UNIT IV

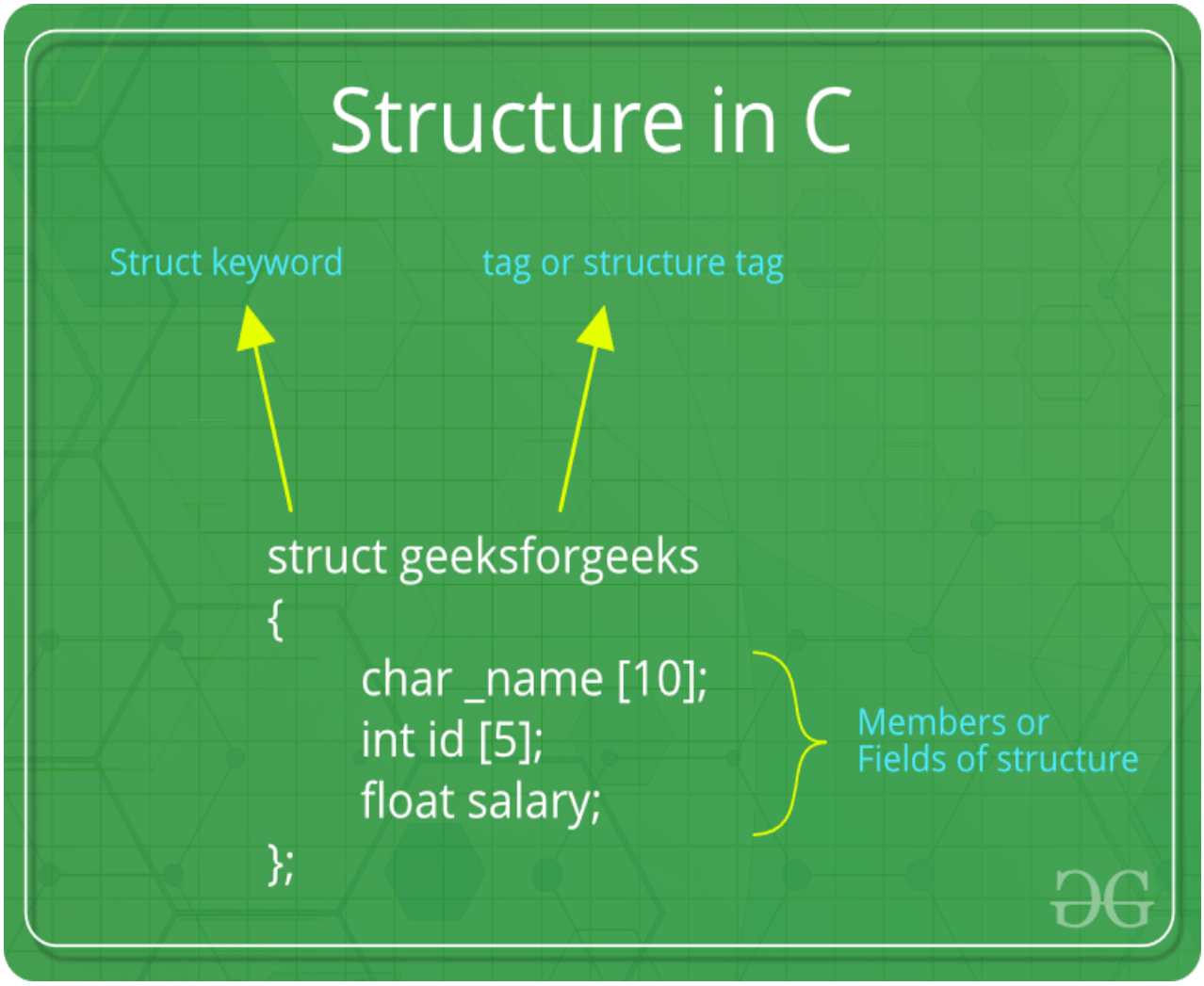
STRUCTURES

Structure - Nested structures – Pointer and Structures – Array of structures – Example Program using structures and pointers – Self-referential structures – Dynamic memory allocation - Singly linked list - typedef.

STRUCTURE

What is a structure?

A structure is a user defined data type in C/C++. A structure creates a data type that can be used to group items of possibly different types into a single type.



How to create a structure?

