Queues

Definition

A queue is a linear list in which data can only be inserted at one end, called the rear, and deleted from the other end, called the front. These restrictions ensure that the data are processed through the queue in the order in which they are received. In other words, a queue is a first in - first out (FIFO) structure

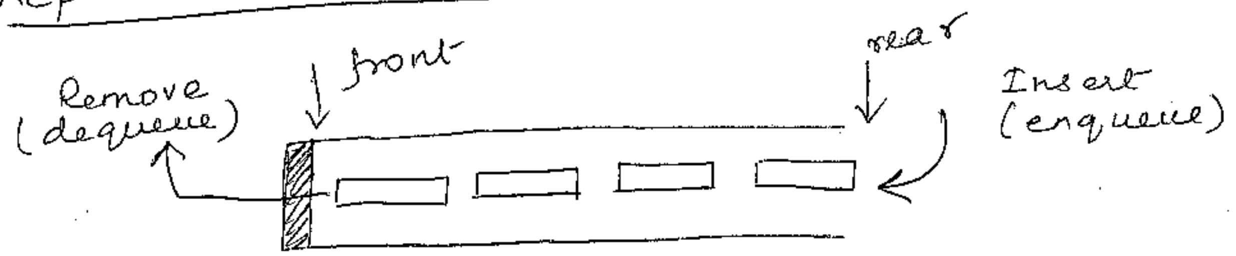
eg

* A line of people waiting for the bus at a bus station is a queue

* A list of calls put on hold to be answered by a telephone operator is queue

* A list of waiting jobs to be processed by a computer is a queue.

Representations of a queue



Computer queue

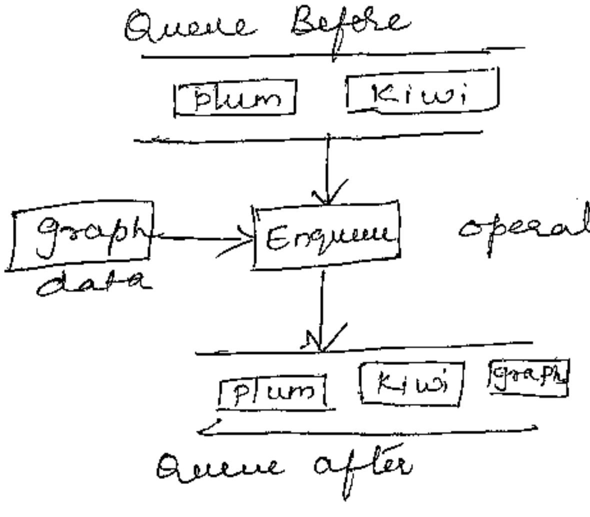
Operations

(1) Enqueue (x)

(2) Dequeue ( )

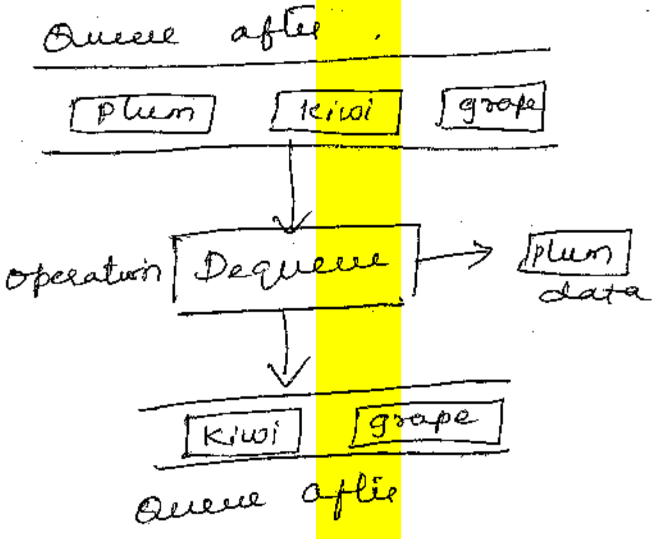
Enqueue (Enqueue inserts an element at the rear of the queue)

The queue insert operation is known as enqueue. After the data have been inserted into the queue, the new element becomes the rear. If there is not enough room for another element in the queue, the queue is in an overflow State.



