UNIT – II:

Arrays: Introduction to Linear and Non- Linear Data Structures, One- Dimensional Arrays, Array Operations, Two- Dimensional arrays, Multidimensional Arrays, Pointers and Arrays, an Overview of Pointers

Linked Lists: Introduction to Lists and Linked Lists, Dynamic Memory Allocation, Basic Linked List Operations, Doubly Linked List, Circular Linked List, Atomic Linked List, Linked List in Arrays, Linked List versus Arrays

---

Definition of Linear Data Structure

The data structure is considered to be **linear** if the data elements construct a sequence of a linear list. The elements are adjacently attached to each other and in a specified order. It consumes linear memory space, the data elements are required to store in a sequential manner in the memory. While implementing the linear data structure the necessary amount of memory is declared previously. It does not make a good utilization of memory and result in memory wastage. The data elements are visited sequentially where only a single element can be directly reached.

The examples included in the linear data structure are array, stack, queue, linked list, etc. An **array** is a group of a definite number of homogeneous elements or data items. **Stack** and **queue** are also an ordered collection of the elements like an array but there is a special condition where stack follows LIFO (Last in first out) order and queue employ FIFO (First in first out) to insert and delete the elements. **Lists** can be defined as a set of variable number data items.

Definition of Non-linear Data Structure

Non-linear data structure does not arrange the data consecutively rather it is arranged in sorted order. In this, the data elements can be attached to more than one element exhibiting the hierarchical relationship which involves the relationship between the child, parent, and grandparent. In the non-linear data structure, the traversals of data elements and insertion or deletion are not done sequentially.

The non-linear data structure utilizes the memory efficiently and does not require the memory declaration in advance. There are the two common examples of the non-linear data structure – **tree** and **graph**. A tree data structure organizes and stores the data elements in a hierarchical relationship.

Comparison:

BASIS FOR COMPARISON	**LINEAR DATA STRUCTURE**	**NON-LINEAR DATA STRUCTURE**
Basic	The data items are arranged in an orderly manner where the elements are attached adjacently.	It arranges the data in a sorted order and there exists a relationship between the data elements.
Traversing of the data	The data elements can be accessed in one time (single run).	Traversing of data elements in one go is not possible.
Ease of implementation	Simpler	Complex
Levels involved	Single level	Multiple level
Examples	Array, queue, stack, linked list, etc.	Tree and graph.
Memory utilization	Ineffective	Effective

Arrays:

An array is a collection of one or more values of the same type. Each value is called an element of the array. The elements of the array share the same variable name but each element has its own unique index number (also known as a subscript). An array can be of any type, For example: int, float, char etc. If an array is of type int then it's elements must be of type int only.

Syntax

The syntax is as follows for declaring an array −

```
datatype array_name [size];
```

Types of arrays

Arrays are broadly classified into three types. They are as follows −

- One – dimensional arrays
- Two – dimensional arrays
- Multi – dimensional arrays

One – dimensional array

The Syntax is as follows −

```
datatype array name [size]
```

For example, int a[5]

Initialization

An array can be initialized in two ways, which are as follows −

- Compile time initialization
- Runtime initialization

Example

Following is the C program on compile time initialization −

```
#include<stdio.h>
int main ( )
{
     int a[5] = {10,20,30,40,50};
    int i;
    printf ("elements of the array are");
    for ( i=0; i<5; i++)
    printf ("%d", a[i]);
}
```

Output

Upon execution, you will receive the following output −

```
Elements of the array are
10 20 30 40 50
```

Example

Following is the C program on runtime initialization −

```
#include<stdio.h>
main ( )
{
     int a[5],i;
    printf ("enter 5 elements");
    for ( i=0; i<5; i++)
        scanf("%d", &a[i]);
printf("elements of the array are");
for (i=0; i<5; i++)
    printf("%d", a[i]);
}
```
Output

The output is as follows −

```
enter 5 elements 10 20 30 40 50
elements of the array are : 10 20 30 40 50
```
Note

The output of compile time initialized program will not change during different runs of the program.

The output of run time initialized program will change for different runs because, user is given a chance of accepting different values during execution.

Example

Following is another C program for one dimensional array −

```
        #include <stdio.h>
        int main(void)
{
  int a[4];
  int b[4] = {1};
  int c[4] = {1,2,3,4};
  int i; //for loop counter
  //printing all elements of all arrays
  printf("\nArray a:\n");
  for( i=0; i<4; i++ )
    printf("arr[%d]: %d ",i,a[i]);
    printf("\nArray b:\n");
  for( i=0; i<4; i++)
    printf("\narr[%d]: %d",i,b[i]);
    printf("\nArray c:\n");
  for( i=0; i<4; i++ )
    printf("arr[%d]: %d ",i, c[i]);
  return 0;
}
```
Output

The output is stated below −

```
Array a:
arr[0]: 8 arr[1]: 0 arr[2]: 54 arr[3]: 0
Array b:
arr[0]: 1 arr[1]: 0 arr[2]: 0 arr[3]: 0
Array c:
arr[0]: 1 arr[1]: 2 arr[2]: 3  arr[3]: 4
```

Array Operations:

Basic Operations:**Following are the basic operations supported by an array.**

- **Traverse** − print all the array elements one by one.
- **Insertion** − Adds an element at the given index.
- **Deletion** − Deletes an element at the given index.
- **Search** − Searches an element using the given index or by the value.
- **Update** − Updates an element at the given index.

In C, when an array is initialized with size, then it assigns defaults values to its elements in following order.

Data Type	Default Value
bool	false
char	0
int	0
float	0.0
double	0.0f
void	
wchar_t	0

Traverse Operation

This operation is to traverse through the elements of an array.

Example:**Following program traverses and prints the elements of an array:**

```
#include <stdio.h>
main() {
   int LA[] = {1,3,5,7,8};
   int item = 10, k = 3, n = 5;
   int i = 0, j = n;
   printf("The original array elements are :\n");
   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }
}
```

When we compile and execute the above program, it produces the following result −

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
```

Insertion Operation

Insert operation is to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array.

Here, we see a practical implementation of insertion operation, where we add data at the end of the array −

Example

Following is the implementation of the above algorithm −

```
#include <stdio.h>
main() {
   int LA[] = {1,3,5,7,8};
   int item = 10, k = 3, n = 5;
   int i = 0, j = n;
      printf("The original array elements are :\n");
   for(i = 0; i<n; i++)
 {
      printf("LA[%d] = %d \n", i, LA[i]);
   }
   n = n + 1;
   while( j >= k)
{
      LA[j+1] = LA[j];
      j = j - 1;
   }
   LA[k] = item;
   printf("The array elements after insertion :\n");
   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }
}
```

When we compile and execute the above program, it produces the following result −

Output

```
The original array elements are : LA[0] = 1 LA[1] = 3 LA[2] = 5 LA[3] = 7 LA[4] = 8
The array elements after insertion : LA[0] = 1 LA[1] = 3 LA[2] = 5 LA[3] = 10 LA[4] = 7
LA[5] = 8
```

For other variations of array insertion operation click here

Deletion Operation

Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

Algorithm

Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Following is the algorithm to delete an element available at the Kth position of LA.

```
1. Start
2. Set J = K
3. Repeat steps 4 and 5 while J < N
4. Set LA[J] = LA[J + 1]
5. Set J = J+1
6. Set N = N-1
7. Stop
```

Example

Following is the implementation of the above algorithm −

```
#include <stdio.h>
void main() {
   int LA[] = {1,3,5,7,8};
   int k = 3, n = 5;
   int i, j;

   printf("The original array elements are :\n");

   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }
      j = k;
   while( j < n) {
      LA[j-1] = LA[j];
      j = j + 1;
   }
   n = n -1;
      printf("The array elements after deletion :\n");
   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }
}
```

When we compile and execute the above program, it produces the following result −

Output

```
The original array elements are : LA[0] = 1 LA[1] = 3 LA[2] = 5 LA[3] = 7 LA[4] = 8
The array elements after deletion : LA[0] = 1 LA[1] = 3 LA[2] = 7 LA[3] = 8
```

Search Operation

You can perform a search for an array element based on its value or its index.

Algorithm

Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Following is the algorithm to find an element with a value of ITEM using sequential search.

```
1. Start
2. Set J = 0
3. Repeat steps 4 and 5 while J < N
4. IF LA[J] is equal ITEM THEN GOTO STEP 6
5. Set J = J +1
6. PRINT J, ITEM
7. Stop
```

Example

Following is the implementation of the above algorithm −

```
#include <stdio.h>
void main() {
   int LA[] = {1,3,5,7,8};
   int item = 5, n = 5;
   int i = 0, j = 0;
      printf("The original array elements are :\n");
             for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }
   while( j < n){
      if( LA[j] == item ) {
         break;
      }
                j = j + 1;
   }

   printf("Found element %d at position %d\n", item, j+1);
}
```

When we compile and execute the above program, it produces the following result −

```
Output
The original array elements are : LA[0] = 1 LA[1] = 3 LA[2] = 5 LA[3] = 7 LA[4] = 8
Found element 5 at position 3
```

Update Operation

Update operation refers to updating an existing element from the array at a given index.

Algorithm

Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Following is the algorithm to update an element available at the Kth position of LA.

1. Start
2. Set LA[K-1] = ITEM
3. Stop

Example

Following is the implementation of the above algorithm −

```
#include <stdio.h>
void main()
{
   int LA[] = {1,3,5,7,8};
   int k = 3, n = 5, item = 10;
   int i, j;
   printf("The original array elements are :\n");
   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }
    LA[k-1] = item;
   printf("The array elements after updation :\n");
   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }
}
```

When we compile and execute the above program, it produces the following result −

Output

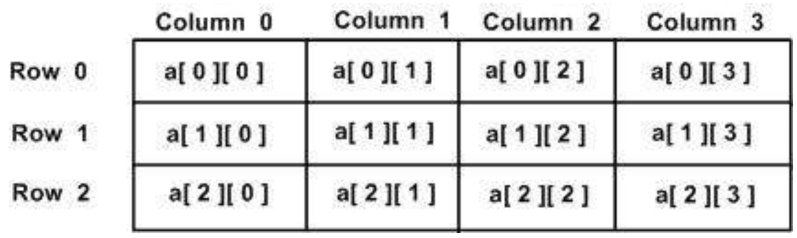
```
The original array elements are :LA[0] = 1 LA[1] = 3 LA[2] = 5 LA[3] = 7 LA[4] = 8
The array elements after updation :LA[0] = 1 LA[1] = 3 LA[2] = 10 LA[3] = 7 LA[4] = 8
```

Two- Dimensional arrays:

The simplest form of multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size [x][y], you would write something as follows −

```
type arrayName [ x ][ y ];
```

Where type can be any valid C data type and arrayName will be a valid C identifier. A two-dimensional array can be considered as a table which will have x number of rows and y number of columns. A two-dimensional array a, which contains three rows and four columns can be shown as follows −



	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Thus, every element in the array a is identified by an element name of the form a[i][j], where 'a' is the name of the array, and 'i' and 'j' are the subscripts that uniquely identify each element in 'a'.

Initializing Two-Dimensional Arrays

Multidimensional arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row has 4 columns.

```
int a[3][4] = {
   {0, 1, 2, 3} ,   /*  initializers for row indexed by 0 */
   {4, 5, 6, 7} ,   /*  initializers for row indexed by 1 */
   {8, 9, 10, 11}   /*  initializers for row indexed by 2 */
};
```

The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to the previous example −

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

Accessing Two-Dimensional Array Elements

An element in a two-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example −

```
int val = a[2][3];
```

The above statement will take the 4th element from the 3rd row of the array. You can verify it in the above figure. Let us check the following program where we have used a nested loop to handle a two-dimensional array −

```
#include <stdio.h>
 int main () {
   /* an array with 5 rows and 2 columns*/
   int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};
   int i, j;
   /* output each array element's value */
  for ( i = 0; i < 5; i++ ) {
    for ( j = 0; j < 2; j++ ) {
      printf("a[%d][%d] = %d\n", i,j, a[i][j] );
    }
  }
    return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
a[0][0]: 0 a[0][1]: 0 a[1][0]: 1 a[1][1]: 2 a[2][0]: 2 a[2][1]: 4 a[3][0]: 3
a[3][1]: 6 a[4][0]: 4 a[4][1]: 8
```

As explained above, you can have arrays with any number of dimensions, although it is likely that most of the arrays you create will be of one or two dimensions.

Multidimensional Arrays:

Array Declaration

A multidimensional array is declared using the following syntax:

type array_name[d1][d2][d3][d4]………[dn];

Example

int table[5][5][20];

int designates the array type integer. table is the name of our 3D array.

Our array can hold 500 integer-type elements. This number is reached by multiplying the value of each dimension. In this case: 5x5x20=500.

float arr[5][6][5][6][5];

Array arr is a five-dimensional array.

It can hold 4500 floating-point elements (5x6x5x6x5=4500).

Can you see the power of declaring an array over variables? When it comes to holding multiple values in C programming, we would need to declare several variables. But a single array can hold thousands of values.

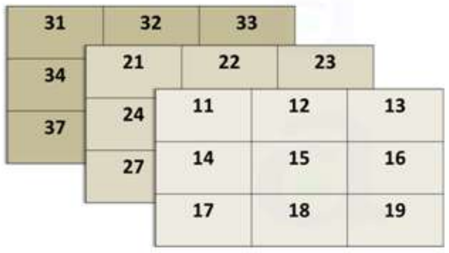
Note: For the sake of simplicity, this tutorial discusses 3D arrays only. Once you grab the logic of how the 3D array works then you can handle 4D arrays and larger.

Explanation of a 3D Array

Let's take a closer look at a 3D array. A 3D array is essentially an array of arrays of arrays: it's an array or collection of 2D arrays, and a 2D array is an array of 1D array.

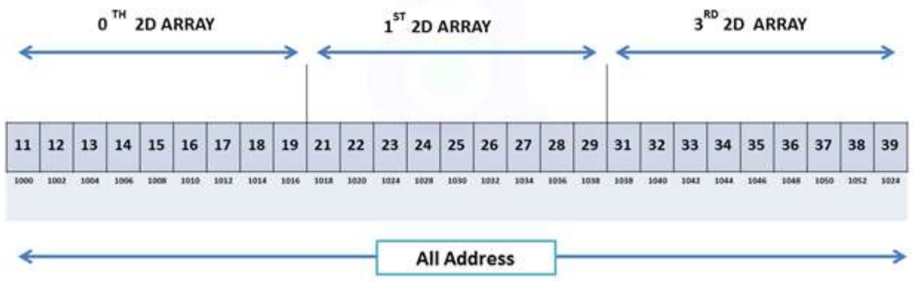
It may sound a bit confusing, but don't worry. As you practice working with multidimensional arrays, you start to grasp the logic.

The diagram below may help you understand:



Three Dimensional Array in c programming language

Address and view



Three Dimensional Array in c programming language

Initializing a 3D Array in C

Like any other variable or array, a 3D array can be initialized at the time of compilation. By default, in C, an uninitialized 3D array contains "garbage" values, not valid for the intended use.

Let's see a complete example on how to initialize a 3D array:

Syntax:

