1 Introduction

Data Structures are the models for organizing the data so that accessing becomes easy. They deal with the study of how data is organized in memory, how it can be manipulated, how efficiently it can be retrieved. and the possible ways in which different data items are logically related.

Dr.Mahesh G, Assoc. Prof., BMSIT & M
Mr.Harish G, Assoc. Prof., Dr. AIT

1.1 Data Structures Classification

The different types of data structures are

✓ **Primitive Data Structures**

Data structures which can be manipulated directly by machine instructions are called primitive data structures.

EX: int, float, char etc.

✓ **Non-Primitive Data Structures**

Data structures which cannot be manipulated directly by machine instructions are called non-primitive data structures.

The different types of non-primitive data structures are

- **Linear Data Structures**: The elements here exhibit either physical adjacency or logical adjacency between them. Here the data is arranged in a linear fashion although the way they are stored in memory need not be sequential.

  EX: Arrays are physically adjacent i.e. elements are stored in consecutive memory locations. Linked Lists are logically adjacent i.e. they are linked by pointers.

- **Non-Linear Data Structures**: They do not exhibit any adjacency between elements but are logically related. Here data is not arranged in a sequential way.

  EX: Trees, Graphs etc.

1.2 Data Structures Operations

The following are the operations that can be performed on data structures

1. **Create** - This operation is used to create a data structure. In most of the programming languages this is usually done using a simple declarations statement. For example, the declaration "int a;" allocates memory for the integer 'a' during compilation of the declaration statement.
2. **Insertion** - Insertion means adding new details or new node into the data structure.
3. **Deletion** - Deletion means removing the existing details or a node from the data structure.

4. **Update** - Update means changing the existing details or contents of a node in the data structure.
5. **Traversal** - Traversing means accessing each node exactly once so that the nodes of a data structure can be processed. Traversing is also called as visiting.
6. **Searching** - Searching means finding the location of node for a given key value.
7. **Sorting** - Sorting means arranging the nodes in a particular order.
8. **Merging** - Merging means combining the nodes from two lists into a single list.

Example: An organization contains a membership file for every department in the organization wherein each record contains the following data for a given member.

Name	Address	Phone Number	Email	Age	DOB

1. Suppose a new person joins the organization, then one would **insert** his or her record to the file.
2. Suppose a person resigns from the organization, then one would **delete** his or her record from the file.
3. Suppose a member has moved to a new location and has a new address, then one would **update** his or her record in the file.
4. Suppose the organization wants to organize a meeting though mailing, then one would **traverse** the file to obtain the name and email of each member.
5. Suppose one wants to find the address for a given name, then one would **search** the file for the record containing the name.
6. Suppose one wants the list of members in increasing order of their age, then one would **sort** the file based on age.
7. Suppose one wants a consolidated list of members from 2 different departments, then one would **merge** the files of 2 different departments into a single file.

1.3 Data Structures Representation

Any data structure can be represented using one of the following two ways.

1. **Sequential Representation** - A sequential representation maintains the data in continuous memory locations which takes less time to retrieve the data but leads to time complexity during insertion and deletion operations. Because of sequential nature, the elements of the list must be freed, when we want to insert a new element or new data at a particular position of the list. To acquire free space in the list, one must shift the data of the list towards the right side from the position where the data

has to be inserted. Thus, the time taken by CPU to shift the data will be much higher than the insertion operation and will lead to complexity in the algorithm. Similarly, while deleting an item from the list, one must shift the data items towards the left side of the list, which may waste CPU time.

2. **Linked Representation** - Linked representation maintains the list by means of a link between the adjacent elements which need not be stored in continuous memory locations. During insertion and deletion operations, links will be created or removed between which takes less time when compared to the corresponding operations of sequential representation.

Dr.Mahesh G Assoc. Prof. BMSIT & M
Mr.Harish G Assoc. Prof. Dr. AIT

1.4 Algorithm Specification

Definition: An algorithm is a finite set of instructions that accomplishes a particular task.

The **criteria's** that must be satisfied are

1. **Input:** Zero or more quantities are externally supplied.
2. **Output:** At least one quantity is produced.
3. **Definiteness:** Each instruction must be clear and unambiguous.
4. **Finiteness:** The algorithm always terminates after a finite number of steps and uses finite sources.
5. **Effectiveness:** Every instruction must be basic enough to be carried out.

