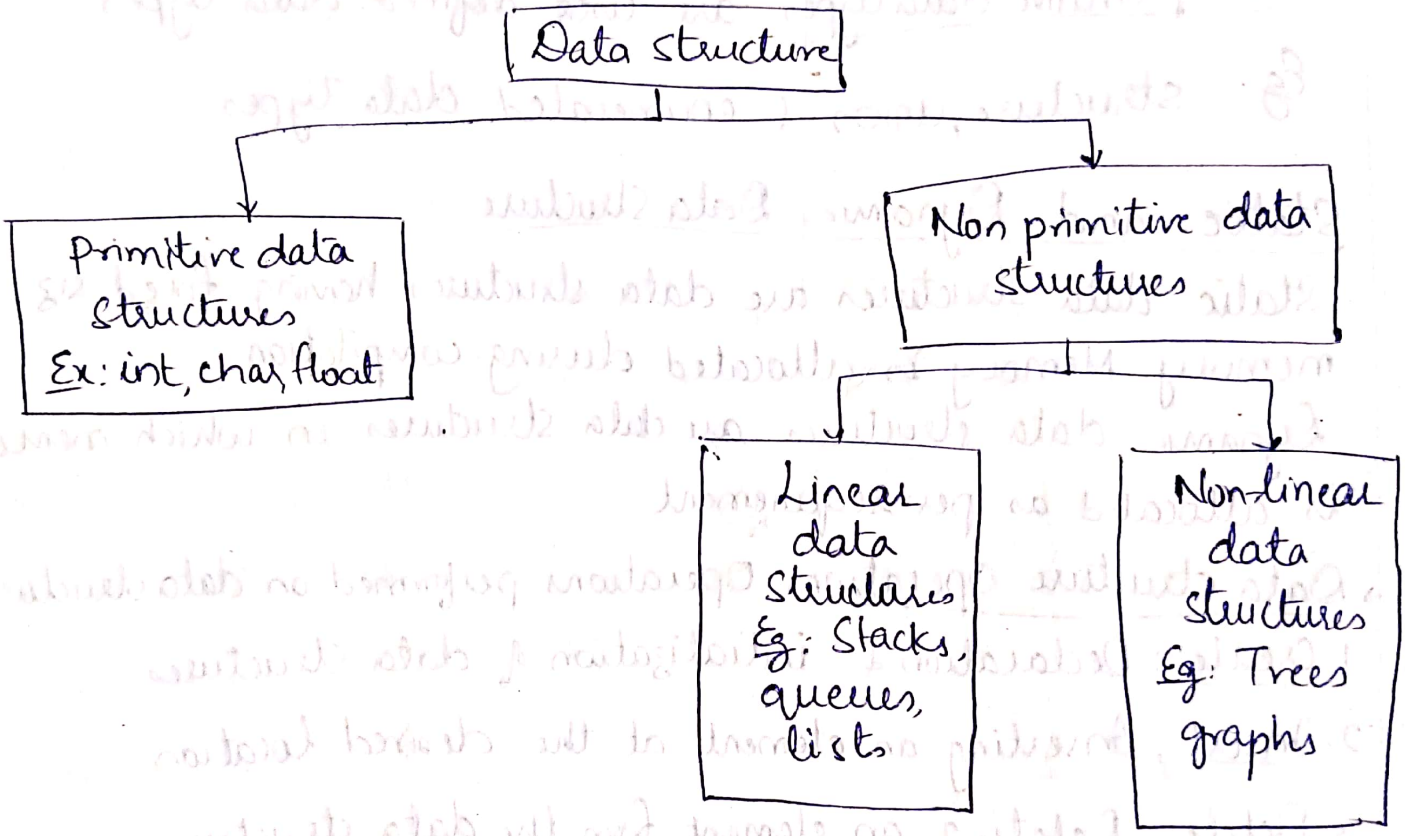
MODULE - 1

Introduction

Data Structure is a collection of set of elements and various operations that can be performed on these elements

→ **Classification of Data Structures**

Data structures can be divided into 2 basic types as shown below



- **Linear data structures** are the data structures in which data is arranged in a list or in a straight sequence.
  Eg: Arrays, Lists.
- **Non-linear data structures** are those in which data may be arranged in hierarchical manner.
  Eg: Trees, Graphs.

Primitive and Non primitive data structures

- Primitive data types are the fundamental data types.
- The non primitive data types can be built using primitive data types

Eg: int, float, double char

Non-primitive data-types are user defined data types

Eg: structure, union & enumerated data types.

Static and Dynamic Data Structure

Static data structures are data structures having fixed size memory. Memory is allocated during compilation

Dynamic data structures are data structures in which memory is allocated as per requirement

→ Data structure Operations: Operations performed on data structures

1. Create: Declaration & initialization of data structures
2. Insert: Inserting an element at the desired location
3. Delete: Deleting an element from the data structure
4. Searching: Searching of a key element from the data structure means finding the location of key element.
5. Sorting: Arranging the elements in a specific order (ascending or descending)
6. Reversing: Changing the order of data structure
7. Merging: Combining 2 data structures together to form a single data structure.
8. Traversal: Visiting each element of the data structure

Data Structures: Data structure is the organization of data in primary memory

Types: Stacks
Queues
trees
Linked Lists

Pointers and Dynamic Memory Allocation

Pointers

A pointer is an address of memory

Two operators used with the pointers are

& the address operator

* the dereferencing operator (value at the addre

A variable which holds the address of another variable or a memory location is called a pointer variable.