```
void main()
{
int i, j, k;
int arr[3][3][3]=
        {
           {
           {11, 12, 13},
           {14, 15, 16},
           {17, 18, 19}
           },
           {
           {21, 22, 23},
           {24, 25, 26},
           {27, 28, 29}
           },
           {
           {31, 32, 33},
           {34, 35, 36},
           {37, 38, 39}
           },
        };

printf(":::3D Array Elements:::\n\n");
for(i=0;i<3;i++)
{
    for(j=0;j<3;j++)
    {
        for(k=0;k<3;k++)
        {
        printf("%d\t",arr[i][j][k]);
        }
        printf("\n");
    }
    printf("\n");
}
getch();
}
```

```
:::3D Array Elements:::

11      12      13
14      15      16
17      18      19

21      22      23
24      25      26
27      28      29

31      32      33
34      35      36
37      38      39
```

In the code above we have declared a multidimensional integer array named “arr” which can hold 3x3x3 (or 27) elements.

Pointers and Arrays:

Before you start with Pointer and Arrays in C, learn about these topics in prior:

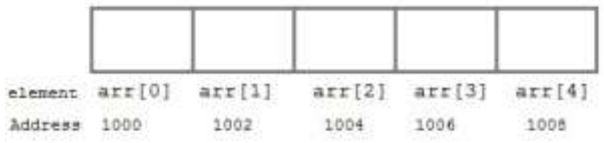
- Array in C
- Pointer in C

When an array in C language is declared, compiler allocates sufficient memory to contain all its elements. Its base address is also allocated by the compiler.

Declare an array arr,

int arr[5] = { 1, 2, 3, 4, 5 };

Suppose the base address of arr is 1000 and each integer requires two bytes, the five elements will be stored as follows:



Variable arr will give the base address, which is a constant pointer pointing to arr[0]. Hence arr contains the address of arr[0] i.e 1000.

arr has two purpose -

- It is the name of the array
- It acts as a pointer pointing towards the first element in the array.

arr is equal to &arr[0] by default

For better understanding of the declaration and initialization of the pointer - click here. and refer to the program for its implementation.

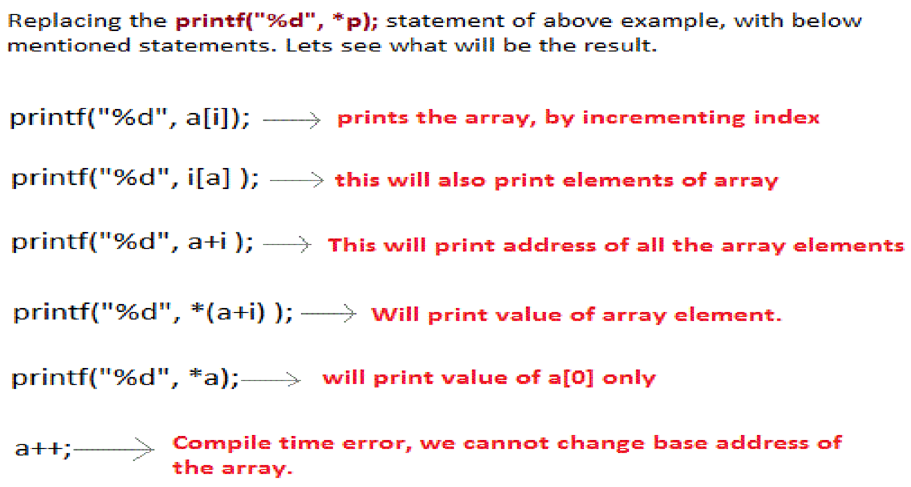
NOTE:

- You cannot decrement a pointer once incremented. p-- won't work.

Pointer to Array
Use a pointer to an array, and then use that pointer to access the array elements. For example,

```
#include<stdio.h>
void main()
{
  int a[3] = {1, 2, 3};
  int *p = a;
  for (int i = 0; i < 3; i++)
  {
    printf("%d", *p);
    p++;
  }
 return 0;
}
```

```
1 2 3
```



Replacing the **printf("%d", *p);** statement of above example, with below mentioned statements. Lets see what will be the result.

printf("%d", a[i]); ⟶ **prints the array, by incrementing index**

printf("%d", i[a]); ⟶ **this will also print elements of array**

printf("%d", a+i); ⟶ **This will print address of all the array elements**

printf("%d", *(a+i)); ⟶ **Will print value of array element.**

printf("%d", *a); ⟶ **will print value of a[0] only**

a++; ⟶ **Compile time error, we cannot change base address of the array.**

Syntax:
*(a+i) //pointer with an array
is same as: a[i]
Pointer to Multidimensional Array
Let's see how to make a pointer point to a multidimensional array. In a[i][j], a will give the base address of this array, even a + 0 + 0 will also give the base address, that is the address of a[0][0] element.
Syntax:
((a + i) + j)

Pointer and Character strings
Pointer is used to create strings. Pointer variables of char type are treated as string.

```
char *str = "Hello";
```

The above code creates a string and stores its address in the pointer variable str. The pointer str now points to the first character of the string "Hello".

- The string created using char pointer can be assigned a value at **runtime**.

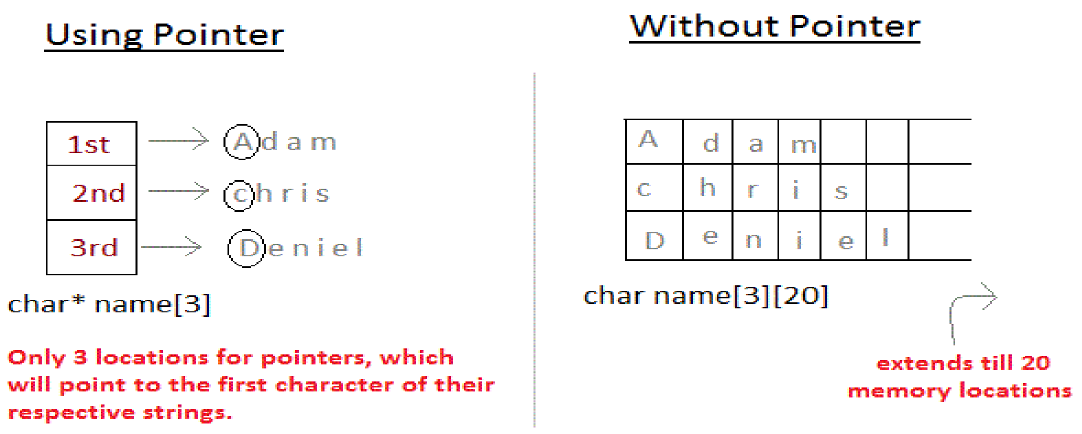
```
char *str;
str = "hello";
```

- The content of the string can be printed using printf() and puts().
- printf("%s", str);
- puts(str);

- str is a pointer to the string and also name of the string. Therefore we do not need to use indirection operator *.

Array of Pointers
Pointers are very helpful in handling character arrays with rows of varying lengths.

```
char *name[3] = {
    "Adam",
    "chris",
    "Deniel"
};
//without pointer
char name[3][20] = {
    "Adam",
    "chris",
    "Deniel"
};
```


Using Pointer
1st
2nd
3rd
Adam
chris
Deniel
char* name[3]
Only 3 locations for pointers, which will point to the first character of their respective strings.
Without Pointer
A d a m
c h r i s
D e n i e l
char name[3][20]
extends till 20 memory locations

In the second approach memory wastage is more, hence it is preferred to use pointer in such cases.

An Overview of Pointers

Introduction to Pointers
Pointer is a fundamental part of C. If you cannot use pointers properly then you have basically lost all the power and flexibility that C allows. The secret to C is in its use of pointers. C uses *pointers* a lot because:

- It is the only way to express some computations.
- It produces compact and efficient code.
- Pointers provided an easy way to represent multidimensional arrays.
- Pointers increase the execution speed.
- Pointers reduce the length and complexity of program.

C uses pointers explicitly with arrays, structures and functions.
A **pointer** is a variable which contains the address in memory of another variable. We can have a pointer to any variable type.
The ***unary*** operator **&** gives the "address of a variable". The ***indirection*** or dereference operator ***** gives the "contents of an object ***pointed to*** by a pointer".

Declaring Pointer Variables
The general syntax of pointer declaration is,

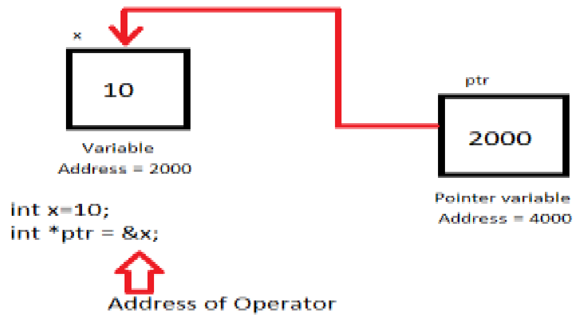
```
datatype *pointer_name;
```

The data type of the pointer and the variable to which the pointer variable is pointing must be the same.
Example

```
int *pointer;
float *p;
char *x;
```

We must associate a pointer to a particular type. We can't assign the address of a short int to a long int.
Initialization of C Pointer variable
Pointer Initialization is the process of assigning address of a variable to a pointer variable. It contains the address of a variable of the same data type. In C language address operator & is used to determine the address of a variable. The & (immediately preceding a variable name) returns the address of the variable associated with it.

```
int a = 10;
int *ptr;       //pointer declaration
ptr = &a;       //pointer initialization
```



Pointer variable always points to variables of the same datatype. For example:

```
float a;
int *ptr = &a;       // ERROR, type mismatch
```

Consider the effect of the following code:

```
#include <stdio.h>
main()
{
        int x = 1, y = 2;
        int *ip;
        ip = &x;
        y = *ip;
        *ip = 3;
}
```

It is worth considering what is going on at the machine level in memory to fully understand how pointer works. Assume for the sake of this discussion that variable x resides at memory location 100, y at 200 and ip at 1000 shown in figure 4.1.

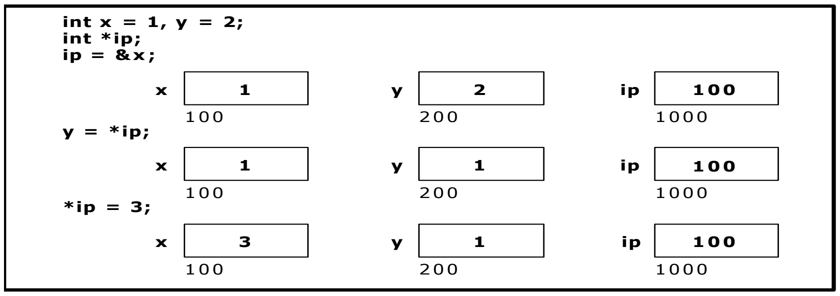


Fig. 4.1 Pointer, Variables and Memory

Now the assignments x = 1 and y = 2 obviously load these values into the variables. ip is declared to be a pointer to an integer and is assigned to the address of x (&x). So ip gets loaded with the value 100.

Next y gets assigned to the contents of ip. In this example ip currently points to memory location 100 -- the location of x. So y gets assigned to the values of x -- which is 1. Finally, we can assign a value 3 to the contents of a pointer (*ip).

IMPORTANT: When a pointer is declared it does not point anywhere. You must set it to point somewhere before you use it. So,

```
int *ip;
*ip = 100;
```

will generate an error (program crash!!). The correct usage is:

```
int *ip;
int x;
ip = &x;
*ip = 100;
++ip;
```

Pointer Expressions and Pointer Arithmetic

In general, expressions involving pointers conform to the same rules as other expressions. This section examines a few special aspects of pointer expressions, such as assignments, conversions, and arithmetic.

Pointer Assignments

You can use a pointer on the right-hand side of an assignment statement to assign its value to another pointer. When both pointers are the same type, the situation is straightforward. For example:

```
int s = 56;
int *ptr1, *ptr2;
ptr1 = &s;
ptr2 = ptr1;
```

Program

```
#include <stdio.h>
int main(void)
{
        int s = 56;
        int *ptr1, *ptr2;
        ptr1 = &s;
        ptr2 = ptr1;
        /* print the value of s twice */
        printf("Values at ptr1 and ptr2: %d %d \n", *ptr1, *ptr2);
        /* print the address of s twice */
        printf("Addresses pointed to by ptr1 and ptr2: %p %p",ptr1,ptr2);
        return 0;
}
```
Output

```
Values at ptr1 and ptr2: 56 56
Addresses pointed to by ptr1 and ptr2: 0240FF20 0240FF20
```
Pointer Arithmetic

We can perform addition and subtraction of integer constant from pointer variable.

Addition

```
        ptr1 = ptr1 + 2;
```
subtraction

```
        ptr1 = ptr1 - 2;
```
We can not perform addition, multiplication and division operations on two pointer variables.

For Example:

```
        ptr1 + ptr2 is not valid
```
However we can subtract one pointer variable from another pointer variable. We can use increment and decrement operator along with pointer variable to increment or decrement the address contained in pointer variable.

For Example:

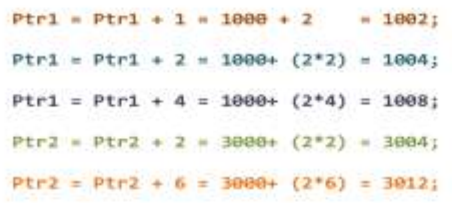
```
        ptr1++;
        ptr2--;
```
Multiplication

```
        Example:
        int x = 10, y = 20, z;
        int *ptr1 = &x;
        int *ptr2 = &y;
        z = *ptr1 * *ptr2 ;
        Will assign 200 to variable z.
```
Division

there is a blank space between '/' and * because the symbol /*is considered as beginning of the comment and therefore the statement fails.

