DATA STRUCTURES USING C

NOTES

Prepared by

Prof. Prabhu Kichadi, M. Tech
9880437187

UNIT – I

CONTENTS

Introduction to Data structures: Definition, Types of data structures, Primitive and Non primitive data structures, Linear and non-linear data structures, Operations on data structures, Dynamic memory allocation, static and dynamic memory allocation, memory allocation and de-allocation functions, malloc, calloc, realloc, and free.
Pointers in C: understanding pointers – Declaring and initializing pointers, accessing address and value of variables using pointers, pointers and arrays, pointer arithmetic, advantages and disadvantages of using pointers.

Introduction to Data structures:

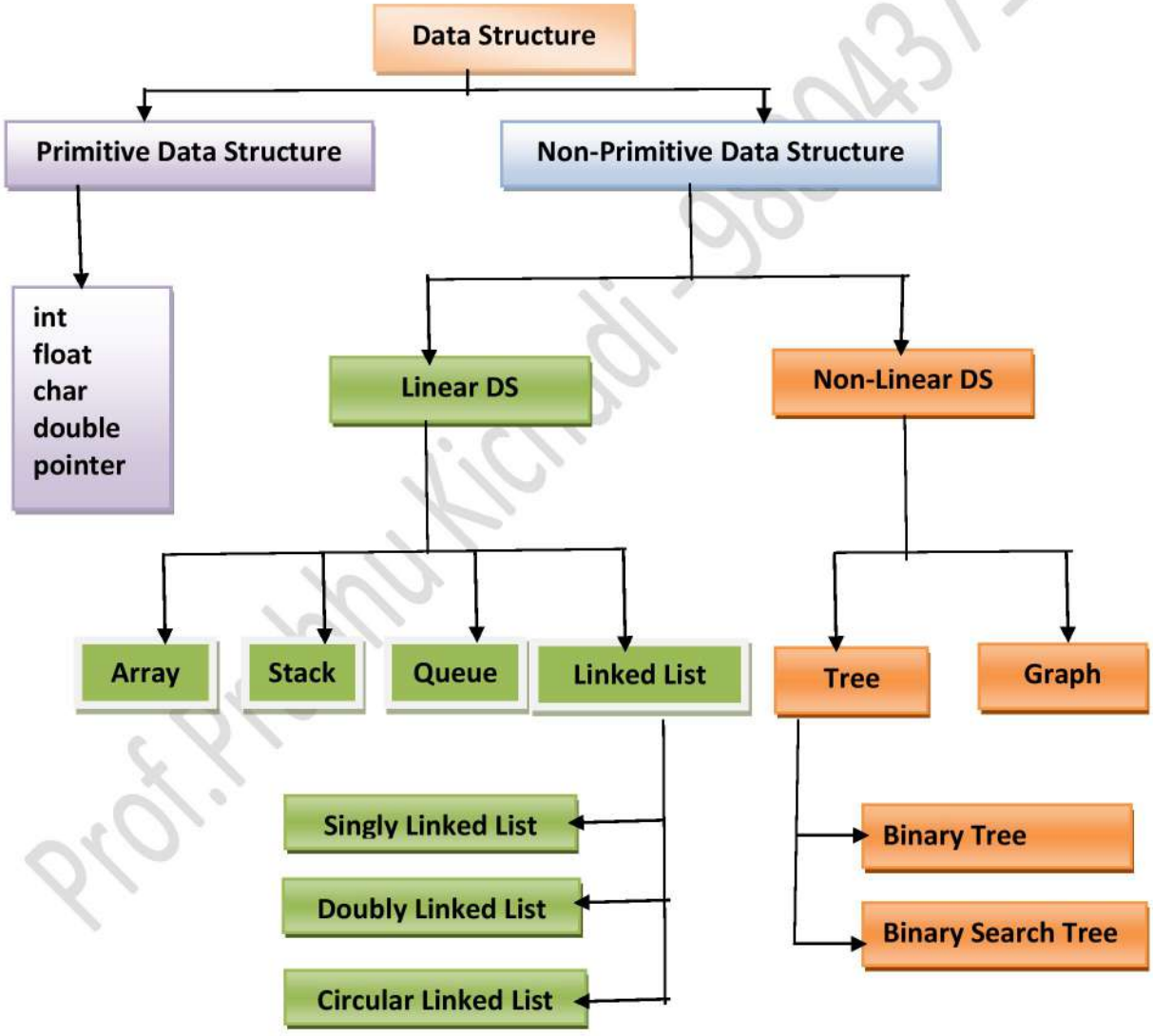
Data Structures: *Data Structures are the programmatic way of storing data into memory so that data can be used efficiently.*

OR

Data structures is the way of organizing the data-items into the memory, so that it can accessed in efficient manner.

Ex: Array, List, Stack, Queue, Tree, Graphs, Files, Pointers, etc.

Classification of data structures:



Primitive data structures: *These are basic structures and are directly operated upon by the machine instructions.*

- These are atomic in nature we cannot hold more than one data-items in those variables & we cannot decompose into more atomic.

- In C primitive data structures are mainly built-in data structures like Integral(int), Floating Point (float, double), Character(char) and Pointers (Derived).

