Data structure?
is it a programming language?

Data structure is a concept of set of algorithms, used to structure the information & can be implemented using any programming language like C, C++, java etc.

→ main focus is on storing data — How efficiently we can store data through this algorithms.

efficient ← time / space

→ we are going to implement this algorithms using C.

→ All these algorithms are called ADT (Abstract data types)

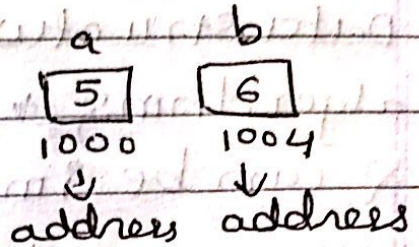
→ Algorithm → set of rules

→ Examples for algorithms: Arrays, stack, queue, linked list, graphs, trees

## Pointers:

is a variable that contains the address of another variable or address of memory location

Ex:




```
int main()
{  int a, b;
   a=5, b=6;
   printf("%d %d", a,b);
}
```

why pointers?

→ ~~Foor~~ For dynamic memory allocation

→ program runs faster

~~Ex:~~

```
~~int~~ main()
{ ~~int~~
```

## Declaring a pointer:

syntax:

```
Datatype * variablename;
```

↓ int, float, char, double

↓ name given to memory location

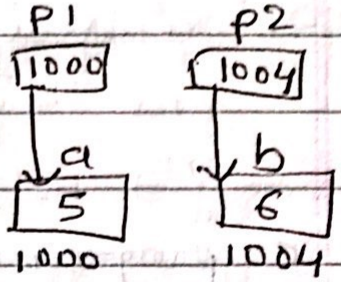
DATE : / /

Ex:

```
int *p;
float *p;
```

Ex;

```
int main()
{
    int *p1, *p2;
    int a, b; a=5, b=6;
    p1 = &a;
    p2 = &b;
    printf("%d %d", *p1, *p2);
}
```




} initializing a pointer.

Initializing a pointer:

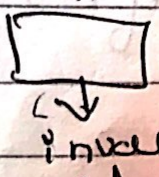
Syntax:

pointer variable = & data variable;

Dangling pointer:

A pointer variable which does not contain valid address is called as Dangling pointer

Ex: int *p;



NULL pointers:
It points nowhere in memory. Special value NULL is assigned to it.

Ex: int *p;
p=NULL;

① C program to implement simple calculator using pointer.

```
#include<stdio.h>
void main()
{   int a,b,sum,sub,div,
    int *p1,*p2;                mul;

    printf("Enter values of a&b");
    scanf("%d%d",&a,&b);
    p1=&a;
    p2=&b;
    sum=*p1+*p2;
    sub=*p1-*p2;
    div=*p1/*p2;
    mul=(*p1)*(*p2);
printf("%d %d %d %d",sum,sub,div,
                        mul);
}
```

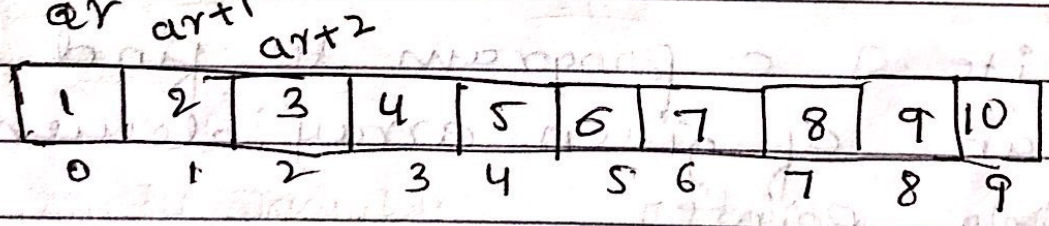
DATE : / /

# Pointers & Arrays

→ every array is a pointer
→ value of array variable = address of first element
→ Elements of the array can be accessed using pointer also

Example:

```
int ar[10];
printf("%d", ar[0]) → Prints value at 0
printf("%d", &ar[0]) → prints address of first element
printf("%d", ar); → prints address of first element.
```