'struct' keyword is used to create a structure. Following is an example.

```
struct address
{
   char name[50];
   char street[100];
   char city[50];
   char state[20];
   int pin;
};
```

How to declare structure variables?

A structure variable can either be declared with structure declaration or as a separate declaration like basic types.

```
// A variable declaration with structure declaration.
struct Point
{
   int x, y;
} p1;  // The variable p1 is declared with 'Point'


// A variable declaration like basic data types
struct Point
{
   int x, y;
};

int main()
{
   struct Point p1;  // The variable p1 is declared like a normal variable
}
```

How to initialize structure members?

Structure members **cannot be** initialized with declaration. For example the following C program fails in compilation.

```
struct Point
```

```
{
  int x = 0;  // COMPILER ERROR:  cannot initialize members here
  int y = 0;  // COMPILER ERROR:  cannot initialize members here
};
```

The reason for above error is simple, when a datatype is declared, no memory is allocated for it. Memory is allocated only when variables are created.

Structure members **can be** initialized using curly braces '{}'. For example, following is a valid initialization.

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

int main()
{
  // A valid initialization. member x gets value 0 and y
  // gets value 1.  The order of declaration is followed.
  struct Point p1 = {0, 1};
}
```

How to access structure elements?

Structure members are accessed using dot (.) operator.

```
#include<stdio.h>

struct Point
{
  int x, y;
};

int main()
{
  struct Point p1 = {0, 1};

  // Accessing members of point p1
  p1.x = 20;
  printf ("x = %d, y = %d", p1.x, p1.y);

  return 0;
}
```

Output:

```
x = 20, y = 1
```

eg:

```
#include<stdio.h>

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

int main()
{
  // Examples of initialization using designated initialization
  struct Point p1 = {.y = 0, .z = 1, .x = 2};
  struct Point p2 = {.x = 20};

  printf ("x = %d, y = %d, z = %d\n", p1.x, p1.y, p1.z);
  printf ("x = %d", p2.x);
  return 0;
}
```

Output:

```
x = 2, y = 0, z = 1

x = 20
```

This feature is not available in C++ and works only in C.

What is an array of structures?

Like other primitive data types, we can create an array of structures.

```
#include<stdio.h>

struct Point
{
  int x, y;
};

int main()
{
  // Create an array of structures
  struct Point arr[10];
```

```
  // Access array members
  arr[0].x = 10;
  arr[0].y = 20;
  printf("%d %d", arr[0].x, arr[0].y);
  return 0;
}
```

Output:

```
10 20
```

What is a structure pointer?

Like primitive types, we can have pointer to a structure. If we have a pointer to structure, members are accessed using arrow (->) operator.

```
#include<stdio.h>
struct Point
{
  int x, y;
};

int main()
{
  struct Point p1 = {1, 2};

  // p2 is a pointer to structure p1
  struct Point *p2 = &p1;

  // Accessing structure members using structure pointer
  printf("%d %d", p2->x, p2->y);
  return 0;
}
```

Output:

```
1 2
```

Limitations of C Structures

In C language, Structures provide a method for packing together data of different types. A Structure is a helpful tool to handle a group of logically related data items. However, C structures have some limitations.

 The C structure does not allow the struct data type to be treated like built-in data types:

 We cannot use operators like +,- etc. on Structure variables. For example, consider the following code:

```
struct number
{
   float x;
};
int main()
{
   struct number n1,n2,n3;
   n1.x=4;
   n2.x=3;
   n3=n1+n2;
   return 0;
}

/*Output:

prog.c: In function 'main':
prog.c:10:7: error:
invalid operands to binary + (have 'struct number' and 'struct number')
  n3=n1+n2;
*/
```

 No Data Hiding: C Structures do not permit data hiding. Structure members can be accessed by any function, anywhere in the scope of the Structure.
 Functions inside Structure: C structures do not permit functions inside Structure
 Static Members: C Structures cannot have static members inside their body
 Access Modifiers: C Programming language do not support access modifiers. So they cannot be used in C Structures.
 Construction creation in Structure: Structures in C cannot have constructor inside Structures.

NESTED STRUCTURE

Nested Structure in C: Struct inside another struct

You can use a structure inside another structure, which is fairly possible. As I explained above that once you declared a structure, the **struct struct_name** acts as a new data type so you can include it in another struct just like the data type

of other data members. Sounds confusing? Don't worry. The following example will clear your doubt.

Example of Nested Structure in C Programming

Lets say we have two structure like this:
Structure 1: stu_address

