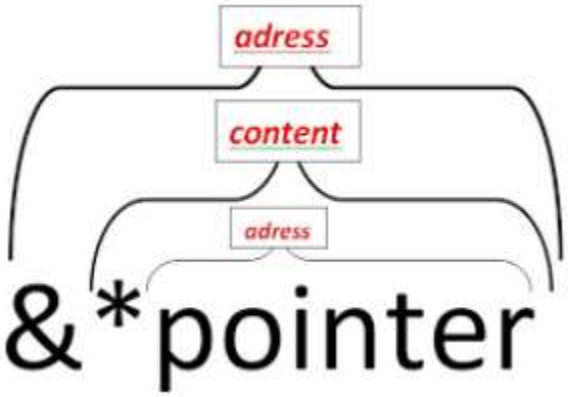
## 3.1 Pointer

Pointer is a derived data type.

A pointer is a variable which contains the memory address of another variable.

A pointer is essentially a simple variable which holds a memory address that points to a value, instead of holding the actual value itself.




For example, an integer variable 'a' holds an integer value, however an integer pointer '*ptr' holds the address of a integer variable.

We can have a pointer to any variable type.

Pointers are used in C program to access the memory and manipulate the address.

A Pointer is used to allocate memory dynamically.

The unary or monadic operator & gives the ``address of a variable''.

The indirection or dereference operator * gives the ``contents of an object pointed to by a pointer''.

## Advantages

- Pointers reduce the length and complexity of a program.
- Pointers makes possible to return more than one value from the function.
- Pointers increase execution speed.
- Pointers reduce the length and complexity of a program.
- The use of a pointer array of character strings results in saving of data storage space in memory.
- Pointers allows passing of arrays and strings to functions more efficient.
- Pointers allow us to use dynamic memory allocation.
- Pointers allow us to resize the data structure whenever needed.
- Pointers save memory.

## Disadvantages

- Pointers variables are slower than normal variables.
- If sufficient memory is not available during runtime for the storage of pointers, then the program may crash
- Pointers always require Free Memory for Dynamically Allocated Memory.
- Pointers are prone to memory leaks.

Syntax#

Datatype *Pointer variable name

Example#

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

The indirection operator (*) Asterisk tells the compiler that the variable named ptr is a pointer variable. The datatype tells the compiler what type the pointer points to.

## 3.1.1 Size of pointer

Pointer always allocates or occupies 2 bytes.

```
#include<stdio.h>
#include<conio.h>
main()
{

int *ptra;
float *ptrf;
double *ptrd;
char *ptrc;

clrscr();

printf("\nsizeof(ptra) = %d",sizeof(ptra));
printf("\nsizeof(ptrf) = %d",sizeof(ptrf));
printf("\nsizeof(ptrd) = %d",sizeof(ptrd));
printf("\nsizeof(ptrc) = %d",sizeof(ptrc));


getch();
}
```

**Ouput#**

```
sizeof(ptra) = 2
sizeof(ptrf) = 2
sizeof(ptrd) = 2
sizeof(ptrc) = 2
```

## 3.1.2 Address operator

Once we declare a pointer variable then we must point it to something. We can do this by assigning the pointer to the address of the variable we want to point as in the following example. Without pointing there is no meaning to declare a pointer.

Example#

```
int *ptr;
int a=20;
ptr=&a;
```


As we know every variable which has stored values in the memory, has a memory address.

Assume 'a' variable has address 65524 and value stored in this address is 20 and 2 bytes memory is allocated to it because int is a 2 byte in size.

ptr = &a; statement assigns address of 'a' variable to 'ptr' pointer variable.

Assume address of 'ptr' is 5000 and now 'ptr' stored 65524 as value in it. 'ptr' is now pointing to memory address of 'a' variable.

Now to access value of 'a' variable, there are two ways:

1. Use 'a' variable directly.
2. Use pointer variable 'ptr' to access 'a' indirectly.

To access variable 'a' directly we know it very well. For example,

printf("%d", a); prints value of a.

To access variable 'a' using pointer variable, use * sign before pointer variable. For example,

printf("%d", *ptr);

Print value of 'a' variable indirectly.

There are two operators used with the pointer:

1. * (asterisk) stands for "contains at".
2. &(ampersand) stands for "address of"

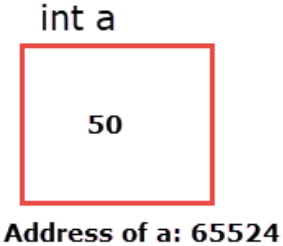
ptr has address of variable 'a' which is 65524, write *ptr as *(65524) and now apply the meaning of *, it makes "contains at 65524" which is value of 'a', 20.

%u control string is used to print the address of the variable.

### Let's understand how pointer stores the value once again with example and image

Every variable associated with three things,

- A variable name
- A variable's memory address
- A variable's value



In above image, we see that the computer has selected memory location 65524 as the place to store the value 50. But it is not necessary that the location number 65524 is fixed, some other time the computer may choose a different location for storing the value 50.