~~ar+1 →~~

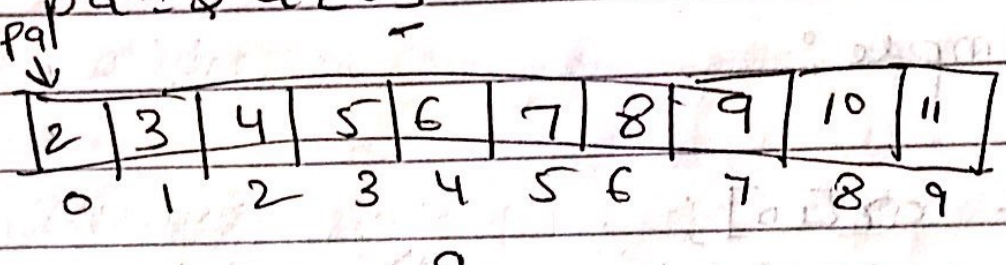
ar+i → points at ith element after ar

~~ar+1~~ ar−i → points at ith element before ar

~~ar+2~~

*ar → value at ar[0]
*(ar+1) → value at ar+1

pointers with arrays
Example:
int *pa; int a[10]
pa = &a[0]




pa → will give address
*pa → value at where pa is pointing

1] write a c program to find sum & mean of given array elements using pointer

```
#include<stdio.h>
void main()
{   float a[10], mean, sum=0, *pa
    int i,n;
```

```
printf("Enter the size of array");
scanf("%d",&n);
printf("Enter array elements");
for(i=0; i<n; i++)
{
    scanf("%f",&a[i]);
}
  pa=a;
for(i=0; i<n; i++)
{
  sum=sum+*pa;
   pa++;
}
mean=sum/n;
printf("sum is=%f, mean=%f", sum,
                              mean);
}
```

pointers & functions:

This concept is similar to pass by address & pass by value

→ pointers can be passed to a function & can be returned from a function

passing pointers to a function [pass by address]

Example: swapping content of two variables using pointers,

```
#include<stdio.h>
void exchange(int *m, int *n)

void main()
{
    int a, b;
    printf(" Enter values of a & b ");
    scanf(" %d %d ", &a, &b);
    exchange(&a, &b);
    printf(" %d %d ", a, b);
}
```

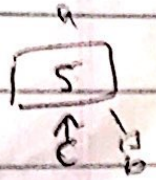
```
void exchange ( int *m, int *n )
{
    int temp;
    temp = *m;
    *m = *n;
    *n = temp;
}
```

~~pointer to pointer~~

Function returning a pointer

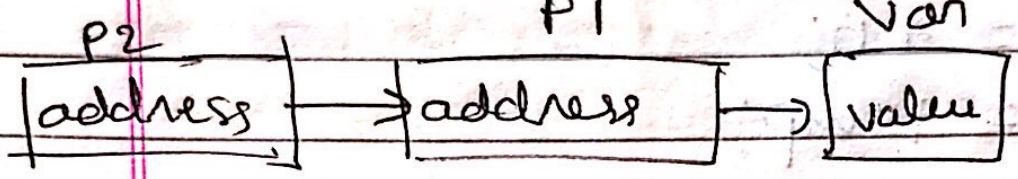
A function can return a pointer

```
#include <stdio.h>
int *display( int *c);
void main ()
{   int a;
    int *b;
  printf("Enter value of a");
  scanf("%d", &a)
   b = display (&a);
   printf("%d", *b);
}
int *display (int *c)
{ return c; // Return address
}
```



Pointer to pointer

A variable that contains the address of another pointer variable is called as pointer to pointer.




```
int var;
int *p1, **p2;
p1 = &var;
p2 = &p1;
```

To access value:

var, *p1, **p2

Union
is also a collection of variables of similar or dissimilar data types.

Syntax

```
union tagname
{
   type1 variable1;
   type2 variable2;
      :
   typen variablen;
};
```

DATE: / /

Difference

→ in structure we can store elements at a time & retrive all of them at a time

→ In structure if there are n memembers then n memory locations will be created.

→ But in union only one memory location is created even if there are n members & ~~one~~ one member at a time can be stored & retrived. i,e can not proces all elements at a time.