Integral DS: Represents any single integral value within the range of respective data type.
Keyword: int
Size: 4 Bytes
Ex: int x = 121;

Floating Point DS: Any atomic numerical data in the form of fractional value.
Keyword: float, double
Size: 8 Bytes
Ex: float x = 12.12;
Character DS: Any atomic data represented in a single quote.
Keyword: char
Size: 1 Bytes
Ex: char ch = 'A';

Non-Primitive data structures: *These are the complex and sophisticated data structures, derived from the primitive data structure and also called user defined data structures.*

We can group homogeneous and/or heterogeneous data-items and relationship between each other.

Ex: Array, List, Files, Tree, Graph etc.

Linear Data Structures: *Data-items stored in a sequential manner or linear order. That is data items stored one after the other memory locations. Such type of data structures are Linear Data Structures.*

Ex: Arrays, Stack, Queues, Linked List.

1. Arrays: In array data-elements are homogeneous and stored in contiguous memory locations, each element is accessed using its index.

2. Stack: In stack data elements are stored continuous memory locations, insertion & deletion happens from the same end, it follows LIFO order.

3. Queue: It is linear data structure it follows FIFO order. Insertion & deletion happens from two separate ends.

4. Linked List: A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. A linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.

Non-Linear Data Structures: *Data-items stored in a random manner or nonlinear order. That is data items are not stored in contiguous memory locations.*

Ex: Trees, Graphs.

Operations on Data Structures:

1. Searching
2. Sorting
3. Insertion
4. Deletion
5. Updation
6. Traversing

Most of the data structures are helpful for storing huge amount of data, occupies memory. But we need to use/fetch the stored data from memory. Based on application requirement we do many operations on data structures, those are,

1. Searching: Searching means to find a particular **data element** (Key Element) from the set of elements in the given data-structure. It is considered as successful when the required element is found.

2. Sorting: Re arranging data-elements in a preferred order or sequence.so that data elements can be easily accessed.
If data elements are numerals, then order can be ascending or descending. If data elements are char, then order can be ascending or descending.

3. Insertion: Adding or creating new data-item into a data structure.

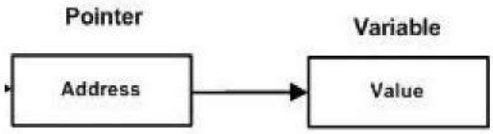
4. Deletion: Removing or deleting existing data-item from a data structure.

5. Updation: Modifying or changing existing data-item value from a data structure.

6. Traversing: Traversing a Data Structure means visiting all the data element exactly once.

Pointers:

Pointer is a variable which stores address of another variable of the same pointer type.



This variable can be of type int, char, array, function, or any other pointer. The size of the pointer depends on the architecture. However, in 32-bit architecture the size of a pointer is 2 bytes.

- Every variable you declare which has a memory location address. Which is represented by using & operator (Address/Reference operator).

Declarations of pointers:

Syntax:

datatype *identifier; **OR** **datatype * pointername;**

Ex: int *p;

Datatype → it must be a valid data type from C, it can be user defined data type also.
* → Asterisk/De-reference operator/Indirection Operator/Pointer Operator.
Identifier → It is the valid pointer variable name; it must be a valid identifier.

Ex: int *ptr;

ptr is a pointer variable it can store only store an address of another int type variable.

```
int *p1;                    //p1 is a int-type pointer
char *c1;                   //c1 is a char-type pointer
float * f1;                 //f1 is float-type pointer
double *d1;                 //d1 is a double-type pointer
struct student *s1;         // s1 is pointer student type variable
struct employee *e1;        // e1 is pointer to employee-type
```

Initialization of pointers:

- After declaring a pointer variable, we need to initialize the pointer.

Syntax:

```
pointer = & variable;
```

& - Address operator is used to get the address of a variable.
* - dereference operator is used to get the values at the address pointed by the pointer.

Ex:

```
int a=10;
int *p;
p = &a;
```

Accessing a variable through its pointer:

Operators used in Pointer's concept:

& - Address Operator **or** Reference operator -> Its in hexadecimal format.
* - Indirection Operator **or** dereference operator or pointer operator.

How to use pointers

1. Define a pointer variable.
2. Declare a variable and assign value of the same-type.
3. Assign the address of a variable to a pointer variable.
4. Access the value at the address through pointer.

Note:

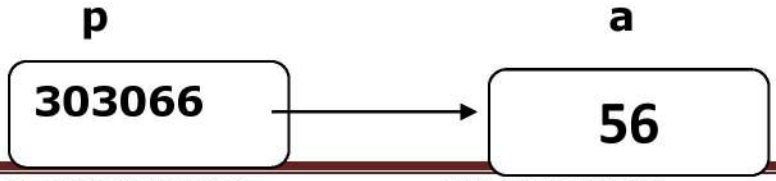
Any type of Pointer size will remain same based on compiler type; it is always 4 bytes. Address of the variable is always positive and it is never known by the programmer.