Declarations

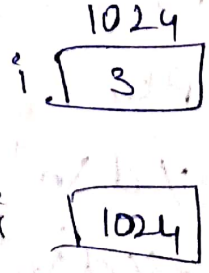
```
int i, *pi;
```

i is an integer variable

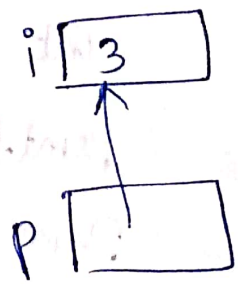
pi is a pointer variable

Initialization:

```
i = 3;
pi = &i;
```



(Or)

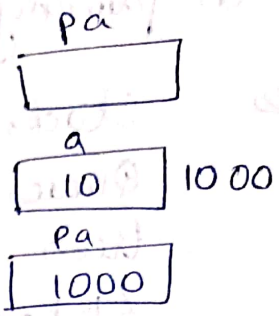


Pi holds the address of variable i.

Data can be accessed using pointer variable with the use of * operator.

For eg.

```
int *pa;
int a=10;
pa = &a;
printf("%d %d \n", *pa, a);
```



output: 10 10

Here a or *pa or *&a refer to the data 10 stored using variable a.

Operations on pointers

1) Arithmetic operations such as addition, subtraction, multiplication and division can be performed using pointers.

Eg:
```
#include<stdio.h>
void main()
{
  int a=10, b=20;
  int *pa=&a, *pb=&b;
  int x = *pa + *pb;
  int y = *pa - *pb;
  int z = *pa * *pb;
  printf("%d+%d=%d \n", *pa, *pb, x);
  printf("%d-%d=%d \n", *pa, *pb, y);
  printf("%d*%d=%d \n", *pa, *pb, z);
}
```

The output for the above program is

10 + 20 = 30

10 - 20 = -10

10 * 20 = 200

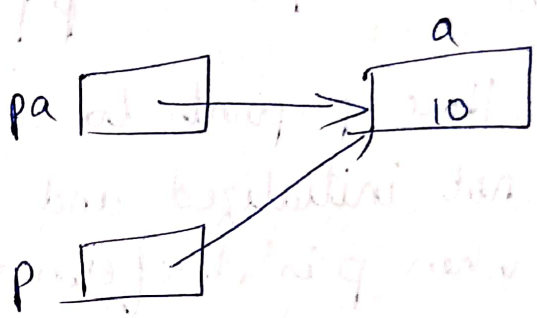
2) Two pointers can be compared

If pa & pb are 2 pointers

pa < pb, pa > pb, pa == pb, pa != pb are allowed

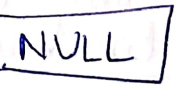
3) Copying 1 pointer to another pointer is allowed.

Eg:
```
int a = 10;
int *pa = &a;
int *p;
p = pa;
```



NULL pointer is a special pointer that points to 'lo' (nowhere) in memory.

```
int *p = NULL;
```



NULL pointer variable does not point to any part of the memory

Pointers can be dangerous

If a pointer is not initialized to NULL or any other legitimate object, it is likely to access an area of memory that is out of range of our program or it may contain a pointer reference to an illegitimate object.

- This may lead to serious errors.

Eg:
```
int *p;
int a = 10;
p = &a;
printf("%d", *p); // output = 10
printf("%d", *p-1); // accessing memory which is out of range
```

Eg: `int *p;` p → Junk address

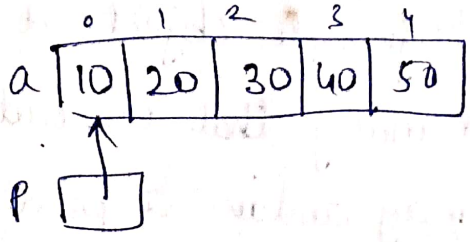
Here p points to an invalid address as it is not initialized and this may crash the program when p is dereferenced

- Dereferencing a NULL pointer may return 0 in some computers or may return data in location 0 producing serious errors
- Size of int and size of pointer all same. A valid integer may be interpreted as a pointer, causing errors.

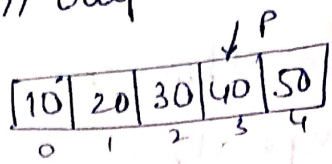
Pointers to Arrays

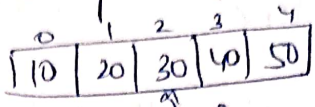
- Pointers can be used to hold the address of an array element and the complete array elements can be accessed using pointer

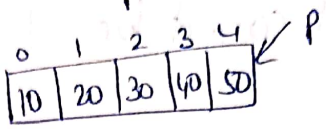
```
int a[5] = {10,20,30,40,50};
int *p;
p = a; // or p = &a[0]
```



```
printf("%d", *p);        // output : 10
p = p+3;
printf("%d", *p);        // output : 40
p--;
printf("%d", *p);        // output = 30
p = &a[4];
printf("%d", *p);        // output = 50
}
```







- In the above example p=a; stores the address of the first element in the array a in p ie, &a[0].
- *p displays the value at the address pointed by p ie, 10
- P value is then changed to p=p+3 and hence it points to the element 3 spaces to the right of the current position ie, p has the value &a[3].
- decrementing p points it to the previous location.
- p = &a[4] assigns the address of a[4] to p.

