MODULE-1

DATA STRUCTURE AND APPLICATION

Introduction

Data structure is way of Organizing data in a memory so that it can be used efficiently

or

Data structure is specialized format for Organizing, storing and retrieving the data.

Types of Data structure

* Data structure can be classified as follows
  1) Primitive Data structure
  2) Non-Primitive Data structure
  3) Linear Data structure
  4) Non-Linear Data structure

(a) Primitive Data-type or Data structure

* Basic data types that are available in most of the Programming Language. Data structures that are directly operated upon by machine-level instructions are known as primitive data types

* For example
  1) Integer (int): used to represent a number without decimal point Eg: 12, 90
  2) Float or (float) double: used to represent a number with decimal point Eg: 12.80
  3) Character (char): used to represent single character Eg: 'c', 'a'
  4) Boolean: used to represent logical values i.e true or false

Non-Primitive Data Structure

* The data structures are derived from the primitive data structures. They stress on formation of sets of homogeneous and heterogeneous data elements.

Example: Arrays, stack, Queue, Linked List, Tree, Graph

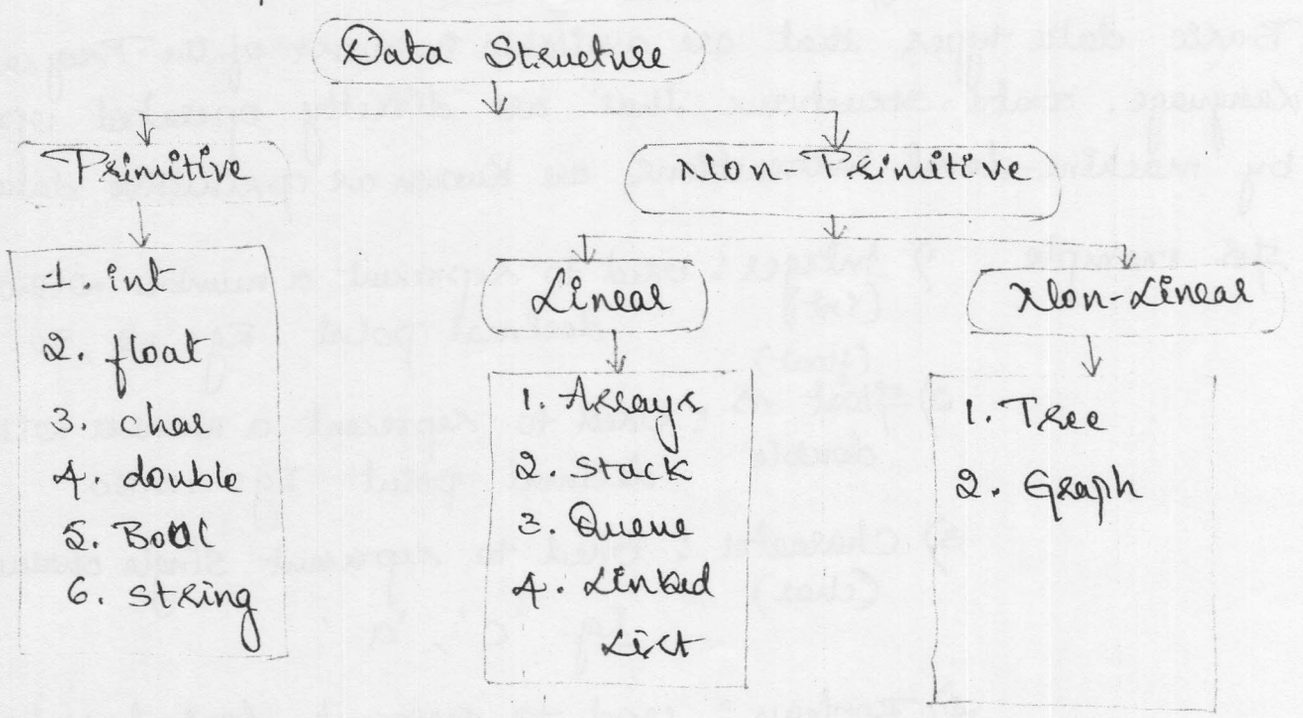
Linear-Data Structure

* In Linear Data Structure, data is arranged in linear fashion. Eg: Arrays, stack Queue, Linked List

Non-Linear Data Structure

* In Non Linear Data structure, data is not arranged in order or linear fashion

Eg: Trees, Graph, table etc.



⇒ Array

Definition: is a data structure, that is used for the storage of Homogeneous data i.e data of the same type. (or)

Array is a collection of elements of the same data type. The individual elements of an array are not name. All the elements of an array share a common name i.e the array name.

* Individual elements of an array accessed using index or subscript. Array index starts with 0 i.e index of the first element of an array is 0.

* Memory space required by an array can be computed as
  (Size of element type) * (Number of elements in an array)

* Arrays are always stored in Continuous memory locations.

Syntax:

arrayname [Size of array];

arrayname → name of an array

[ ] → Subscript operator

Size of array → Size or Maximum capacity of an array

Example: ① int array1[5]; /* integer array which can store 5 elements */

array1

1	10	20	50	30
0	1	2	3	4

array1 → name of an array

0 1 2 3 4 → index

② char array2[3];

array2

'A'	'B'	\0
0	1	2

Types of Array

1) Single-dimensional Array
2) Multi-dimensional array

Single-dimensional Arrays

- Fixed number of elements of the same data type Organized as a Simple linear Sequence.

Syntax: type arrayname [size];

where arrayname is name given to an array

size specifies number of elements

[ ] → Subscript Operator

Example: int array[5];
float array2[5];

Initializing elements of Single-dimensional array

- The elements of an array can be initialized by using initialization list. An initialization list is a comma-seperated list of initializer enclosed within braces.

Syntax: type arrayname[size] = $\underbrace{\{v_1, v_2, v_3 \ldots v_n\}}_{\text{initializers}}$;

The number of initializers in the initialization list should be less than or at most equal to the value of size specifier.