Example# Variable and its address

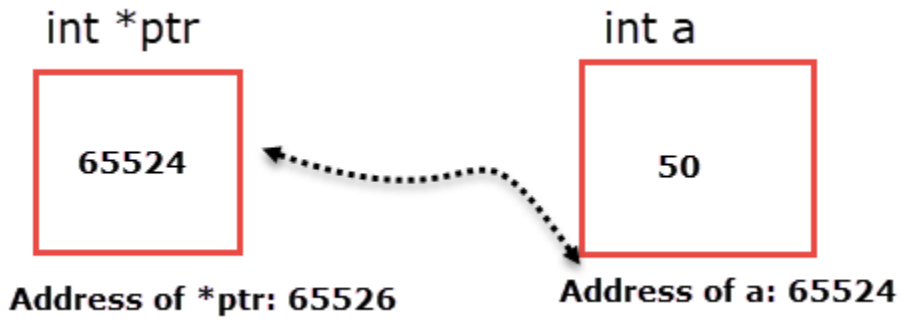
```
#include<stdio.h>
#include<conio.h>

main()
{
int A=24;
clrscr();
        printf("\n\nA = %d Address of A = %u",A,&A);
getch();
}
```

**Output#**
A = 24  Address of A = 65524

In the case of pointer, we initialize the pointer with the address of another variable's address.

In our below example, we initialize pointer '*ptr' to address of variable 'a'.




```
int  *ptr;
ptr=&a;
```

Example# Pointer and its pointing values

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

main()
{
int *ptr;
int a;
```

```
clrscr();

a=45;
ptr=&no;
printf ("\n A is %d Address of &A is %u", a, &a);
printf ("\n *ptr is %d Address of &Ptr is %u  Value of ptr = %u", *ptr,&ptr,ptr);

getch();
}
```

**Output#**

```
A is 45 Address of &A is 65524
*ptr is 45 Address of 65522 is %u  Value of ptr = 65524
```

Example# Change in pointer value

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

main()
{
int no,*ptr;
clrscr();
        no=45;
        printf("\n\nBefore No - %d *Ptr - %d",no,*ptr);
        ptr=&no;
        *ptr=200;
        printf("\n\nAfter pointer No - %d  *Ptr - %d",no,*ptr);
getch();
}
```

**Output#**

```
Before No – 45  *Ptr – 45
After pointer No – 45 *ptr - 45
```

## 3.2 Pointer Arithmetic

Pointer Arithmetic starts with increment and decrement operators.

Example#

```
int *ptr;
ptr++;
```

In the above example, pointer *ptr will be of 2 bytes. And when we increment it, it will increment by 2 bytes because int is also of 2 bytes.

Example#

```
float*ptr;
ptr++;
```

In the above example, pointer *ptrwill be of 2 bytes. But now, when we increment it, it will increment by 4 bytes because float is of 4 bytes.

Let's see more example

| Data Type | Initial Address | Operation | Address after Operations | Required Bytes |
|---|---|---|---|---|
| int | 65524 | ++ | 65526 | 2 |
| int | 65524 | - - | 65522 | 2 |
| char | 65524 | ++ | 65525 | 1 |
| char | 65524 | - - | 65523 | 1 |
| float | 65524 | ++ | 65528 | 4 |
| float | 65524 | - - | 65520 | 4 |
| long | 65524 | ++ | 65528 | 4 |
| long | 65524 | - - | 65520 | 4 |

According to the rule of pointers, we can add or subtract address and integer number to get a valid address. We can even subtract two addresses but we cannot add two addresses

Example#

| Expression | Example | Result |
|---|---|---|
| Address + Number | 65524+10 | Address |
| Address – Number | 65524-10 | Address |

| Expression | Example | Result |
|---|---|---|
| Address – Address | 65524-1920 | Number |
| Address + Address | 65524+65528 | Illegal |

Example#Pointer Increment, Difference and comparison

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

void main(){

int *ptr=(int *)1000;
int *ptr1=(int *)2000;
int *ptr2=(int *)3000;

clrscr();

ptr=ptr+1;
printf("New Value of ptr : %u",ptr);

printf("\nDifference : %d",ptr2-ptr1);

if(ptr2 > ptr1)
  printf("\nPtr2 is far from ptr1");
else
  printf("\nPtr1 is far from ptr2");

getch();
}
```

**Ouptut#**

```
New Value of ptr : 1002
Difference : 500
Ptr2 is far from ptr1
```

In above example,

We Incremented a pointer to an integer data caused its value to be incremented by 2.Ptr2-Ptr1 will gives us number of integer numbers that can be stored and at last we have compared two pointers of different data types.

Example# Char Array with pointer and incremented by 1

```
#include<stdio.h>
#include<conio.h>
main()
{
```

```
 char s[] = "I love C Language";
 char* ptr = s;

 clrscr();