Memory allocation

Memory Allocation Techniques
- Static allocation
- Dynamic allocation

SRI DEVI G.M
ASST. PROFESSOR

Static Allocation	Dynamic Allocation
1) memory is allocated during compilation ie, compile time	1) Memory is allocated during execution i.e, run time
2) Memory space allocated is fixed and cannot be altered	2) Memory allocated can be increased or reduced during execution
3) Eg: User declarations and definitions Eg: int a;	3) Memory allocated using predefined functions like malloc(), calloc(), realloc() and free().

Dynamic Memory Allocation is the procen of allocating memory space during execution time (ie, run time)

Functions
- malloc
- calloc
- realloc
- free

SRI DEVI.G.M
ASST PROFESSOR

1) malloc: allocates memory of required size

Syntax: datatype *p;

p = (datatype *) malloc (size);

p is a pointer variable of type datatype. datatype can be int, float, char, void or user defined

Size - number of bytes to be allocated.

malloc function returns the first byte of allocated space if memory is allocated succesfully else it returns NULL. malloc does not initialize the memory.

2) calloc

Syntax datatype *p;

p = (data type *) calloc (n, size);

n - number of blocks to be allocated

size - number of bytes in each block

Calloc allocates n blocks of memory where each block is of specified size. Calloc initializes the memory allocated to 0's.

3) realloc

- realloc changes the size of the memory allocated using malloc() or calloc
- realloc() can extend or delete the allocated memory at the end of the block.

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

p - pointer to a block of previously allocated memory either using malloc() or calloc().

size - new size of block.

free

free( ) function is used to deallocate the block of memory which is allocated by using the function malloc( ) or calloc( ).

```
free(p);
```

p- pointer to memory block prenously allocated

Macro to allocate memory for one or more items of any datatype using malloc

```
#define MALLOC(p,n,type)
  p=(type*) malloc(n*sizeof(type));
  if(p==NULL)
  {
    printf("\nsufficient memory\n");
    exit(0);
  }
```

Array

- An array is a consecutive set of memory locations. It is a set of pairs <index, value> such that each index that is defined has a value associated with it.
- An array is a series of entities or a sequence of entities of the same type
- An array is a data structure

An array of integers

10 20 30 40 50

An array of real numbers

35.6 20.8 50.7 63.5 80.2 11.12

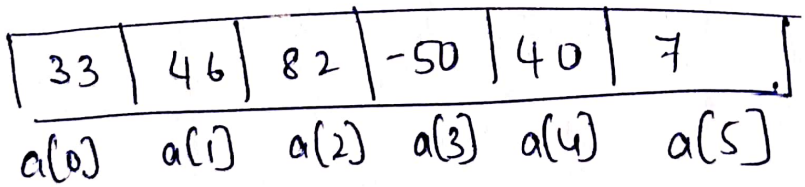
An array of characters

Hello world

- To store an array of integers in memory, we have to first create an array and store the integers in that array. as shown below

```
int a[6] = {33, 46, 82, -50, 40, 7};
```

- The above declaration int a[6] allocates 6 blocks of memory and stores the integers in those blocks.
- It is diagrammatically represented as



Array size = number of blocks * size of each block

= 6 * 2

= 12 bytes

An array can be initialized directly as shown above or using a forloop

Arrays in C

Array declaration

datatype name[size];

Eg: int list[5], *plist[5];

declares 2 arrays each containing 5 elements.

- The first array defines five integers
- Second array defines five pointers to integers.
- Arrays start at index 0. list[0], list[1], ... list[4] are the names of the five array elements, each of which contains an integer value
- The compiler allocates five consecutive memory locations each large enough to hold a single integer
- The address of the first element list[0] is called the base address.
- Memory address of list[i] is $\alpha + i * sizeof(int)$ where α is base address
- In the above example, list is a pointer to list[0].

list + i equals & list[i]

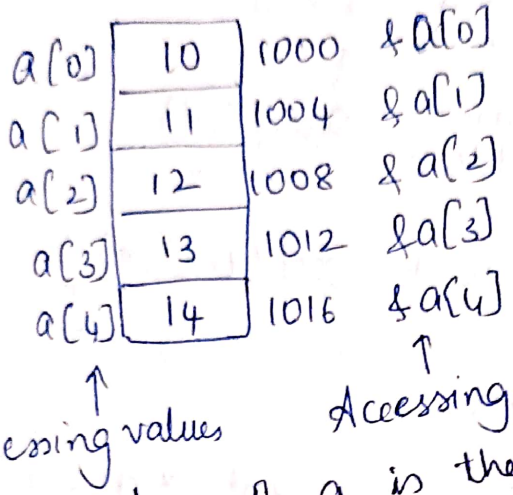
*(list + i) equals list[i]

SRI DEVI. G. M
ASST. PROFESSOR

Consider the following initialization statement

int a[5] = {10, 11, 12, 13, 14}

Compiler allocates memory locations and stores the values as shown below



Value of a is the base address i.e., 1000

&a[0] or a+0 = 1000

&a[1] or a+1 = 1004

&a[2] or a+2 = 1008

⇒ &a[i] or (a+i) or (i+a) gives address of ith item.

&a[4] or a+4 = 1016

*&a[0] or *(a+0) or a[0] = 10

*&a[1] or *(a+1) or a[1] = 11

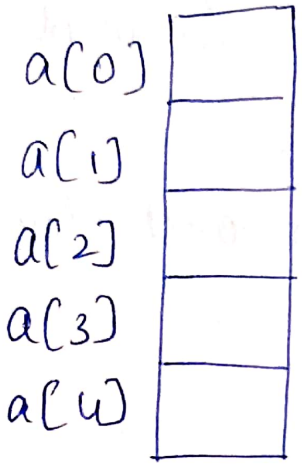
*&a[4] or *(a+4) or a[4] = 14

⇓

*&a[i] or *(a+i) or *(i+a) or a[i] or i[a]

gives value of ith item.