```
        Z=5*-*Ptr2/ *Ptr1;
```
If Ptr1 and Ptr2 are properly declared and initialized pointers, then the following statements are valid:

```
        Y=*Ptr1**Ptr2;
        Sum=sum+*Ptr1;
        *Ptr2=*Ptr2+10;
        *Ptr1=*Ptr1+*Ptr2;
        *Ptr1=*Ptr2-*Ptr1;
```



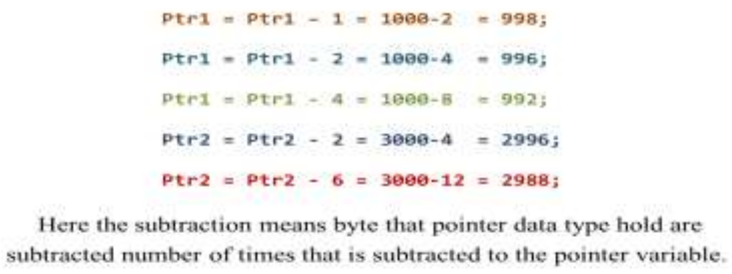
```
Ptr1 = Ptr1 + 1 = 1000 + 2     = 1002;
Ptr1 = Ptr1 + 2 = 1000+ (2*2) = 1004;
Ptr1 = Ptr1 + 4 = 1000+ (2*4) = 1008;
Ptr2 = Ptr2 + 2 = 3000+ (2*2) = 3004;
Ptr2 = Ptr2 + 6 = 3000+ (2*6) = 3012;
```

Here addition means bytes that pointer data type hold are subtracted number of times that is subtracted to the pointer variable.

if Ptr1 and Ptr2 are properly declared and initialized pointers then, 'C' allows adding integers to a pointer variable.

EX:

```
        int a=5, b=10;
        int *Ptr1,*Ptr2;
        Ptr1=&a;
        Ptr2=&b
```
If Ptr1 & Ptr2 are properly declared and initialized, pointers then 'C' allows to subtract integers from pointers. From the above example,



```
Ptr1 = Ptr1 - 1 = 1000-2  = 998;
Ptr1 = Ptr1 - 2 = 1000-4  = 996;
Ptr1 = Ptr1 - 4 = 1000-8  = 992;
Ptr2 = Ptr2 - 2 = 3000-4  = 2996;
Ptr2 = Ptr2 - 6 = 3000-12 = 2988;
```
Here the subtraction means byte that pointer data type hold are subtracted number of times that is subtracted to the pointer variable.

If Ptr1 & Ptr2 are properly declared and initialize pointers, and both points to the elements of the same type. "Subtraction of one pointer from another pointer is also possible".
NOTE: this operation is done when both pointer variable points to the elements of the same array.
EX:
P2- P1 (It gives the number of elements between p1 and p2)
Pointer Increment and Scale Factor
We can use increment operator to increment the address of the pointer variable so that it points to next memory location.
The value by which the address of the pointer variable will increment is not fixed. It depends upon the data type of the pointer variable.
For Example:
```
        int *ptr;
        ptr++;
```

It will increment the address of pointer variable by 2. So if the address of pointer variable is 2000 then after increment it becomes 2002.
Thus the value by which address of the pointer variable increments is known as scale factor. The scale factor is different for different data types as shown below:

Char	1 Byte
Int	2 Byte
Short int	2 Byte
Long int	4 Byte
Float	4 Byte
Double	8 Byte
Long double	10 Byte

Write a C program to compute the sum of all elements stored in an array Using pointers.
Program
```
/*program to compute sum of all elements stored in an  array */
#include<stdio.h>
#include<conio.h>
main ()
```

```
{
            int a [10], i, sum=0,*p;
            printf ("enter 10 elements \n");
            for (i=0; i<10; i++)
            scanf ("%d", & a[i]);
            p = a;
            for (i = 0; i<10; i++)
            {
            sum = sum+(*p);
            p++;
            }
            printf ("the sum is % d", sum);
            getch ();
}
```

Output

```
enter 10 elements
1
2
3
4
5
6
7
8
9
10
the sum is  55
```

Write a C program using pointers to determine the length of a character String.

Program

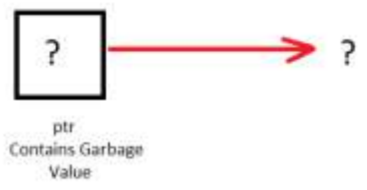
```
/*program to find the length of a char string */
#include<stdio.h>
#include<conio.h>
main ()
{
            char str[20], *ptr ;
            int l=0;
            printf("enter a string \n");
            scanf("%s", str);
            ptr=str;
            while(*ptr!='\0')
            {
            l++;
            ptr++;
            }
            printf("the length of the given string is %d \n", l);

 }
```

Output

```
enter a string
atnyla.com
the length of the given string is 10
Press any key to continue .
```

Null Pointers

While declaring a pointer variable, if it is not assigned to anything then it contains garbage value. Therefore, it is recommended to assign a NULL value to it,



A pointer that is assigned a NULL value is called a **NULL pointer in C**

- NULL Pointer is a pointer which is pointing to nothing.
- The NULL pointer points the base address of the segment.
- In case, if you don't have an address to be assigned to a pointer then you can simply use NULL
- The pointer which is initialized with the NULL value is considered as a NULL pointer.
- NULL is a macro constant defined in following header files – stdio.h, alloc.h, mem.h, stddef.h, stdlib.h

Defining NULL Value

```
#define NULL 0
```

Below are some of the variable representations of a NULL pointer.

```
        float *ptr  = (float *)0;
        char  *ptr  = (char *)0;
        double *ptr = (double *)0;
        char *ptr   = '\0';
        int *ptr    = NULL;
```

Example of NULL Pointer

```
#include <stdio.h>
int main()
{
  int  *ptr = NULL;
  printf("The value of ptr is %u",ptr);
  return 0;
}
```

Output :

```
The value of ptr is 0
```

Generic Pointers

When a variable is declared as being a pointer to type **void,** it is known as a *generic pointer*. Since you cannot have a variable of type **void**, the pointer will not point to any data and therefore cannot be dereferenced. It is still a pointer though, to use it you just have to cast it to another kind of pointer first. Hence the term ***Generic pointer***.

This is very useful when you want a pointer to point to data of different types at different times.

Void pointer is a specific pointer type – void * – a pointer that points to some data location in storage, which doesn't have any specific type. Void refers to the type. Basically the type of data that it points to is can be any. If we assign address of char data type to void pointer it will become char Pointer, if int data type then int pointer and so on. Any pointer type is convertible to a void pointer hence it can point to any value.

Why Void Pointers is important

1. Suppose we have to declare integer pointer, character pointer and float pointer then we need to declare 3 pointer variables.
2. Instead of declaring different types of pointer variable it is feasible to declare single pointer variable which can act as an integer pointer, character pointer.

Declaration of Void Pointer

```
        void * pointer_name;
```

Void Pointer Example :

```
        void *ptr;    // ptr is declared as Void pointer
        char c;
        int i;
        float f;
        ptr = &c;  // ptr has address of character data
        ptr = &i;  // ptr has address of integer data
        ptr = &f;  // ptr has address of float data
```

Explanation :

void *ptr;

1. **Void pointer** declaration is shown above.
2. We have declared 3 variables of integer, character and float type.
3. When we assign **the address of the integer** to the void pointer, the pointer will become Integer Pointer.
4. When we assign **the address of Character** Data type to void pointer it will become Character Pointer.
5. Similarly, we can assign the address of any data type to the void pointer.
6. It is capable of storing the address of any data type

Example of Generic Pointer

Here is some code using a void pointer:

```
#include<stdlib.h>
 int main()
{
   int x = 4;
   float y = 5.5;
    //A void pointer
   void *ptr;
   ptr = &x;
    // (int*)ptr - does type casting of void
   // *((int*)ptr) dereferences the typecasted
   // void pointer variable.
   printf("Integer variable is = %d", *( (int*) ptr) );
    // void pointer is now float
   ptr = &y;
   printf("\nFloat variable is= %f", *( (float*) ptr) );
   return 0;
}
```

Another Example

```
#include"stdio.h"
int main()
{
  int i;
  char c;
  void *the_data;
  i = 6;
  c = 'a';
  the_data = &i;
  printf("the_data points to the integer value %d\n", *(int*) the_data);
  the_data = &c;
  printf("the_data now points to the character %c\n", *(char*) the_data);
  return 0;
}
```

Passing Arguments to Functions using Pointer

A function has a physical location in memory that can be assigned to a pointer. This address is the entry point of the function and it is the address used when the function is called. Once a pointer points to a function, the function can be called through that pointer. Function pointers also allow functions to be passed as arguments to other functions.

Pointer as a function parameter list is used to hold the address of argument passed during the function call. This is also known as call by reference. When a function is called by reference any change made to the reference variable will affect the original variable.

Program

```
#include<stdio.h>
void swap(int *a, int *b); // function prototype
int main()
{
    int p=10, q=20;
    printf("Before Swapping:\n\n");
    printf("p = %d\n",p);
    printf("q = %d\n\n",q);
    swap(&p,&q); //passing address of p and q to the swap function
    printf("After Swapping:\n\n");
    printf("p = %d\n",p);
    printf("q = %d\n",q);
    return 0;
}

//pointer a and b holds and points to the address of p and q
void swap(int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

Output

```
Before Swapping:
p = 10
q = 20

After Swapping:
p = 20
q = 10
Press any key to continue . . .
```

The address of memory location m and n are passed to the function swap and the pointers *a and *b accept those values.

So, now the pointer a and b points to the address of m and n respectively.

When, the value of pointers are changed, the value in the pointed memory location also changes correspondingly.

Hence, changes made to*a and *bare reflected in m and n in the main function.

This technique is known as **Call by Reference** in C programming.
Simple Example of Pointer to Function

Program

```
#include<stdio.h>
int add(int x, int y)
{
 return x+y;
}

int main( )
{
        int (*functionPtr)(int, int);
        int s;
        functionPtr = add;   //
         s = functionPtr(20, 45);
        printf("Sum is %d",s);
        getch();
        return 0;
}
```

Output

```
Sum is 65
```

Explanation

It is possible to declare a pointer pointing to a function which can then be used as an argument in another function. A pointer to a function is declared as follows,

```
 type (*pointer-name)(parameter);
```

Example :

```
        int (*add)();   //legal declaration of pointer to function
        int *add();   //This is not a declaration of pointer to function
```

A function pointer can point to a specific function when it is assigned the name of the function.

```
        int add(int, int);
        int (*s)(int, int);
        sr = add;
```

sr is a pointer to a function sum. Now sum can be called using function pointer s with the list of parameter.

```
        sr(10, 20);
```

Function returning Pointer

A function can also return a pointer to the calling function. In this case you must be careful, because local variables of function doesn't live outside the function. They have scope only till inside the function. Hence if you return a pointer connected to a local variable, that pointer be will pointing to nothing when function ends.

Program

This program will check who is larger among two number, it is not for quality checking

```
#include<stdio.h>
int* checklarger(int*, int*);
void main()
{
        int num1 ;
        int num2;
        int *ptr;
        printf("Enter Two number: \n");
        scanf("%d %d",&num1,&num2);
        ptr = checklarger(&num1, &num2);
        printf("%d is larger \n",*ptr);
}

int* checklarger(int *m, int *n)
{
         if(*m > *n)
                return m;
        else
                return n;
}
```

Output

```
Enter Two number:
        546    1213
1213 is larger
```

Address of the Function

We can fetch the address of an array by the array name, without indexes, Similarly We can fetch the address of a function by using the function's name without any parentheses or arguments. To see how this is done, read the following program, which compares two strings entered by the user. Pay close attention to the declarations of checkString( ) and the function pointer p, inside main( ).

```
#include<stdio.h>
void checkString(char *a, char *b,
int (*cmp)(const char *, const char *));
int main(void)
{
        char strng1[80], strng2[80];
        int (*ptr)(const char *, const char *); /* function pointer */
        ptr = strcmp; /* assign address of strcmp to ptr */
        printf("Enter two strings.\n");
        gets(strng1);
        gets(strng2);
        checkString(strng1,strng2,ptr); /* pass address of strcmp via ptr */
        return 0;
}

void checkString(char *m, char *n,
int (*cmp) (const char *, const char *))
{
        printf("Testing for equality.\n");
        if(!(*cmp)(m, n))
        {
                printf("Equal \n");
        }
        Else
        {
                 printf("Not Equal \n");
        }
}
```

```
Output 1:
Enter two strings.
          atnyla
          atnyla
          Testing for equality.
          Equal

Output 2:
Enter two strings.
atnyla
atnlla
Testing for equality.
Not Equal
Press any key to continue . . .
```

Explanation

First, examine the declaration for ptr in main( ). It is shown here:

```
    int (*ptr)(const char *, const char *);
```

This declaration tells the compiler that ptr is a pointer to a function that has two const char * parameters, and returns an int result. The parentheses around ptr are necessary in order for the compiler to properly interpret this declaration. You must use a similar form when declaring other function pointers, although the return type and parameters of the function may differ.

```
     void checkString(char *m, char *n,
int (*cmp) (const char *, const char *))
```

Next, examine the checkString( ) function. It declares three parameters: two character pointers, m and n, and one function pointer, cmp. Notice that the function pointer is declared using the same format as was ptr inside main( ). Thus, cmp is able to receive a pointer to a function that takes two const char * arguments and returns an int result. Like the declaration for ptr, the parentheses around the *cmp are necessary for the compiler to interpret this statement correctly.

When the program begins, it assigns ptr the address of strcmp( ), the standard string comparison function. Next, it prompts the user for two strings, and then it passes pointers to those strings along with ptr to check( ), which compares the strings for equality. Inside checkString( ), the expression

```
(*cmp)(a, b)
```

calls strcmp( ), which is pointed to by cmp, with the arguments m and n. The parentheses around *cmp are necessary. This is one way to call a function through a pointer. A second, simpler syntax, as shown here, can also be used.

```
cmp(a, b);
```

The reason that you will frequently see the first style is that it tips off anyone reading your code that a function is being called through a pointer (that is, that cmp is a function pointer, not the name of a function). Also, the first style was the form originally specified by C.

Note that you can call checkString( ) by using strcmp( ) directly, as shown here:

```
checkString(s1, s2, strcmp);
```

Pointer and Arrays

There is a close association between pointers and arrays. Let us consider the following statements:

```
int x[5] = {11, 22, 33, 44, 55};
int *p = x;
```

The array initialization statement is familiar to us. The second statement, array name x is the starting address of the array. Let we take a sample memory map as shown in figure 4.2.:

From the figure 4.2 we can see that the starting address of the array is 1000. When x is an array, it also represents an address and so there is no need to use the (&) symbol before x. We can write int *p = x in place of writing int *p = &x[0].

The content of p is 1000 (see the memory map given below). To access the value in x[0] by using pointers, the indirection operator * with its pointer variable p by the notation *p can be used.

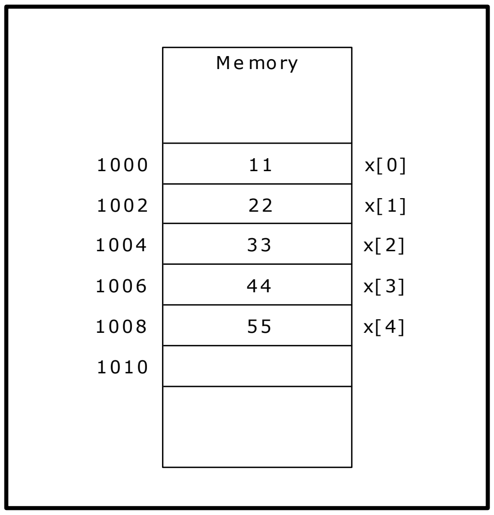


Figure 4.2. Memory map - Arrays

The increment operator ++ helps you to increment the value of the pointer variable by the size of the data type it points to. Therefore, the expression p++ will increment p by 2 bytes (as p points to an integer) and new value in p will be 1000 + 2 = 1002, now *p will get you 22 which is x[1].