Example: int array1[3] = {1, 2};

array 1	1	2	0
	0	1	2

Here number of initialize is two. But size of an array is 3 One space is empty. so it will be initialized to zero since it is an integer array.

⇒ Reading, Storing Accessing elements

* An iteration statement is used for storing and reading the elements of a one dimensional array. The code snippet illustrates a method to read, store and access the elements.

```
#include<stdio.h>
main()
{
   int n, a[10], i;
   printf("enter the number of elements \n");
   scanf("%d", &n);
   printf("enter the array elements \n");
   for(i=0; i<n; i++)
        scanf("%d", &a[i]);
   printf("The array elements are \n");
   for(i=0; i<n; i++)
        printf("%d \t", a[i]);
}
```

Output:

```
enter the number of elements
5
enter the array elements
10
20
30
40
50
The array elements are
10    20    30    40    50
```

Memory representation

a	10	20	30	40	50					
	0	1	2	3	4	5	6	7	8	9

⟹ Operations on Arrays

1) Insert
2) Delete
3) Simple merge
4) Search

Insertion Operation

* Consider an array a[10] having 3 elements in it initially and a[0]=1, a[1]=2, a[2]=3 and want to insert a number 45 at location 1 i.e a[0]=45. So we have to move elements one step ~~before~~ so after insertion a[1]=1 which was a[0] initially, and a[2]=2 and a[3]=3.
* The code snippet illustrate a method where element can be inserted at given position.

```
# include <stdio.h>
main()
{
    int a[20], pos, i, n, val;
    printf("in enter the number of elements in");
    scanf("%d", &n);
    printf("in enter the array elements in");
    for(i=0; i<n; i++)
        scanf("%d", &a[i]);
    printf("in enter the location where you want to insert in");
        scanf("%d", &pos);
    printf("in enter the value to insert in");
        scanf("%d", &val);
    for(i=n-1; i>=pos-1; i--)
        a[i+1] = a[i];
```

```
        a[pos-1] = val;
        n++;
    printf("Resultant array is \n");
    for(i=0; i<=n; i++)
        printf("%d\n", a[i]);
}
```

<u>Output</u>: enter the number of elements

5

enter the array elements

2
5
4
3
8

enter the location where you want to insert

4

enter the value to insert

10

Resultant array is

2
5
4
10
3
8

Delete operation

* To delete element from an array, shift the elements in the array. consider an array a[5] having 4 elements in it initially and a[0] = 1, a[1] = 2, a[2] = 3 and want to delete element in the position = 2 i.e a[1]. So we have move elements one step below. So after deletion a[0] = 1, a[1] = 3

a	1	2	3	5	
	0	1	2		

← position to be deleted is 2

a	1	3		5	
	0	1	2	3	

Move 5 to a[2]

a	1	3	5		
	0	1	2	3	4

```
# include<stdio.h>
main()
{
    int a[20], pos, n, i;
  printf("enter the number of elements\n");
   scanf("%d", &n);
  printf("enter the array elements\n");
   for(i=0; i<n; i++)
       scanf("%d", &a[i]);
  Printf("enter the position where you want to delete\n");
       scanf("%d", &pos);
    for(i=pos-1; i<n-1; i++)
        a[i] = a[i+1];
    n--;
  Printf("Resultant array is\n");
```

```
    for (i=0; i<n; i++)
        printf("%d\n", a[i]);
}
```

Output:

enter the number of elements in an array

5

enter the array elements

4
6
8
10
7

enter the position where you want to delete

2

Resultant array is

4
8
10
7

SIMPLE MERGE

* Merge is one of the operation, to combine two arrays, ~~After~~ i.e combining Array elements in sorted manner.

For example: a1 | 2 | 4 | 5 | 7 | 8 |

a2 | 5 | 6 | 7 | 8 | 9 | 10 |

* first time

SIMPLE MERGE:-

* Merge is a operation, which combines the elements of two arrays and store it in a resultant array. This should be in sorted manner.

* For example:

array 1

10	6	5	4	8	2
0	1	2	3	4	5

array 2

1	7	4	12	14
0	1	2	3	4

* if merge operation is applied on array1 & array2, it should combine both the arrays and store it in a resultant array. This should be sorted.

* In order to merge, compare the first element of array1 with first element of array2, if a1[0] <= a2[0], then copy a1[0] to res[0], else copy a2[0] to res[0]. i.e

```
if(a1[0]<=a2[0])
    res[0] ← a1[0]
else
    res[i] ← a2[0]
```

Above operation will applied for n elements in array. if any elements are remaining in array1 or array2 copy to the resultant array by using.

```
while(i<n1)
{
    res[k] = arr1[i];
    i++;
    k++;
}
```

// If any remaining elements in array1 copy to resultant array

```
while(j<n2)
{
    res[k] = arr2[j];
    j++;
    k++;
}
```

// If any remaining elements in array2 copy to resultant array.

```
#include <stdio.h>
int main()
{
    int a1[10], a2[20], res[30];
    int i, j, k, n1, n2;
    Printf("\n enter the number of elements in array1 \n");
    Scanf("%d", &n1);
    Printf("\n enter the array elements \n");
    for(i=0; i<n1; i++)
    {       scanf("%d", &arr1[i]);
    Printf("\n enter the number of elements in array2 \n");
    Scanf("%d", &n2);
    for(i=0; i<n2; i++)
            Scanf("%d", &arr2[i]);
    i=0, j=0, k=0;
    while(i<n1 && j<n2)
    {
        if(a1[i] <= a2[j])
        {
            res[k] = a1[i];
            i++; k++;
        }
        else
        {
            res[k] = a2[j];
            j++; k++;
        }
    }

    while (i<n1)
    {
        res[k] = a1[i];
        i++;
        k++;
    }
```

...rch Operation

...he process of finding a particular item in the large amount of data called Searching. The two techniques are shown below.