  //there is no need of initialization in for loop
  for(; *ptr != '\0'; ptr++) {
    printf("%c", *ptr);
  }

getch();
}
```

**Output#**

I love C Language

Example# int Array with pointer and incremented by 2

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

main()
{
 int a[] ={11,22,33,44,55};
 int *ptr = a;
 int i;
 clrscr();

  printf("\nPos\tValue\tAddress");
  printf("\n====================");

   for(i=0; i<5; i++) {
     printf("\n%d\t%d\t%u",i, *ptr,ptr);
     ptr++;
   }

getch();
}
```

**Output#**

```
Pos    Value  Address
====================
0      11     65516
1      22     65518
2      33     65520
3      44     65522
4      55     65524
```

One pointer variable can be assigned to another pointer variable but both must be of the same type.
For example#

```
int  x, *y, *z;
*y=&x;
z=y;
```

Now z and y pointers point to the same memory location.

Example# Swapping of two values using pointer

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

void main()
{
        int a, b, t, *p1, *p2;
        clrscr();
        printf("\n Enter two numbers :");
        scanf("%d  %d", &a, &b);
        printf("\n Before exchange a=%d  b=%d", a, b);
        p1=&a;          /* address of a is assigned to p1 */
        p2=&b;          /* address of b is assigned to p2 */
t=*p1;
        *p1= *p2;
        *p2=t;
        printf("\n After exchange a=%d  b=%d", a, b);
        getch();
}
```

**Output#**

```
Enter two numbers: 5  7
Before exchange a=5  b=7
After exchange a=7  b=5
```

## 3.3 Void pointer

A void pointer is pointer which has no specified data type. The keyword 'void' precedes the pointer variable, because the data type is not specific. It is also known as a generic pointer. The void pointer can be pointed to any type of variable with proper type casting. Two bytes of memory is assigned to void pointer when declared.

A void pointer is generally used as function parameters, when the parameter or return type is unknown.

Example# void pointer

```
#include<stdio.h>
void main()
{
        int rollno=12;
        char name[10]="nirav";
        /* assign address of rollno to ptr */
        void *ptr=&rollno;
        /* before display , typecasting the pointer varaible */
        printf("\n %d", *(int *)ptr);
        /* assign address of name to ptr */
        (char *)ptr=name;
        /* before display , typecasting the pointer varaible */
        printf("\n %s", (char *)name);

}
```

**Output#**

```
12
Nirav
```

Description#

printf("\n %d", *(int *)ptr);

Here value pointed by ptr will be type cast to integer before being display.

printf("\n %s", (char *)name);

Here value pointed by ptr will be type cast to char before being display.

## 3.4 Arrays and Pointer

Array itself act as a pointer because when we declare array, compiler allocates the address (base address) of the first element and memory to all the elements and stores the values in contiguous memory location. Array name acts as a constant pointer and it points to the first element.

Example# Pointer with 1 D array

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int n[3];
        int i;
        clrscr();
        for(i=0; i<3; i++)
        {
                printf("\n Enter array value for element n[%d] :" , i);
                scanf("%d", &n[i]);
        }
        printf(" \n value     address");
        for(i=0; i<3; i++)
        {
                printf(" \n %d       %u", *(n+i), &n[i]);
        }
        getch();
}
```

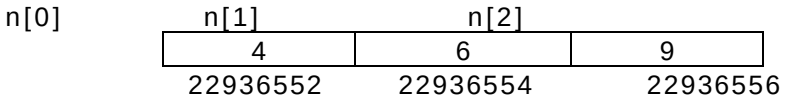
**Output:**

```
Enter array value for element n[0] : 4
Enter array value for element n[1] : 6
Enter array value for element n[2] : 9
value  address
4       2293552
6       2293554
9       2293556
```

Description#

*(n+i) is a pointer concept which allow printing the value of array n. staring address of first element of the array is 2293552

In the first iteration,
*(22936552+ 0) is interpreted as contains at 22936552 which is 4,
In the second iteration,
*(22936552+ 1) is interpreted as contains at 22936554 which is 6,
In the last iteration,
*(22936552+ 2) is interpreted as contains at 22936556 which is 9,
Pictorial representation of the array is:




**Example# Access pointer using array**

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

void main()
{
        int n[3], *ptr;
        int i;
        clrscr();
        for(i=0; i<3; i++)
        {
                printf("\n Enter array value for element n[%d] :" , i);
                scanf("%d", &n[i]);
        }
        ptr = n;    /*  address of first element is assigned to ptr */
        /* or ptr = &n[0];  */
        printf(" \n element    value     address");
        for(i=0; i<3; i++)
        {
                printf(" \n n[%d]    %d    %d    %u", i, ptr[i], *(ptr+i), &n[i]);
        }
        getch();
}
```

**Output#**