Dequeue

The queue delete operation is known as dequeue. The data at the front of the queue are returned to the user and removed from the queue. If there are no data in the queue when a dequeue is attempted, the queue is in an underflow state.



Applications of Queue

① When a resource is shared among multiple consumers.

eg: CPU Scheduling, Disk Scheduling

② What data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes.

eg IO Buffers, pipes, file I/O etc.

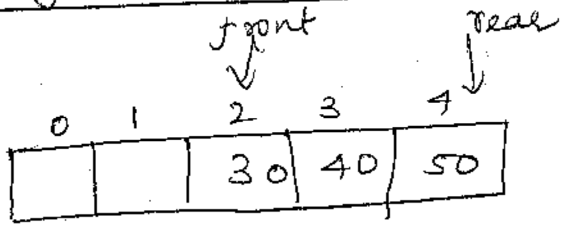
Implementation of queue using array

```
#include <stdio.h>
~~#define queuearr[~~
# define Max 10

int queuearr [Max];
int rear = -1
int front = -1;

void main()
{
    int choice;
    while(1)
    {
     printf("1. Insert \n");
     printf("2. Delete \n");
      printf("3. Display \n");
      printf("4. Quit \n");
      printf("Enter your choice :");
      scanf("%d", &choice);

      switch(choice)
      {
         case 1 : insert();
                  break;

         case 2 : delete();
                  break;
```

```
          case 3 : display ( );
                      break ;

           case 4:  exit ( )
           default:  printf ("wrong choice \n");

        }
     }
}


insert ( )
{
   int item ;
    if (rear == Max - 1)
    printf ("Queue overflow \n");
    else
    {
      if ( front == - 1)
      front = 0 ;
      printf ("Insert the element in queue : ");
      scanf ("%d", & item);
      rear = rear + 1;
      queuearr [rear] = item ;

    }
}
```

```
delete()
{
   if (front == -1 || front > rear)
   {
      printf ("Queue underflow\n");
      return;

   }
   else
   {
      printf ("Element deleted from queue is : %d\n",
                                          queuearr[front]);

      front = front + 1;
   }
}
display()
{
   int i;
   if (front == -1)
      printf ("Queue is empty\n");
   else
   {
      printf (" Queue is :\n");
      for(i= front; i <= rear; i++)
         printf ("%d", queuearr[i]);
         printf ("%d", queuearr[i]);
      printf ("\n");
   }
}
```

Disadvantage of Ordinary Queue



The above situation arises when 5 elements say 10, 20, 30, 40 and 50 are inserted and then deleted first two items 10 and 20. Now, if we try to insert an item we get the message "Queue Overflow".

In the above situation, rear insertion is denied even if space is available at the front end. This is because in our insert function we test whether before inserting an element, rear is equal to Queue_Size-1. If so, we say "Queue is full and can't insert". This is a disadvantage. This disadvantage can be overcome using 2 methods.

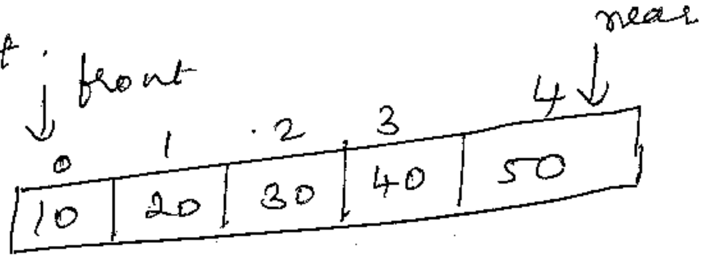
* Shift left method

(or)

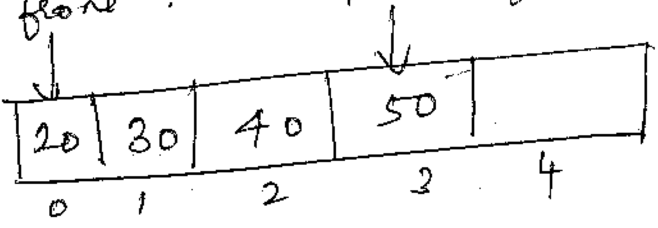
* Using circular representation of queue.