Consider the following expressions:

```
*p++;
*(p++);
(*p)++;
```

How would they be evaluated when the integers 10 & 20 are stored at addresses 1000 and 1002 respectively with p set to 1000.

p++ :	The increment ++ operator has a higher priority than the indirection operator * . Therefore p is increment first. The new value in p is then 1002 and the content at this address is 20.
*(p++):	is same as *p++.
(*p)++:	*p which is content at address 1000 (i.e. 10) is incremented. Therefore (*p)++ is 11.

Note that, *p++ = content at incremented address.

Example:

```
#include <stdio.h>
main()
{
        int x[5] = {11, 22, 33, 44, 55};
        int *p = x, i;                    /*  p=&x[0] = address of the first element */
        for (i = 0; i < 5; i++)
        {
```

```
        printf ("\n x[%d] = %d", i, *p);        /* increment the address*/
        p++;
    }
}
```

Output:

```
x [0] = 11
x [1] = 22
x [2] = 33
x [3] = 44
x [4] = 55
```

The meanings of the expressions p, p+1, p+2, p+3, p+4 and the expressions *p, *(p+1), *(p+2), *(p+3), *(p+4) are as follows:

P = 1000 P+1 = 1000 + 1 x 2 = 1002 P+2 = 1000 + 2 x 2 = 1004 P+3 = 1000 + 3 x 2 = 1006 P+4 = 1000 + 4 x 2 = 1008	*p = content at address 1000 = x[0] *(p+1) = content at address 1002 = x[1] *(p+2) = content at address 1004 = x[2] *(p+3) = content at address 1006 = x[3] *(p+4) = content at address 1008 = x[4]

Pointers and strings:

A string is an array of characters. Thus pointer notation can be applied to the characters in strings. Consider the statements:

```
    char tv[20] = "ONIDA";
    char *p = tv;
```

For the first statement, the compiler allocates 20 bytes of memory and stores in the first six bytes the char values as shown below:

Variable	tv[0]	tv[1]	tv[2]	tv[3]	tv[4]	tv[5]
Value	'O'	'N'	'I'	'D'	'A'	'\0'
Address	1000	1001	1002	1003	1004	1005

The statement:

```
    char *p = tv;                    /* or  p = &tv[0] */
```

Assigns the address 1000 to p. Now, we will write a program to find the length of the string tv and print the string in reverse order using pointer notation.

Example:

```
#include <stdio.h>
main()
{
    int n, i;
    char tv[20] = "ONIDA";                 /* p = 1000 */
    char *p = tv, *q;                      /* p = &tv[0], q is a pointer */
    q = p;
    while (*p != '\0')          /* content at address of p is not null character */
        p++;
    n = p - q;                             /* length of the string */
    --p;                 /* make p point to the last character A in the string */
    printf ("\nLength of the string is %d", n);
    printf ("\nString in reverse order: \n");
```

```
          for (i=0; i<n; i++)
          {
                    putchar (*p);
                    p--;
          }
}
```

Output:

```
Length of the string is 5
String in reverse order: ADINO
```

Linked Lists: Introduction to Lists and Linked Lists

Linked lists and arrays are similar since they both store collections of data. One way to think about linked lists is to look at how arrays work and think about alternate approaches.

Array is the most common data structure used to store collections of elements. Arrays are convenient to declare and provide the easy syntax to access any element by its index number. Once the array is set up, access to any element is convenient and fast.

The **disadvantages** of arrays are:

• The size of the array is fixed. Most often this size is specified at compile time. This makes the programmers to allocate arrays, which seems "large enough" than required.

• Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.

• Deleting an element from an array is not possible.

Linked lists have their own strengths and weaknesses, but they happen to be strong where arrays are weak. Generally array's allocates the memory for all its elements in one block whereas linked lists use an entirely different strategy.

Linked lists allocate memory for each element separately and only when necessary.

Dynamic Memory Allocation:

The concept of **dynamic memory allocation in c language** *enables the C programmer to allocate memory at runtime*. Dynamic memory allocation in c language is possible by 4 functions of stdlib.h header file.

1. malloc()
2. calloc()
3. realloc()
4. free()

Before learning above functions, let's understand the difference between static memory allocation and dynamic memory allocation.

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.

Now let's have a quick look at the methods used for dynamic memory allocation.

malloc()	allocates single block of requested memory.
calloc()	allocates multiple block of requested memory.
realloc()	reallocates the memory occupied by malloc() or calloc() functions.
free()	frees the dynamically allocated memory.

malloc() function in C

The malloc() function allocates single block of requested memory.

It doesn't initialize memory at execution time, so it has garbage value initially.

It returns NULL if memory is not sufficient.

The syntax of malloc() function is given below:

```
ptr=(cast-type*)malloc(byte-size)
```

Let's see the example of malloc() function.

```
#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));  //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);
return 0;
}
```

Output

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

calloc() function in C

The calloc() function allocates multiple block of requested memory.
It initially initialize all bytes to zero.
It returns NULL if memory is not sufficient.
The syntax of calloc() function is given below:

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

Let's see the example of calloc() function.

```
#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));  //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);
return 0;
}
```

Output

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

realloc() function in C

If memory is not sufficient for malloc() or calloc(), you can reallocate the memory by realloc() function. In short, it changes the memory size.

Let's see the syntax of realloc() function.

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

free() function in C

The memory occupied by malloc() or calloc() functions must be released by calling free() function. Otherwise, it will consume memory until program exit.

Let's see the syntax of free() function.

```
free(ptr)
```

Basic Linked List Operations:

A linked list is a non-sequential collection of data items. It is a dynamic data structure. For every data item in a linked list, there is an associated pointer that would give the memory location of the next data item in the linked list.

The data items in the linked list are not in consecutive memory locations. They may be anywhere, but the accessing of these data items is easier as each data item contains the address of the next data item.

Advantages of linked lists:

Linked lists have many advantages. Some of the very important advantages are:

1. Linked lists are dynamic data structures. i.e., they can grow or shrink during the execution of a program.
2. Linked lists have efficient memory utilization. Here, memory is not pre-allocated. Memory is allocated whenever it is required and it is de-allocated (removed) when it is no longer needed.
3. Insertion and Deletions are easier and efficient. Linked lists provide flexibility in inserting a data item at a specified position and deletion of the data item from the given position.
4. Many complex applications can be easily carried out with linked lists.

Disadvantages of linked lists:

1. It consumes more space because every node requires a additional pointer to store address of the next node.
2. Searching a particular element in list is difficult and also time consuming.

Types of Linked Lists:

Basically we can put linked lists into the following four items:

1. Single Linked List.
2. Double Linked List.
3. Circular Linked List.
4. Circular Double Linked List.

A single linked list is one in which all nodes are linked together in some sequential manner. Hence, it is also called as linear linked list.

A double linked list is one in which all nodes are linked together by multiple links which helps in accessing both the successor node (next node) and predecessor node (previous node) from any arbitrary node within the list. Therefore each node in a double linked list has two link fields (pointers) to point to the left node (previous) and the right node (next). This helps to traverse in forward direction and backward direction.

A circular linked list is one, which has no beginning and no end. A single linked list can be made a circular linked list by simply storing address of the very first node in the link field of the last node.

A circular double linked list is one, which has both the successor pointer and predecessor pointer in the circular manner.

Comparison between array and linked list:

ARRAY	**LINKED LIST**
Size of an array is fixed	Size of a list is not fixed
Memory is allocated from stack	Memory is allocated from heap
It is necessary to specify the number of elements during declaration (i.e., during compile time).	It is not necessary to specify the number of elements during declaration (i.e., memory is allocated during run time).
It occupies less memory than a linked list for the same number of elements.	It occupies more memory.
Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.	Inserting a new element at any position can be carried out easily.
Deleting an element from an array is not possible.	Deleting an element is possible.

Trade offs between linked lists and arrays:

FEATURE	**ARRAYS**	**LINKED LISTS**
Sequential access	efficient	efficient
Random access	efficient	inefficient
Resigning	inefficient	efficient
Element rearranging	inefficient	efficient
Overhead per elements	none	1 or 2 inks

Applications of linked list:

1. Linked lists are used to represent and manipulate polynomial. Polynomials are expression containing terms with non zero coefficient and exponents. For example:

   $P(x) = a_0 X^n + a_1 X^{n-1} + \ldots\ldots + a_{n-1} X + a_n$

2. Represent very large numbers and operations of the large number such as addition, multiplication and division.
3. Linked lists are to implement stack, queue, trees and graphs.
4. Implement the symbol table in compiler construction

Single Linked List:

A linked list allocates space for each element separately in its own block of memory called a "node". The list gets an overall structure by using pointers to connect all its nodes together like the links in a chain.

Each node contains two fields; a "data" field to store whatever element, and a "next" field which is a pointer used to link to the next node.

Each node is allocated in the heap using malloc(), so the node memory continues to exist until it is explicitly de-allocated using free(). The front of the list is a pointer to the "start" node. A single linked list is shown in figure 6.2.1.

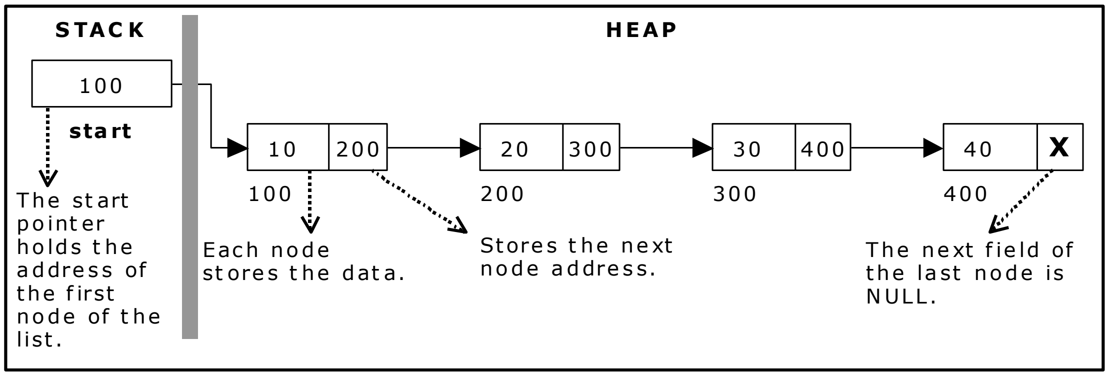

STACK
HEAP
100
start
10
200
20
300
30
400
40
X
100
200
300
400
The start pointer holds the address of the first node of the list.
Each node stores the data.
Stores the next node address.
The next field of the last node is NULL.

Figure 6.2.1. Single Linked List

The beginning of the linked list is stored in a "**start**" pointer which points to the first node. The first node contains a pointer to the second node. The second node contains a pointer to the third node, ... and so on. The last node in the list has its next field set to NULL to mark the end of the list. Code can access any node in the list by starting at the **start** and following the next pointers.

The **start** pointer is an ordinary local pointer variable, so it is drawn separately on the left top to show that it is in the stack. The list nodes are drawn on the right to show that they are allocated in the heap.

Implementation of Single Linked List:

Before writing the code to build the above list, we need to create a **start** node**,** used to create and access other nodes in the linked list. The following structure definition will do (see figure 6.2.2):

- Creating a structure with one data item and a next pointer, which will be pointing to next node of the list. This is called as self-referential structure.
- Initialise the start pointer to be NULL.

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

typedef struct slinklist node;

node *start = NULL;
```

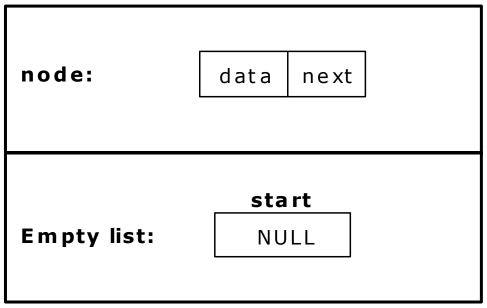


Figure 6.2.2. Structure definition, single link node and empty list

The basic operations in a single linked list are:

- Creation.
- Insertion.
- Deletion.
- Traversing.

Creating a node for Single Linked List:

Creating a singly linked list starts with creating a node. Sufficient memory has to be allocated for creating a node. The information is stored in the memory, allocated by using the malloc() function. The function getnode(), is used for creating a node, after allocating memory for the structure of type node, the information for the item (i.e., data) has to be read from the user, set next field to NULL and finally returns the address of the node. Figure 6.2.3 illustrates the creation of a node for single linked list.

```
node* getnode()
{
     node* newnode;
     newnode = (node *) malloc(sizeof(node));
     printf("\n Enter data: ");
     scanf("%d", &newnode -> data);
     newnode -> next = NULL;
     return newnode;
}
```

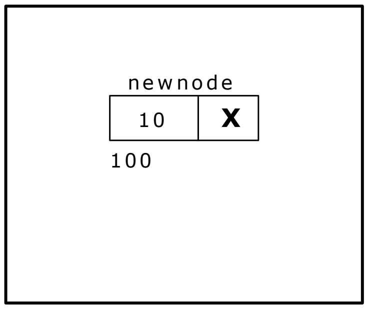


Figure 6.2.3. new node with a value of 10

Creating a Singly Linked List with 'n' number of nodes:

The following steps are to be followed to create 'n' number of nodes:

- Get the new node using getnode().
  newnode = getnode();
- If the list is empty, assign new node as start.
  start = newnode;
- If the list is not empty, follow the steps given below:
  - The next field of the new node is made to point the first node (i.e. start node) in the list by assigning the address of the first node.
  - The start pointer is made to point the new node by assigning the address of the new node.
- Repeat the above steps 'n' times.

Figure 6.2.4 shows 4 items in a single linked list stored at different locations in memory.

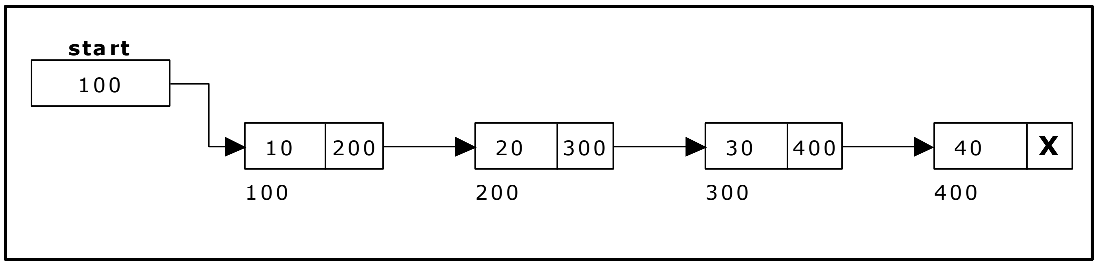


Figure 6.2.4. Single Linked List with 4 nodes

The function createlist(), is used to create 'n' number of nodes:

```
void createlist(int n)
{
        int i;
        node *newnode;
        node *temp;
        for(i = 0; i < n ; i++)
        {
                newnode = getnode();
                if(start == NULL)
                {
                        start = newnode;
                }
                else
                {
                        temp = start;
                        while(temp -> next != NULL)
                                temp = temp -> next;
                        temp -> next = newnode;
                }
        }
}
```

Insertion of a Node:

One of the most primitive operations that can be done in a singly linked list is the insertion of a node. Memory is to be allocated for the new node (in a similar way that is done while creating a list) before reading the data. The new node will contain empty data field and empty next field. The data field of the new node is then stored with the information read from the user. The next field of the new node is assigned to NULL. The new node can then be inserted at three different places namely:

- Inserting a node at the beginning.
- Inserting a node at the end.
- Inserting a node at intermediate position.

Inserting a node at the beginning:

The following steps are to be followed to insert a new node at the beginning of the list:

- Get the new node using getnode().
  newnode = getnode();
- If the list is empty then *start = newnode.*
- If the list is not empty, follow the steps given below:
  newnode -> next = start;
  start = newnode;

The function insert_at_beg(), is used for inserting a node at the beginning
Figure 6.2.5 shows inserting a node into the single linked list at the beginning.

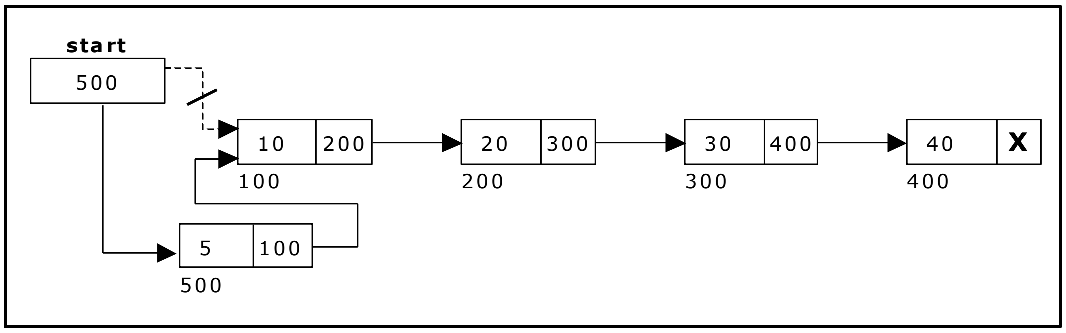


Figure 6.2.5. Inserting a node at the beginning

Inserting a node at the end:

The following steps are followed to insert a new node at the end of the list:

- Get the new node using getnode()

```
newnode = getnode();
```

- If the list is empty then *start = newnode*.
- If the list is not empty follow the steps given below:

```
temp = start;
while(temp -> next != NULL)
        temp = temp -> next;
temp -> next = newnode;
```

The function insert_at_end(), is used for inserting a node at the end.
Figure 6.2.6 shows inserting a node into the single linked list at the end.

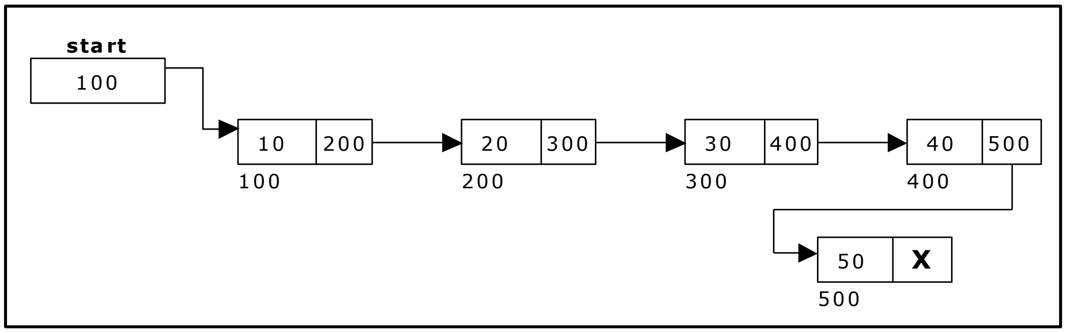


Figure 6.2.6. Inserting a node at the end.

Inserting a node at intermediate position:

The following steps are followed, to insert a new node in an intermediate position in the list:

- Get the new node using getnode().

```
newnode = getnode();
```

- Ensure that the specified position is in between first node and last node. If not, specified position is invalid. This is done by countnode() function.

- Store the starting address (which is in start pointer) in temp and prev pointers. Then traverse the temp pointer upto the specified position followed by prev pointer.

- After reaching the specified position, follow the steps given below:

```
prev -> next = newnode;
newnode -> next = temp;
```

- Let the intermediate position be 3.

The function insert_at_mid(), is used for inserting a node in the intermediate position.

Figure 6.2.7 shows inserting a node into the single linked list at a specified intermediate position other than beginning and end.

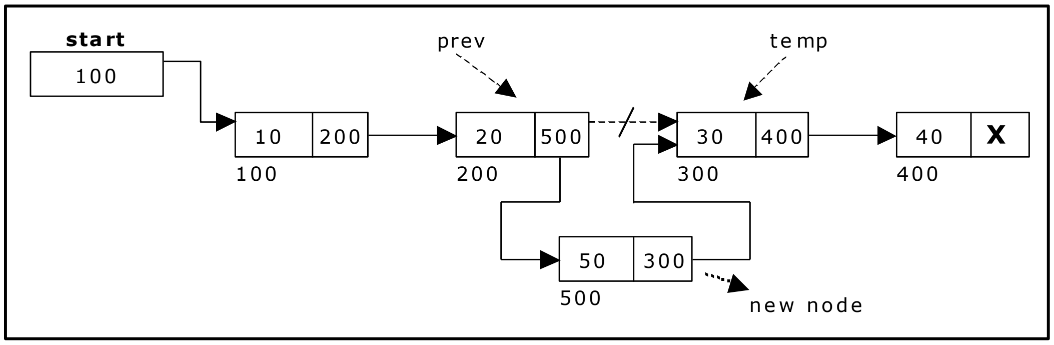


Figure 6.2.7. Inserting a node at an intermediate position.

Deletion of a node:

Another primitive operation that can be done in a singly linked list is the deletion of a node. Memory is to be released for the node to be deleted. A node can be deleted from the list from three different places namely.

- Deleting a node at the beginning.
- Deleting a node at the end.
- Deleting a node at intermediate position.

Deleting a node at the beginning:

The following steps are followed, to delete a node at the beginning of the list:

- If list is empty then display 'Empty List' message.
- If the list is not empty, follow the steps given below:

```
temp = start;
start = start -> next;
free(temp);
```

The function delete_at_beg(), is used for deleting the first node in the list.
Figure 6.2.8 shows deleting a node at the beginning of a single linked list.

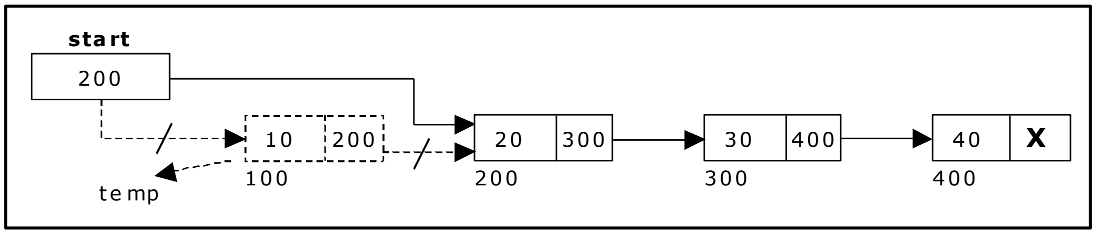


Figure 6.2.8. Deleting a node at the beginning.

Deleting a node at the end:

The following steps are followed to delete a node at the end of the list:

- If list is empty then display 'Empty List' message.
- If the list is not empty, follow the steps given below:

```
temp = prev = start;
while(temp -> next != NULL)
{
        prev = temp;
        temp = temp -> next;
}
prev -> next = NULL;
free(temp);
```
The function delete_at_last(), is used for deleting the last node in the list. Figure 6.2.9 shows deleting a node at the end of a single linked list.

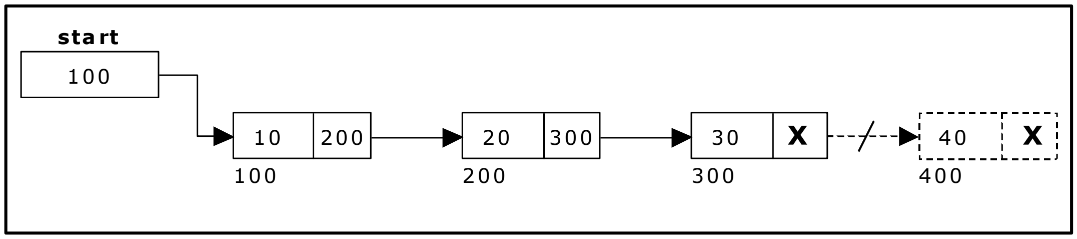


Figure 6.2.9. Deleting a node at the end.

Deleting a node at Intermediate position:

The following steps are followed, to delete a node from an intermediate position in the list (List must contain more than two node).

- If list is empty then display 'Empty List' message
- If the list is not empty, follow the steps given below.

```
if(pos > 1 && pos < nodectr)
{
        temp = prev = start;
        ctr = 1;
        while(ctr < pos)
        {
                prev = temp;
                temp = temp -> next;
                ctr++;
        }
        prev -> next = temp -> next;
        free(temp);
        printf("\n node deleted..");
}
```

The function delete_at_mid(), is used for deleting the intermediate node in the list. Figure 6.2.10 shows deleting a node at a specified intermediate position other than beginning and end from a single linked list.

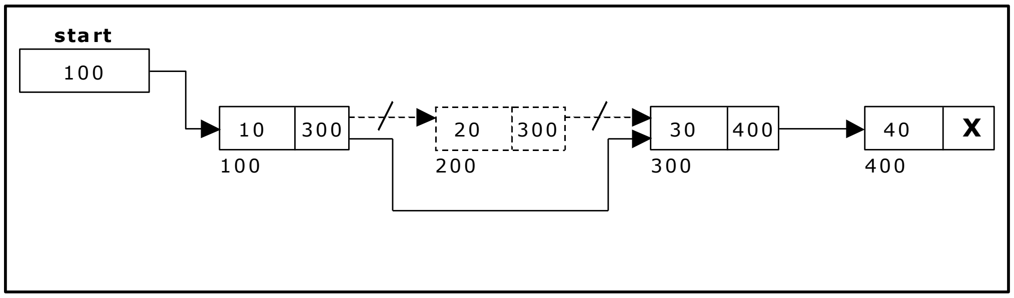


Figure 6.2.10. Deleting a node at an intermediate position.

Traversal and displaying a list (Left to Right):

To display the information, you have to traverse (move) a linked list, node by node from the first node, until the end of the list is reached. Traversing a list involves the following steps:

- Assign the address of start pointer to a temp pointer.
- Display the information from the data field of each node.

The function *traverse*() is used for traversing and displaying the information stored in the list from left to right.

```
void traverse()
{
	node *temp;
	temp = start;
	printf("\n The contents of List (Left to Right): \n");
	if(start == NULL )
		printf("\n Empty List");
	else
	while(temp != NULL)
	{
		printf("%d ->", temp -> data);
		temp = temp -> next;
	}
	printf("X");
}
```

Alternatively there is another way to traverse and display the information. That is in reverse order. The function rev_traverse(), is used for traversing and displaying the information stored in the list from right to left.

```
void rev_traverse(node *st)
{
	if(st == NULL)
	{
		return;
	}
	else
	{
		rev_traverse(st -> next);
		printf("%d ->", st -> data);
	}
}
```

Counting the Number of Nodes:

The following code will count the number of nodes exist in the list using *recursion*.

```
int countnode(node *st)
{
	if(st == NULL)
		return 0;
	else
		return(1 + countnode(st -> next));
}
```

Doubly Linked List

A double linked list is a two-way list in which all nodes will have two links. This helps in accessing both successor node and predecessor node from the given node position. It provides bi-directional traversing. Each node contains three fields:

- Left link.
- Data.
- Right link.

The left link points to the predecessor node and the right link points to the successor node. The data field stores the required data.

Many applications require searching forward and backward thru nodes of a list. For example searching for a name in a telephone directory would need forward and backward scanning thru a region of the whole list.

The basic operations in a double linked list are:

- Creation.
- Insertion.
- Deletion.
- Traversing.

A double linked list is shown in figure 6.3.1.

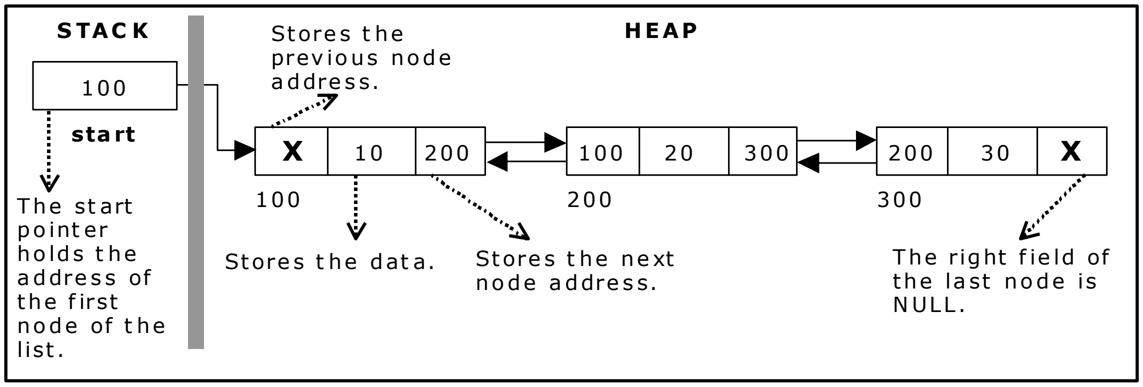

STACK
HEAP
100
start
The start pointer holds the address of the first node of the list.
Stores the previous node address.
X
10
200
100
20
300
200
30
X
100
200
300
Stores the data.
Stores the next node address.
The right field of the last node is NULL.

Figure 6.3.1. Double Linked List

The beginning of the double linked list is stored in a "**start**" pointer which points to the first node. The first node's left link and last node's right link is set to NULL.
The following code gives the structure definition:

```
struct dlinklist
{
        struct dlinklist *left;
        int data;
        struct dlinklist *right;

};

typedef struct dlinklist node;
node *start = NULL;
```

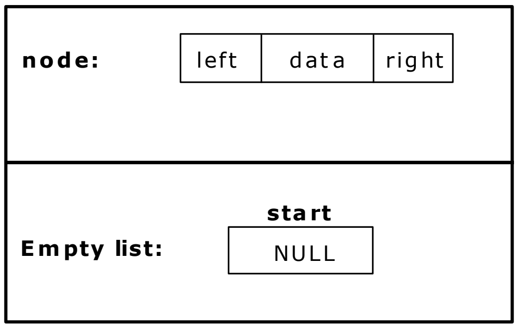


Figure 6.4.1. Structure definition, double link node and empty list

Creating a node for Double Linked List:

Creating a double linked list starts with creating a node. Sufficient memory has to be allocated for creating a node. The information is stored in the memory, allocated by using the malloc() function. The function getnode(), is used for creating a node, after allocating memory for the structure of type node, the information for the item (i.e., data) has to be read from the user and set left field to NULL and right field also set to NULL (see figure 6.2.2).