Ex1:

```
int *p;     /* Pointer Declaration */
int a;      /* Actual variable Declaration */

a = 56;     /* Initialize variable */
p = &a;     /* Store Address of variable in Pointer */

printf("\nAddress of a = %d", &a);          /* Address of a variable */
printf("\nValue at the address = %d", *p);  /* Value at the Address */
```



&p-> 2020 &a-> 303066

```
printf("a = %d", a);        => 56
printf("p = %u", p);        => 303066
printf("&a = %u", &a);      => 303066
printf("*p = %d", *p);      => 56
printf("&p = %u", &p);      => 2020
```

Advantages of Pointers:

- Memory can be allocated at Run-time or dynamically.
- Direct access of memory.
- helps for manual memory management and efficient use of memory.
- Give ease access of array, structure elements.
- Pointers provide a way to return more than one value from functions.
- Affection on actual parameters if we use pointer access from other functions.
- Pointers provide a way to return more than one value to the functions
- Reduces the storage space and complexity of the program
- Reduces the execution time of the program.
- Reduces the execution time of the program
- Provides an alternate way to access array elements
- Pointers can be used to pass information back and forth between the calling function and called function.
- Pointers allows us to perform dynamic memory allocation and deallocation.
- Pointers helps us to build complex data structures like linked list, stack, queues, trees, graphs etc.
- Pointers allows us to resize the dynamically allocated memory block.
- Addresses of objects can be extracted using pointers

Disadvantages of Pointers:

- Uninitialized pointers might cause segmentation fault.
- Dynamically allocated block needs to be freed explicitly. Otherwise, it would lead to memory leak.
- Pointers are slower than normal variables.
- If pointers are updated with incorrect values, it might lead to memory corruption

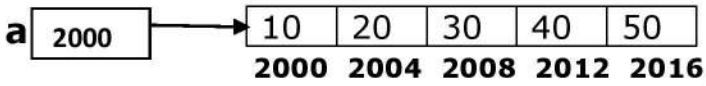
Pointer & Array:

- Array is derived & non-primitive linear data structure, that holds homogeneous data-elements in contiguous memory locations.
- In C, array name holds base address of an entire array but it cannot be accessed as like pointer.

Ex: **int a[5];**

a is a pointer of int type that points 5 integers memory locations, it holds base address of an array ie 2000 is the base address (starting address).

- Elements of an array can be accessed using pointer concept also.



Ex:

```
#include<stdio.h>
int main()
{
   int a[ ] = {10,20,30,40,50};
   int *p, i;

   p = a;

   for(i=0; i<5; i++)
   {
      printf("\na[%d] = %d",i, *p);
      p++;
   }
   return 0;
}
```

Operations on Pointers: [IMP]

Pointer Arithmetic:

- C supports pointer arithmetic for only four operators **++, --, +, -.**
- Since pointers hold addresses, we cannot perform operations on addresses but we can do on values pointed by the pointers.
- C support, we can move pointer forward & backward using increment & decrement operators.

1. *Addition of integers to pointers of the same type is allowed.*

Ex:
```
int *p;
p = p + 2;
```

2. *A pointer can be decremented or incremented.* Ex: p++, p--.
3. *Subtraction of two pointers of the same type is allowed. But higher-lower(addresses).*

Ex: p1-p2; (p1->2000, p2->1996)

Ex:

```
#include<stdio.h>
int main()
{
   int a[] = {10,20,30,40,50};
   int *p;

   p = a;

   printf("First Element : %d", *p);
   //Move the pointer to next element
   p++;
   printf("\nSecond Element : %d", *p);
   p = p + 3;
   printf("\nFifth Element : %d", *p);

   getch();
   return 0;
}
```

Output:

```
First Element : 10
Second Element : 20
Fifth Element : 50
```


Pointers & Functions:

- It is possible to design functions with pointers.
- We can define function parameters as pointers.
- Then we need to pass addresses of variables to functions.

Programs on pointer & function: (Pass by value and Pass by reference)

- In this example swap() function is defined with 2 pointer parameters of int type, hence if we need to pass addresses of 2 int variables to this function. Those pointers now have access to actual parameters from different section.

1. Program to swap two numbers using function & pointer.

```
#include<stdio.h>
void swap(int *p1, int *p2);

int main()
{
   int a, b;

   a = 10;
   b = 20;

   printf("\nBefore Swap : a = %d\t b = %d",a,b);

   swap(&a, &b);  /* Passing Address of a & b */

   printf("\n\nAfter Swap : a = %d\t b = %d",a,b);

   getch();
   return 0;
}

void swap(int *p1, int *p2)  /* Address will assign to pointers p1 & p2  */
{
   int temp;

    temp = *p1;
   *p1 = *p2;
   *p2 = temp;
}
```

Output:

Before Swap : a = 10 b = 20

After Swap : a = 20 b = 10

Pointer & Strings:

- String is group of characters enclosed in a double quote and ends with a null character.

- in C it is allowed we can create a pointer of type char, which is used to hold address of any char variable, or char array address.

- Char array name holds base address of a string, so we can store base address to a char pointer.

Program1:

```
#include<stdio.h>
int main()
{
   char name[25];
   char *temp;

   //Storing name array base address to char pointer.
   temp = name;

   printf("\nEnter a name : ");
   fflush(stdin);
   gets(name);

   while(*temp != '\0')
   {
      putch(*temp);      //Accessing char value through pointer temp
      temp++;            //Moving temp pointer to next memory location
   }

   getch();
   return 0;
}
```

Output:

```
Enter a name : Raju
Raju
```

Program2: String concatenation using pointers.

```
#include<stdio.h>
char* concat(char *p1, char *p2);
```

```
int main()
{
   char s1[30],s2[20];
   char *temp;

   printf("\nEnter First String : ");
   fflush(stdin);
   gets(s1);

   printf("\nEnter Second String : ");
   fflush(stdin);
   gets(s2);

   temp = concat(s1,s2);

   printf("\nConcatenated String : %s",temp);

   getch();
   return 0;
}

char* concat(char *p1, char *p2)
{
   char *temp=p1;
   while(*p1 != '\0')
   {
      p1++;
   }
   while(*p2 != '\0')
   {
      *p1 = *p2;
      p1++;
      p2++;
   }
   *p1 = '\0';

   return temp;
}
```

Output:

```
Enter First String : Good

Enter Second String : Morning
```

Concatenated String : Good Morning

Pointer & Structure:

Accessing Structure Members with Pointer

- To access members of structure using the structure variable, we used the dot . operator.
- But when we have a pointer of structure type, we use arrow -> to access structure members.
- We can create a structure variable, then we need to assign address of struct variable to struct pointer variable of the same type.