```
Enter array value for element n[0] : 4
Enter array value for element n[1] : 6
Enter array value for element n[2] : 9
Element  value      address
n[0]     4   4    2293552
n[1]     6   6     2293554
n[2]     9   9     2293556
```

Description#

ptr = n assigns address or first element to ptr and it is same as ptr = &n[0].
Now ptr points to the array n. We can access each elements just by incrementing the ptr by 1.

ptr[i] is referred to the element value at ith position and it is same as *(ptr+i).
Address of each element is print by &n[i].

Once pointer variable points to the array, address of the next element is obtained by incrementing the pointer ptr++ and contains (value) of that address is obtained by *ptr.
Consider the above revised example.

## 3.5 Accessing Two-Dimensional Array using Pointer

Using pointer, two dimensional arrays can be also accessed. Elements of the 2-D numeric array are represented by row and column number.

For example#

int arr[3][2];

Array elements can also be represented using pointer concept.

To access 1st element of 2nd row we write arr[0][1] and is equivalent to *(*(arr+0)+1) or *(arr[0]+1)

It row is specified by i index and column by j then we can write as arr[i][j] or *(*(arr + i) +j) or *(arr[i]+j)

Example# 2D array using pointer

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int arr[3][2];
        int i,j;
        clrscr();
        for(i=0; i<3; i++)
        {
                for(j=0; j<2; j++)
                {
                    printf("\n Enter array value for element arr[%d][%d]" , i,j);
                    scanf("%d", &arr[i][j]);
                }
        }
        printf("\n the array contains \n");
        for(i=0; i<3; i++)
        {
                for(j=0; j<2; j++)
                {
                   printf(" \n %d  %d   %d", arr[i][j], *(*(arr+i)+j) , *(arr[i]+j));
                }
        }
getch();
}
```

**Output#**

```
Enter array value for element arr[0][0] 4
Enter array value for element arr[0][1] 5
Enter array value for element arr[1][0] 6
Enter array value for element arr[1][1] 7
Enter array value for element arr[2][0] 8
```

```
Enter array value for element arr[2][1] 9
4  4  4
5  5  5
6  6  6
7  7  7
8  8  8
9  9  9
```

## 3.6 Pointer with String

String is a collection of characters. It can also be accessed using pointer. String array itself act as a pointer because when we declare an array, compiler allocate the address (base address) of the first character in the array and memory to all the elements and store values in contiguous memory location. Array name is acts as a constant pointer and it points to the first element.

Example# Access String using pointer

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

void main()
{
        char name[15];
        char *ptr;
        clrscr();
        ptr=name;
        puts("Enter city name ");
        scanf("%s" , &name);
        printf("\n %s" , ptr);
        getch();
}
```

**Output #**

```
Enter city name : Surat
Surat
```

Example# To Access string using pointer , print each character in different position

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

void main()
{
        char name[10];
        char *ptr;
        clrscr();
        ptr=name;    /* address of 1st character is assigned to ptr */
/*    or ptr=&name[0];   */
        puts("Enter city name ");
        scanf("%s" , &name);
        for(i=0 ; ptr[i]!='\0' ;i++)
        {
                printf("\n%c  %c   %c   %u", *(name+i), ptr[i], *(ptr+i), ptr[i]);
        }
        getch();
```

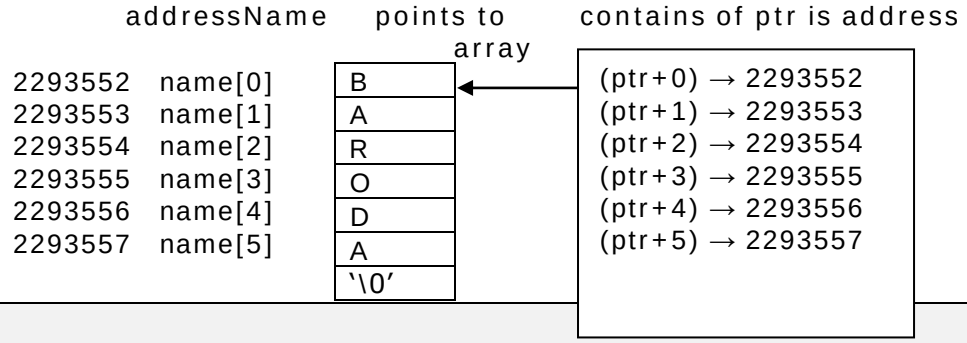
```
}
```

**Output#**

```
Enter city name : baroda
b b b   2293552
a a a   2293553
r r r   2293554
o o o   2293555
d d d   2293556
a a a   2293557
```

Description#

ptr pointer points to name array and each character of the array is referenced by incrementing the ptr pointer.
When i=0,
ptr+i points the first character of the string array and *(ptr+i) gives the contains at address pointed by ptr pointer.
For i=1,
ptr points to next character in the array name and *(ptr+i) gives the contains at address pointed by ptr pointer.



## 3.9 Array of pointers

Some time we need array of pointers.