1) Linear Search

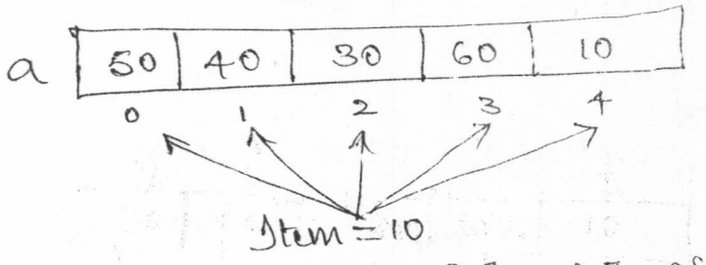
2) Binary Search

...EAR SEARCH (Sequential Search)

A Linear Search is a simple Searching technique. In this technique we Search for a given Key item in the list linear order (Sequential order) i.e one after the other. The item to be searched is called Key item.

if Key = 10 and the list is 20, 10, 40, 25 after searching key is present. If Key = 100 after searching we say that key is not present.

...ocedure : Assume 10 is the item to be searched in the list of items 50, 40, 30, 60, 10. Compare Key item 10 with list of items i.e



* Compare Key = 10 with a[0], a[1], a[2], a[3], a[4]. If it matches with any of the element, then Search is Successful. In this case, Search is "Success"

* If Key = 75, compare a[0], a[1], a[2], a[3], a[4] 75 is not there. Then Search is unsuccess.

```
for(i=0; i<5; i++)
{
    if (Key == a[i])
    {
        printf("success");
        exit(0);
    }
}
```

```
while(j<n2)
{
    res[k] = a2[j];
    k++;
    j++;
}

printf("\n Merged array is :");
  for(i=0; i< n1+n2; i++)
      printf(" %d", res[i]);
}
```

<u>output</u>

```
enter the number of elements in array 1
5
enter the array elements
10 12 5 6 1

enter the number of elements in array 2
4
enter the array elements
7 6 4 8
```

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

int main()
{
    int i, n, key, a[10];
    printf("enter the number of elements\n");
    scanf("%d", &n);
    printf("enter the array elements\n");
    for(i=0; i<n; i++)
        scanf("%d", &a[i]);
    printf("enter the key to be searched\n");
    scanf("%d", &key);
    for(i=0; i<n; i++)
    {
        if (a[i]==key)
        {
            printf("Search success");
            return 0;
        }
    }
    printf("Search is unsuccess");
}
```

<u>Output:</u>

```
enter the number of elements
5
enter the array elements
10  20  50  40  30
enter the key to be search
30
Search success
```

```
enter the number of elements
5
enter the array elements
10  20  50  40  30
enter the key to be searched
70
Search is unsuccess.
```

Binary Search

* It is a simple searching technique, used to search key element By dividing array into two parts. Here list of elements should be in sorted manner.

Procedure: * Initialize three variables low = 0, high = n-1. calculate the mid position i.e mid = (low + high) / 2

* The key to be searched is to be ~~searched~~ compared with middle element. If condition is true, then search success

```
if (key == a[mid])
{
    printf(" Search is success");
    return;
}
```

* If this condition is false, key may be in the left part of the array or in the right part of the array.

* If key is less than the middle element, key will be in the left part and compare from "low to mid-1" Otherwise right part of the array has to be compared from mid+1 to high. i.e

```
if (key < a[mid])
    high = mid-1;
else
    low = mid+1;
```

All the above statements need to be executed repeatedly untill low crosses high i.e (low <= high)

```
# include <stdio.h>

main()
{
    int i, n, key, a[20], pos;
    printf("enter the value of n\n");
    scanf("%d", &n);
    printf("enter n values\n");
    for(i=0; i<n; i++)
        scanf("%d", &a[i]);
    printf("enter item to search\n");
    scanf("%d", &key);
    low=0;
    high=n-1;
    while(low<=high)
    {
        mid=(low+high)/2;
        if (key==a[mid])
        {
            pos=mid;
            printf("Search is Success & found at position %d", pos);
            return;
        }
        if (key < a[mid])
            high=mid-1;
        else
            low=mid+1;
    }
    printf("Search is unsuccess\n");
}
```

SORTING

* The process of arranging the given elements so that they are in ascending or descending order is called sorting.

For example: consider unsorted elements: 10, 50, 25, 20, 15

After sorting in Ascending order: 10, 15, 20, 25, 50

After sorting in descending Order: 50, 25, 20, 15, 10

Selection Sort

* Selection sort is a sorting technique, here first find the smallest item in the list and exchange it with the first item. Obtain the second smallest element exchange with the second element and so on.
* Finally all the elements will be arranged in ascending order. Since the next least item is selected and exchanged appropriately so that elements are finally sorted.

For example: consider the elements 45 20 40 5 15

a	45	20	40	5	15
	0	1	2	3	4

	After pass 1	After pass 2	After pass 3	After pass 4
a[0] = 45	5	5	5	5
a[1] = 20	20	15	15	15
a[2] = 40	40	40	20	20
a[3] = 5	45	45	45	40
a[4] = 15	15	20	40	45

* To find smallest element position,

```
pos = i
for(j = i+1 ; j<n; j++)
{
    if (a[j] > a[pos])
        pos = j;
}
```

* After finding the position of the smallest number, it should be exchanged with i^{th} position.

* After finding the position of the smallest number, it should be exchanged with i^{th} position.