Ex program1: Accessing members of the structure using pointer.

```
#include <stdio.h>
struct Employee
{
   char empName[25];
   int empId;
   float empSal;
};

int main()
{
   struct Employee e;
   struct Employee *ePtr;

   // Assigning address of e structure to ePtr.
   ePtr = &e;

   printf("\nEnter Employee Name : ");
   fflush(stdin);
   gets(ePtr->empName);
```

```
    printf("\nEnter Employee ID : ");
    scanf("%d", &ePtr->empId);

    printf("\nEnter Employee Salary : ");
    scanf("%f", &ePtr->empSal);


    //Accessing members of structure using pointer ePtr & Arrow operator

    printf("\n\nEmployee Details using Pointer ...\n");
    printf("\nEmployee Name : %s", ePtr->empName);
    printf("\nEmployee Id : %d",ePtr->empId);
    printf("\nEmployee Salary : %f",ePtr->empSal);

    getch();
    return 0;
}
```

Output:

```
Enter Employee Name : Raju
Enter Employee ID : 101
Enter Employee Salary : 350000

Employee Details using Pointer ...
Employee Name : Raju
Employee Id : 101
Employee Salary : 350000.000000
```

In this example,

&e.empName is equivalent to ePtr->emp_name ➔ (*ePtr).empname.
&e.empId is equivalent to &ePtr->empId
e.empId is equivalent to ePtr->empId is equivalent to (*ePtr).empId

Pointer to Pointer (Double Pointer)

Pointer to pointer is a double pointer, A pointer variable stores an address of another pointer variable of the same type.

- A pointer to a pointer is a form of multiple indirections, or a chain of pointers. Normally, a pointer contains the address of a variable.
- When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.

Double pointer declaration syntax:

```
datatype **variablename;
```

Ex: int **p;

Example Program:

```
#include<stdio.h>
int main()
{
   int *p1;
   int **p2;
   int a = 22;
```

```
    p1 = &a;
    p2 = &p1;

    printf("\nValue of a = %d",a);
    printf("\nValue Pointed by p1 = %d", *p1);
    printf("\nValue Pointed by p2 = %d", **p2);

    getch();
    return 0;
}
```

Output:

Value of a = 22
Value Pointed by p1 = 22
Value Pointed by p2 = 22

Memory Allocation:
Static Memory Allocation & Dynamic Memory Allocation

Static Memory Allocation: *Allocation of memory at compile time and size cannot be varied during run time. Size is fixed when it is declared. Size is known in advance.*
Ex: int a; //4 bytes of memory

Dynamic Memory Allocation: *Allocation of memory at run time and size can be varied as and when it is required. Size is known at run-time.*

Differences between static memory allocation & dynamic memory allocation [IMP].

Static Memory Allocation	Dynamic Memory Allocation
1. Memory is allocated during compile time	Memory is allocated during run-time
2. Size is fixed	Size can be changed
3. Size is known in advance	Size is known at run-time
4. Use when memory is sufficient	Use when memory is not sufficient
5. Execution is faster because memory is allocated at compile time.	Execution is slower because memory has to allocated at run time
6. Memory is allocated on stack segment	Memory is allocated on Heap segment
7. No readymade functions available.	Readymade functions for dynamic memory allocation – malloc () , calloc()
Ex: primitive variable declarations	Ex: arrays & structures created by malloc() & calloc()

Memory allocation functions

There are library functions in ***<stdlib.h>*** header files which are used for dynamic memory allocation. Those are,
These are used for dynamic array creation, dynamic structures, linked list creations.

Memory allocation functions

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

Memory deallocation Functions

4. free( )

1. malloc( )

Syntax:

```
void *   malloc(int bytes);
```

or

```
ptr = (datatype *) malloc(size);
```

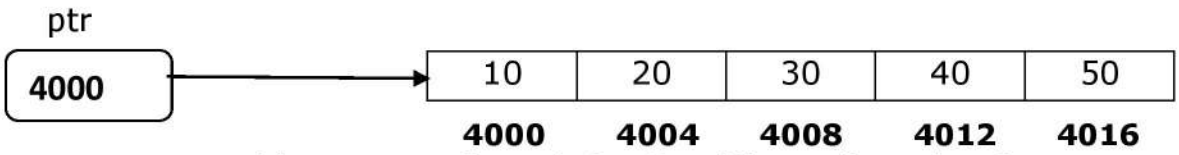
ptr - is a variable of type datatype.
datatype - can be any primitive or user defined data type.
size - is the total number of bytes required.

- This function allocates & reserves a block of memory specified in bytes at run time.
- It returns base address of allocated memory if memory is available, otherwise it returns NULL value if memory is not available.

Ex:

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

- In this example, sizeof(int) returns 4 bytes, * 5 = 20 bytes memory will be reserved. The starting address will be stored in ptr. Now ptr holds base address.