Syntax#

Datatype *pointer Name[ArraySize];

Example#

int *ptr[10];

This declares ptr as an array of 10 integer pointers. Thus, each element in ptr, now holds a pointer to an int value.

Example# Integer pointer array

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

main()
{
 int a[] ={11,22,33,44,55};
 int *ptr[5];
 int i;
 clrscr();

  for(i=0;i<5;i++) //here we assign address to each position of array
  {
   ptr[i]=&a[i];
  }

  printf("\nPos\tValue\tAddress");
  printf("\n====================");

   for(i=0; i<5; i++) {
     printf("\n%d\t%d\t%u",i, *ptr[i],ptr[i]);
   }

getch();
}
```

**Output#**

```
Pos   Value  Address
====================
0     11     65516
1     22     65518
2     33     65520
3     44     65522
```

```
4     55     65524
```

Example# Pointer Character array

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

main()
{
 char *names[4]={"Jayul","Bhavin","Manav","Rehan"};
 int i;
 clrscr();

 for(i=0;i<4;i++)
 {
  printf("\n%s",names[i]);
 }

getch();
}
```

**Output#**

```
Jayul
Bhavin
Manav
Rehan
```

## 3.10 Pointers and Functions

Like normal variable passing, C programming allows passing of a pointer to a function.

When a pointer is passed as an argument to a function, address of the memory location is passed instead of the value.

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.

This is because pointer stores the location of the memory, and not the value.

Example# Passing pointer

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

void salaryIncrement(int  *sal, int inc)
{
   *sal = *sal+inc;
}

void main()
{
   int salary=0, inc=0;
   clrscr();
   printf("Enter current salary:");
   scanf("%d", &salary);
   printf("Enter Increment:");
   scanf("%d", &inc);
   salaryIncrement(&salary, inc);
   printf("After increment Final salary: %d", salary);
   getch();
}
```

**Output#**

```
Enter current salary:10000
Enter Increment:200
After increment Final salary: 10200
```

Example# Passing array to pointer argument

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

int getSum(int *arr, int size);
void setArray(int *arr,int size);
void printArray(int *arr,int size);
```

```
void main () {
  int arr[100];
  int n;
  int sum=0;
  clrscr();
  printf("\nEnter array limit =>");
  scanf("%d",&n);
  setArray(arr,n);
  printArray(arr,n);
  sum= getSum(arr,n) ;
  printf("\nSum of Values is: %d", sum);
  getch();
}

void setArray(int *arr, int size)
{
int i;

        printf("\nEnter %d values =>\n",size);
        for(i=0;i<size;i++)
        {
        scanf("%d",&arr[i]);
        }
}

void printArray(int *arr, int size)
{
int i;
        printf("\nValues\n");
        for(i=0;i<size;i++)
        {
        printf("%d ",arr[i]);
        }
}

int getSum(int *arr, int size) {
  int  i, sum = 0;

  for (i = 0; i < size; ++i) {
    sum += arr[i];
  }

  return sum;
}
```

**Output#**

```
Enter array limit =>5
```

```
Enter 5 values =>
11 22 33 44 55

Values
11 22 33 44 55
Sum of Values is: 165
```

Example# Swapping using Function

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

void swap(int *x,int *y)
{
 int z;
 z=*x;
 *x=*y;
 *y=z;
}
main()
{
int a,b;

clrscr();

printf("\nEnter a and b =>");
scanf("%d %d",&a,&b);

printf("\nBefore swap A = %d and B = %d",a,b);
swap(&a,&b);
printf("\nAfter swap A = %d and B = %d",a,b);