```
struct stu_address
{
    int street;
    char *state;
    char *city;
    char *country;
}
```

Structure 2: stu_data

```
struct stu_data
{
   int stu_id;
   int stu_age;
   char *stu_name;
   struct stu_address stuAddress;
}
```

As you can see here that I have nested a structure inside another structure.

Assignment for struct inside struct (Nested struct)

Lets take the example of the two structure that we seen above to understand the logic

```
struct  stu_data  mydata;
mydata.stu_id = 1001;
mydata.stu_age = 30;
mydata.stuAddress.state = "UP"; //Nested struct assignment
..
```

How to access nested structure members?

Using chain of "." operator.
Suppose you want to display the city alone from nested struct –

```
printf("%s",  mydata.stuAddress.city);
```

POINTER AND STRUCTURES

A pointer is a variable which points to the address of another variable of any data type like int, char, float etc. Similarly, we can have a pointer to structures, where a pointer variable can point to the address of a structure variable. Here is how we can declare a pointer to a structure variable.

```
struct dog
{
   char name[10];
   char breed[10];
   int age;
   char color[10];
};

struct dog spike;

// declaring a pointer to a structure of type struct dog
struct dog *ptr_dog
```

This declares a pointer ptr_dog that can store the address of the variable of type struct dog. We can now assign the address of variable spike to ptr_dog using & operator.

Accessing members using Pointer

There are two ways of accessing members of structure using pointer:

1. Using indirection (*) operator and dot (.) operator.
2. Using arrow (->) operator or membership operator.
3. Using Indirection (*) Operator and Dot (.) Operator

At this point ptr_dog points to the structure variable spike, so by dereferencing it we will get the contents of the spike. This means spike and *ptr_dog are functionally equivalent. To access a member of structure write *ptr_dog followed by a dot(.) operator, followed by the name of the member. For example:

(*ptr_dog).name – refers to the name of dog

(*ptr_dog).breed – refers to the breed of dog

and so on.

Parentheses around *ptr_dog are necessary because the precedence of dot(.) operator is greater than that of indirection (*) operator.

Using arrow operator (->)

The above method of accessing members of the structure using pointers is slightly confusing and less readable, that's why C provides another way to access members using the arrow (->) operator. To access members using arrow (->) operator write pointer variable followed by -> operator, followed by name of the member.

1 ptr_dog->name - refers to the name of dog

2 ptr_dog->breed - refers to the breed of dog

and so on. Here we don't need parentheses, asterisk (*) and dot (.) operator. This method is much more readable and intuitive.

We can also modify the value of members using pointer notation.

```
strcpy(ptr_dog->name, "new_name");
```

Here we know that the name of the array (ptr_dog->name) is a constant pointer and points to the 0th element of the array. So we can't assign a new string to it using assignment operator (=), that's why strcpy() function is used.

```
--ptr_dog->age;
```

In the above expression precedence of arrow operator (->) is greater than that of prefix decrement operator (--), so first -> operator is applied in the expression then its value is decremented by 1.

The following program demonstrates how we can use a pointer to structure.

```
#include<stdio.h>

struct dog
{
   char name[10];
   char breed[10];
   int age;
   char color[10];
};

int main()
{
   struct dog my_dog = {"tyke", "Bulldog", 5, "white"};
```

```
    struct dog *ptr_dog;
    ptr_dog = &my_dog;

    printf("Dog's name: %s\n", ptr_dog->name);
    printf("Dog's breed: %s\n", ptr_dog->breed);
    printf("Dog's age: %d\n", ptr_dog->age);
    printf("Dog's color: %s\n", ptr_dog->color);

    // changing the name of dog from tyke to jack
    strcpy(ptr_dog->name, "jack");

    // increasing age of dog by 1 year
    ptr_dog->age++;

    printf("Dog's new name is: %s\n", ptr_dog->name);
    printf("Dog's age is: %d\n", ptr_dog->age);

    // signal to operating system program ran fine
    return 0;
}
```

Expected Output:

```
Dog's name: tyke
Dog's breed: Bulldog
Dog's age: 5
Dog's color: white

After changes

Dog's new name is: jack
Dog's age is: 6
```

ARRAY OF STRUCTURES

C Structure is collection of different datatypes(variables) which are grouped together. Whereas, array of structures is nothing but collection of structures. This is also called as structure array in C.

EXAMPLE PROGRAM FOR ARRAY OF STRUCTURES IN C:

This program is used to store and access “id, name and percentage” for 3 students. Structure array is used in this program to store and display records for many students. You can store “n” number of students record by declaring structure variable as ‘struct student record[n]“, where n can be 1000 or 5000 etc.

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

struct student
{
    int id;
    char name[30];
    float percentage;
};

int main()
{
    int i;
    struct student record[2];

    // 1st student's record
    record[0].id=1;
    strcpy(record[0].name, "Raju");
    record[0].percentage = 86.5;

    // 2nd student's record
    record[1].id=2;
    strcpy(record[1].name, "Surendren");
    record[1].percentage = 90.5;

    // 3rd student's record
    record[2].id=3;
    strcpy(record[2].name, "Thiyagu");
    record[2].percentage = 81.5;

    for(i=0; i<3; i++)
    {
        printf("     Records of STUDENT : %d \n", i+1);
        printf(" Id is: %d \n", record[i].id);
        printf(" Name is: %s \n", record[i].name);
        printf(" Percentage is: %f\n\n",record[i].percentage);
    }
    return 0;
}
```

OUTPUT

Records of STUDENT : 1

Id is: 1

Name is: Raju
Percentage is: 86.500000

Records of STUDENT : 2
Id is: 2
Name is: Surendren
Percentage is: 90.500000

Records of STUDENT : 3
Id is: 3
Name is: Thiyagu
Percentage is: 81.500000

EXAMPLE PROGRAM FOR DECLARING MANY STRUCTURE VARIABLE IN C:

In this program, two structure variables “record1″ and “record2″ are declared for same structure and different values are assigned for both structure variables. Separate memory is allocated for both structure variables to store the data.

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

struct student
{
    int id;
    char name[30];
    float percentage;
};

int main()
{
    int i;
    struct student record1 = {1, "Raju", 90.5};
    struct student record2 = {2, "Mani", 93.5};

    printf("Records of STUDENT1: \n");
    printf("  Id is: %d \n", record1.id);
    printf("  Name is: %s \n", record1.name);
    printf("  Percentage is: %f \n\n", record1.percentage);

    printf("Records of STUDENT2: \n");
    printf("  Id is: %d \n", record2.id);
    printf("  Name is: %s \n", record2.name);
```

```
    printf("  Percentage is: %f \n\n", record2.percentage);

    return 0;
}
```

OUTPUT

Records of STUDENT1:
Id is: 1
Name is: Raju
Percentage is: 90.500000
Records of STUDENT2:
Id is: 2
Name is: Mani
Percentage is: 93.500000

SELF REFERENTIAL STRUCTURE

Self-Referential structures are those structures that have one or more pointers which point to the same type of structure, as their member.

Example:

```
struct node {
   int data1;
   char data2;
   struct node* link;
};

int main()
{
   struct node ob;
   return 0;
}
```

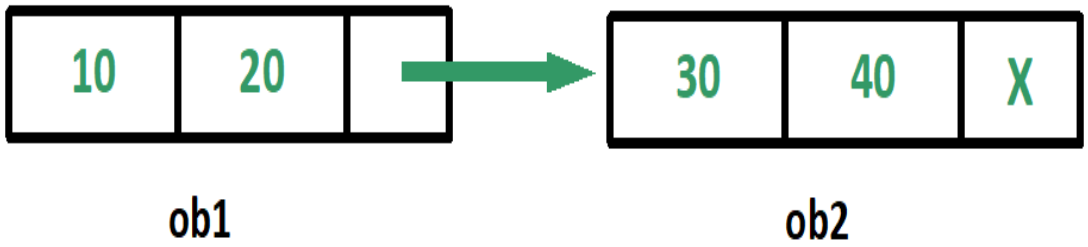
In the above example ‘link’ is a pointer to a structure of type ‘node’. Hence, the structure ‘node’ is a self-referential structure with ‘link’ as the referencing pointer.
An important point to consider is that the pointer should be initialized properly before accessing, as by default it contains garbage value.

Types of Self Referential Structures

1. Self-Referential Structure with Single Link
2. 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.