Shift left

After deleting the element from the front, Shift all remaining elements to the left.



After deleting 10, shift remaining elements to the left.



Note that each time an item is deleted, all the elements towards right are moved to left by one position and rear is decremented by 1. But, shifting the data is costly in terms of compute time if the data being stored is very Large.

So, this method is not recommended.

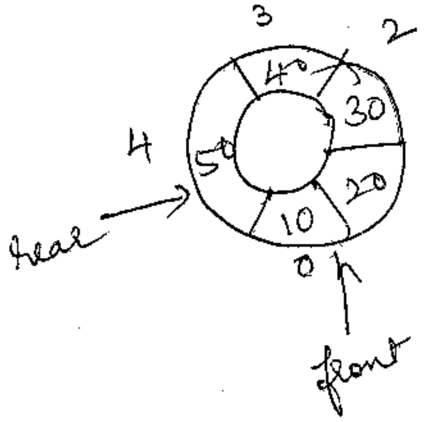
Circular Queue

In circular queue, the elements of a given queue can be stored efficiently in an array so as to "wrap around" so that end of the queue is followed by the front of queue.

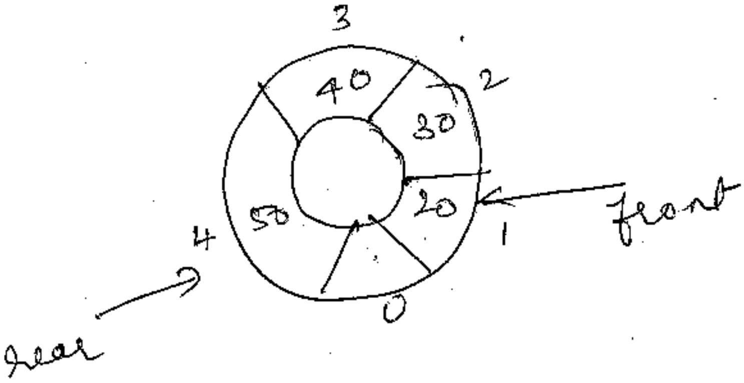
To wrap around the end of the queue to follow the front of queue the following operation is performed. We increment rear by 1 as usual and then perform the modulus operation using operator %. as below

$$rear \leftarrow (rear + 1) \ \% \ Queue\_Size$$

Assume Queue_Size is 5 and 5 elements are inserted into queue. The circular representation of queue as shown as follow



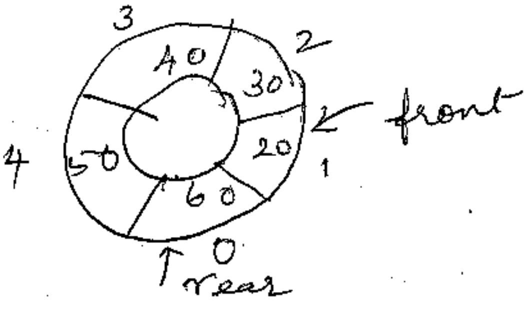
In the above queue, item 10 is the first element. So during deletion, the item 10 has to be deleted. This is achieved by incrementing front by 1 so that front contains the index of the second element.