getch();
}
```

**Output#**

```
Enter a and b =>22 3

Before swap A = 22 and B = 3
After swap A = 3 and B = 22
```

Description#

In above example, we need to pass the address of both variables to the function. We can say Parameter passing as Reference.

Address of first variable 'a' will be collected in *x pointer variable and second variable 'b' will be collected in *y pointer variable.

## Difference between Call by Value and Call by Reference

| Call By Value | Call By Reference |
|---|---|
| Using call by value the called function creates a new set of variables and copies the values of arguments into them. | Using call by reference, the formal arguments in the called function become aliases to the actual arguments in the calling function |
| Original value is not modified. | Original value is modified. |
| The function does not have access to the actual variable in the calling program and can only work on the copies of values | This means that when the function is working with its own arguments, it is actually working on the original data. |
| Actual and formal arguments will be created in different memory location | Actual and formal arguments will be created in same memory location |
| Example#<br>add(a, b); | Example#<br>Swap(&a, &b); |

## 3.11 Dynamic Memory Allocation

When simple array is declared, array size is also specified.

Example#
int num[60];

Compiler allocates the 120 bytes memory during program compilation. This is called static memory allocation. When we actually run the program, we might provide 10 numbers only. This would result in wastage of 100 bytes memory. Sometime you also need to gives more than 60 numbers.

We may need to decrease or increase the array size but we cannot do this during program execution.

Hence to allocate the memory during program execution or dynamically, C provides library functions malloc() and alloc().

Using these functions, we can allocate the memory dynamically for pointer type's variables only. free() function is used to de-allocate the memory. These functions reside in "alloc.h" header file.

### 3.11.1 malloc() function

It allocates the requested memory and returns a pointer to it.

This function reserves a block of memory of given size and returns a pointer of type void.

Syntax#
ptr-variable = (data-type *) malloc(size);
size is the size of data-type in bytes.

Example#
int *n = (int *) malloc(2);

Integer is a 2 byte in size so two byte is allocated to n pointer variable.
char *name = (char *) malloc(10);

### 3.11.2 calloc() function

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

Syntax#
ptr-variable = (data-type *) calloc(n, size);
n is the numbers of byte and size is a size of data type.

Example#
int *n = (int *) calloc(2, sizeof(int));
*name = (char *) calloc(10, sizeof(char));

## Difference between Calloc VS Malloc

| calloc() | malloc() |
|---|---|
| calloc() initializes the allocated memory with 0 value. | malloc() initializes the allocated memory with garbage values. |
| Syntax#<br>(cast_type *)calloc(blocks , size_of_block); | Syntax#<br>(cast_type *)malloc(Size_in_bytes); |

## 3.11.3 free() function

Used to free the memory allocated to pointer variables.

After memory allocation using malloc() or calloc(), when the program finish it execution memory still remains allocated .We have to free the memory before end of the program using free() function.

Syntax#
free(ptr data-type);

Example#

```
int *ptr;
ptr = (int*) malloc(num * sizeof(int));
free(ptr);
```

## 3.11.4 realloc()

Change the size of previously allocated space.

Syntax#
ptr = realloc(ptr, newsize);

Example#

```
int *ptr;
int newsize;
ptr = (int*) malloc(20 * sizeof(int));

newsize=50;
ptr = realloc(ptr, newsize);
```

Example# Run time memory allocation to one Dimension integer array , demo of Calloc() and free() function

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