```
temp = a[pos];
a[pos] = a[i];
a[i] = temp;
```

* The above procedure has to be performed for each of i in the range $0 \le i < n-1$

```
#include<stdio.h>

main()
{
    int n, i, j, temp, a[20], pos;

    printf("enter the number of items");
    scanf("%d", &n);

    printf("enter the items to sort\n");
    for(i=0; i<n; i++)
        scanf("%d", &a[i]);

    for(i=0; i<n-1; i++)
    {
        pos=i;
        for(j=i+1; j<n; j++)
        {
            if(a[j]<a[pos])
                pos=j;
        }

        temp = a[pos];
        a[pos] = a[i];
        a[i] = temp;
    }

    printf("The sorted items are\n");
    for(i=0; i<n; i++)
        printf("%d\n", a[i]);
}
```

45	40	20	15	5
0	1	2	3	4

$i=0 \quad 0<4$

$pos = i$

$j=1 \quad 1<5$

$a[j] < a[pos]$

$a[1] < a[0]$ (F)

$j=2 \quad 2<5$

BUBBLE SORT

* It is a sorting technique, two successive items are compared and exchanged whenever first element greater than or equal to next element. i.e $A[i] >= A[i+1]$

For example: 40, 50, 30, 20, 10

	1st pass				2nd pass			3rd pass		4th pass
A[0] = 40	40	40	40	40	30	30	30	20	20	10
A[1] = 50	50	30	30	30	40	20	20	30	10	20
A[2] = 30	30	50	20	20	20	40	10	10	30	30
A[3] = 20	20	20	50	10	10	10	40	40	40	40
A[4] = 10	10	10	10	50	50	50	50	50	50	50

* In first part 40 is compared with 50 and they are in order. So, no exchange has been done. Next 50 is compared with 30 and they are exchanged since 50 is greater than 30. If we proceed in the same manner, at the end of the first pass the largest item occupies the last position
* In the second pass 30 is compared with 40 and they are in order. So, no exchange has been done. Next 40 is compared with 20. and they are exchanged since 40 is greater than 20. If we proceed in the same manner, at the end of second pass the largest item occupies the last position.
* The above procedure is repeated until all the elements in the array are sorted.

```
#include <stdio.h>

int main()
{
        int a[25], i, j, n, temp;

        printf("\n Enter the number of integers
                to be sorted: ");

        scanf("%d", &n);
```

```
    printf("Enter the numbers: ");
    for(i=0; i<n; i++)
    {
        scanf("%d", &a[i]);
    }

    for(i=0; i<n; i++)
    {
        for(j=0; j<n-i-1; j++)
        {
            if (a[j]>a[j+1])
            {
                temp = a[j];
                a[j]= a[j+1];
                a[j+1] = temp;
            }
        }
    }

    printf("The sorted array using Bubble
            Sort is : \n");
    for(i=0; i<n; i++)
    {
        printf("%d\t", a[i]);
    }

    return 0;
}
```

Output:

```
Enter the number of integers to be sorte: 5
Enter the numbers: 40 50 30 20 10
The sorted array using Bubble sort is:
10 20 30 40 50
```

⇒ STRING

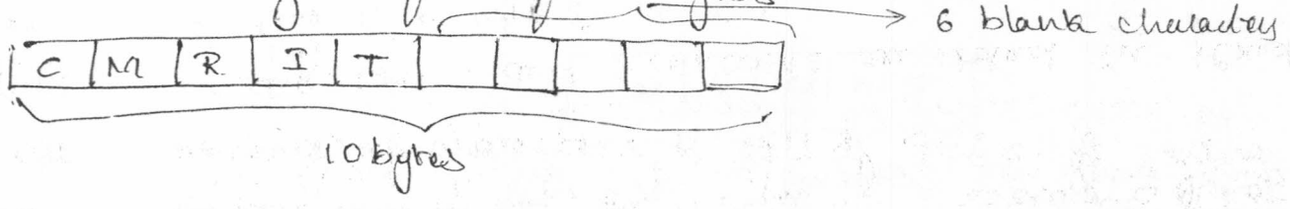
* An array (sequence) of characters is called String. A string is most conveniently represented using a sequence of storage locations in memory.

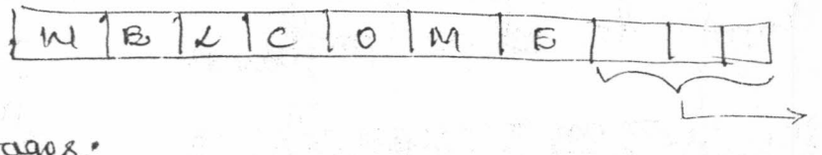
Storage representation of String
- → Fixed length string
- → Variable length String
  - → Length controlled string
  - → Delimited String

① Fixed Length String

* Here the length of the string is always fixed. In fixed length string is a whose storage requirement is known, when the string is created.
* Once the string is defined, the length of a string can not be changed.

For example: The string CMRIT and WBCOME are stored in fixed format whose size is fixed for 10 bytes





Disadvantages:

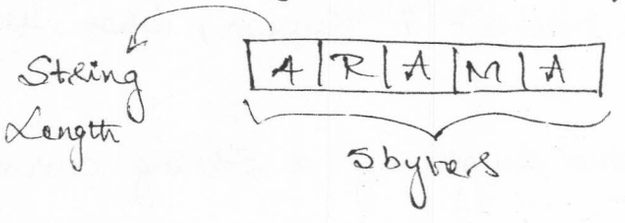
1) If length reserved for string is too small, it is not possible to store the larger data.
2) If length reserved for string is too large, too much memory wasted
3) Once the string is defined length of string cannot be changed.

⇒ Variable Length format

* Variable length strings don't have a pre-defined length. The storage structure for a string can expand or shrink to accommodate any size of data. But, there should be a mechanism to indicate the end of the data. The common techniques
  1) Length controlled string
  2) Delimited string.

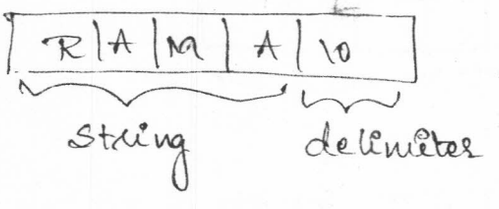
Length controlled string

* Here, length of the string is stored as part of the string itself. This technique uses a count that specifies string length. The count is normally stored as the first byte followed by the string.



Delimited String

* String ends with a delimiter is known as Delimited string



Basic Terminology

1) **String:** Finite sequence of zero or more characters is called string.

2) **Length:** The number of characters in a string is called its length.

3) **empty string or null string:** The string with zero characters is called the empty string or the null string.

4) **Concatenation:** Let S_1 and S_2 be strings. The string consisting of the characters of S_1 followed by the characters S_2 is called the concatenation of S_1 & S_2
* It will be denoted by $S_1 || S_2$

For example:

① "$S_1 || S_2$"

② "THE" || "END" = "THEEND"

③ "THE" || "□" || "END" = "THE END"

5) **Substring:** A string y is called a substring of a string S if there exist strings x and z such that

$$S = x || y || z$$

if x is empty string, then y is called initial substring of S
if z is empty string, then y is called terminal substring.

Strchr

* function searches for the first occurrence from the beginning of the string and following values are returned.
  a) On Success, a pointer to the character is returned
  b) On failure, NULL is returned.

* Prototype:

```
char *strchr (char *s, char ch);
```

where s is the first string
ch is character to be searched

For example:

```
#include<stdio.h>
int main()
{
    char *s;
    char buf[20] = "This is a text"
    s = strchr (buf, 't');
    Printf("found t at %s", s);
}
```

Output: found t at text

Strrchr

* This function searches for the first occurrence from the end of the string and i) On success, a pointer to the character is returned
  b) On failure, NULL is returned

* Prototype: char *strrchr (char *s, char ch)

where s is the string
ch is the character to be searched

* For example:

```
#include <stdio.h>
int main()
{
    char *s;
```

```
char buf[20] = "This is text";
p = strchr(buf, 't');
printf("found t at %s", p);
}
```

Output: found t at t

Strstr

* This function searches the substring in string for the first occurence from the begining. The following values are returned:

  1) On success, a pointer to the characted is returned
  2) On failure, NULL is returned.

* Prototype: `char *strstr(char *str, char *sub_str)`

  where str is the string

  sub_str is substring be searched in str.

* For example:

```
char s1[50]="abefgnpanm";
char s2[5] = "ab";
char *s6;
s6 = strstr(s1, s2);
printf("%s\n", s6);
```

Output: abefgnpanm

Str Spn

* This functions keeps comparing each character of String S1 in order from left to right with contents of string S2.
* As long as the corresponding character of S1 is available in S2, the search continues with next character.
* The search stops when a character in S1 does not match with any of the characters in S2. The function returns the following values:

  1) Returns number of characters in S1 that have so far successfully matched with one of the characters of S2.
  2) return 0, if no characters in S1 match with any of the characters of S2.

* Syntax:

```
size_t strspn( const char *str1,
               const char *str2)
```

str1 - is the main string.
str2 - string containing the list of characters to match in str1

* Example:

```
const char s1[] = "ABCDEFGH01234";
const char s2[] = "ABCD";

len = strspn (s1, s2);

printf("%d", len);
```

Output: 4

⟹ STRUCTURES

* Structure is a collection of one or more ~~characters~~ one or more variables of same or different data types grouped together under the single name.

* Structure is a user defined data type that can store related information about an entity.

* Syntax:

```
struct structure_name
{
        data type var_name 1;
        :
        data type var_name 2;
};
```

* Initialization of structure is done after declaration

```
struct structure_name structure_variable_name;
```

For example:

```
struct person
{
    char name[10];
    int age;
    float salary;
};
```

→ name of the structure

} members of the structure

```
struct person P;
```

→ variable of structure

* Accessing the members of structure

* Inorder to access the members of the structure we require structure variable name and . operator

Syntax:

structure_variable . member
name name

* For example: inorder to store name of person

```
strcpy(p.name, "xyz");
p.age = 10;
p.salary = 35000;
```

* The above can also be read dynamically using scanf

~~scanf("%d", &emp 1/. employee~~

```
scanf("%s", p.name);
scanf("%d", p.age);
scanf("%f", p.salary);
```

/* C program to read an employee number, name, age salaray and print details using structure */

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

struct employee
{
    int empnumber;
    char empname[20];
    int age;
    float salary;
};
```

Date
Page

```
void main()
{
    struct employee emp1;

    printf("enter employee number, name, age, salary\n");
    scanf("%d %s %d %f", &emp1.empnumber,
                emp1.name, &emp1.age, &emp1.salary);

    printf("employee number is %d\n", emp1.empnumber);
    printf("employee name is %s\n", emp1.emp name);
    printf("employee name is %d\n", emp1.age);
    printf("employee name is %f\n", emp1.salary);

}
```

output:

```
enter employee number, name, age, salary
12, nav 25 28000

employee number is 12
employee name is nav
employee age is 25
employee salary is 28000
```

⟹ STRUCTURES

* Structure is a collection of one or more ~~structures~~ one or more variables of same or different data types grouped together under the single name.

* Structure is a user defined data type that can store related information about an entity.

* Syntax:

```
struct structure_name
{
    data type var_name 1;
    :
    data type var_name 2;
};
```

* Initialization of structure is done after declaration

```
struct structure_name structure_variable_name;
```

For example:

```
struct person
{
    char name[10];
    int age;
    float salary;
};
```

→ name of the structure

} members of the structure

```
struct person P;
```

→ variable of structure

⇒ Self Referential Structures:

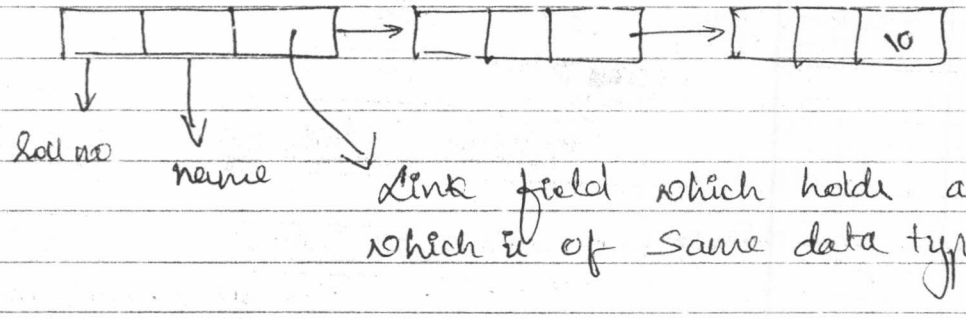
* Structure that contains a pointer to a structure of the same type
* Can be linked together to form useful data structures such as list, queues, stacks & trees.