```
node* getnode()
{
     node* newnode;
     newnode = (node *) malloc(sizeof(node));
     printf("\n Enter data: ");
     scanf("%d", &newnode -> data);
     newnode -> left = NULL;
     newnode -> right = NULL;
     return newnode;
}
```

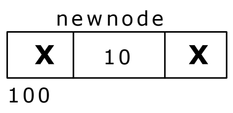


Figure 6.4.2. new node with a value of 10

Creating a Double Linked List with 'n' number of nodes:
The following steps are to be followed to create 'n' number of nodes:

- Get the new node using getnode().
  newnode =getnode();
- If the list is empty then *start = newnode*.
- If the list is not empty, follow the steps given below:
  - The left field of the new node is made to point the previous node.
  - The previous nodes right field must be assigned with address of the new node.
- Repeat the above steps 'n' times.

The function createlist(), is used to create 'n' number of nodes:

```
void createlist(int n)
{
        int i;
        node *newnode;
        node *temp;
        for(i = 0; i < n; i++)
        {
                newnode = getnode();
                if(start == NULL)
                {
                        start = newnode;
                }
                else
                {
                        temp = start;
                        while(temp -> right)
                                temp = temp -> right;
                        temp -> right = newnode;
                        newnode -> left = temp;
                }
        }
}
```

Figure 6.4.3 shows 3 items in a double linked list stored at different locations.

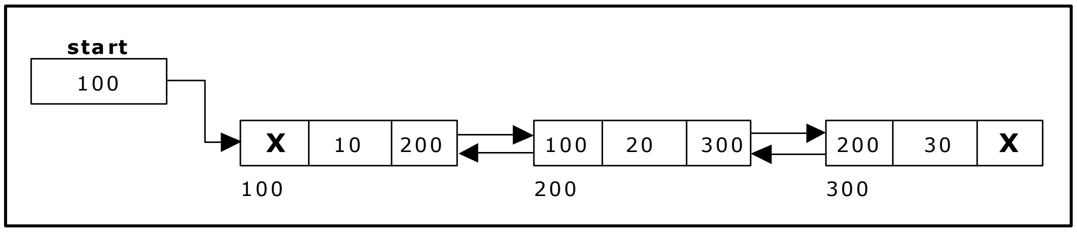


Figure 6.4.3. Double Linked List with 3 nodes

Inserting a node at the beginning:

The following steps are to be followed to insert a new node at the beginning of the list:

- Get the new node using getnode().
  ```
  newnode=getnode();
  ```
- If the list is empty then *start = newnode*.
- If the list is not empty, follow the steps given below:
  ```
  newnode -> right = start;
  start -> left = newnode;
  start = newnode;
  ```

The function dbl_insert_beg(), is used for inserting a node at the beginning. Figure 6.4.4 shows inserting a node into the double linked list at the beginning.

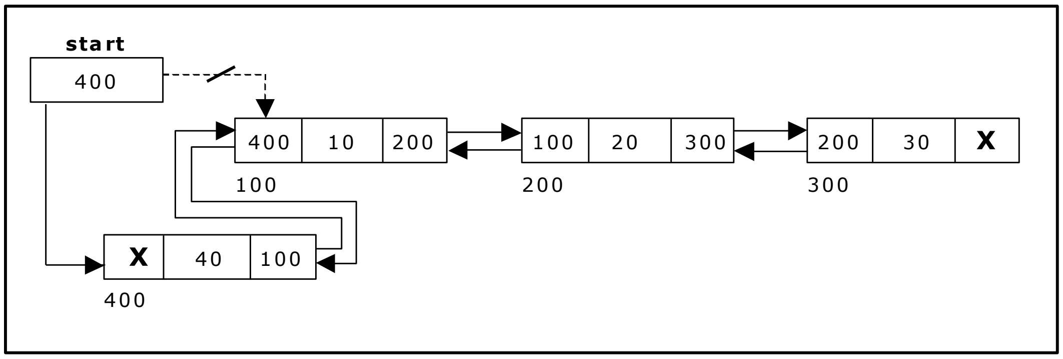


Figure 6.4.4. Inserting a node at the beginning

Inserting a node at the end:

The following steps are followed to insert a new node at the end of the list:

- Get the new node using getnode()
  ```
  newnode=getnode();
  ```
- If the list is empty then *start = newnode*.
- If the list is not empty follow the steps given below:
  ```
  temp = start;
  while(temp -> right != NULL)
          temp = temp -> right;
  temp -> right = newnode;
  newnode -> left = temp;
  ```

The function dbl_insert_end(), is used for inserting a node at the end. Figure 6.4.5 shows inserting a node into the double linked list at the end.

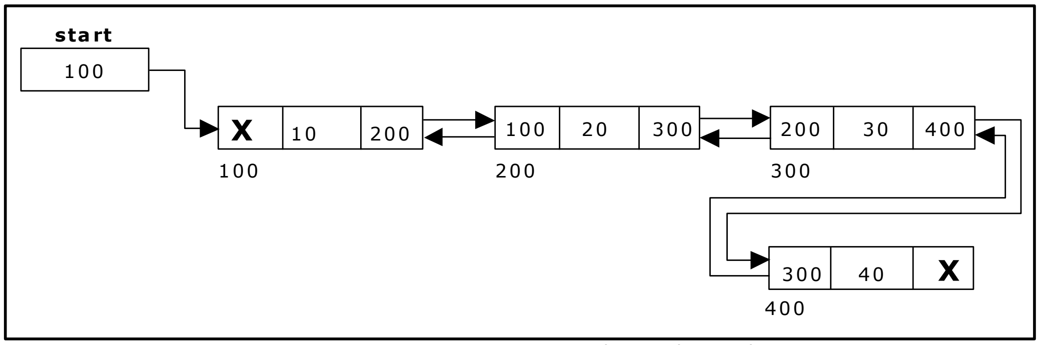


Figure 6.4.5. Inserting a node at the end

Inserting a node at an intermediate position:

The following steps are followed, to insert a new node in an intermediate position in the list:

- Get the new node using getnode().

  newnode=getnode();
- Ensure that the specified position is in between first node and last node. If not, specified position is invalid. This is done by countnode() function.

- Store the starting address (which is in start pointer) in temp and prev pointers. Then traverse the temp pointer upto the specified position followed by prev pointer.
- After reaching the specified position, follow the steps given below:

  newnode -> left = temp;
  newnode -> right = temp -> right;
  temp -> right -> left = newnode;
  temp -> right = newnode;

The function dbl_insert_mid(), is used for inserting a node in the intermediate position. Figure 6.4.6 shows inserting a node into the double linked list at a specified intermediate position other than beginning and end.

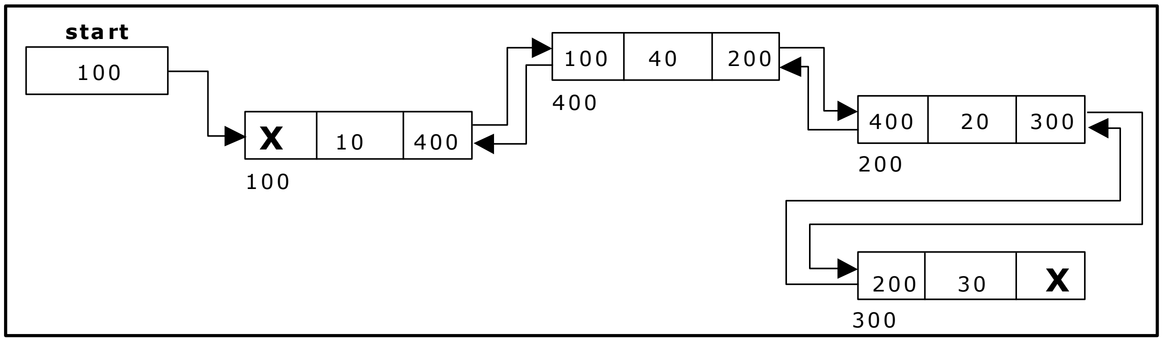


Figure 6.4.6. Inserting a node at an intermediate position

Deleting a node at the beginning:

The following steps are followed, to delete a node at the beginning of the list:

- If list is empty then display 'Empty List' message.
- If the list is not empty, follow the steps given below:

  temp = start;
  start = start -> right;
  start -> left = NULL;
  free(temp);

The function dbl_delete_beg(), is used for deleting the first node in the list. Figure 6.4.6 shows deleting a node at the beginning of a double linked list.

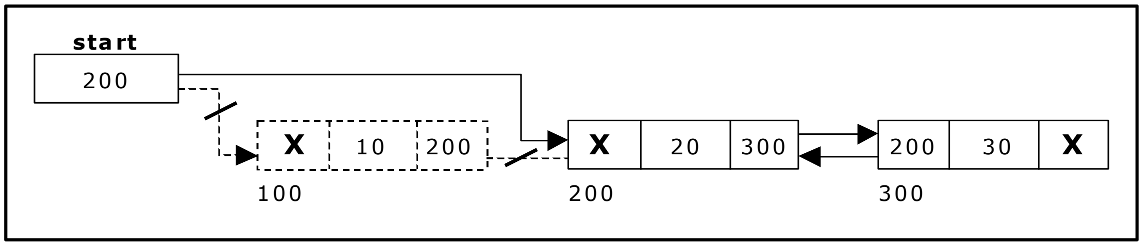


Figure 6.4.6. Deleting a node at beginning

Deleting a node at the end:

The following steps are followed to delete a node at the end of the list:

- If list is empty then display 'Empty List' message
- If the list is not empty, follow the steps given below:

```
temp = start;
while(temp -> right != NULL)
{
        temp = temp -> right;
}
temp -> left -> right = NULL;
free(temp);
```

The function dbl_delete_last(), is used for deleting the last node in the list. Figure 6.4.7 shows deleting a node at the end of a double linked list.

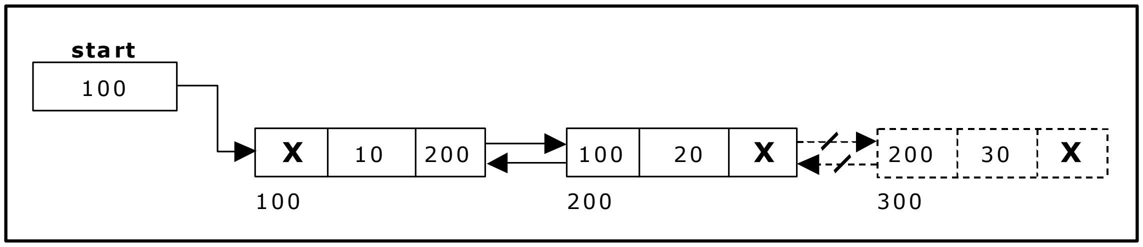


Figure 6.4.7. Deleting a node at the end

Deleting a node at Intermediate position:

The following steps are followed, to delete a node from an intermediate position in the list (List must contain more than two nodes).

- If list is empty then display 'Empty List' message.
- If the list is not empty, follow the steps given below:
  - Get the position of the node to delete.
  - Ensure that the specified position is in between first node and last node. If not, specified position is invalid.
  - Then perform the following steps:

```
if(pos > 1 && pos < nodectr)
{
        temp = start;
        i = 1;
        while(i < pos)
        {
                temp = temp -> right;
                i++;
```

```
                    }
                    temp -> right -> left = temp -> left;
                    temp -> left -> right = temp -> right;
                    free(temp);
                    printf("\n node deleted..");
          }
```

The function delete_at_mid(), is used for deleting the intermediate node in the list. Figure 6.4.8 shows deleting a node at a specified intermediate position other than beginning and end from a double linked list.

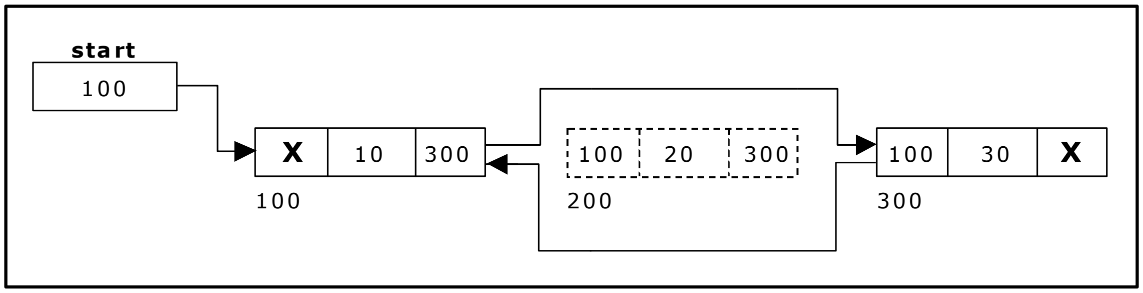


Figure 6.4.8 Deleting a node at an intermediate position

Traversal and displaying a list (Left to Right):

To display the information, you have to traverse the list, node by node from the first node, until the end of the list is reached. The function *traverse_left_right*() is used for traversing and displaying the information stored in the list from left to right.

The following steps are followed, to traverse a list from left to right:

- If list is empty then display 'Empty List' message.
- If the list is not empty, follow the steps given below:

```
temp = start;
while(temp != NULL)
{
        print  temp -> data;
        temp = temp -> right;
}
```

Traversal and displaying a list (Right to Left):

To display the information from right to left, you have to traverse the list, node by node from the first node, until the end of the list is reached. The function *traverse_right_left*() is used for traversing and displaying the information stored in the list from right to left.

The following steps are followed, to traverse a list from right to left:

- If list is empty then display 'Empty List' message.
- If the list is not empty, follow the steps given below:

```
temp = start;
while(temp -> right != NULL)
        temp = temp -> right;
while(temp != NULL)
{
        print temp -> data;
        temp = temp -> left;
}
```

Counting the Number of Nodes:
The following code will count the number of nodes exist in the list (using recursion).