```
void main()
{
        int *arr=NULL ;
        int i, size;
        clrscr();
        printf("\n Enter array size :");
        scanf("%d", &size);
        arr=(int *) calloc(size,sizeof(int));
/* OR  //n=(int *) malloc(sizeof(int));   */
        if(arr==NULL)
        {
                printf("\n mem. allocation problem");
                exit(0);
        }
        for(i=0;i<size;i++)
        {
                printf("\n Enter value for arr[%d]:", i);
                scanf("%d",(arr+i));
        }
        for(i=0;i<size;i++)
        {
                printf(" %d  ",arr[i]);
        }
        free(arr);  /* to free the memory reserved by arr*/
        getch();
}
```

**Output#**

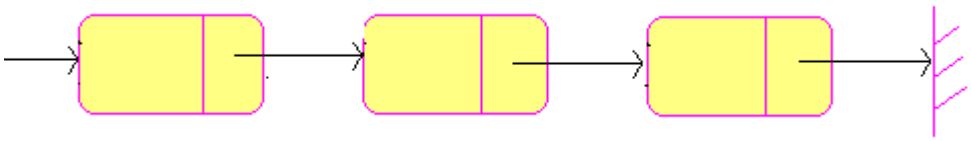
```
Enter array size: 5
Enter value for arr[0] :5
Enter value for arr[1] :15
Enter value for arr[2] :2
Enter value for arr[3] :4
Enter value for arr[4] :12
5  15  2  4  12
```

## 3.12 Drawbacks of Array

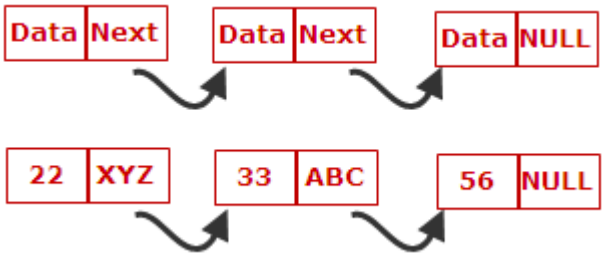
- It does not have dynamic structures.
- Resizing the array is a time-consuming operation.
- No simple method to insert additional elements or delete elements anywhere in the array.
- To remove an item, we have to move one by one and next item is shifted to that removed item.
- If inserting element on first position then all elements are move next.
- To access any element, we must know its index.

## 3.12.1 LinkedList

- Linked list is a data structure of linear type.
- A simple way to represent a linear list is to expand each node to contain a link or pointer to the next node. This representation is called a one-way chain or singly linked linear list.
- A series of structures which are connected with pointers can be used to implement the data structures. Each structure contains data with one or more pointers to neighboring structures.



- There are several variants of the linked list structure:

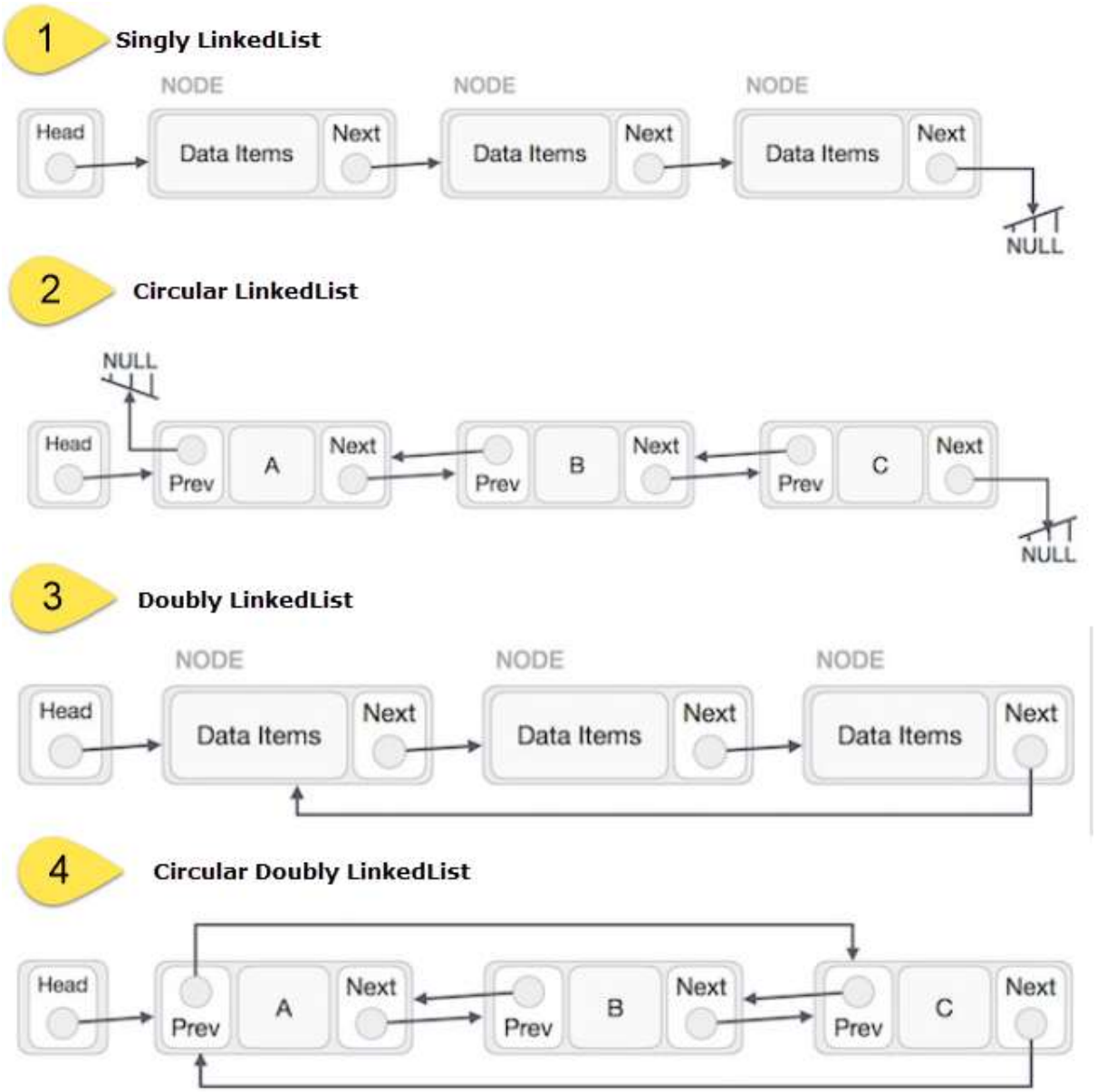



- Lined list is defined as an ordered collection of homogeneous data elements and each element in the list is called NODE.
- It is ordered collection of nodes.
- In a Linear list, each node contains 2 area data portion and a link or pointer to the next node.
- The data are contains the data of that location.
- The pointer contains the address of location where next information is stored.
- Each of which contains some data.
- Connected using pointers.
- Each node contains address of next node in list.
- Last node points to NULL and first node in list is called head.
- Last node in list is called tail.

## Applications of LinkedList

1. Polynomial representation, automatic polynomial manipulations are performed by linked list.
2. Addition and subtractions operations of polynomial are easily implemented using linked list.
3. Symbol table creation.
4. Multiple precision arithmetic and representation of sparse matrices.

## There are 4 types of LinkList



## Array VS LinkedList

| Array | LinkedList |
|---|---|
| An array is a static data structure | A link list is a dynamic data structure |
| It is a consistent set of a fixed number of data items. | It is an ordered set consisting of a variable number of data items. |
| Specified fixed size during declaration. | Dynamically , there is no need to specify size |
| Memory is allocated at design time. | Memory is allocated at run time. |
| Stored consecutively | Stored randomly |
| Access possilbe is by the array index or subscript. | Direct or randomly accessed. |
| For Insertion and deletion of element shifting is required. | Insertion and deletion Easier, fast and efficient and no need of shifting. |
| Memory requiredless | Memory required More |
| Only one data type | It can have more than one data types. |
| It is difficult to delete or insert items by rearranging the links. | It is easier to delete or insert items by rearranging the links. |

## Operation on LinkedList

- To create a linked list
- Traversing a linked list
- Insert New node at beginning (at first)
- Insert new node at end
- Insert new node at any location or in between the list
- Inserting a node in to an ordered linear list.
- Delete a first node (at beginning)
- Delete a last node (at end)
- Delete a node on basis of node number
- Copy of the list
- Searching element in linked list
- Merging operation of two linked list

Example#

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

struct node
{
int x;
struct node *next;
}*Head;

void create() //Create List
{
        struct node *q,*tmp;
        int y;
```

```
        printf("\nEnter Value =>");
        scanf("%d",&y);

        tmp=(struct node*)malloc(sizeof(tmp));
        tmp->x=y;
        tmp->next=NULL;

        if(Head==NULL)
        {
                Head=tmp;
                printf("\nAdded at first");
        }
        else
        {
                 q=Head;
                 while(q->next!=NULL)
                {
                                q=q->next;
                }

                 q->next=tmp;
                 printf("\nAdded at Last");
        }
}


void print() //print
{
struct node *q;

        if(Head==NULL)
        {
                printf("\nHead is null");
        }
        else
        {
                 q=Head;
                printf("\nValues are\n");
                while(q!=NULL)
                {
                        printf("%d-->",q->x);
                        q=q->next;
                 }

        }
}

void insf() //insert first
```

```
{
        struct node *tmp;
        int y;
        printf("\nEnter value =>");
        scanf("%d",&y);

        tmp=(struct node*)malloc(sizeof(tmp));
        tmp->x=y;

        tmp->next=Head;
        Head=tmp;

        printf("\nAdded at First");
}
```