Base address = 1000 i.e. &a[0].



1000 = 1000 + 4 * 0

1004 = 1000 + 4 * 1

1008 = 1000 + 4 * 2

1012 = 1000 + 4 * 3

1016 = 1000 + 4 * 4 ← i

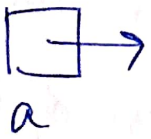
↑ base addres

↑ sizeof(int)

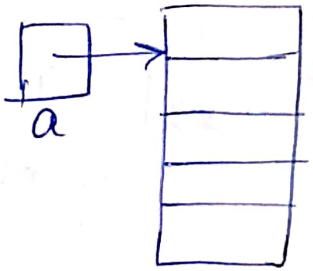
address of a[i] = α + sizeof(int) * i

Pictural representations

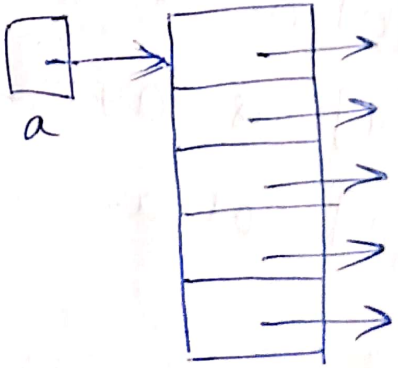
int *a;



int a[5]



int *a[5]



Function to display array elements using pointers

```
void display (int *ptr, int n)
{
   int i;
   printf("Array contents");
   for(i=0; i<n; i++)
      printf("%d\t", *(ptr+i));
}
```

Dynamically Allocated Arrays

Memory for array can be allocated dynamically during run time using malloc as shown below.

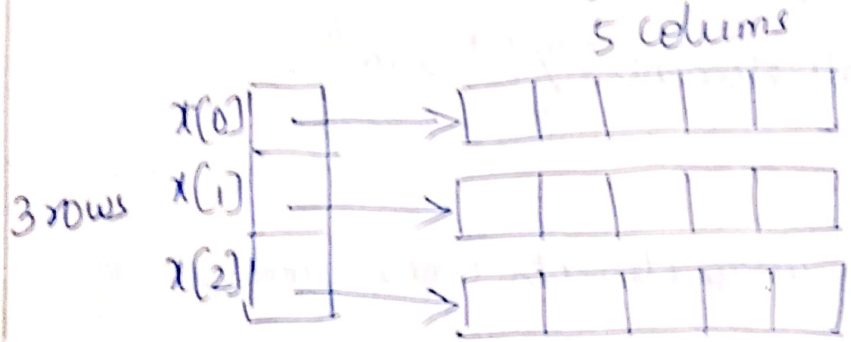
```
int *p,n,i;
printf("Enter the number of elements");
scanf("%d",&n);
```

```
p = malloc (n * sizeof (int));
printf ("Enter elements");
for (i = 0; i<n; i++)
    scanf ("%d", &p[i]);
```

Two-dimensional arrays

Two-dimensional array can be represented as

```
int x[3][5];
```



Arrays of arrays representation

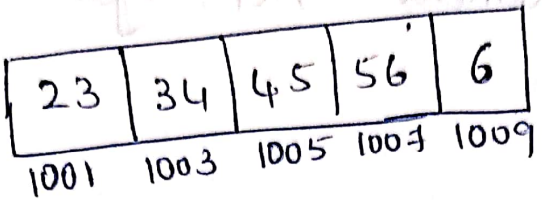
- C finds the element x[i][j] by first accessing the pointer in x[i].
- This pointer gives the addres of zeroth element of row i of the array
- Addres of j^{th} element of row i is found by adding j * sizeof (int) to this pointer.
- Memory can be allocated dynamically for a two-dimensional array as shown below.

```
int **p;
p = makeArray (5, 10);
```

SRI DEVI.G.M
ASST. PROFESSOR

Pointers & arrays

Eg1: `int num[5] = { 23, 34, 45, 56, 6};`



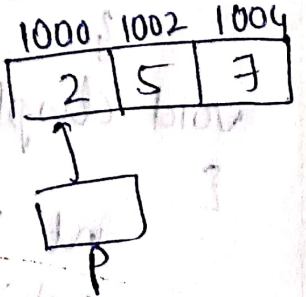

```
main()
{
   int num[5] = { 23,34,45, 56,6};
   int i=0;
   while (i<5)
   { printf ("Element %d, Addren %u", i, &num[i]);
     i++;
   }
}
```

O/P:

Element 0 Addren 1001
Element 1 Addren 1003
" 2 " 1005
" 3 " 1007
" 4 " 1009

Eg2:

```
main()
{
   int num[] = {2, 5, 7};
   int *p;
   p = num;   or   p = &num[0];
```



```
while (i<3)
  {
    printf ("Address =%u", &num[i]);
    printf ("Element = %d", *p);
    i++;
    p++;
  }
}
```

O/P Address = 1000 Element = 2
Address = 1002 Element = 5
Address = 1004 Element = 7

Name of the array holds the address of the first element in the array

Hence num ⇒ &num[0] ⇒ 1000

Eg3:

```
main()
{
  int num[]={2,5,7};
  display (&num[0],3); //display(num,3);
}
void display (int *j, int n)
{
  int i=1;
  while(i<=n)
  {
    printf("Element = %d", *j);
  } i++; j++;
}
```