- We can access address using (ptr+i) **OR** &ptr[i], i is the index for address.
- We can access values using *(ptr+i) **OR** ptr[i], i is the index for address.

Ex:

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
   int *ptr,i,n;

   printf("\nEnter Number of Elements : ");
   scanf("%d", &n);

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

   for(i=0; i<n; i++)
   {
      printf("\nEnter Element : ");
      scanf("%d", (ptr+i));
   }

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

   getch();
   return 0;
}
```

Output:

```
Enter Number of Elements : 5
Enter Element : 10
Enter Element : 20
Enter Element : 30
Enter Element : 40
Enter Element : 50
```

Displaying elements : 10 20 30 40 50

2. calloc( )

Syntax:

```
void *  calloc(int num, int size);
```

or

```
ptr = (datatype *) calloc(int num, int size);
```

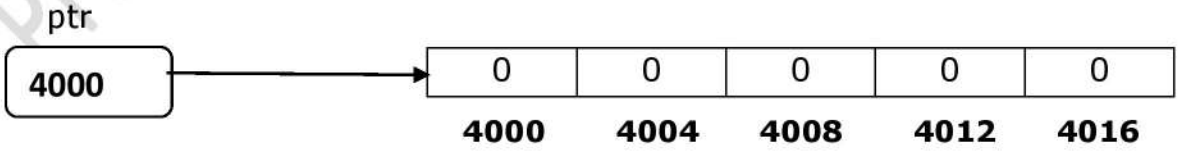
ptr - is a variable of type datatype.
datatype - can be any primitive or user defined data type.
size - It is the data type size in bytes.
num – It is number of blocks to be allocated.

- This function allocates & reserves multiple blocks of memory specified in bytes at run time and **initializes all locations to default values**(0 for int, 0.0 for float, etc)
- It returns base address of allocated memory if memory is available, otherwise it returns ***NULL*** value if memory is not available.

Ex:

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

- In this example, sizeof(int) returns 4 bytes, * 5 = 20 bytes memory will be reserved. The starting address will be stored in ptr. Now ptr holds base address.



We can access address using (ptr+i) **OR** &ptr[i], i is the index for address.
- We can access values using *(ptr+i) **OR** ptr[i], i is the index for address.

Ex:

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
   float *ptr;
   int i,n;

   printf("\nEnter Number of Salaries : ");
   scanf("%d", &n);

   ptr = (float *)calloc(n, sizeof(float));

   for(i=0; i<n; i++)
   {
      printf("\nEnter Salary : ");
      scanf("%f", (ptr+i));
   }

   printf("\nDisplaying Salaries : ");
   for(i=0; i<n; i++)
   {
      printf("%f\t", *(ptr+i));
   }

   getch();
   return 0;
}
```

Output

```
Enter Number of Salaries : 5
Enter Salary : 25000.00
Enter Salary : 350000.50
Enter Salary : 26000.00
Enter Salary : 16000.11
Enter Salary : 230000.25

Displaying Salaries : 25000.000000      350000.500000   26000.000000
16000.110352    230000.250000
```

3. realloc( )

Syntax:

```
void *   realloc(void *, int size);
```

or

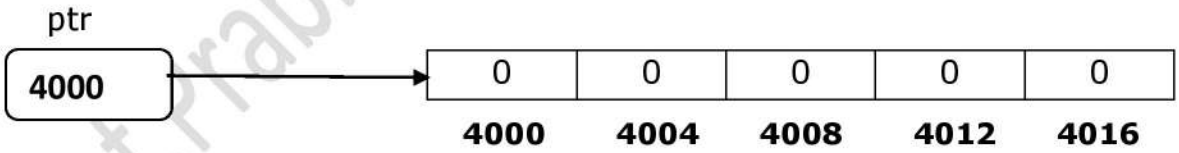
```
ptr = (datatype *) realloc(ptr, int size);
```

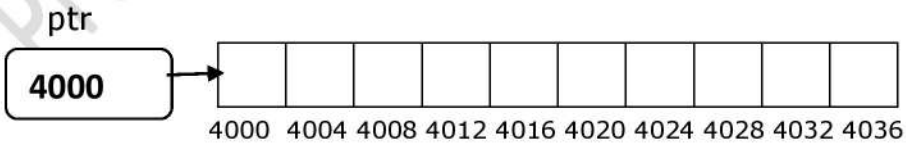
ptr - is a pointer variable of type datatype.
datatype - can be any primitive or user defined data type.
size - It is the data type size in bytes.
num – It is number of blocks to be allocated.
ptr inside – ptr is pointer which allocated memory using malloc or calloc previously.

- realloc() function is used to extend or delete already allocated memory using malloc or calloc.

Ex:

```
int *ptr, oldsz,newsz;
oldsize = 5;
ptr = (int *)malloc(oldsz*sizeof(int));  // allocates 20 bytes
ptr = (int *)realloc(ptr,newsz); //extends 20 bytes to 40 bytes.
```





Ex:

```
#include<stdio.h>
#include<stdlib.h>
int main()
```

```
{
   int *ptr,i,oldsz,newsz;

   printf("\nEnter Number of Elements : ");
   scanf("%d", &oldsz);

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

   printf("\nEnter New Size(Number of elements) : ");
   scanf("%d", &newsz);

   ptr = (int *)realloc(ptr, newsz);

   for(i=0; i<newsz; i++)
   {
      printf("\nEnter Element : ");
      scanf("%d", (ptr+i));
   }

   printf("\nDisplaying elements : ");
   for(i=0; i<newsz; i++)
   {
      printf("%d\t", *(ptr+i));
   }

   getch();
   return 0;
}
```

Output

```
Enter Number of Elements : 5
Enter New Size(Number of elements) : 6

Enter Element : 1
Enter Element : 2
Enter Element : 3
Enter Element : 4
Enter Element : 5
Enter Element : 6

Displaying elements: 1   2      3      4      5      6
```

Memory deallocation functions:

4. free( ) :

Syntax:

```
void *  free(void *);
```

or

```
free(ptr);
```

free(fptr) is used to deallocate or free the memory which is previously reserved using malloc or calloc function.