```
#include <stdio.h>

struct node {
    int data1;
    char data2;
    struct node* link;
};

int main()
{
    struct node ob1; // Node1

    // Initialization
    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
```

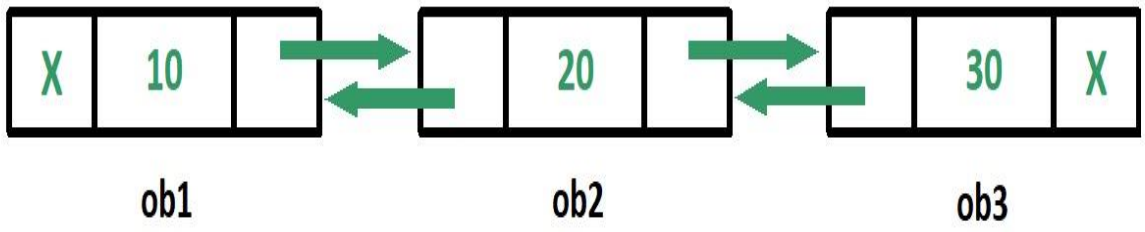
```
    printf("%d", ob1.link->data1);
    printf("\n%d", ob1.link->data2);
    return 0;
}
```

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.
The connections made in the above example can be understood using the following figure.




```
#include <stdio.h>

struct node {
    int data;
    struct node* prev_link;
    struct node* next_link;
};

int main()
{
    struct node ob1; // Node1

    // Initialization
    ob1.prev_link = NULL;
    ob1.next_link = NULL;
    ob1.data = 10;
```

```
    struct node ob2; // Node2

    // Initialization
    ob2.prev_link = NULL;
    ob2.next_link = NULL;
    ob2.data = 20;

    struct node ob3; // Node3

    // Initialization
    ob3.prev_link = NULL;
    ob3.next_link = NULL;
    ob3.data = 30;

    // Forward links
    ob1.next_link = &ob2;
    ob2.next_link = &ob3;

    // Backward links
    ob2.prev_link = &ob1;
    ob3.prev_link = &ob2;

    // Accessing  data of ob1, ob2 and ob3 by ob1
    printf("%d\t", ob1.data);
    printf("%d\t", ob1.next_link->data);
    printf("%d\n", ob1.next_link->next_link->data);

    // Accessing data of ob1, ob2 and ob3 by ob2
    printf("%d\t", ob2.prev_link->data);
    printf("%d\t", ob2.data);
    printf("%d\n", ob2.next_link->data);

    // Accessing data of ob1, ob2 and ob3 by ob3
    printf("%d\t", ob3.prev_link->prev_link->data);
    printf("%d\t", ob3.prev_link->data);
    printf("%d", ob3.data);
    return 0;
}
```

Output:

```
10    20    30

10    20    30
```

```
10   20   30
```

In the above example we can see that ‘ob1’, ‘ob2’ and ‘ob3’ are three objects of the self-referential structure ‘node’. And they are connected using their links in such a way that any of them can easily access each other’s data. This is the beauty of the self-referential structures. The connections can be manipulated according to the requirements of the programmer.

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

It Dynamically allocate memory in your C program using standard library functions: malloc(), calloc(), free() and realloc().

As you know, an array is a collection of a fixed number of values. Once the size of an array is declared, you cannot change it.

Sometimes the size of the array you declared may be insufficient. To solve this issue, you can allocate memory manually during run-time. This is known as dynamic memory allocation in C programming.

To allocate memory dynamically, library functions are `malloc()`, `calloc()`, `realloc()` and `free()` are used. These functions are defined in the `<stdlib.h>` header file.

C malloc()

The name "malloc" stands for memory allocation.

The `malloc()` function reserves a block of memory of the specified number of bytes. And, it returns a pointer of `void` which can be casted into pointers of any form.

Syntax of malloc()