* In the self-referential structure one of the elements of the structure variable is able to reference another subsequent structure variable of the same type.

For example:

```
Struct node
{
    int roll_no;
    char name[25];
    struct node *link;
};
```

* Here link is field or member of structure, which will be pointing to same data type



Union and its Defination

* Union is a derived data type like structure. So union can also be treated as a collection of variables under a single name.
* All these variables may contain data items of similar or dissimilar data types.
* Each variable in the union is called a member or field.

* The syntax, declaration and use of union is similar to the structures but its functionality is totally different.

Syntax:

```
union tag-name
{
    type member1;
    type member2;
    :
}
```

For example:

```
union item
{
    int i;
    double d;
    char c;
}
union item x;
```

Note: defination of structure and union does not allocate memory

* Once the variable x is declared memory will be allocated.

* To access the members of union, the same syntax used to access various members of structure can be used.
* By using dot operator, we can access various members. i.e

```
x.i
x.d
x.c
```

* The variable x is associated with union can also be declared as a pointer i.e

```
union item *x;
```

* For the Pointer variables, the indirection operator * and dot operator or '->' can be used to access the member. i.e

```
x->i      or  (*x).i
x->d          (*x).d
x->c          (*x).c
```

/* Program to access various members of a union */

```
#include<stdio.h>
void main()
{
    union Student
    {
        int marks;
        char grade;
        float per;
    };
    union Student x;
```

⇒ Self Referential Structures:

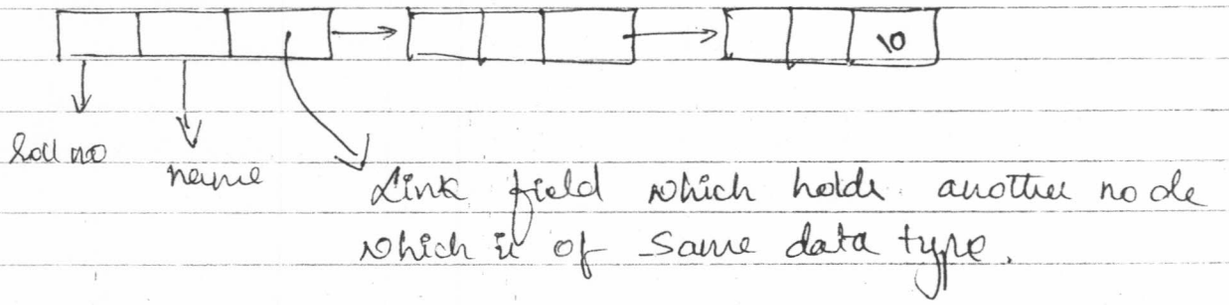
* Structure that contains a pointer to a structure of the same type
* Can be linked together to form useful data structures such as list, queues, stacks & trees.

* In the self-referential structure one of the elements of the structure variable is able to reference another subsequent structure variable of the same type.

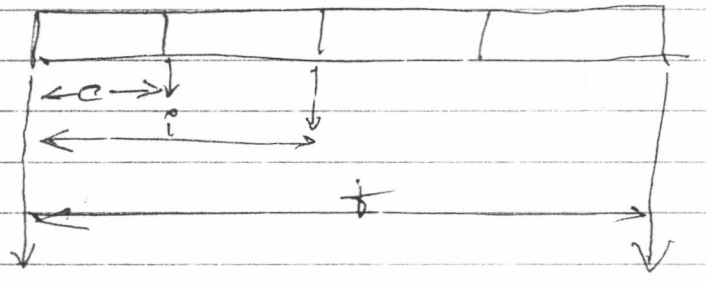
For example:

```
Struct node
{
    int roll_no;
    char name[25];
    struct node *link;
};
```

* Here link is field or member of structure, which will be pointing to same data type



* Suppose int occupy 2 bytes, char occupy 1 byte, float occupy 4 bytes. In this example, the member d occupies more memory and so 4 bytes are allocated for the variable X.

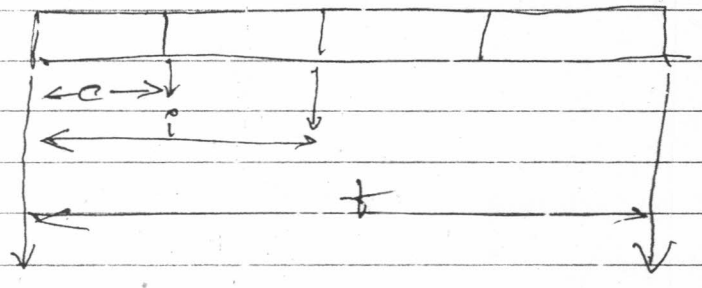


* In this, memory will be shared by all the variables.
* member which has highest size, will be allocated with memory.

Difference Between Structure and Union

STRUCTURE	UNION
1) Keyword struct is used to define structure	1) Keyword Union is used to define a union
2) When a variable is associated with a structure, the compiler allocates memory to each of the variable.	2) When a variable associates with a union the compiler allocates memory by considering the size of the largest memory.
3) The size of structure will be greater than or equal to the sum of sizes of its members	3) The size of union will be equal ~~than~~ to size of largest member.
4) Each member with in a structure is assigned unique storage area of locations	4) Memory allocated is shared by individual members.
5) The address of each member will be in ascending order. This indicates that memory for each member will start at different offset values.	5) The address is same for all the members of a union. every member begins at different offset values.
6) Individual member can be accessed at a time	6) Only one member can be accessed

* Suppose int occupy 2 bytes, char occupy 1 byte, float occupy 4 bytes. In this example, the member d occupies more memory and so 4 bytes are allocated for the variable x.