Sparse Matrices

A sparse matrix is a matrix that contains many zero entries

Eg:

row \ col	0	1	2	3	4	5
0	5	0	0	0	1	0
1	0	0	2	-5	0	11
2	2	0	0	0	0	15
3	3	2	0	0	0	0
4	0	0	0	0	0	0

Matrix A

We can use an array of triples <row, col, value> to represent a sparse matrix as shown below

	row	col	value
a[0]	5	6	9
[1]	0	0	5
[2]	0	4	1
[3]	1	2	2
[4]	1	3	-5
[5]	1	5	11
[6]	2	0	2
[7]	2	5	15
[8]	3	0	3
[9]	3	1	2

Structure for triple format

```
typedef struct
{
    int row;
    int col;
    int value;
}Triple;

Triple a[20];
```

a[0].row contains the number of rows, a[0].col contains the number of columns and a[0].value contains the total number of nonzero entries. Other positions store the triples representing nonzero entries. row index is in the field 'row' column index in the field 'col' and value is in the field 'value'

Program to read a sparse matrix and convert it to triple format

```
#include<stdio.h>  //Insert the structure for triple format;
void main()
{
    int a[10][10], m, n, i, j, k=1;
    Triple ta[10];
    printf("Enter the order of the sparse matrix\n");
    scanf("%d%d", &m, &n);
    ta[0].row=m;
    ta[0].col = n;
    printf("Enter elements into the sparse matrix\n");
    for( i=0; i<m ; i++)
    {
        for( j=0; j<n; j++)
        {
```

SRI DEVI.G.M
ASST. PROFESSOR

ASST. PROFESSOR

```
            scanf("%d", &a[i][j]);
            if(a[i][j] != 0)
            {
                ta[k].value = a[i][j];
                ta[k].row = i;
                ta[k].col = j;
                k++;
            } //end of if
        } //inner for loop closed
    } //outer for loop closed

    ta[0].value = k-1;
    printf("Matrix A \n");
    for(i=0; i<m; i++)
    {
        for(j=0; j<n; j++)
        {
            printf("%d\t", a[i][j]);
        }
        printf("\n");
    }
    printf("Triple format of matrix a");
    Printf("Row \t col \t value \t\n");
    for(i=0; i<k; i++)
    {
        printf("%d \t %d \t %d \n", ta[i].row, ta[i].col,
                                    ta[i].value);
    }
}
```

Transpose of a sparse matrix in triple format

```
void transpose(Triple a[], Triple b[])
{
    int n, j, i, curb;
    n = a[0].value;
    b[0].row = a[0].col;
    b[0].col = a[0].row;
    b[0].value = n;
    if (n>0)
    {
        curb = 1;
        for(i=0; i<a[0].col; i++)
            for(j=1; j<=n; j++)
                if (a[j].col == i)
                {
                    b[curb].row = a[j].col;
                    b[curb].col = a[j].row;
                    b[curb].value = a[j].value;
                    curb++;
                }
    }
}
```

Polynomials using arrays

A polynomial is a sum of terms, where each term has a form ax^e where x is the variable, a is the coefficient and e is the exponent

Eg:

$$A(x) = 3x^{20} + 2x^5 + 4$$

$$B(x) = x^4 + 10x^3 + 3x^2 + 1$$

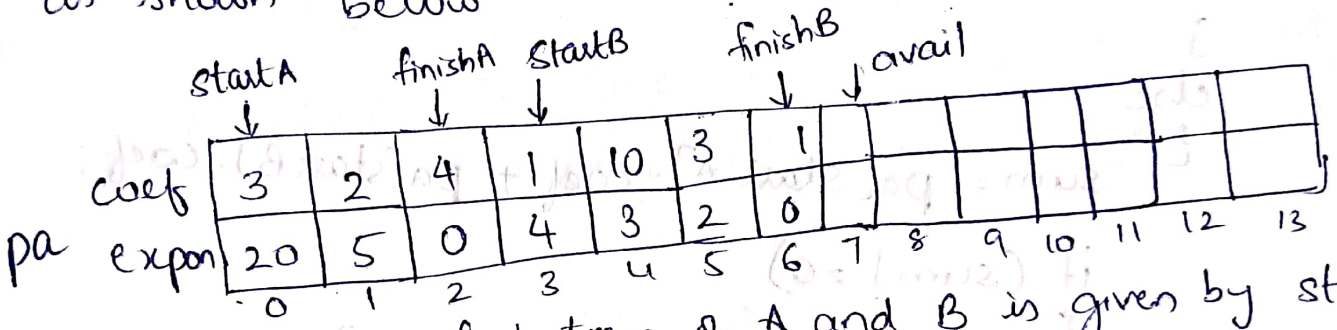
The largest exponent of a polynomial is called its degree.

Polynomial representation

```
typedef struct
{
   int coef;
   int expon;
} poly;
poly pa[MAX];
```

Consider 2 polynomials A and B given in the example

These 2 polynomials can be stored in the array pa as shown below



The index of the first term of A and B is given by startA and startB respectively. finishA and finishB give the index of the last term of A and B.

The index of the next free location in the array is given by avail

Here startA = 0, finishA = 2, startB = 3, finishB = 6, avail = 7.