Ex:

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

Differences between malloc and calloc [IMP].

Malloc	calloc
1.Allocates single & contiguous locations	Allocates multiple blocks
2.No default initialization - Garbage	Default initialization takes place.
3.More time efficiency	Less time efficiency
4. ptr = (datatype *)malloc(size)	Ptr = (datatype *)calloc(n, size)
5. used for arrays	Used for arrays & structures

Assignment Programs:

1. How dynamic arrays are created using malloc, calloc (program).
2. Write a c program to find largest number from a given dynamic array or use dynamic array concept.
3. Write a C program to print sum of all array elements using dynamic array.
4. Write a C program to read one dimensional array, print sum of all elements along with inputted array elements using Dynamic Memory Allocation.

1. **Write a C Program to demonstrate the Dynamic Memory Allocation for Structure by reading and printing n student details.**

Code:

```
#include<stdio.h>
#include<stdlib.h>
typedef struct student
{
   int id;
   char name[25];
}student;

int main()
{
   int n,i;
   student *s;

   printf("\nEnter Number of students : ");
   scanf("%d", &n);

   s = (student *)malloc(n*sizeof(student));

   for(i=0; i<n; i++)
   {
      printf("\nEnter Student Id : ");
      scanf("%d",&(s+i)->id);
      printf("\nEnter Student Name : ");
      fflush(stdin);
      gets((s+i)->name);
   }

   printf("\nStudent details are...\n");
   for(i=0; i<n; i++)
   {
      printf("\n***************\n");
      printf("\nId = %d", (s+i)->id);
      printf("\nName = %s", (s+i)->name);
   }

   getch();
   return 0;
}
```

Output

```
Enter Number of students : 2
```

```
Enter Student Id : 101
Enter Student Name : Rajesh

Enter Student Id : 102
Enter Student Name : Akash

Student details are...

***************

Id = 101
Name = Rajesh
***************

Id = 102
Name = Akash
```


QUESTION BANK

UNIT-I

Introduction to Data Structures and Pointers

2 Marks

1. Define data structure. List out its types.
2. Define pointer. How to initialize pointer.
3. List out primitive and non-primitive data structures.
4. What is pointer to pointer how to declare it.
5. What is dynamic and static memory allocation.
6. What are primitive and non-primitive data structures? Give examples.
7. List advantages of pointer.
8. State the use of calloc( ) function with syntax.
9. List applications of data structures.

5/10 Marks

10. What are operations on data structures? List out.
11. Explain with syntax malloc( ) , calloc( ), realloc( ) and free( ).
12. Write a C program to print sum of array elements using dynamic memory allocation.
13. What is dynamic memory allocation? Explain DMA functions.
14. Explain pointer and structure.
15. Discuss classification of data structures with example.
16. Explain dynamic memory allocation.
17. Differentiate static and dynamic memory allocation.
18. What are advantages and disadvantages of pointers.
19. Write a note on pointers and arrays.
20. Demonstrate call by reference and call by value with an example.
21. Write a program to implement dynamic array find smallest and largest element in an array.

Note: Refer the DS Lab journal programs on above concepts program may ask for 5 or 10 marks.

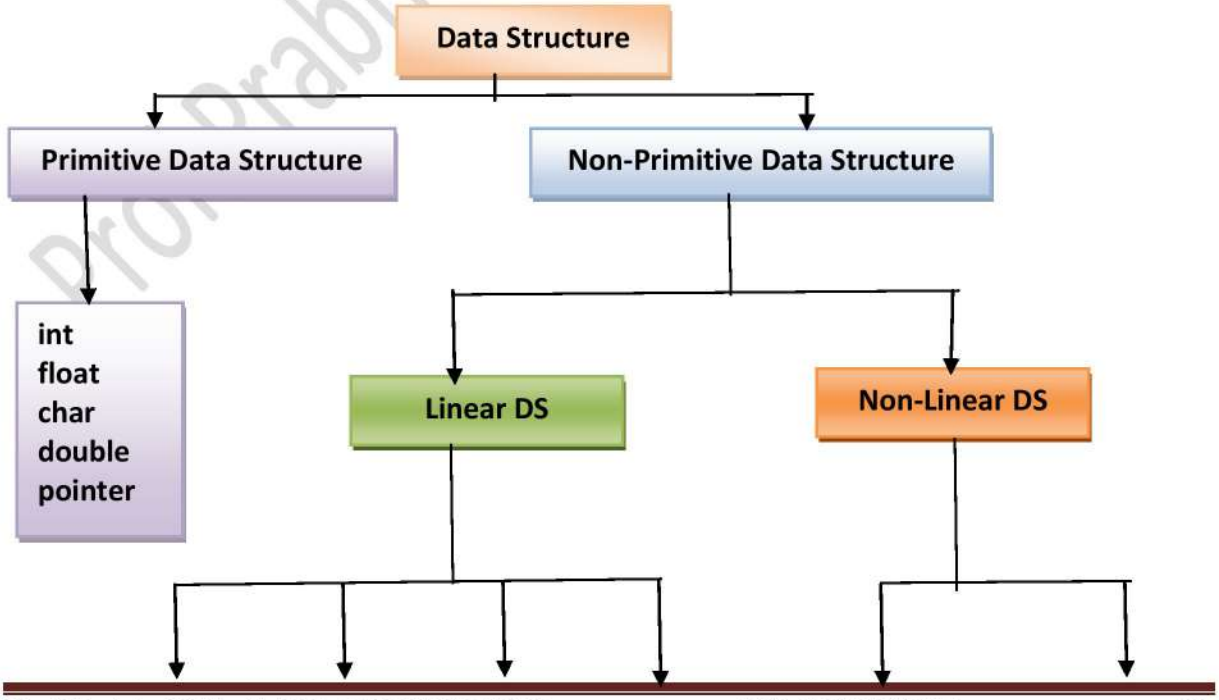
QP Drive Link:

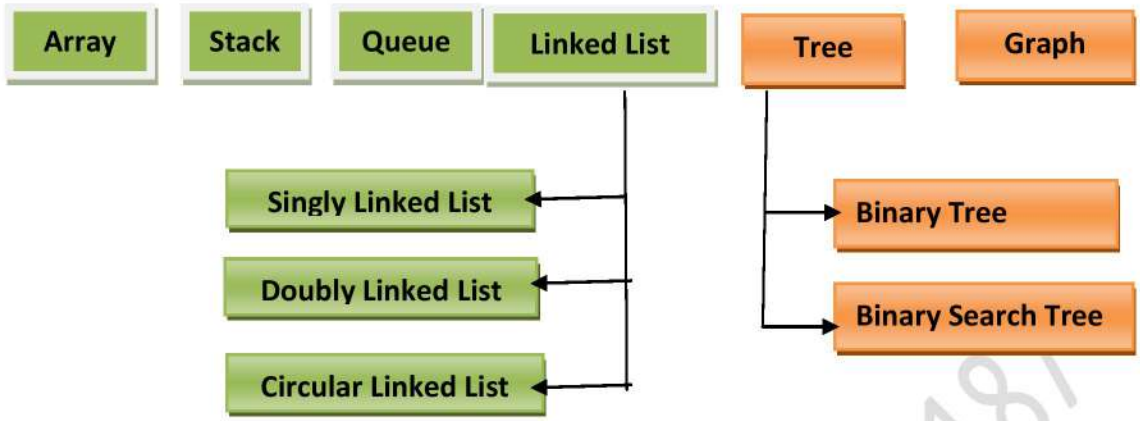
https://drive.google.com/drive/folders/1F0CEAtoFmji9Gx9snSGZYqEksw4VkjSW

1. Define data structure. List out its types.

:- *Data Structures are the programmatic way of storing data into memory so that data can be used efficiently.*

Classification of data structures:





2. Define pointer. How to initialize pointer.

Pointer is a variable which stores address of another variable of the same pointer type.
Declarations of pointers:

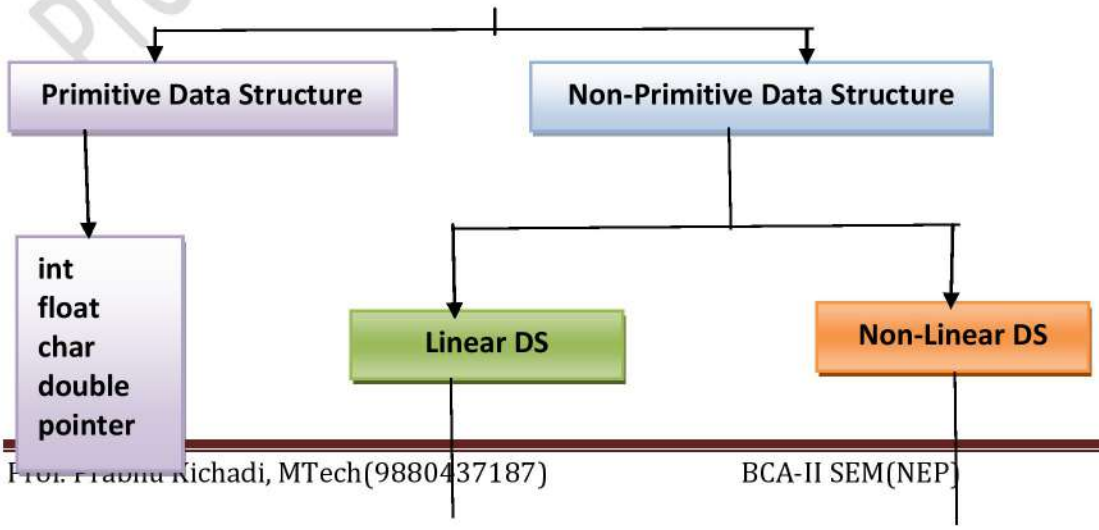
Syntax:

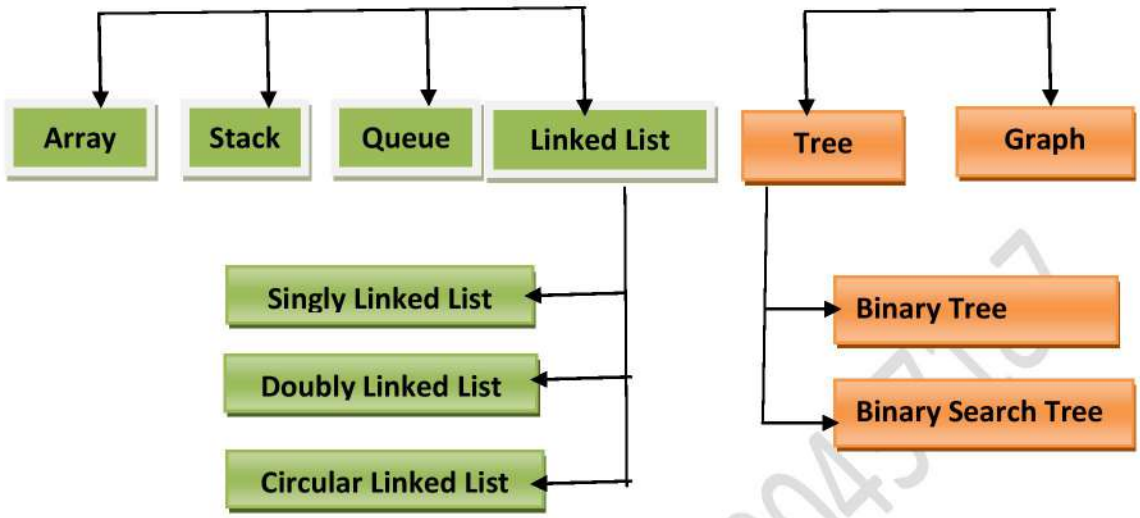
datatype *identifier; OR **datatype * pointername;**

Ex: int *p;

Datatype → it must be a valid data type from C, it can be user defined data type also.
* → Asterisk/De-reference operator/Indirection Operator/Pointer Operator.
Identifier → It is the valid pointer variable name; it must be a valid identifier.

3. List out primitive and non-primitive data structures.





4. What is pointer to pointer how to declare it.

Pointer is a variable which stores address of another variable of the same pointer type.

Declarations of pointers:

Syntax:

datatype *identifier; **OR** **datatype * pointername;**

Ex: int *p;

Datatype → it must be a valid data type from C, it can be user defined data type also.
* → Asterisk/De-reference operator/Indirection Operator/Pointer Operator.
Identifier → It is the valid pointer variable name; it must be a valid identifier.

Ex: int *ptr;

5. What is dynamic and static memory allocation.

Static memory allocation: Allocation of memory at compile time and size cannot be varied during run time. Size is fixed when it is declared. Size is known in advance.
Ex: int a; //4 bytes of memory

Dynamic memory Allocation: Allocation of memory at run time and size can be varied as and when it is required. Size is known at run-time.

6. What are primitive and non-primitive data structures? Give examples.

Primitive data structures: *These are basic structures and are directly operated upon by the machine instructions.*

- These are atomic in nature we cannot hold more than one data-items in those variables & we cannot decompose into more atomic.
- In C primitive data structures are mainly built-in data structures like Integral(int), Floating Point (float, double), Character(char) and Pointers (Derived).

Non-Primitive data structures: *These are the complex and sophisticated data structures, derived from the primitive data structure and also called user defined data structures.*

- We can group homogeneous and/or heterogeneous data-items and relationship between each other.
- **Ex:** Array, List, Files, Tree, Graph etc.

7. List advantages of pointer.

- Memory can be allocated at Run-time or dynamically.
- Direct access of memory.
- helps for manual memory management and efficient use of memory.
- Give ease access of array, structure elements.
- Pointers provide a way to return more than one value from functions.
- Affection on actual parameters if we use pointer access from other functions.

8. State the use of calloc() function with syntax.

This function allocates & reserves multiple blocks of memory specified in bytes at run time and **initializes all locations to default values**(0 for int, 0.0 for float, etc) and returns null if no memory available.

Syntax :

```
void *  calloc(int num, int size);
```

9. List applications of data structures.

Storing list of data elements belonging to same data type.

Auxiliary storage for other data structures.

Storage of binary tree elements of fixed count.
Storage of matrices.

5/10 Marks

1. What are operations on data structures? List out.

1. Searching
2. Sorting
3. Insertion
4. Deletion
5. Updation
6. Traversing

1. Searching: Searching means to find a particular data element (Key Element) from the set of elements in the given data-structure.
2. Sorting: Re arranging data-elements in a preferred order or sequence.so that data elements can be easily accessed.
3. Insertion: Adding or creating new data-item into a data structure.
4. Deletion: Removing or deleting existing data-item from a data structure.
5. Updation: Modifying or changing existing data-item value from a data structure.
6. Traversing: Traversing a Data Structure means visiting all the data element exactly once.

2. Explain with syntax malloc( ) , calloc( ), realloc( ) and free( ).

12. Write a C program to print sum of array elements using dynamic memory allocation.
13. What is dynamic memory allocation? Explain DMA functions.
14. Explain pointer and structure.
15. Discuss classification of data structures with example.
16. Explain dynamic memory allocation.
17. Differentiate static and dynamic memory allocation.
18. What are advantages and disadvantages of pointers.
19. Write a note on pointers and arrays.
20. Demonstrate call by reference and call by value with an example.
21. Write a program to implement dynamic array find smallest and largest element in an array.