Note: In Computational theory a program does not have to satisfy the 4th criterion (Finiteness).Example, an operating system continues in a wait loop until more jobs are entered and will not terminate unless the system crashes. Since the programs that we discuss will always terminate, we use algorithm and program interchangeably here after.

Methods of specifying an algorithm

1. **Using a natural language:** a clear description of algorithms is surprisingly difficult because of the inherent ambiguity associated with any natural language.
2. **Flowchart:** is a method of expressing an algorithm by a collection of geometric shapes containing descriptions of the algorithm's steps. They work well only if the algorithm is small and simple.
3. **Pseudo code:** is a mixture of a natural language and programming language like constructs(C) giving a semi-formal description of each step to be carried out by the computer.

2. Pointers

variable that holds the address of another variable

2.1 Memory Organization

Memory is organized as a sequence of byte sized locations (1 byte = 8bits). Each byte in memory is associated with a unique address. Addresses are positive integer values that range from zero to some positive integer constant corresponding to last location in the memory.

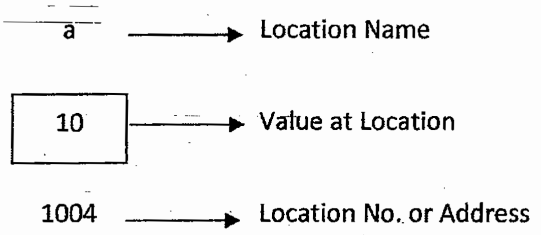
Every object (data or program code) that is loaded into memory is associated with a valid range of addresses, i.e. each variable and each function in a program starts at a particular location, and spans across consecutive addresses from that point onwards depending on the size of the data item.

Consider the statement int a = 10; This tells the compiler to

1. Reserve space in memory to hold a integer value.
2. Associate the name 'a' with this memory location.
3. Store the value 10 at this location.

This is as shown in the following memory map.

Dr.Mahesh G Mr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT



2.2 The Address Operator (&)

This operator when used as a prefix to a variable name gives the address of that variable.

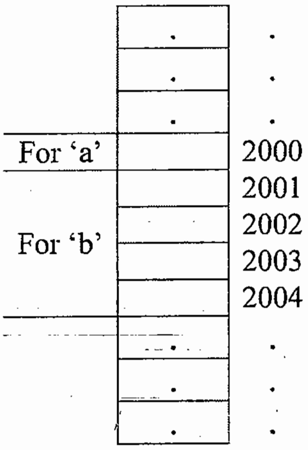
EX: Program to print the address of a variable along with its value.

```
#include<stdio.h>
void main( )
{
        int a;
        a = 10;
        printf("The address of a is %u\n", &a);
        printf("The value of a is %d\n", a);
}
```

Here %u is used for printing address values, because memory addresses are unsigned integers.

Consider the declaration,
char a;
int b;

Assuming char requires 1 byte and int requires 4 bytes, memory space may be reserved in the following way.



&a = 2000
&b = 2001
i.e. a variables address is the first byte occupied by the variable.

2.3 The Indirection Operator (*)

This operator when used as a prefix to the address of a variable gives the value of that variable (contents at that address).

EX: Program to print the address of a variable and then print its value using * operator.

```
#include<stdio.h>
void main( )
{
        int a;
        a = 10;
        printf("The address of a is %u\n", &a);
        printf("The value of a is %d\n", *(&a));
}
```

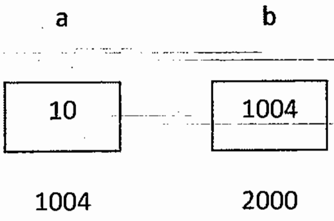
Note: The address and indirection operators are complimentary i.e. when both are applied on a variable, the variable is got back. [*(&a) = a]

2.4 Pointer Variables or Pointers

The address operator (&) returns the address of a variable. This address can be collected or stored in another variable.

EX: b = &a;

Here the compiler must also reserve space for the variable 'b'. The memory map for this is as shown.



Note that 'b' is not a ordinary variable. It is a variable which contains the address of another variable 'a'. Such type of variable is called a pointer variable.

Definition: A pointer is a variable that contains the address of another variable.