Now if we want to insert an item 60, we have to increment rear by 1. In the above figure, the value of rear is 4. If we increment rear by 1, its value will be 5. But, we have assumed that queue is circular. So, after incrementing by 1, it should be 0 instead 5. This is achieved by taking modulus as shown below

$$\boxed{rear = (rear + 1) \ \% \ Queue\_Size}$$

After executing the above statement, the value of rear will be 0 so that item 60 can be inserted at 0th position



We increment front by 1 each time an item is deleted using the statement

$$\boxed{\text{front} = (\text{front} + 1) \% \text{ Queue\_size}}$$

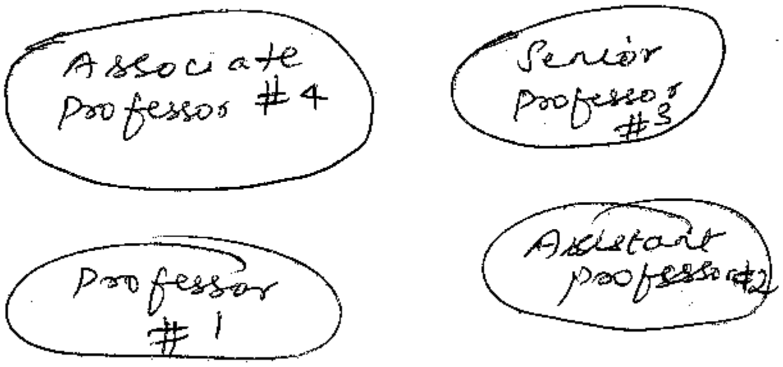
```
# include <stdio.h>
# include <conio.h>
# define  max  5
 void  insert( );
 void  del( );
 void  display( );
 int  cq[max], rear = -1, front = -1;

 void  main( )
 {
     int  choice;
     clrscr( );
     do
      {
          Printf ("1. Insert \n");
          Printf ("2. delete \n");
          Printf ("3. display \n");
          Printf ("4. exit \n");
          Printf ("Enter your  choice ");
          Scanf ("%d", &choice);
      Switch(choice)
      {
      case 1 :  insert() ; break ;
      case 2 :  del() ; break ;
      case 3 :  display() ; break ;
      case 4 :  exit(0) ;
      }
   } while (1) ;
}
void   insert ( )
  {
      int ele ;
      if( (rear+1) % max == front)
         Printf (" Queue full") ;
      else
      {
         if (front = = -1)
           front = 0 ;
         Printf ( "Enter the element \n") ;
         Scanf ( "%d", & ele) ;
         rear = (rear+1) % max ;
      }  cq[rear] = ele ;
  }
```

```
void del()
{
        if ( front = = -1 )
        printf ("Queue Empty");
       else
       {
        printf ("Deleted Element %d", cq[front]);
        if (front = = rear)
         {
            rear = -1;
            front = -1;
         }
        else
          front = (front +1) % max;
    }
}
void display()
{
     int i;
     if (front = = -1)
      printf ("Queue Empty");
      printf ("Content of the queue");
     for(i= front; i!= rear; i= (i+1)% max)
       {
       printf ("%d", cq[i]);
       }
       printf ("%d", cq[i]);
}
```

Priority Queue

A Special form of queue from which items are removed according to their designated priority and not the order in which they entered.

eg.



items entered the queue is Sequential Order but will be removed in the order #3, #1, #4, #2

eg → Network routing - give priority to packets with strickest quality-of-Service requirements

In a priority queue each item (element) has an associated priority & data

The main action that must be very efficient in a priority queue is finding the item with highest priority (usually the one with the maximum or minimum key value)

↳ A priority queue is a Collection of zero or more elements → each element has a priority or value.

- unlike the FIFO queues, the order of deletion from ~~that~~ a priority queue (eg. who gets served next) is determined by the element priority

- Elements are deleted by increasing or decreasing order of priority rather than order in which they arrived in a queue

- operations performed on priority queues
  - * Find an element
  - * Insert a new element
  - * Delete an element etc

Two kinds of (Min, Max) priority queues exist

↳ In a Min Priority queue, find/delete operation ~~operation~~ finds/delete the element with minimum priority

↳ In a Max priority queue, find/delete operation finds/delete the element with maximum priority

— ~~A~~,

Priority Queues

A priority queue is a Collection of elements, each one having an assigned to one-and-the same.