→ In union, memory is allocated by considering size of largest member of union

# Dynamic memory management in C

Static memory allocation [Compile time]

→ if memory is reserved or allocated for the variables at compile time then it is called as static memory allocation.

Dynamic memory [Runtime]

→ if memory is reserved or allocated for the variables at runtime then it is called as dynamic memory allocation

Ex:

```
int x;
int a[10];
```

i) malloc() ii) calloc() iii) Realloc() iv) free()

→ All are defined in <stdlib.h>

i) malloc()

→ check if its pointing to NULL

Syntax:

```
ptr = (data type *) malloc (size);
```

→ it returns void pointer.

void → tells that data type is not important I have given you space.

Ex:

```
int *ptr;
ptr = (sizeof)
ptr = (int *) malloc (sizeof(int));
*ptr = 5;
```

Ex:

```
int *ptr
ptr = (int *) malloc (no of items * sizeof(int));
ptr[0] = 2;
```

generic pointer.

ii) calloc ( )

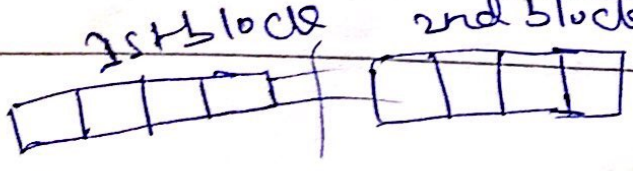
This function is used to allocate multiple blocks of memory & mainly used for arrays.

Syntax:

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

Ex:

```
ptr = (int *) calloc (2, 4);
```



(1) program to read & display marks of 3 subjects using calloc() function

```
#include<stdio.h>
void main()
{
    int *marks, i;

    marks = (int *) calloc(3, sizeof(int));

    for(i=0; i<3; i++)
    {
        printf("Enter %d subject marks", i);
        scanf("%d", &marks[i]);
    }

    for(i=0; i<3; i++)
    {
        printf("%d", marks[i]);
    }
}
```

malloc

(2) program to find sum of even & odd numbers in a given ~~array~~ stored using malloc.

Realloc ()

→ used to extend size of already allocated memory or to delete memory at the end of block.

→ Before using this function memory should have allocated using malloc or calloc

Syntax :

ptr = (datatype *) realloc (ptr, size)

→ ptr is points to previously allocated memory using either malloc or calloc

→ size is new size of block.

```
int *ptr
ptr = (int *) malloc (4);
ptr = (int *) realloc (8);
```

Subhi

iv) free()

used to deallocate allocated blocks of memory which is allocated by using functions such as malloc(), calloc(), or realloc()

syntax:

```
free(ptr);
ptr=NULL;
```

Difference between malloc & calloc

| malloc() | calloc |
|---|---|
| 1] it does not initialize allocated memory | 1] it initializes deallocated memory to zero |
| 2] takes one argument i.e size | 2] takes 2 arguments i.e size & no of blocks |
| 3] efficiency is higher | 3] efficiency is less compared to malloc |
| 4] allocates memory even if memory is not available contiguously | 4] allocates required blocks contiguously. |

Pointer to Array of Structure

```
#include<stdio.h>
struct student
{
    char name[20], USN[20];
    int marks, rollno;
};
struct student stud[30];
struct student *s;

void main()
{   int i;
    s=&stud;
    printf("Enter no of students");
    scanf("%d", &n);
    printf("Enter student details");
    for(i=0; i<n; i++)
    {
    scanf("%s%s%d%d", (s+i)->name,
        (s+i)->USN, &(s+i)->marks,
        &(s+i)->rollno);
    }
    printf("student details are");
    for(i=0; i<n; i++)
    { printf("%s%s%d%d", (s+i)->name,
        (s+i)->USN, (s+i)->marks,
        (s+i)->rollno);
}   }
```

Allocating memory dynamically to a structure.