* In this, memory will be shared by all the variables.
* member which has highest size, will be allocated with memory.

Date
Page

Pointers

→ A pointer is a variable that contains address of another variable.

→ The data is stored in the memory at specific addresses. Using pointers we can manipulate data.

Advantages and Disadvantages of pointers :

Advantages :

* Data in one function can be modified by other function by passing the addresses
* Data can be accessed faster
* More than one value can be returned from a function using pointers.
* supports dynamic allocation of memory.

Disadvantages

* Un-initialized pointers or pointers containing invalid addresses can cause system crash
* Incorrect usage of pointers may cause bugs which are difficult to identify.

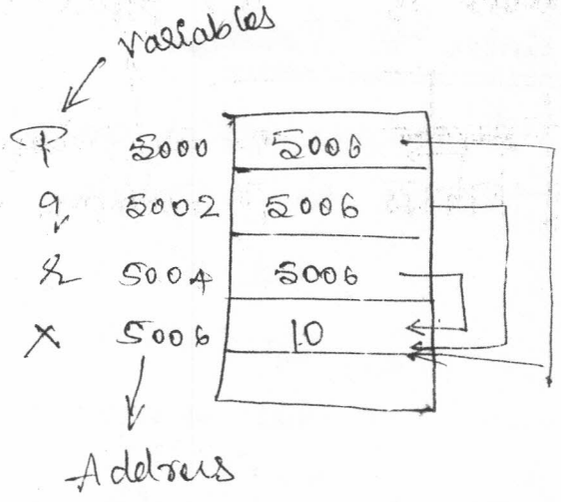
* Here the variable x is declared as integer data variable. Since p is a pointer variable of type integer, it should contain address of integer variable.
* Using the statement p = &x the valid address is stored in the pointer variable and hence the pointer variable p is initialized.

⇒ Accessing Variables through Pointers

* The value of a variable can be accessed using pointer variable using unary operator *. This operator is called indirection operator or dereferencing operator.

Consider the following declaration

```
x = 10;
p = &x;
q = &x;
p = &x;
```




```
Printf("&p = %d, p = %d, *p = %d\n", &p, p, *p);
```

output :- &p = 5000, p = 5006, *p = 10

```
Printf("&q = %d, q = %d, *q = %d", &q, q, *q);
```

output :- &q = 5002, q = 5006, *q = 10

```
/* Program to add two numbers using Pointers */
#include <stdio.h>
void main()
{
    int a, b, sum;
```

```
int *pa, *pb
pa = &a;
pb = &b;
printf("enter the value of a &b");
scanf("%d %d", &a, &b); /* scanf("%d %d", pa, pb);
sum = *pa + *pb;
printf("sum = %d \n", sum);
}
```

Pointer to Pointers

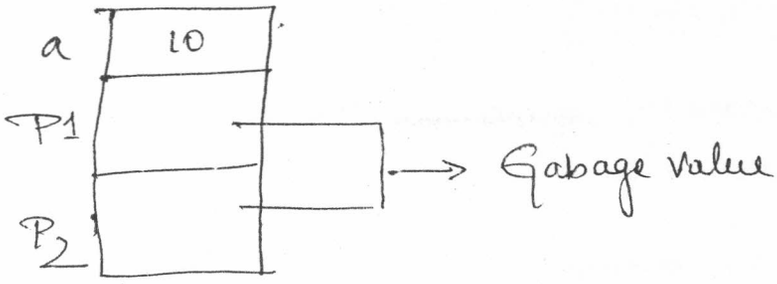
* A variable which contains address of a pointer variable is called pointer to a pointer. consider the following declaration.

```
int a;
int *p1;
int **p2;
```

* The first declaration instructs the compiler to allocate the memory for the variable in which integer data can be stored.
* The second declaration tells the compiler to allocate a memory for variable P1 in which address of an integer variable can be stored.
* The third declaration tells the compiler to allocate memory for the variable P_2 in which address of a pointer variable which points to an integer can be stored.
* Let

```
a=10;
P1 = &a;
P2 = &P1;
```

memory Organization after executing statement a = 10



After executing P1 = &a

Dynamic memory management functions

* The memory management functions that are used to allocate & dealloc memory are shown below

memory management functions
- → malloc
- → calloc
- → realloc
- → free

* Using the functions malloc, calloc and realloc memory space can be allocated where as using the function free() space can be deleted.

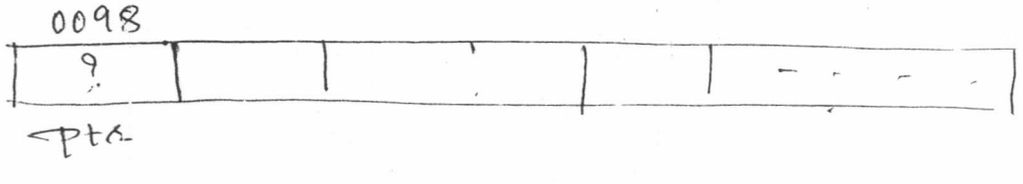
① malloc ( )

* function allows the program to allocate memory explicitly as & when required and the exact amount needed during execution.
* This function allocates a block of memory. The size of the block is number of bytes specified in the parameter.
* It is defined in stdlib.h header file. Syntax is

```
#include<stdlib.h>
:
ptr = (data type *) malloc(size);
```

where ptr is a pointer variable of type datatype
datatype can be any of the basic data type
size is the number of bytes required.