Insertion and deletion are 2 basic operations to be performed for this type of queue as well. However, deletion operation is different from normal queue.

We classify 2 types of priority queues based upon the way in which the elements are deleted.

(a) Ascending priority Queue (Min priority Queue) - element with minimum priority or value is deleted first.

(b) Descending priority Queue (Max priority queue) - element with maximum priority or value is deleted first.

Priority Queue Operations

There are 2 operations defined for a priority Queue

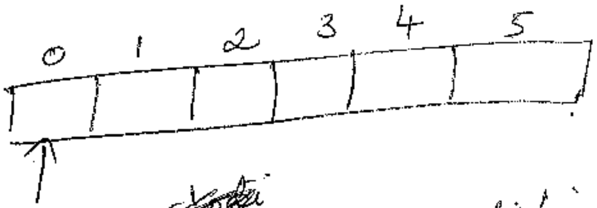
1. enpqueue: enter an element, x into the queue irrespective of its priority.
2. depqueue: delete an element from the queue whose priority is the highest.

An efficient implementation for the priority queue is to use a heap.

The below discussion is based on implementing priority Queue using array and Store the queue elements in an unsorted manner.

Priority Queue Insertion - enqueue

The priority queue design, especially insertion, is same as the traditional queue. One small difference, however, is that we shall not use 2 pointers front and rear, but manage with only one pointer, index



index = -1 ~~index~~ Initial Condition

10	25	15	17	11	21

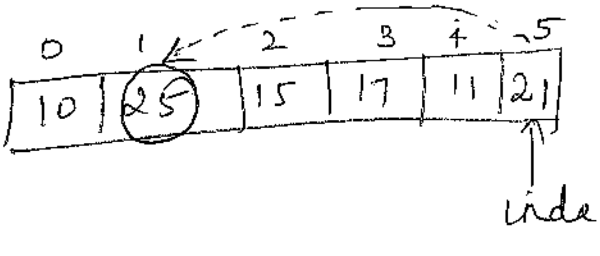
After inserting 10, 25, 15, 17, 11, 21

we shall assume that the elemental values are the priorities. The task of rear pointer is taken by the variable index. When index reaches N-1 it signifies that queue

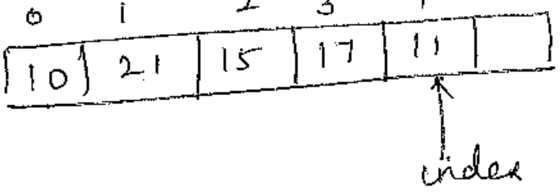
Priority Queue Deletion - depqueue

We shall assume a decending Priority queue for our discussion. This means, the largest element must be removed. Since the elements are not stored in any particular order, we must find the largest element and remove it.

To delete the element, first find the largest element and save it in temporary variable, k. Copy the last element to occupy the position of the largest element. In our example, the largest element 25 is at position 1. That 25 and 21 are swapped. Then decrement the index by one and return the largest element (ie) the element with the highest priority.



0	1	2	3	4	5
10	25	15	17	11	21

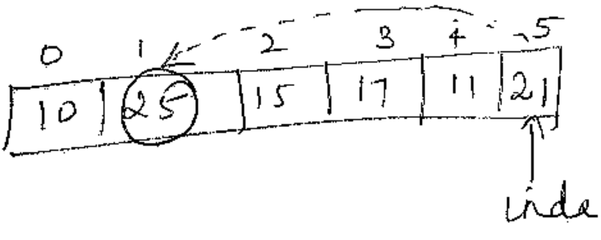


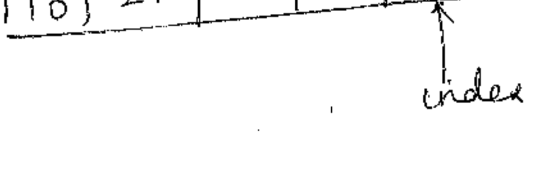
0	1	2	3	4	
10	21	15	17	11	

Priority Queue Deletion - depqueue

We Shall assume a decending Priority queue for our discussion. This means, the largest element must be removed. Since the elements are not stored in any particular Order, we must find the Largest element and remove it.