Polynomial addition

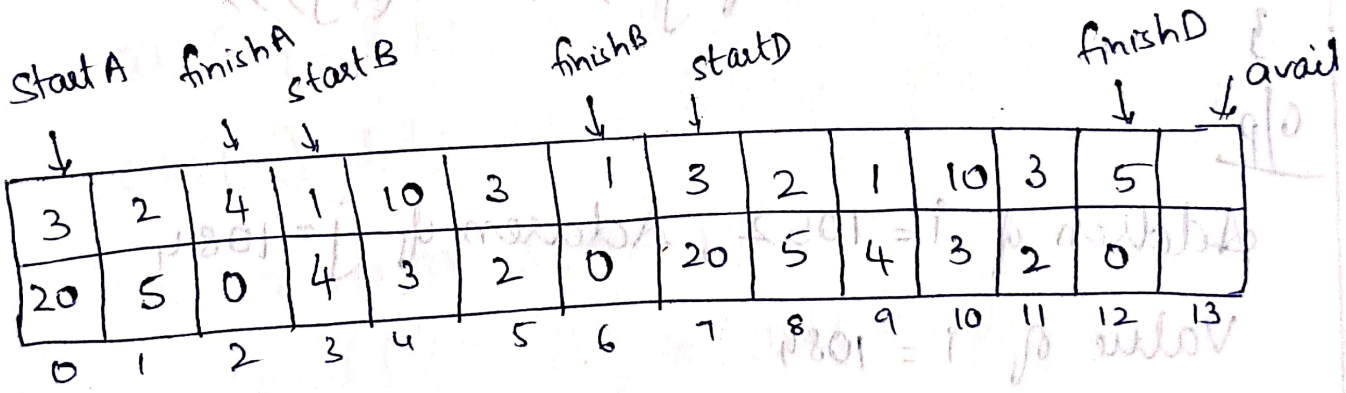
```
void poly_add (int startA, int finishA, int startB, int finishB)
{
   int sum;
   startD = avail;
   while ( startA <= finishA && startB <= finishB)
   {
```

SRI DEVI.G.M
ASST PROFESSOR

```
if (pa[startA].expon < pa[startB].expon)
{
    pa[avail].expon = pa[startB].expon;
    pa[avail].coef = pa[startB].coef;
    avail++; startB++;
}
else if (pa[startA].expon > pa[startB].expon)
{
    pa[avail].expon = pa[startA].expon;
    pa[avail].coef = pa[startA].coef;
    avail++; startA++;
}
else
{
    sum = pa[startA].coef + pa[startB].coef;
    if (sum != 0)
    {
        pa[avail].expon = pa[startA].expon;
        pa[avail].coef = sum;
        avail++;
    }
    startA++;
    startB++;
}
for(; startA <= finishA; startA++)
{
    pa[avail].expon = pa[startA].expon;
    pa[avail].coef = pa[startA].coef;
    avail++;
}
```

```
for(; startB<=finishB; startB++)
{
    pa[avail].expon = pa[startB].expon;
    pa[avail].coefs = pa[startB].coefs;
    avail++;
}
finishD = avail;
avail++;
}
```

The above function adds the 2 polynomials A and B are the result is stored in the array starting at position startD and ending at finishD as shown below



$A(x) = 3x^{20} + 2x^5 + 4$

$B(x) = x^4 + 10x^3 + 3x^2 + 1$

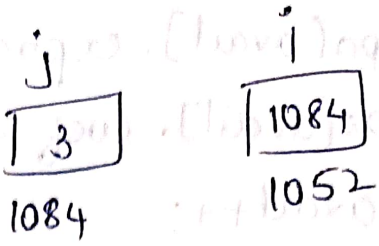
sum

$D(x) = 3x^{20} + 2x^5 + x^4 + 10x^3 + 3x^2 + 5$

SRI DEVI.G.M
ASST. PROFESSOR

Pointer examples

i)
```
main()
{
    int j=3, *i;
    i = &j
    printf("Addres of i = %u, Addres of j = %u \n",
                                    &i, &j);
    printf("Value of i = %u", i);
    printf("\nvalue of j using pointer variable is
                                    %d", *i);
    printf("value of j = %d", *(&j));
}
```



o/p:

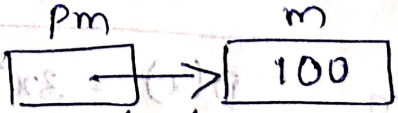
Addres of i = 1052, Addres of j = 1084

Value of i = 1084

value of j using pointer variable is 3

value of j = 3

ii)
```
main()
{
    int m=122, *pm;
    pm = &m;
    *pm = 100;
    printf("value of m = %d", *pm);
}
```



output

value of m = 100

Output

Element = 2
Element = 5
Element = 7

Pointer to pointer

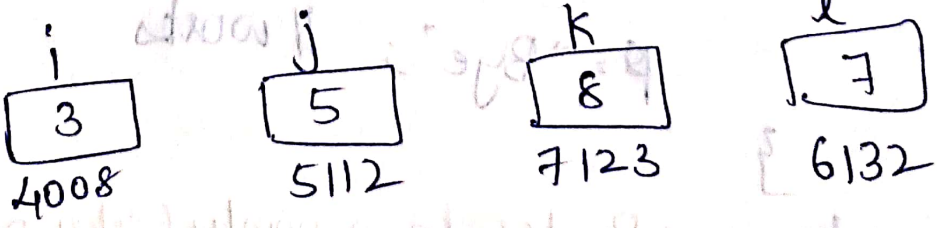
A variable containing the address of a pointer variable is called pointer to a pointer.

