFront(struct node** headRef, int value)
{
```

```
          struct node* head = *headRef;
          struct node *newNode;
          newNode = malloc(sizeof(struct node));
          newNode->data = value;
          newNode->next = head;
          head = newNode;
          *headRef = head;
}
void insertAtEnd(struct node* head, int value)
{
          struct node *newNode;
          newNode = malloc(sizeof(struct node));
          newNode->data = value;
          newNode->next = NULL;
          struct node *temp = head;
          while(temp->next != NULL)
          {
                    temp = temp->next;
          }
          temp->next = newNode;
}
void deleteFromFront(struct node** headRef)
{
          struct node* head =  *headRef;
          head = head->next;
          *headRef = head;
}
void deleteFromEnd(struct node* head)
{
          struct node* temp = head;
          while(temp->next->next!=NULL)
```

```
            {
                temp = temp->next;
            }
            temp->next = NULL;
        }
        void deleteFromMiddle(struct node* head, int position)
        {
            struct node* temp = head;
             int i;
            for(i=2; inext != NULL)
            {
                temp = temp->next;
            }
            temp->next = temp->next->next;
        }
        void main()
        {
            struct node *head;
            struct node *one = NULL;
            struct node *two = NULL;
            struct node *three = NULL;
             /* Allocate memory */
            one = malloc(sizeof(struct node));
            two = malloc(sizeof(struct node));
            three = malloc(sizeof(struct node));
            /* Assign data values */
            one->data = 1;
            two->data = 2;
            three->data = 3;
            /* Connect nodes */
            one->next = two;
            two->next = three;
            three->next = NULL;
            /* Save address of first node in head */
            head = one;
            display(head); // 1 --->2 --->3 --->
            insertAtFront(&head, 4);
            display(head); // 4 --->1 --->2 --->3 --->
            deleteFromFront(&head);
            display(head); // 1 --->2 --->3 --->
            insertAtEnd(head, 5);
            display(head); // 1 --->2 --->3 --->5 --->
            deleteFromEnd(head);
            display(head); // 1 --->2 --->3 --->
            int position = 3;
            insertAtMiddle(head, position, 10);
            display(head); // 1 --->2 --->10 --->3 --->
            deleteFromMiddle(head, position);
            display(head); // 1 --->2 --->3 --->
    }
```

TYPEDEF

The typedef keyword allows us to create a new data type name from an existing data type. It does not create a new data type but introduces a new name for existing type. A typedef declaration does not reserve storage. We cannot use the typedef specifier inside a function definition.

Syntax

```
typedef  dataype newdatatype;
```

Example:

```
typedef int INTEGER;
INTEGER num=29;
```

Here, INTEGER is the new name of data type int.

Program

```
#include<stdio.h>
#include<conio.h>
typedef   char    uchar;
void main()
{
   uchar ch = 'a';
   printf("ch: %c\n", ch);
   getch();
}
```

Output

```
ch : a
```

typedef in structure

We can use typedef with a structure. If the typedef keyword is given before the keyword struct, then the struct becomes a new type. For example,

```
typedef struct student
{
        int regno;
        char name[20];
};
```

Now, student is the new data type. The variables of the structure student can be declared as follows:

```
student stud;
```

Program

```
#include <stdio.h>
#include <string.h>
typedef struct student
{
        int id;
        char name[20];
        float percentage;
} status;
void main()
{
        status record;
        record.id=1;
         strcpy(record.name, "Raju");
        record.percentage = 86.5;
        printf(" Id is: %d \n", record.id);
        printf(" Name is: %s \n", record.name);
        printf(" Percentage is: %f \n", record.percentage);
        getch();
}
```

Output

```
Id is: 1

Name is: Raju
Percentage is: 86.500000
```

TYPEDEF

The typedef keyword allows us to create a new data type name from an existing data type. It does not create a new data type but introduces a new name for existing type. A typedef declaration does not reserve storage. We cannot use the typedef specifier inside a function definition.

Syntax

```
typedef  dataype newdatatype;
```

Example:

```
typedef int INTEGER;
INTEGER num=29;
```

Here, INTEGER is the new name of data type int.

Program

```
#include<stdio.h>
#include<conio.h>
typedef   char    uchar;
void main()
{
    uchar ch = 'a';
    printf("ch: %c\n", ch);
    getch();
}
```

Output

```
ch : a
```

typedef in structure

We can use typedef with a structure. If the typedef keyword is given before the keyword struct, then the struct becomes a new type. For example,

```
typedef struct student
{
        int regno;
        char name[20];
};
```

Now, student is the new data type. The variables of the structure student can be declared as follows:

```
student stud;
```

Program

```
#include <stdio.h>
#include <string.h>
typedef struct student
{
        int id;
        char name[20];
        float percentage;
} status;
void main()
{
        status record;
        record.id=1;
         strcpy(record.name, "Raju");
        record.percentage = 86.5;
        printf(" Id is: %d \n", record.id);
        printf(" Name is: %s \n", record.name);
        printf(" Percentage is: %f \n", record.percentage);
        getch();
}
```

Output

```
Id is: 1

Name is: Raju
Percentage is: 86.500000
```