To delete the element, first find the Largest element and Save it in temporary variable, k. Copy the Last element to occupy the position of the Largest element. In our example, the Largest element 25 is at position 1. That 25 and 21 are swapped. Then decrement the index by one and return the Largest element (ie) the element with the highest priority.





Pointer

Every variable has a memory location and every memory location has its address defined which can be accessed using ampersand(&) operator, which denotes as address in memory.

Consider the following example, which will print the address of the variables defined

```
#include <Stdio.h>
void main( )
{
    int Var1;
    char Var2 [10];

    Printf ("Address of Var1 Variable: %x\n", &Var1);
    Printf ("Address of Var2 variable: %x\n", &Var2);
}
```

Definition

A pointer is a Variable whose value is the address of another variable. (ie) direct address of the memory location.

The general form of a pointer variable declaration is

```
type * var_name;
```

eg

```
int *ip;
double *dp;
float *fp;
char * ch
```

The actual data type of the value of all pointers, whether integer, float, character or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

The following ~~from~~ program is the example of how to define a pointer variable, assign the address of a variable to a pointer and access the value at the address available in the pointer variable.

```
#include <stdio.h>

void main()
{
    int var=20;
    int *ip;
    ip = &var;
   Printf("Address of a var : %x\n", &var);
   Printf("Address stored in ip variable : %x\n", ip);

   Printf("value of *ip variable : %d\n", *ip);

}
```

NULL Pointers in C

It is always a good practice to assign a NULL value to a pointer variable in case you do not have exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a null pointer.

The NULL pointer is a constant with a value of zero defined in several standard libraries.

```
#include <Stdio.h>
 Void main()
 {
     int *ptr =NULL;
     printf("The value of ptr is :%x|n", ptr);
 }
```

On most of the OS, programs are not Permitted to access memory at address 0 because that memory is reversed by the OS. However, the memory address 0 has special significance. it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it assumed to point to nothing.

Pointer Arithmetic

There are four arithmetic operators that can be used on pointers ++, --, + and -

Let us consider that ptr is an integer pointer which points to the address 1000 Assuming 32bit integers, let us perform the following arithmetic operation on the pointer

$\boxed{\text{ptr++}}$

After the above operation, the ptr will Point the location 1004 because each time ptr is incremented, it will point to the next integer location which is 4 bytes next to the current location. This operation will move the pointer to next memory location without impacting actual value at the memory location. If ptr points to a character whose address is 1000, then above operation will point to the location 1001 because next character will be available 1001.

```
#include<stdio.h>
 const int MAX = 3;
 void main()
   { int var[]= {10, 100, 200};
     int i, *ptr;
     ptr = var;
     for(i=0; i<MAX; i++)
     {
     printf("Address of var[%d]=%x\n", i, ptr);
     Printf("value of var[%d]=%d\n", i, *ptr);
     ptr++;
    }
 }
```

Array of Pointers

There may be a situation when we want to maintain an array, which can store pointers to an int or char or any other data type available. Following is the declaration of an array of pointers to an integer.

```
int *ptr[3];
```

This declares ptr as an array of 3 integer pointers. Thus, each element in ptr, now holds a pointer to an int value. Following examples makes use of three integers, which will be stored in an array of pointers as follows.

```
#include <stdio.h>
void main()
{
  int var[] = {10, 100, 200};
  int i, *ptr[3];

  for(i=0 ; i<3 ; i++)
  {
     ptr[i] = &var[i];
  }
  for (i=0 ; i<3 ; i++)
  {  printf("value of var[%d] = %d\n", i, *ptr[i]);
  }
}
```