For example: int *ptr;
ptr = (int *) malloc (sizeof(int));

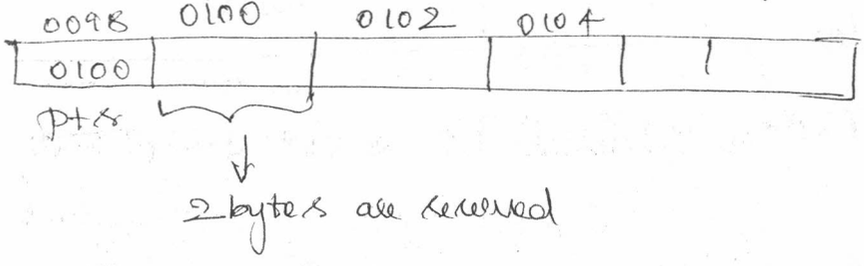


execute the statement

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

Since 2 bytes of free memory space is available, the function malloc allocates a block of 2 bytes of memory and returns the address of the first byte.

The returned address is stored in the pointer.



Example:

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

void main()
{
    int n, *p

    printf("enter the number of elements\n");
    scanf("%d", &n);
    ptr = (int *) malloc(sizeof(int)* n);
    if (ptr==NULL)
    {
        printf("Insufficient memory");
        return;
    }
    printf("enter N elements\n");
    for (i=0; i<n; i++)
        scanf("%d", ptr+i);
    printf("The given elements are\n");
    for(i=0; i<n; i++)
        printf("%d", *(ptr+i));
}
```

- Calloc

* Calloc stands for contiguous allocation of multiple blocks and is mainly used to allocate memory for arrays.
* The number of blocks determined by n. ~~The number of blocks~~ The size of each block is equal to the number of bytes specified in the parameter i.e size.
* The total number of bytes allocated is n * size and all bytes will be intialized to 0.
* The syntax:

```
# include<stdlib.h>
.....
.....
ptr = (data type *) calloc(n, size);
```

where

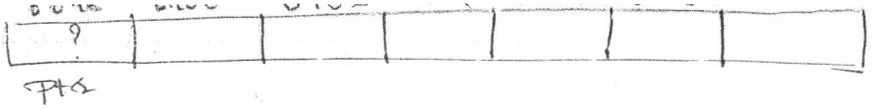
* ptr is pointer variable of type data-type
* data-type can be any of the basic data type or user defined data type.
* n is the number of blocks to be allocated
* size is the number of bytes in each block.

* On successful allocation, the function return the address of first byte of allocated memory. Since the address is returned, the return type is a void pointer, By type casting appropriately we can use it to store integer float etc.

* If specified size of memory is not available, the condition is called overflow of memory. In such case, the function returns NULL.

For example: int *ptr;

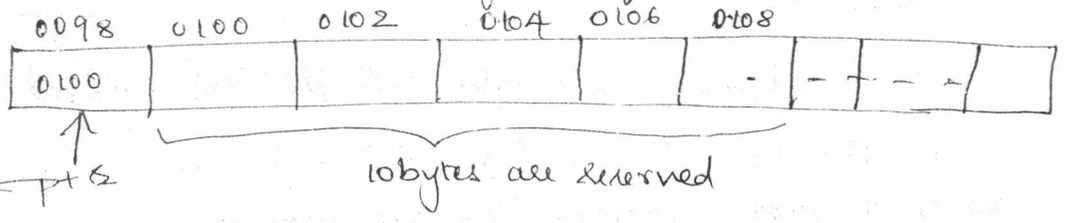
```
ptr = (int *) calloc( 5, sizeof(int));
```



execute the statement

```
ptr=(int *) calloc(5, sizeof(int));
```

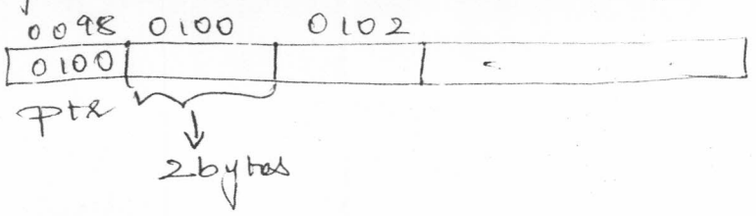
Since 5*2=10 bytes of free memory space is available, the function calloc(), allocates 5 blocks of 2 bytes each, initialize each byte to 0 and returns the address of the first byte.




```
/*program to accept n number of elements using dynamic arrays.
#include<stdio.h>
#include <stdlib.h>
int main()
{
    int *a, i, j, n;
  printf("enter the number of elements\n");
  scanf("%d", &n);
      a = (int *) malloc (n, sizeof(int));
      if (a==NULL)
      {
          printf("Insufficient memory\n");
          return;
      }

  printf("enter the array elements");
  for(i=0; i<n; i++)
          scanf("%d", &a[i]);
  printf("array elements are ");
  for(i=0; i<n; i++)
          printf("%d", a[i]);
}
```

Here ptr is allocated with 2 bytes of memory



* now if we want to extend memory, use realloc function

```
ptr = (int *) realloc(ptr, 10);
```

```
/* Program to show usage of realloc() function */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void main()
{
    char *str;
    str = (char *) malloc(10);
    strcpy(str, "computer");
    str = (char *) realloc(str, 40);
    strcpy(str, "computer Science & Engineering");
}
```

free

* This function is used to de-allocate the allocated block of memory which is allocated by using the functions calloc(), malloc() & realloc().

* It is responsibility of a programmer to de-allocate memory whenever it is not required by the program and initialize ptr to NULL

realloc

* Sometimes, the allocated memory may not be sufficient and we may require additional memory space. In another situation, where allocated memory may be much larger and we want to reduce.
* In both the situations, the size of allocated memory can be changed using realloc() and process is called reallocation of memory.
* realloc() changes the size of the block by extendenting or deleting the memory at the end of the block.
* If the existing memory can be extended, ptr value will not be changed.
* If the memory can not be ~~changed~~ extended, the function allocates completely new block.

Syntax: #include<stdlib.h>

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

where

ptr is a pointer to block of memory which is allocated previously

Size is new size of block.

* On successful allocation, the function returns the address of first byte of allocated memory.
* If memory can not be allocated, function returns NULL

For example: int *ptr;

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