```
ptr = (castType*) malloc(size);
```

Example

```
ptr = (float*) malloc(100 * sizeof(float));
```

The above statement allocates 400 bytes of memory. It's because the size of `float` is 4 bytes. And, the pointer `ptr` holds the address of the first byte in the allocated memory.
The expression results in a `NULL` pointer if the memory cannot be allocated.

C calloc()

The name "calloc" stands for contiguous allocation.

The `malloc()` function allocates memory and leaves the memory uninitialized. Whereas, the `calloc()` function allocates memory and initializes all bits to zero.

Syntax of calloc()

```
ptr = (castType*)calloc(n, size);
```

Example:

```
ptr = (float*) calloc(25, sizeof(float));
```

The above statement allocates contiguous space in memory for 25 elements of type `float`.

C free()

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

Syntax of free()

```
free(ptr);
```

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

Example 1: malloc() and free()

```
// Program to calculate the sum of n numbers entered by the user

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

int main()
{
   int n, i, *ptr, sum = 0;

   printf("Enter number of elements: ");
   scanf("%d", &n);

   ptr = (int*) malloc(n * sizeof(int));

   // if memory cannot be allocated
   if(ptr == NULL)
   {
      printf("Error! memory not allocated.");
      exit(0);
   }

   printf("Enter elements: ");
   for(i = 0; i < n; ++i)
   {
      scanf("%d", ptr + i);
      sum += *(ptr + i);
   }

   printf("Sum = %d", sum);

   // deallocating the memory
   free(ptr);

   return 0;
}
```

Here, we have dynamically allocated the memory for `n` number of `int`.

Example 2: calloc() and free()

```c
// Program to calculate the sum of n numbers entered by the user

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

int main()
{
   int n, i, *ptr, sum = 0;
   printf("Enter number of elements: ");
   scanf("%d", &n);

   ptr = (int*) calloc(n, sizeof(int));
   if(ptr == NULL)
   {
      printf("Error! memory not allocated.");
      exit(0);
   }

   printf("Enter elements: ");
   for(i = 0; i < n; ++i)
   {
      scanf("%d", ptr + i);
      sum += *(ptr + i);
   }

   printf("Sum = %d", sum);
   free(ptr);
   return 0; }
```

C realloc()

If the dynamically allocated memory is insufficient or more than required, you can change the size of previously allocated memory using the `realloc()` function.

Syntax of realloc()

```
ptr = realloc(ptr, x);
```

Here, `ptr` is reallocated with a new size `x`.

Example 3: realloc()

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

int main()
{
   int *ptr, i , n1, n2;
   printf("Enter size: ");
   scanf("%d", &n1);

   ptr = (int*) malloc(n1 * sizeof(int));

   printf("Addresses of previously allocated memory: ");
   for(i = 0; i < n1; ++i)
       printf("%u\n",ptr + i);

   printf("\nEnter the new size: ");
   scanf("%d", &n2);

   // rellocating the memory
   ptr = realloc(ptr, n2 * sizeof(int));

   printf("Addresses of newly allocated memory: ");
   for(i = 0; i < n2; ++i)
       printf("%u\n", ptr + i);

   free(ptr);

   return 0;
}
```

When you run the program, the output will be:

```
Enter size: 2
Addresses of previously allocated memory:26855472
26855476

Enter the new size: 4
Addresses of newly allocated memory:26855472
26855476
26855480
26855484
```

SINGLY LINKED LIST

What is Singly Linked List (SLL)?

The simplest kind of linked list is a singly liked list (SLL) which has one link per node. It has two parts, one part contains data and other contains address of next node. The structure of a node in a SLL is given as in C:

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

The program is given below that will perform insertion, deletion and display a singly linked list.

```
#include<stdio.h>

#include<conio.h>

#include<process.h>

struct node

{

   int data;

   struct node *next;

}*start=NULL,*q,*t;

int main()

{

   int ch;

   void insert_beg();

   void insert_end();

   int insert_pos();
```

```
    void display();
    void delete_beg();
    void delete_end();
    int delete_pos();
    while(1)
    {
        printf("\n\n---- Singly Linked List(SLL) Menu ----");
        printf("\n1.Insert\n2.Display\n3.Delete\n4.Exit\n\n");
        printf("Enter your choice(1-4):");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1:
                printf("\n---- Insert Menu ----");
                printf("\n1.Insert at beginning\n2.Insert at end\n3.Insert at specified
position\n4.Exit");
                printf("\n\nEnter your choice(1-4):");
                scanf("%d",&ch);
                switch(ch)
                {
                    case 1: insert_beg();
                            break;
                    case 2: insert_end();
                            break;
                    case 3: insert_pos();
```

```
                    break;
                case 4: exit(0);
                default: printf("Wrong Choice!!");
            }
            break;
        case 2: display();
            break;
        case 3: printf("\n---- Delete Menu ----");
            printf("\n1.Delete from beginning\n2.Delete from end\n3.Delete
from specified position\n4.Exit");
            printf("\n\nEnter your choice(1-4):");
            scanf("%d",&ch);
            switch(ch)
            {
                case 1: delete_beg();
                    break;
                case 2: delete_end();
                    break;
                case 3: delete_pos();
                    break;
                case 4: exit(0);
                default: printf("Wrong Choice!!");
            }
            break;
        case 4: exit(0);
```

```
                default: printf("Wrong Choice!!");
        }
    }
    return 0;
}
void insert_beg()
{
    int num;
    t=(struct node*)malloc(sizeof(struct node));
    printf("Enter data:");
    scanf("%d",&num);
    t->data=num;
    if(start==NULL)        //If list is empty
    {
        t->next=NULL;
        start=t;
    }
    else
    {
        t->next=start;
        start=t;
    }
}
void insert_end()
```

```
{
   int num;
   t=(struct node*)malloc(sizeof(struct node));
   printf("Enter data:");
   scanf("%d",&num);
   t->data=num;
   t->next=NULL;
   if(start==NULL)        //If list is empty
   {
      start=t;
   }
   else
   {
      q=start;
      while(q->next!=NULL)
      q=q->next;
      q->next=t;
   }
}
int insert_pos()
{
   int pos,i,num;
   if(start==NULL)
   {
```

```
        printf("List is empty!!");
        return 0;
    }
    t=(struct node*)malloc(sizeof(struct node));
    printf("Enter data:");
    scanf("%d",&num);
    printf("Enter position to insert:");
    scanf("%d",&pos);
    t->data=num;
    q=start;
    for(i=1;i<pos-1;i++)
    {
        if(q->next==NULL)
        {
            printf("There are less elements!!");
            return 0;
        }
        q=q->next;
    }
    t->next=q->next;
    q->next=t;
    return 0;
}
void display()
```

```
{
   if(start==NULL)
   {
      printf("List is empty!!");
   }
   else
   {
      q=start;
      printf("The linked list is:\n");
      while(q!=NULL)
      {
         printf("%d->",q->data);
         q=q->next;
      }
   }
}
void delete_beg()
{
   if(start==NULL)
   {
      printf("The list is empty!!");
   }
   else
   {
```

```
        q=start;
        start=start->next;
        printf("Deleted element is %d",q->data);
        free(q);
    }
}
void delete_end()
{
    if(start==NULL)
    {
        printf("The list is empty!!");
    }
    else
    {
        q=start;
        while(q->next->next!=NULL)
        q=q->next;
        t=q->next;
        q->next=NULL;
        printf("Deleted element is %d",t->data);
        free(t);
    }
}
int delete_pos()
```

```
{
    int pos,i;
    if(start==NULL)
    {
        printf("List is empty!!");
        return 0;
    }
    printf("Enter position to delete:");
    scanf("%d",&pos);
    q=start;
    for(i=1;i<pos-1;i++)
    {
        if(q->next==NULL)
        {
            printf("There are less elements!!");
            return 0;
        }
        q=q->next;
    }
    t=q->next;
    q->next=t->next;
    printf("Deleted element is %d",t->data);
    free(t);
    return 0;
```

}

Typedef

Use of typedef in Structure

typedef makes the code short and improves readability. In the above discussion we have seen that while using structs every time we have to use the lengthy syntax, which makes the code confusing, lengthy, complex and less readable. The simple solution to this issue is use of typedef. It is like an alias of struct.

Code without typedef

```
struct home_address {
  int local_street;
  char *town;
  char *my_city;
  char *my_country;
};
...
struct home_address var;
var.town = "Agra";
```

Code using tyepdef

```
typedef struct home_address{
  int local_street;
  char *town;
  char *my_city;
  char *my_country;
}addr;
..
..
addr var1;
var.town = "Agra";
```

Instead of using the struct home_address every time you need to declare struct variable, you can simply use addr, the typedef that we have defined.