**void insl() //insert first**

```
{

struct node *tmp,*q;
int y;
printf("\nEnter value =>");
scanf("%d",&y);

tmp=(struct node*)malloc(sizeof(tmp));
tmp->x=y;
tmp->next=NULL;
q=Head;
        while(q->next!=NULL)
        {
        q=q->next;
        }
        q->next=tmp;
        printf("\nAdded at Last");
}
```

**void delf() //delete first**

```
{
        struct node *tmp;
        tmp=Head;
        Head=Head->next;
        free(tmp);
        printf("\nHead deleted and New Head created");
}
```

**void dell() //delete last**

```
{
struct node *tmp,*q;
q=Head;
```

```
        while(q->next->next!=NULL)
        {
        q=q->next;
        }
        tmp=q->next;
        q->next=NULL;
        free(tmp);
        printf("\nLast element deleted");
}



main()
{
int ch;
clrscr();

Head=NULL;

while(1)
{
        printf("\n press 1 for Create ");
        printf("\n press 2 for Print ");
        printf("\n press 3 for Exit ");
        printf("\n press 4 for Insert first");
        printf("\n press 5 for Insert last");
        printf("\n press 6 for Delete first");
        printf("\n press 7 for Delete last");

        printf("\n Enter >");
        scanf("%d",&ch);

switch(ch)
{
case 1:
        create();
        break;

case 2:
        print();
        break;

case 3:
        exit(0);

case 4:
        insf();
        break;
```

```
case 5:
        insl();
        break;

case 6:
        delf();
        break;

case 7:
        dell();
        break;

default:
        printf("wrong operation");
}

getch();
}
getch();
}
```

**Output#**

```
press 1 for Create
press 2 for Print
press 3 for Exit
press 4 for Insert first
press 5 for Insert last
press 6 for Delete first
press 7 for Delete last

Enter >1

Enter Value =>11

Added at first

press 1 for Create.....menu

Enter >1

Enter Value =>12

Added at Last

press 1 for Create.....menu

Enter >1
```

```
Enter Value =>13

Added at Last

press 1 for Create......menu

Enter >1

Enter Value =>14

Added at Last

press 1 for Create......menu

Enter >1

Enter Value =>15

Added at Last


press 1 for Create.....

 Enter >1

Enter Value =>15

Added at Last

 press 1 for Create....menu

 Enter >2

Values are
11-->12-->13-->14-->15-->

 press 1 for Create....menu

Enter >4

Added at First

 press 1 for Create....menu
 Enter >5

Enter value =>999

Added at Last
```

```
 press 1 for Create....menu

 Enter >2

Values are
101-->11-->12-->13-->14-->15-->999-->

press 1 for Create....menu

Enter >6

Head deleted and New Head created

press 1 for Create....menu

Enter >2

Values are

11-->12-->13-->14-->15-->999-->

press 1 for Create....menu

Enter >7

Last element deleted

press 1 for Create....menu

Enter >2

Values are

11-->12-->13-->14-->15-->

press 1 for Create .....me
Enter >3
```