2.5 Pointer Declaration

The syntax for declaring a pointer variable is

datatype * Name_of_the_Pointer_Variable;   int * P

Here datatype can be

- ✓ Primitive datatypes such as int, float char, etc
- ✓ User defined datatypes

Name_of_the_Pointer_Variable is any valid variable name.

Note: The datatype of a pointer variable must be same as the type of the variable that it points to.

- ✓ int *P means P is a pointer to an integer i.e. it can hold the address of a integer variable .
- ✓ float *P means P is a pointer to a float i.e. it can hold the address of a floating point variable.
- ✓ char *P means P is a pointer to a character i.e. it can hold the address of a character variable.

EX:

Valid in 'C'	Invalid in 'C'
int *P;	int *P;
int a = 10;	float a = 3.4;
P = &a;	P = &a;

2.6 Accessing a Variable through its Pointer

We know that by using a pointer variable, we can obtain the address of another variable.

Dr.Mahesh G Mr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

EX:

```
int a, b;
int *P;
a = 10;          // Assume 'a' is stored at address 1000
P = &a;          // P will contain address of a i.e. 1000
```

We know that the value stored at any address can be obtained using the indirection operator (*).

EX: The value stored at address 1000 can be obtained by *P [same as *(&a)]. Here P means the address 1000 and *P means the contents at address 1000 which is 10.

EX: Program to illustrate the use of indirection operator '*' to access the value pointed by a pointer.

```
#include<stdio.h>
void main( )
{
        int a, b;
        int *P;
        a = 10;
        P = &a;
        b = *P;
        printf("%d\n", a);
        printf("%d\n", b);
        printf("%d\n", *P);
}
```

OUTPUT

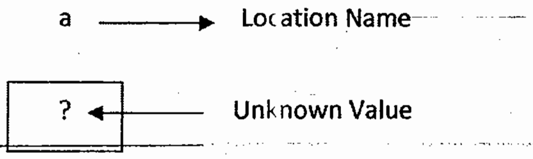
```
10
10
10
```

2.7 Initialization of Pointer Variables

Consider the following variable declaration, where the variable is uninitialized.

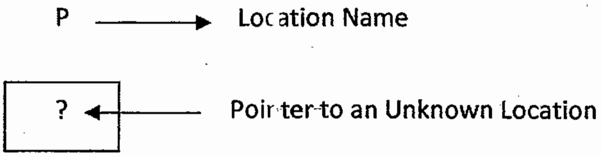
int a;

This declaration tells the compiler to reserve space in memory and associate name 'a' to that location. Because it is uninitialized, the value at this location is unknown or some garbage value. This is as shown.



The same thing applies to pointer variables also. Uninitialized pointers will have unknown memory addresses or unknown values that will be interpreted as memory locations.

int *P;



Uninitialized pointers cause errors which are very difficult to debug. Hence it is always good practice to assign a valid memory address or NULL value to a pointer.

EX:

```
int a;
int *P = &a;
int *Q = NULL;
```

Note: NULL is a pointer value which when assigned to a pointer variable indicates that the pointer does not point to any part of the memory.

2.8 Pointers and Functions

EX: Program to swap two numbers

```
#include<stdio.h>
void main( )
{
        int a, b, temp;
        printf("Enter 2 numbers\n");
        scanf("%d%d", &a, &b);
        printf("BEFORE SWAPPING \n");
        printf("Value of a = %d \n",a);
        printf("Value of b = %d \n",b);
        temp = a;
        a = b;
        b = temp;
        printf("AFTER SWAPPING\n");
        printf("Value of a = %d \n",a);
        printf("Value of b = %d \n",b);
}
```

EX: Program to swap two numbers using Functions

```
#include<stdio.h>
void swap( int *c, int *d)
{
        int temp;
        temp = *c;
        *c = *d;
        *d = temp;
}
void main( )
{
        int a, b, temp;
        printf("Enter 2 numbers\n");
        scanf("%d%d", &a, &b);
        printf("BEFORE SWAPPING\n");
        printf("Value of a = %d \n",a);
        printf("Value of b = %d \n",b);
        swap(&a, &b);
        printf("AFTER SWAPPING\n");
        printf("Value of a = %d \n",a);
        printf("Value of b = %d \n",b);
}
```

Dr.Mahesh G Mr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

Note:

- ✓ If 'a' and 'b' are passed by value, then the effect of swapping will not be seen in the main function. The variables 'a' and 'b' will retain their values.
- ✓ When only one value needs to be sent from the called function to the calling function, we can use the 'return' statement.
- ✓ When more than one value needs to be sent from the called function to the calling function, we have to use call by reference.