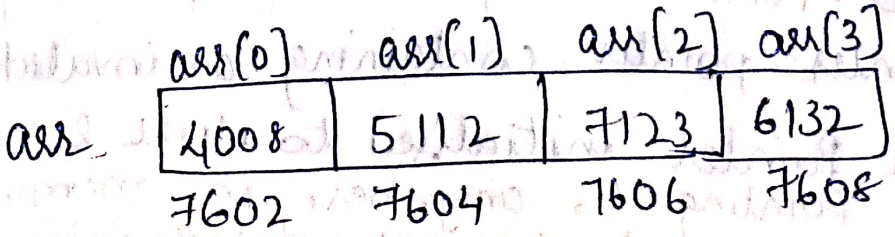
Eg. int p; integer variable
int *p1; pointer
int **p2; pointer to pointer
p1 = &p;
p2 = &p1;

Array of pointers

```
void main()
{
    int * arr[4];
    int i=3, j=5, k=8, l=7,m;
    arr[0] = &i;
    arr[1] = &j;
    arr[2] = &k;
    arr[3] = &l;
    for (m=0;m<3;m++)
        printf("%d", *(arr[m]));
}
```

output: 3 5 8 7





Pointers and strings

We cannot assign a string to another string but a char pointer can be assigned to another char pointer.

Eg: main ()
{
char str1[] = "Hello";
char str2[10];
char *s = "How are you";
char *q;
str2 = str1; //error
q = s //works

- Once a string is defined it cannot be initialized to another set of characters

Eg: void main()
{
char str1[] = "Hello";
char *p = "Hello";
str1 = "Bye"; //error
p = "Bye"; //works
}

Pointer types: Pointer to a pointer (refer previous page)

1. Dangling pointer: pointer containing an invalid address
2. NULL pointer: Pointer initialized to NULL & not pointing to anywhere in memory
3. Void pointer: Can point to any datatype.

Structures

- Various primitive data types in C are

  int - integer data type
  float - floating point data type
  Char - character data type
  double - double precision floating point data type
  void - null data type

- Other data types that can be derived from the basic data types are called derived data types. Various derived data types are

  Arrays
  Pointers
  Enumerated
  Structures
  Unions

SRI DEVI.G.M
ASST. PROFESSOR

⇒ Type definition

- typedef keyword allows the programmer to create a new data type name for an already existing data type.
- The purpose of typedef is to redefine the name of an existing data type
- General syntax of typedef is

```
typedef   data-type   new_name1, new_name2 ... ;
```

- typedef: keyword
- data-type: Existing data type (int, float..)
- new_name: user_defined data types (marks, amount..)

Eg: typedef int MARKS;

Here we are creating a new name "MARKS" for the existing data type int. MARKS is called user-defined data type

The new data type names that are defined by the user using already existing data types are called user-defined data types.

Ex1: typedef int MARKS;
MARKS sub1, sub2, sub3;

Ex2: typedef int BIGARRAY[1000] //user-defined data type
BIGARRAY a; //a is an array of 1000 integers

Ex3: typedef char STRING[80] //user defined data type
STRING s; //s is an array of 80 characters.

-Program to compute simple interest using various typedef definition

```
#include<stdio.h>
typedef float AMOUNT;          //typedef definitions
typedef float TIME;
typedef float RATE;
void main()
{
    AMOUNT p,si;
    TIME t;
    RATE r;
    printf("Enter p, t and r");
    scanf("%f %f %f", &p, &t, &r);
    si=(p*t*r)/100;
    printf("SI = %f \n", si);
}
```

SRI DEVI.G.M
ASST. PROFESSOR

AMOUNT, TIME and RATE are user-defined data types

Advantages of typedef

- Provides a meaningful way of declaring the variables
- Increases the readability of the program
- Helps us to understand the source code and saves time and energy spent in understanding the program.

⇒ Structures

Def: A structure is a collection of one or more variables of same data type or dissimilar data types grouped together under a single name. Structure allows to group data of different data types

Eg: Student information to be grouped may consist of

Name of the student
University marks
USN.

General syntax of a structure

```
struct tag_name
{
    type1 member1;
    type2 member2;
        :
};
```

SRI DEVI.G.M
ASST. PROFESSOR

Example

```
struct student
{
    char name[10];
    int usn;
    float marks;
};
```

struct is the keyword used to define a structure. The tag_name is used as shorthand for declarations. The variables in

a structure such as member1, member2, ... declared within braces are called members of the structure. The members can be of any data type such as int, char, float etc. There should be semicolon at the end of closing brace

→ Structure definition / declaration

A structure can be declared using three different ways

1) Tagged structures
2) Structure without tag
3) Typedefined structures

SRI DEVI.G.M
ASST. PROFESSOR

1) Tagged structure

In structure definition, the keyword struct can be followed by an identifier. This identifier is called tag_name. The structure definition associated with a tag_name is called tagged structure.

/* structure definition */

Syntax

```
struct tag_name
{
  type1 member1;
  type2 member2;
    :
};
```

Example

```
struct student
{
  char name[10];
  int usn;
  float marks;
};
```

- struct is the keyword
- student is the name of the structure. It is also called tag name.
- struct student represents the data type. It is called a derived data type.
- Memory is allocated for structure only when the variables are declared.

Structure variables are declared as shown below

Syntax:

struct tagname variable_list;

Example

struct student S1, S2; /* structure declaration */

S1 and S2 are variables of type struct student.