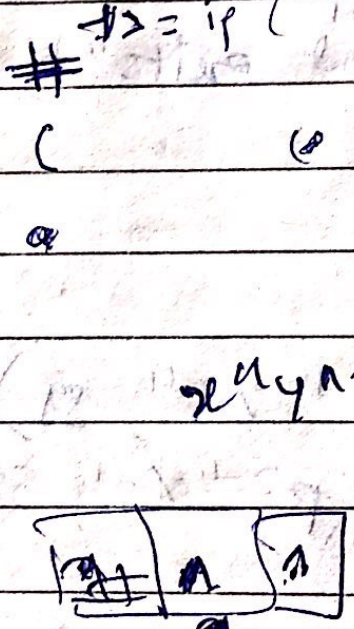
```
struct student
{ char name[20], USN[20];
  int rollno, marks;
};
struct student *s;

void main()
s= (struct student*) malloc (sizeof
                    (struct student));
```

stack:

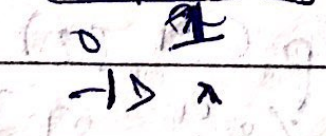
```
void main()
{
    FILE *fp1, fp2, fp3;

    fp1 = fopen("text1.txt", "r");
    fp2 = fopen("text2.txt", "r");
    fp3 = fopen("text3.txt", "w");

    if(fp1==NULL || fp2==NULL)
    { printf("file does not exits");
      exit(0);
    }

    while(fscanf(fp1, "%s", s1)==1)
    { if(fscanf(fp2 "%s", s2)==1)
      {
          if(strcmp(s1,s2)==0)
            printf("%s", s1);
          else
            fprintf(fp3, "%s\n%s\n", s1, s2);
      } else fprintf(fp3, "%s", s1);
    }
    while(fscanf(fp2, "%s", s2)==1)
      fprintf(fp3, "%s", s2);
```

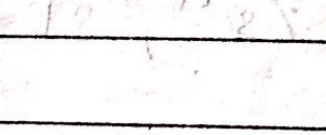
ADT: are entities that are definations of data & operations but do not have implementation details.
[ that means we know what are store, what operations to be perform & way in which data is stored ]

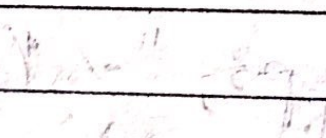
(a+b)

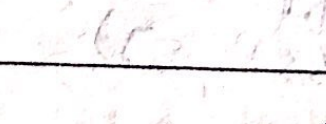


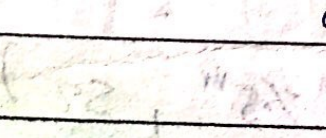


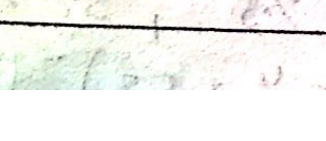












pgm to convert infix to postfix

```
int sp(char c)
{
  switch(c)
  {
    case '#' : return -1;
    case '(' : return 1;
    case '+' :
    case '-' : return 3;
    case '*' :
    case '/' : return 5;
    case '^' : return 7;
  }
}
int ip(char c)
{
  switch(c)
  {
    case '(' : return 6;
    case '+' :
    case '-' : return 2;
    case '*' :
    case '/' : return 4;
    case '^' : return 8;
  }
}
```

```
void convert (char infix[], char postfix[])
{ int i=0, j=0;
  char symb;
  int top=0;
  char stk[30];
  str[top] = '#';
  while(infix[i] != '\0')
  {
    symb = infix[i++];
    if(symb >='0' && symb <='9') ||
      (symb >='a' && symb <='z'))
      postfix[j++] = symb;
    else
    {
      if (symb == ')')
      { while(stk[top] != '(')
        { postfix[j++] = stk[top--];
        }
        top--;
      }
      else if (symb == '(')
        stk[++top] = symb;
      else
      {
```

DATE : / /

```
while(sp(stk[top]) >= ip(symb))
    postfix[j++] = stk[top--];

    str[++top] = symb;
  }
 }
}

while(stk[top] != '#')
  postfix[j++] = stk[top--];
  postfix[j] = '\0';
}

main()
{ char infix[30], postfix[30];
  pf("enter infix exp");
  sf("%s", infix);
  convert(infix, postfix);
  pf("%s" postfix);
}
```