2.9 Functions returning Pointers

EX: Program to find bigger of two numbers

```
#include<stdio.h>
int* bigger(int *c, int *d )
{
        if(*c > *d)
                return c;
        else
                return d;
}

void main( )
{
        int a, b;
        int *big;

        printf(“Enter 2 numbers\n”);
        scanf(“%d%d”, &a, &b);

        big = bigger(&a, &b);
        printf(“Bigger of two numbers is %d \n”, *big);
}
```

Dr.Mahesh G Mr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

2.10 Pointer to a Function

EX: Program to add two numbers

```
#include<stdio.h>
int add(int c, int d )
{
        int res;
        res = c + d;
        return res;
}

void main( )
{
        int a, b, res;

        int (*ptr)(int, int);       // declaring pointer to a function

        ptr = add;                  // storing the address of a function in pointer to a function

        printf(“Enter 2 numbers\n”);
        scanf(“%d%d”, &a, &b);

        res = (*ptr)(a, b);         // calling the function using pointer

        printf(“Result of addition is %d \n”, res);
}
```

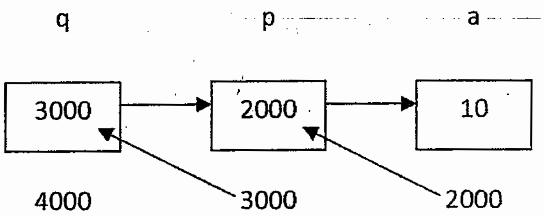
2.11 Pointer to a Pointer

Just as we have pointers to int, float or char, we can also have pointers to pointers. This is because, a pointer is also a variable. Hence, we can always have another variable (pointer) which can contain the address of the pointer variable.

EX:

```
int a = 10;     // integer variable
int *p;         // pointer to an integer
int **q;        // pointer to a pointer to an integer
p = &a;
q = &p;
```

This is pictorially as shown below.



The above shows a 2-level indirection, however, there is no limit as to how many levels of indirection we can use.

Here,

- ✓ p is the address of 'a' (2000)
- ✓ *p is the contents or value at address 2000 i.e. 10
- ✓ q is the address of p (3000)
- ✓ *q is the contents or value at address 3000 (2000)
- ✓ **q is the contents or value at address 2000 (10)
- ✓ a = *p = **q

3. Memory Allocation

3.1 Static Memory Allocation

It is a process where in the memory space is allocated during the compilation time. Here the allocated memory space cannot be expanded or reduced to accommodate more or less data, i.e. the size of the allocated memory space is fixed and it cannot be altered during execution.

EX: Consider the declaration int a[10];

During compilation, the compiler will allocate 10 memory locations for the variable 'a' and once defined cannot be changed. During execution we cannot have more than 10 data items, and if we use 3 locations, another 7 locations are wasted.

Dr.Mahesh G	Mr.Harish G
Assoc. Prof.	Assoc. Prof.
BMSIT & M	Dr. AIT

Disadvantages:

In this type of memory allocation, the data structures require a fixed amount of storage. Since the amount of storage is fixed,

- If the data structure in use, uses only a small amount of memory, rest of the memory is wasted.
- If the data structure in use, tries to use more memory than actually allocated for it results in an overflow.

Because of these limitations, this type of memory allocation can be used in applications where the data size is fixed and known before processing.

3.2 Dynamic Memory Allocation

It is the process of allocating memory space during run time. This type of memory allocation can be used in applications where the storage requirement is unpredictable.
The following are the functions using which additional memory space can be allocated or unwanted space can be deleted, thereby optimizing the use of storage space.

1. **malloc**
   Description: This function allocates and reserves a block of memory, specified in bytes and returns a pointer to the first byte of the allocated space.
   Syntax:

   ptr = (datatype *) malloc (size);

   Where - ptr is a pointer of type datatype
   - datatype can be any of the basic data type or user defined data type.
   - size is the number of bytes required.