- Once the variables are declared memory is allocated for S1 & S2

Number of bytes allocated for variable S1 is

10 bytes for name
2 bytes for usn
4 bytes for marks

} Size of structure student 10+2+4 = 16 bytes.

SRI DEVI.G.M
ASST. PROFESSOR

2) Structure without tag

The structure definition without tag name is called structure without tag.

Syntax

```
struct
{
    type1 member1;
    type2 member2;
    :
} Variable list;
```

Example

```
struct
{
    char name[10];
    int usn;
    float marks;
} S1, S2;
```

In the absence of tag name, variables must be declared immediately after the right brace as shown above

3) Type defined Structure

The structure definition associated with keyword typedef is called typedefined structure.

Syntax

```
typedef struct
{
   type1 member1;
   type2 member2;
   :
} TYPE_ID;
```

Example

```
typedef struct
{
   char name[10];
   int usn;
   float marks;
} STUDENT;
```

- typedef and struct are keywords
- The closing brace must end with type definition name

In the example, STUDENT is the new data type using which variables can be declared as shown below

Syntax: TYPE_ID V1, V2, ..Vn;

Example: STUDENT S1, S2;

SRI DEVI.G.M
ASST. PROFESSOR

Here, student is tagname and STUDENT is the user-defined data type.

⇒ Structure Initialization

The syntax of initializing structure variables is shown below

struct tag_name variable = {V1, V2, ..Vn};

↓ values for members

Examples

```
struct student
{  char name[10];
   int usn;
   int marks;
} a= {"abc", 1, 20};
```

Variable a has values name = "abc", salary = 1 and marks = 20.

- The variable a can also be initialized as shown below

```
struct student
{
    char name[10];
    int usn;
    int marks;
};
struct student a = {"abc", 1, 20};
```

- Initialization during structure declaration with more than one variable

```
struct student
{
    char name[10];
    int usn;
    int marks;
} a, b = {"xyz", 2, 24};
```

Here, a is not initialized and only b is initialized.

SRI DEVI.G.M
ASST. PROFESSOR

>
```
struct student
{
    char name[10];
    int usn;
    int marks;
} a = {"abc", 1, 20}, b = {"xyz", 2, 24};
```

Here both a and b are initialized

> Partial initialization

```
struct student a = {"abc", 1};
```

Here only name and usn of variable a are initialized.

⇒ Accessing structures

A member of a structure can be accessed by specifying the variable name followed by a dot, followed by the member name

Syntax: variable. member

To access the member name of variable a, we specify a.name

Program to display various members of a structure:

```
#include<stdio.h>
typedef struct
{
    char name[10];
    int usn;
    float marks;
} STUDENT;
Void main()
{
    STUDENT a;
    Printf(" Enter student details ");
    scanf("%s %d %f", a.name, &a.usn, &a.marks);
    Printf(" The student details are ");
    printf("Name: %s", a.name);
    printf("USN: %d", a.usn);
    printf("Marks: %f", a.marks);
    getch();
}
```

SRI DEVI. G.M
ASST. PROFESSOR

Unions: A union is similar to a structure which is also a collection of data items of similar or dissimilar data types. The data items are called fields or members. All the members of the union share the same memory space Hence all members have same addresses. Only one member is active at any point during execution

General format / syntax

```
union tagname
{
    type1 member1;
    type2 member2;
      :
};
```

Eg:
```
typedef union
{
    int i;
    float f;
    char c;
} item;
```

item x; → declaration of variable of type item.

The compiler reserves memory whose size is that of the largest member. In the above example float is the largest datatype of the member of the union. Hence, sizeof(float) = 4 bytes of memory is reserved for the variable x.

```
#include <stdio.h>
void main()
{
    typedef union
    {
        int marks;
        char grade;
        float percentage;
    } STUDENT;
    STUDENT x;
    x.marks = 100;
    printf ("Marks: %d \n", x.marks);
    x.grade = 'A';
    printf ("Grade : %c \n", x.grade);
    x.percentage = 99.5;
    printf ("Percentage : %f \n", x.percentage);
}
```

Output

```
Marks :100
Grade : A
Percentage : 99.5.
```

After statement x.marks=100, the member marks will hold the value 100 and other members should not access the data. If accessed, it will be treated as garbage value

Difference between structure and union

Structure	Union
1. The keyword struct is used to define a structure	1. The keyword union is used to define a union
2. For a structure variable, the compiler allocates the memory for each member. The size of structure is equal to the sum of sizes of its members	2. For a union variable, the compiler allocates the memory by considering the size of the largest member. So, size of union is equal to the size of largest member.
3. Altering the value of a member will not affect the other members of the structure	3. Altering the value of any of the members will alter other member values.
4. The address of each member will be in ascending order. The memory for each member starts at different offset values.	4. The address is same for all the members of a union. Every member begins at offset zero.
5. Individual members can be accessed at any time since separate memory is reserved for each member	5. Only one member can be accessed at a time since memory is shared by each member
Eg: struct student { char name[10]; int usn; };	Eg: union student { char name[10]; int usn; };

Self-referential Structures

A self-referential structure is one in which one or more of its components is a pointer to itself.

Eg:
```
typedef struct {
        int data;
        struct list *link;
    } list;
```

data is an integer, link is a pointer to a list structure.

Value of link is either the address in memory of an instance of list or the null pointer.

```
list item1, item2, item3;
item1.data = 10;
item2.data = 20;
item3.data = 30;
item1.link = item2.link = item3.link = NULL;
item1.link = &item2;
item2.link = &item3;
```