Pointers with Strings

```
#include <stdio.h>
main ()
{
 char name [] = "Pointers";
 char * ptr;
    ptr = & name[0];
   while (*ptr != '\0')
   {
       printf ("%c", *ptr);
        ptr++;
    }
}
```

Pointers as Function Arguments

```
#include <stdio.h>
void swap( int *, int *);
void main ()
{
   int no1, no2;
   no1 = 5, no2 = 10;
   swap (&no1, &no2)
   printf ("no1 = %d, no2 = %d", no1, no2);
}
```

```
Void swap(int *a, int *b)
 {
     int temp;
     temp= *a;
     *a = *b;
     *b = temp;
 }
```

Function Returning pointers

Function can return address as they return integers, float or character type data. Return type must explicitly be declared using pointers.

```
#include <stdio.h>
 int func();
Void main ()
 {
     int * b;
     b= func();
     printf("b =%x\n *b=%d", b, *b);
 }

int * func( )
{ Static int j=50
     return(&j);
}
```

Dynamic memory allocation

The process of allocating memory during program execution is called dynamic memory allocation.

There are 4 library function under "stdlib.h" for dynamic memory allocation. They are

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

malloc()

The name malloc stands for "memory allocation". The function malloc() reserves a block of memory of specified size and return a pointer of type void which can be casted into pointer of any form

Syntax

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

Here, ptr is pointer of cast-type. The malloc() function returns a pointer to an area of memory with size of byte size. If the space is insufficient, allocation fails and returns NULL pointer.

eg

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

void main()
{
    char *mem_alloc;
    mem_alloc = (char *) malloc (20* sizeof(char));
    if (mem_alloc == NULL)
    {
        printf("Could not able to allocate requested
                memory");
    }
    else
    {
        strcpy(mem_alloc, "Kayarvizhy");
    }
    printf("Dynamically allocated memory content:
            %s", mem_alloc);

    free(mem_alloc);
}
```

Calloc( )

The name calloc stands for "contiguous allocation". The only difference between malloc() and calloc() is that malloc() allocate single block of memory whereas calloc() allocates multiple block of memory each of same size and sets all bytes to zero.

Syntax

```
ptr = (cast-type *) calloc (n, element-size);
```

This statement will allocate contiguous space in memory for an array of n elements.

eg

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

This statement allocates contiguous space in memory for an array of 25 elements each of size of float (ie) 4 bytes.

free ( )

Dynamically allocated memory with either calloc() or malloc() does not get return on its own. The programmer must use free() explicitly to release space

```
free (ptr);
```

This statement cause the space in memory pointer by ptr to be deallocated

C Program to find Sum of n elements entered by user. To perform this program, allocat memory dynamically.

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

  void main()
   {
       int n, i, *ptr, sum=0;
       Printf("Enter number of elements: ");
       Scanf("%d", &n);
       ptr = (int *) calloc (n, sizeof(int));
       if (ptr == NULL)
       {
           printf("Error! memory not allocated");
           exit(0);
       }
       printf ("Enter elements of array: ");

       for(i=0; i<n; i++)
        {
             Scanf(" %d", ptr+i);
             Sum= Sum + *(ptr+i);
        }

       Printf(" Sum=%d", Sum);
       free(ptr);
   }
```

realloc()

If the previously allocated memory is insufficient or more than sufficient. Then, you can change memory size previously allocated using realloc().

Syntax

ptr = realloc (ptr, newsize);

Here, ptr is reallocated with size of newsize

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

void main ()
 {
     int *ptr, i, n1, n2;
     Printf ("Enter ~~the~~ size of array");
     Scanf ("%d", &n1);
     ptr = (int *) malloc (n1* sizeof(int));
     Printf (" Addr of previously allocated memory");
     for (i=0; i<n1; i++)
          printf ("%u", ptr+i);
     Printf ("Enter new size of array");
     Scanf ("%d", &n2);
ptr = realloc (ptr, n2);
for(i=0; i<n2; i++)
      Printf("%u", ptr+i);

}
```