   Example: ptr = (int *) malloc (sizeof (int))
   On successful execution of this statement, a memory space equivalent to size of int is reserved and the address of the first byte of the memory allocated is assigned to the pointer ptr of type int.
   Return Value – On success, it returns a pointer of type void to the newly allocated block of memory. On failure i.e. if the specified size of memory is not available, it returns NULL.

2. **calloc**

Description – This function allocates multiple blocks of same size, initializes all locations to zero and returns a pointer to the first byte of allocated space.

Syntax

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

Where - ptr is a pointer of type datatype
- datatype can be any of the basic data type or user defined data type.
- size is the number of bytes required.
- n is the number of blocks to be allocated of size bytes.

Example: ptr = (int *) calloc (200, sizeof (int))

On successful execution of this statement, a memory space equivalent to size of 200 int (array of 200 integers) is reserved and the address of the first byte of the memory allocated is assigned to the pointer ptr of type int.

Return Value – On success, it returns a pointer of type void to the newly allocated block of memory. On failure i.e. if the specified size of memory is not available, it returns NULL.

3. **realloc**

Description – This function is used to alter the size of the previously allocated space which is allocated either by using malloc or calloc function.

Syntax:

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

Where - ptr is the starting address of allocated memory obtained previously by calling malloc, calloc, or realloc functions.

- size is the number of bytes required for reallocation. The size specified may be larger or smaller than the previously allocated memory.

Example:

If the original allocation is done by the statement
ptr = malloc(size);
Then reallocation of space may be done by the statement
ptr = realloc(ptr, newsize)

Dr.Mahesh G Mr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

On successful execution of this statement, realloc allocates a new memory space of size newsize to the pointer variable ptr and returns a pointer to the first byte of the new memory block.

Return Value – On success, it returns a pointer to the newly allocated block of memory. On failure i.e. if the specified size of memory is not available, it returns NULL.

Note:

- The storage space allocated dynamically has no name and therefore its contents can be accessed only through a pointer.
- It is the responsibility of a programmer to de-allocate memory whenever it is not required by the application.
- In case of realloc( ), the contents of the old block will be copied into the newly allocated space and so, this function guarantees that the earlier contents are not lost.

4. **free**

Description – This function de-allocates (frees) the allocated block of memory which is allocated by using the functions malloc( ), calloc( ) or realloc( ).

Syntax:

free(ptr);

Where ptr is a pointer to a memory block which has already been created by invoking one of the 3 functions malloc( ), calloc( ) or realloc( ).

Return Value – None.

3.3 Problems with Dynamic Memory Allocation

1. Memory Leakage

This is a problem where in a part of the memory is reserved but is not accessible to any of the applications.

Example: Consider the following program segment.

```
main ( )
{
    int *a;
    a = (int *)malloc(sizeof (int) );
    *a =10;
    a = (int *)malloc(sizeof (int) );
    *a =20;
}
```

Dr.Mahesh G Assoc. Prof. BMSIT & M **Mr.Harish G** Assoc. Prof. Dr. AIT

Here, memory for the variable 'a' is allocated twice. However, 'a' contains the address of most recently allocated memory, thereby making the earlier allocated memory inaccessible i.e. the memory location where the value 10 is stored is inaccessible to any of the application and is not possible to free so that it can be reused.

2. Dangling Pointer

~~Any pointer p~~ointing to a destroyed object or ~~which does not contain a~~ valid address is called a dangling pointer.

Example: Consider the following program segment

```
main ( )
{
    int *a;
    a =(int *)malloc(sizeof(int));
    *a=20;
    .......
    .......
    free(a);
}
```

Here, if we de-allocate the memory for the variable 'a' using free(a), the memory location pointing to by it is returned to the free pool. Now the pointer variable 'a' can be used, but the contents pointed by that cannot be used, because the pointer variable 'a' does not contain a valid address now and ~~is called a~~ dangling pointer.