```
int countnode(node *start)
{
        if(start == NULL)
                return 0;
        else
                return(1 + countnode(start ->right ));
}
```

Circular Linked List

Circular Single Linked List:

It is just a single linked list in which the link field of the last node points back to the address of the first node. A circular linked list has no beginning and no end. It is necessary to establish a special pointer called *start* pointer always pointing to the first node of the list. Circular linked lists are frequently used instead of ordinary linked list because many operations are much easier to implement. In circular linked list no null pointers are used, hence all pointers contain valid address.
A circular single linked list is shown in figure 6.6.1.

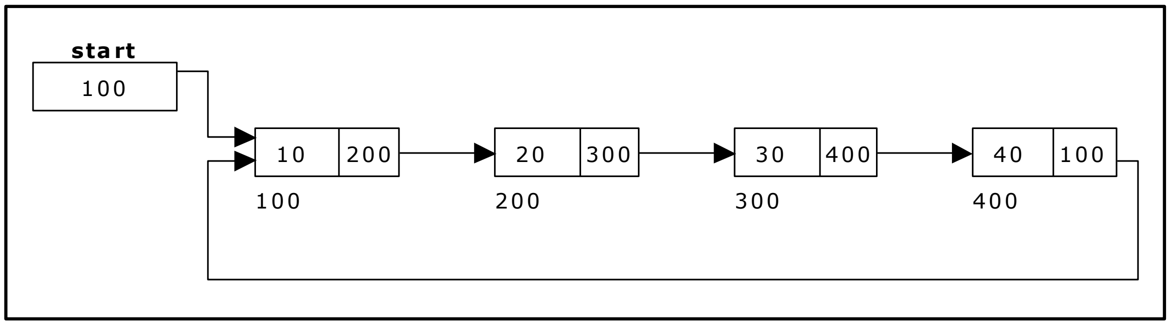


Figure 6.6.1. Circular Single Linked List

The basic operations in a circular single linked list are:

- Creation.
- Insertion.
- Deletion.
- Traversing.

Creating a circular single Linked List with 'n' number of nodes:

The following steps are to be followed to create 'n' number of nodes:

- Get the new node using getnode().
  newnode = getnode();
- If the list is empty, assign new node as start.
  start = newnode;
- If the list is not empty, follow the steps given below:
  ```
  temp = start;
  while(temp -> next != NULL)
          temp = temp -> next;
  ```

```
        temp -> next = newnode;
```

- Repeat the above steps 'n' times.
- newnode -> next = start;

The function createlist(), is used to create 'n' number of nodes:

Inserting a node at the beginning:
The following steps are to be followed to insert a new node at the beginning of the circular list:

- Get the new node using getnode().
```
    newnode = getnode();
```
- If the list is empty, assign new node as start.
```
    start = newnode;
    newnode -> next = start;
```
- If the list is not empty, follow the steps given below:
```
    last = start;
    while(last -> next != start)
            last = last -> next;
    newnode -> next = start;
    start = newnode;
    last -> next = start;
```

The function cll_insert_beg(), is used for inserting a node at the beginning. Figure 6.6.2 shows inserting a node into the circular single linked list at the beginning.

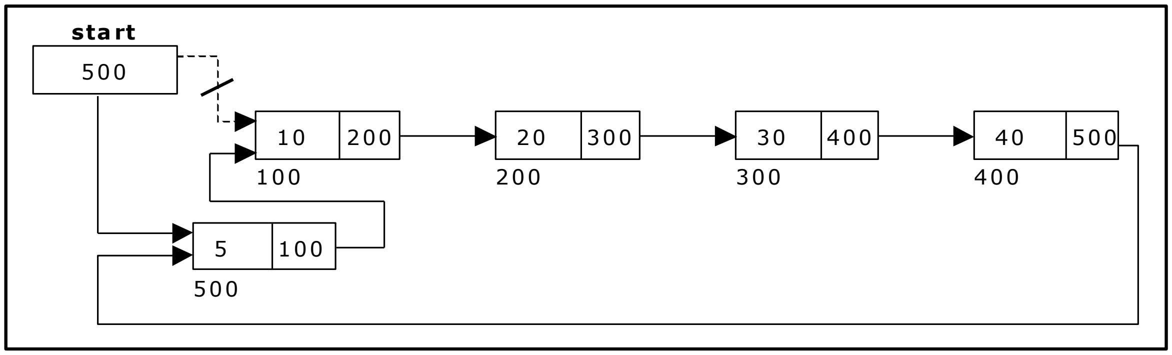


Figure 6.6.2. Inserting a node at the beginning

Inserting a node at the end:
The following steps are followed to insert a new node at the end of the list:

- Get the new node using getnode().
```
    newnode = getnode();
```
- If the list is empty, assign new node as start.
```
    start = newnode;
    newnode -> next = start;
```
- If the list is not empty follow the steps given below:
```
    temp = start;
    while(temp -> next != start)
            temp = temp -> next;
    temp -> next = newnode;
    newnode -> next = start;
```

The function cll_insert_end(), is used for inserting a node at the end.
Figure 6.6.3 shows inserting a node into the circular single linked list at the end.

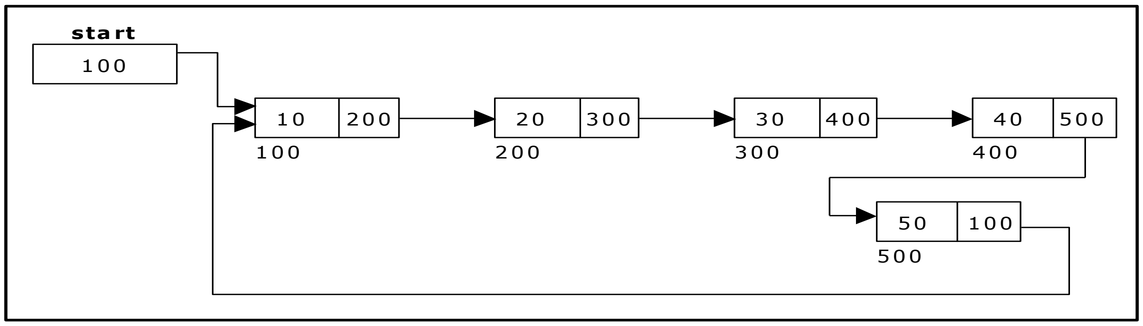


Figure 6.6.3 Inserting a node at the end.

Deleting a node at the beginning:

The following steps are followed, to delete a node at the beginning of the list:

- If the list is empty, display a message 'Empty List'.
- If the list is not empty, follow the steps given below:

```
last = temp = start;
while(last -> next != start)
        last = last -> next;
start = start -> next;
last -> next = start;
```

- After deleting the node, if the list is empty then *start = NULL.*

The function cll_delete_beg(), is used for deleting the first node in the list. Figure 6.6.4 shows deleting a node at the beginning of a circular single linked list.

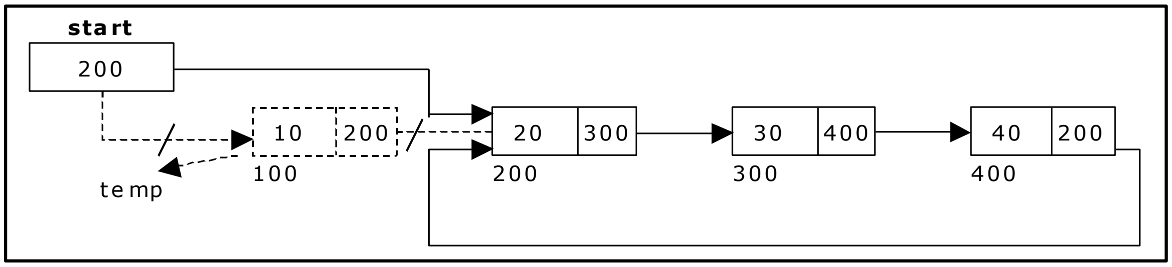


Figure 6.6.4. Deleting a node at beginning.

Deleting a node at the end:

The following steps are followed to delete a node at the end of the list:

- If the list is empty, display a message 'Empty List'.
- If the list is not empty, follow the steps given below:

```
temp = start;
prev = start;
while(temp -> next != start)
{
        prev = temp;
        temp = temp -> next;
}
prev -> next = start;
```

- After deleting the node, if the list is empty then *start = NULL.*

The function cll_delete_last(), is used for deleting the last node in the list. Figure 6.6.5 shows deleting a node at the end of a circular single linked list.

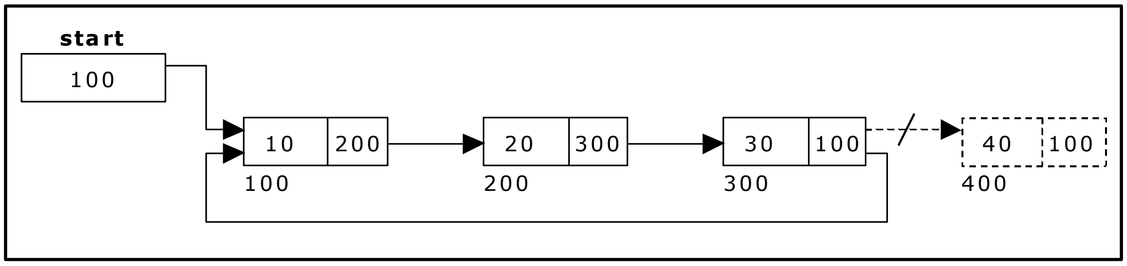


Figure 6.6.5. Deleting a node at the end.

Traversing a circular single linked list from left to right:
The following steps are followed, to traverse a list from left to right:

- If list is empty then display 'Empty List' message.
- If the list is not empty, follow the steps given below:

```
temp = start;
do
{
        printf("%d ", temp -> data);
        temp = temp -> next;
} while(temp != start);
```

Circular Double Linked List:

A circular double linked list has both successor pointer and predecessor pointer in circular manner. The objective behind considering circular double linked list is to simplify the insertion and deletion operations performed on double linked list. In circular double linked list the *right* link of the right most node points back to the *start* node and *left* link of the first node points to the last node.

A circular double linked list is shown in figure 6.8.1.

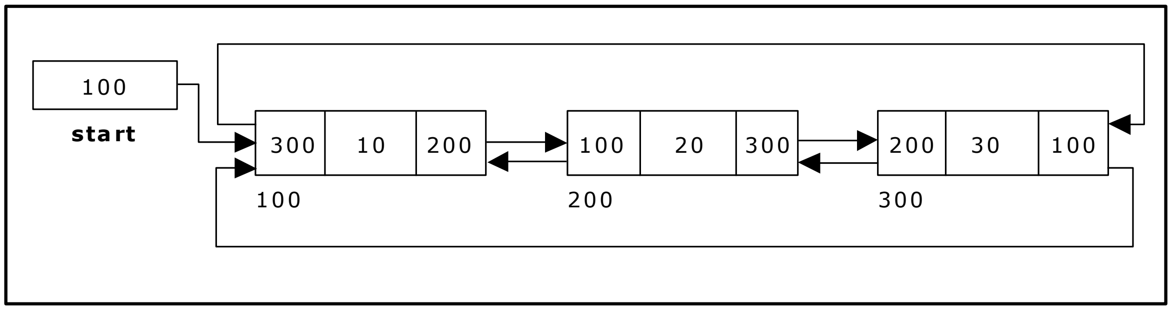


Figure 6.8.1. Circular Double Linked List

The basic operations in a circular double linked list are:

- Creation.
- Insertion.
- Deletion.
- Traversing.

Creating a Circular Double Linked List with 'n' number of nodes:
The following steps are to be followed to create 'n' number of nodes:

- Get the new node using getnode().

```
newnode = getnode();
```

- If the list is empty, then do the following
  ```
  start = newnode;
  newnode -> left = start;
  newnode ->right = start;
  ```
- If the list is not empty, follow the steps given below:
  ```
  newnode -> left =  start -> left;
  newnode -> right = start;
  start -> left->right = newnode;
  start -> left = newnode;
  ```
- Repeat the above steps 'n' times.

The function cdll_createlist(), is used to create 'n' number of nodes:

Inserting a node at the beginning:
The following steps are to be followed to insert a new node at the beginning of the list:

- Get the new node using getnode().
  ```
  newnode=getnode();
  ```
- If the list is empty, then
  ```
  start = newnode;
  newnode -> left = start;
  newnode -> right = start;
  ```
- If the list is not empty, follow the steps given below:
  ```
  newnode -> left = start -> left;
  newnode -> right = start;
  start -> left -> right = newnode;
  start -> left = newnode;
  start = newnode;
  ```

The function cdll_insert_beg(), is used for inserting a node at the beginning. Figure 6.8.2 shows inserting a node into the circular double linked list at the beginning.

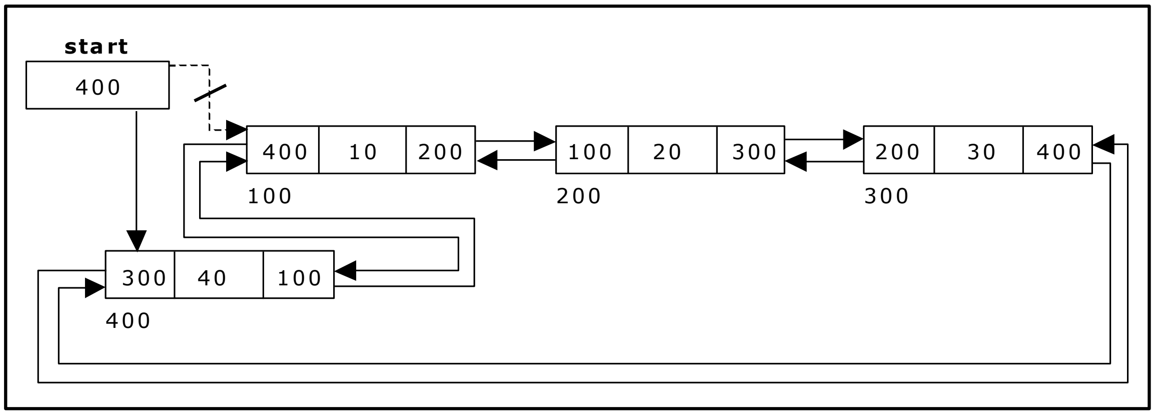


Figure 6.8.2. Inserting a node at the beginning

Inserting a node at the end:
The following steps are followed to insert a new node at the end of the list:

- Get the new node using getnode()
  ```
  newnode=getnode();
  ```
- If the list is empty, then
  ```
  start = newnode;
  newnode -> left = start;
  ```

newnode -> right = start;

- If the list is not empty follow the steps given below:

  newnode -> left = start -> left;
  newnode -> right = start;
  start -> left -> right = newnode;
  start -> left = newnode;

The function cdll_insert_end(), is used for inserting a node at the end. Figure 6.8.3 shows inserting a node into the circular linked list at the end.

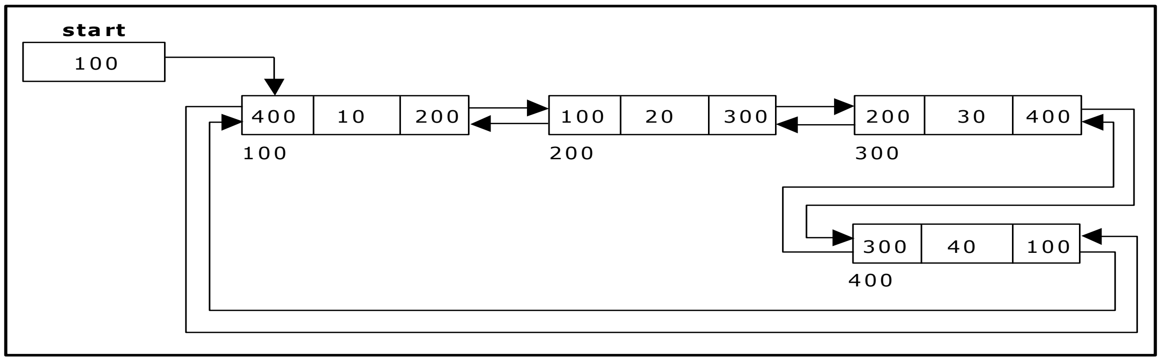


Figure 6.8.3. Inserting a node at the end

Inserting a node at an intermediate position:

The following steps are followed, to insert a new node in an intermediate position in the list:

- Get the new node using getnode().

  newnode=getnode();
- Ensure that the specified position is in between first node and last node. If not, specified position is invalid. This is done by countnode() function.

- Store the starting address (which is in start pointer) in temp and prev pointers. Then traverse the temp pointer upto the specified position followed by prev pointer.
- After reaching the specified position, follow the steps given below:

  newnode -> left = temp;
  newnode -> right = temp -> right;
  temp -> right -> left = newnode;
  temp -> right = newnode;
  nodectr++;

The function cdll_insert_mid(), is used for inserting a node in the intermediate position. Figure 6.8.4 shows inserting a node into the circular double linked list at a specified intermediate position other than beginning and end.

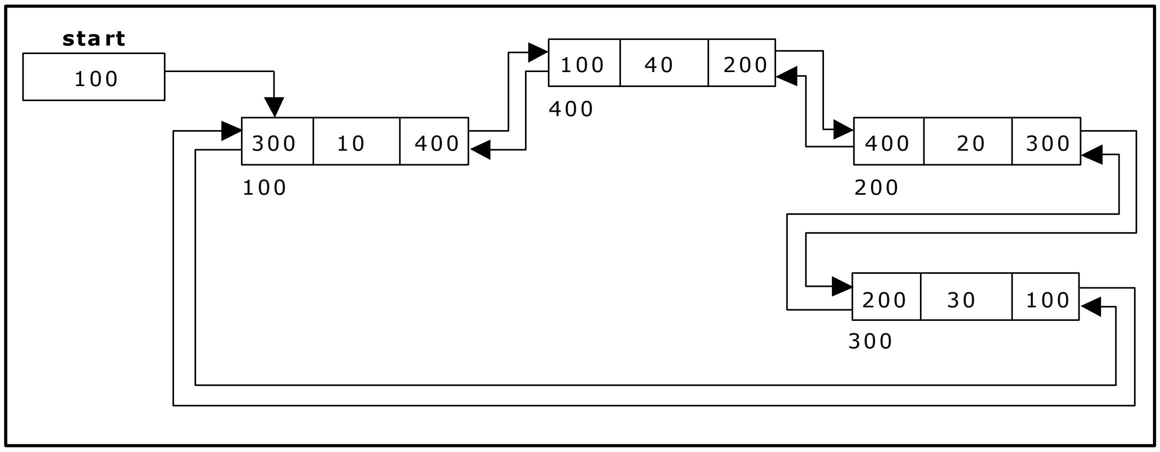


Figure 6.8.4. Inserting a node at an intermediate position

Deleting a node at the beginning:

The following steps are followed, to delete a node at the beginning of the list:

- If list is empty then display 'Empty List' message.
- If the list is not empty, follow the steps given below:

```
temp = start;
start = start -> right;
temp -> left -> right = start;
start -> left = temp -> left;
```

The function cdll_delete_beg(), is used for deleting the first node in the list. Figure 6.8.5 shows deleting a node at the beginning of a circular double linked list.

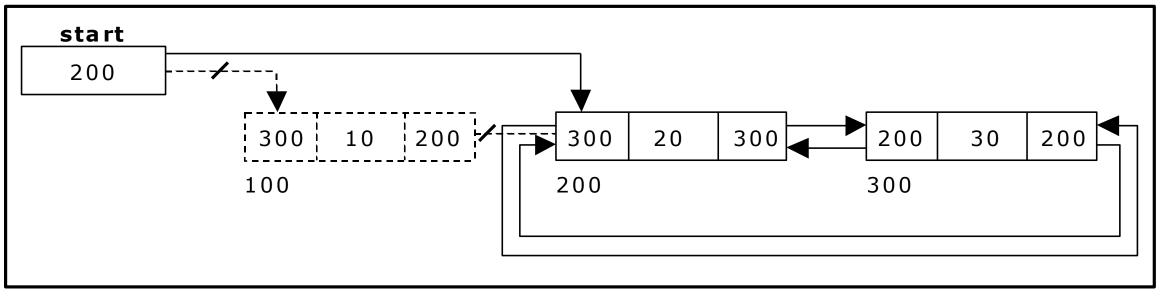


Figure 6.8.5. Deleting a node at beginning

Deleting a node at the end:

The following steps are followed to delete a node at the end of the list:

- If list is empty then display 'Empty List' message
- If the list is not empty, follow the steps given below:

```
temp = start;
while(temp -> right != start)
{
        temp = temp -> right;
}
temp -> left -> right = temp -> right;
temp -> right -> left = temp -> left;
```

The function cdll_delete_last(), is used for deleting the last node in the list. Figure 6.8.6 shows deleting a node at the end of a circular double linked list.

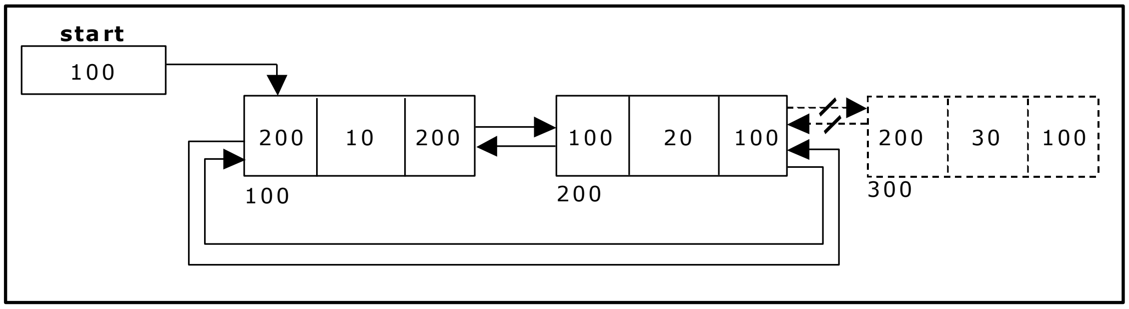


Figure 6.8.6. Deleting a node at the end

Deleting a node at Intermediate position:

The following steps are followed, to delete a node from an intermediate position in the list (List must contain more than two node).

- If list is empty then display 'Empty List' message.
- If the list is not empty, follow the steps given below:
  - Get the position of the node to delete.
  - Ensure that the specified position is in between first node and last node. If not, specified position is invalid.
  - Then perform the following steps:

```
if(pos > 1 && pos < nodectr)
{

        temp = start;
        i = 1;
        while(i < pos)
        {
                temp = temp -> right ;
                i++;
        }
        temp -> right -> left = temp -> left;
        temp -> left -> right = temp -> right;
        free(temp);
        printf("\n node deleted..");
        nodectr--;
}
```

The function cdll_delete_mid(), is used for deleting the intermediate node in the list.

Figure 6.8.7 shows deleting a node at a specified intermediate position other than beginning and end from a circular double linked list.

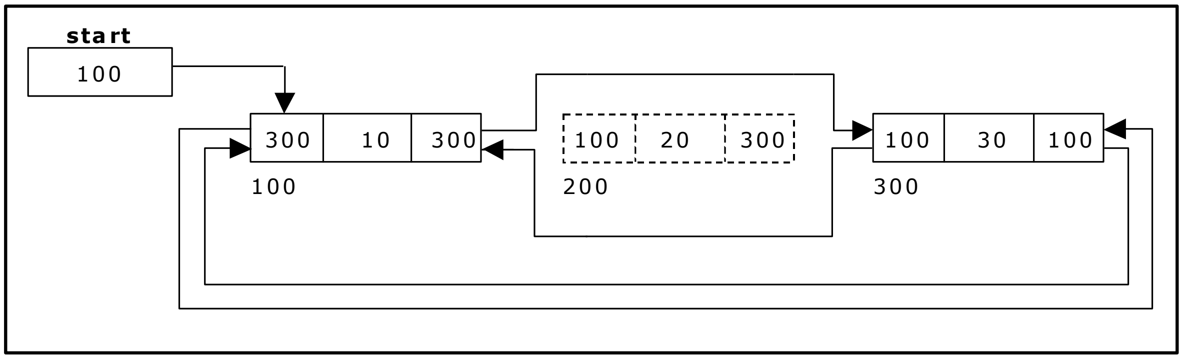


Figure 6.8.7. Deleting a node at an intermediate position

Traversing a circular double linked list from left to right:

The following steps are followed, to traverse a list from left to right:

- If list is empty then display 'Empty List' message.
- If the list is not empty, follow the steps given below:

```
temp = start;
Print  temp -> data;
temp = temp -> right;
while(temp != start)
{
        print  temp -> data;
        temp = temp -> right;
}
```

The function cdll_display_left _right(), is used for traversing from left to right.

Traversing a circular double linked list from right to left:
The following steps are followed, to traverse a list from right to left:

- If list is empty then display 'Empty List' message.
- If the list is not empty, follow the steps given below:

```
temp = start;
do
{
        temp = temp -> left;
        print temp -> data;
} while(temp != start);
```

The function cdll_display_right_left(), is used for traversing from right to left.

Atomic Linked List,

An atomic data type contains only the data items and not the pointers. Thus, for a list of data items several atomic type nodes may exist, each with a single data item corresponding to one of the legal data types. Their list is maintained using a list node which contains pointers to these atomic nodes and a type indicator indicating the type of atomic node to which it points. Whenever a list node is inserted in a list, its address is stored in the next free element of the list of pointers.

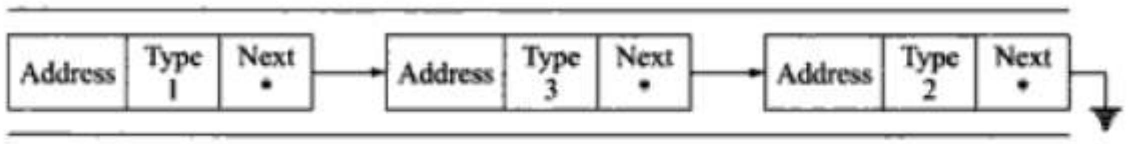


Fig. 4.13

Linked List in Arrays

Linked lists can be implemented without using pointers. For example, consider an ordered list of integers given by L = (10, –5, 0, 99). This list can be stored in an array, say "Ele". The concept of link can be implemented by using another array, "Next". The ith element of "L" is guaranteed to be stored in the ith index of an array "Ele".

The node in a linked list contains two parts—"**data**" and "**next**". These two parts of node are split and stored in two arrays "**Ele**" and "**Next**". If **Ele[i]** represents the data part of the node then **Next[i]** denotes the next part of that node. In this case the actual physical address is not denoted by next. Rather Next[i] is an integer and if Next[i] is j then the node next to the one represented by ith index of "Ele" and the "Next" is the node represented by jth index of "Ele" and "Next". If Next[i] = –1 then, the node under consideration is assumed to be the last node. Initially these arrays are unused and so a variable "free" is set to 0. The "free" function keeps track of the available parts in the arrays.

Figure 4.14 illustrates the initial situation of arrays:

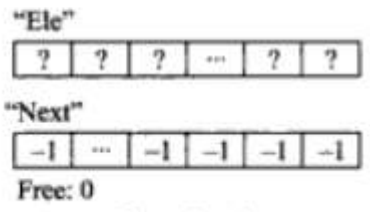


Fig. 4.14

When the first element of our list L is added to the array, **Ele[0]** contains 10, **Next[0]** remains –1, "**free**" becomes 1 and new variable "**start**" is required to remember which index in these arrays represent first node in the list. Figure 4.15 illustrates the addition of first node in the list.

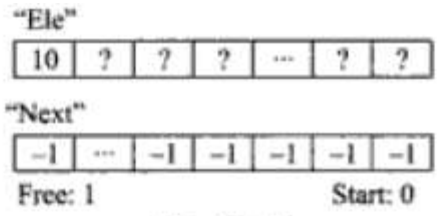


Fig. 4.15

When an attempt is made to add the second element of the list, the existing element is traversed from "start". As in our case, the value is 0. Then the Next[0] is –1, this is the end of the list. So the new node is added after this node. The index where the new node is to be stored in "Ele" and "Next" is found by inspecting "free". Next [0] is updated by current value of "free" and "free' is also incremented.

Figure 4.16 illustrates adding second element in the list.

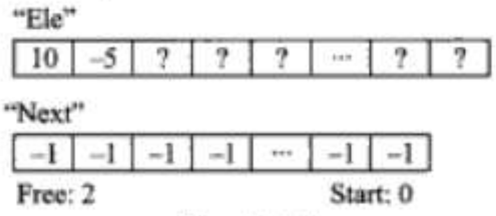


Fig. 4.16

Note that Ele[1] is set to –5, Next[0] is set to 1 and Next[1] is set to –1. Another addition will need to traverse the list from "Start", start = 0. As Next[0] = 1 the next node can be found in index 1 of the array "Ele" and "Next". The Next[1] is found, the value is –1, i.e. the node is the last node. As before, Next[1] is set to the current value of "free". Ele[free] is set to –1, and "free" is incremented. Figure 4.17 and 4.18 illustrate the addition of third and fourth nodes to the list.

"Ele"

10	−5	0	?	?	···	?

"Next"

1	2	−1	−1	···	−1	−1	−1	−1

Free: 3 Start: 0

Fig. 4.17 *Adding Third Node to the List*

"Ele"

10	−5	0	99	?	···	?	?	?

"Next"

1	2	3	−1	···	−1	−1	−1

Free: 4 Start: 0

Fig. 4.18 *Adding Fourth Node to the List*

Now if we want to delete the second node in the linked list, the next field of the first node is to be changed, in respect of the next field of second node. The array will be shown as given in Fig. 4.19.

"Ele"

10	?	0	99	?	···	?	?

"Next"

2	?	3	−1	?	···	−1	−1

Free: 4 Start: 0

Fig. 4.19 *Deletion of the Second Node from the List*

Therefore, it can be seen that no shifting of elements is done. Only the "next" array is updated. Say, if we want to again **add** an element with value 56 to the list after deletion (Fig. 4.20).

"Ele"

10	?	0	99	56	···	?	?	?

"Next"

2	?	3	4	−1	···	−1	−1	−1

Free: 5 Start: 0

Fig. 4.20 *Adding Another Node to the List*

It can be observed that the unused node 2 (i.e. the element with index 1 in the arrays "Ele" and "Next") could not be reused. The variable "free" keeps on increasing without paying any attention to the unused nodes. Figure 4.21 illustrates the deletion of the first node in the list. The value of the "Next" array with index "start" has to be made the new value of "start". In this example start = 0 and Next[0] = 2 before deletion, so the new value of "start" become 2 and Next [0] becomes undefined after deletion.

"Ele"

?	?	0	99	56	···	?	?	?

"Next"

?	?	3	4	−1	···	−1	−1	−1

Free: 5 Start: 0

Fig. 4.21