3.4 Comparison of Malloc and Calloc

Sl no.	Malloc ( )	Calloc ( )
1	Syntax: ptr = (datatype*) malloc (size) (Takes only one argument which is the size of the block)	Syntax: ptr = (datatype*) calloc (n, size) (Takes 2 arguments, first is number of blocks to be allocated and second is the size of each block)
2	Allocates a block of memory of specified size.	Allocates multiple blocks of memory, each block with same size.
3	Allocated space will not be initialized.	Each byte of the allocated space is initialized to zero.
4	Since no initialization takes place, time efficiency is higher than calloc( )	calloc( ) is computationally more expensive because of zero filling but, occasionally, more convenient than malloc( )
5	This function can allocate the required size of memory even if the memory is not available contiguously but available at different locations.	This function can allocate the required number of blocks contiguously. If the required memory cannot be allocated contiguously, it returns NULL.

3.5 Comparison of Static and Dynamic Memory Allocation

Sl no.	Static Memory Allocation	Dynamic Memory Allocation
1	Memory is allocated during compilation time.	Memory is allocated during run time.
2	The size of the allocated memory space is fixed and it cannot be altered during execution.	The size of the allocated memory space is not fixed and it can be altered (increased / decreased) during execution.
3	This type of memory allocation can be used in applications where the data size is fixed and known before processing.	This type of memory allocation can be used in applications where the storage requirement is unpredictable.
4	Execution is faster because memory is already allocated and only data manipulation is done.	Execution is slower because memory is has to be allocated and only then data manipulation is done.
5	Part of memory allocated is stack memory or global/static memory.	Part of memory allocated is heap memory
6	Ex: arrays	Ex: Dynamic arrays, linked lists

Dr.Mahesh G	Mr.Harish G
Assoc. Prof.	Assoc. Prof.
BMSIT & M	Dr. AIT

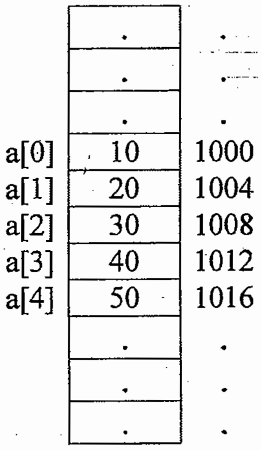
4. Arrays

Arrays is a collection of items of similar data type. All elements of an array share a common name and each element in the array is identified by using the subscript/index.

4.1 Representation

The elements of an array are stored in consecutive memory locations. For example, consider the array declaration (with initialization), int a[5] = {10, 20, 30, 40, 50};

When the compiler encounters this statement, it allocates 5 consecutive memory locations, each large enough to hold a single integer and assigns the values 10, 20, etc to appropriate locations. The memory map of this is as shown below.



	.	.
	.	.
	.	.
a[0]	10	1000
a[1]	20	1004
a[2]	30	1008
a[3]	40	1012
a[4]	50	1016
	.	.
	.	.
	.	.

UB represents the upper bound and is the largest index in the array (for above UB = 4) and LB represents the lower bound and is the smallest index in the array (for above LB = 0). In general the number of elements or the length of the array can be obtained using the formula,

Length = UB - LB + 1

Example: 4 - 0 + 1 = 5

One way of obtaining the address of each element of an array is by using the '&' operator. For example, &a[0] gives the address of the first element, &a[1] gives the address of the second element and so on.

However, the address of the first element (also called as the base address) is stored in the name of the array i.e. a = &a[0]. Hence the name of the array 'a' can be considered as a pointer, pointing to the first element of the array.

A second way of obtaining the address of each element of an array is by using the name of the array which is basically a pointer. For example, 'a' gives the address of the first element, 'a+1' gives the address of the second element, 'a+2' gives the address of the third element and so on. Meaning 'a' is same as &a[0], 'a+1' is same as &a[1] and so on.

Note: Adding an integer 'n' to a pointer, makes the pointer to point to a value 'n' elements away i.e. if 'a' is a pointer and 'n' is a integer, a + n = a + n * (sizeof(one element)).

EX: In the above memory map 'a' = 1000,
        a+3 = a + 3 * sizeof(one element);
Here a = 1000, assuming integer takes 4 bytes, sizeof(one element) = 4 and hence,
a+3 = 1000 + 3 * 4 = 1000 + 12 = 1012 = Address of a[3] i.e &a[3]

*In general, the address of a[i] can be obtained as &a[i] or 'a+i and the value of a[i] can be obtained as a[i] or *(a+i).*

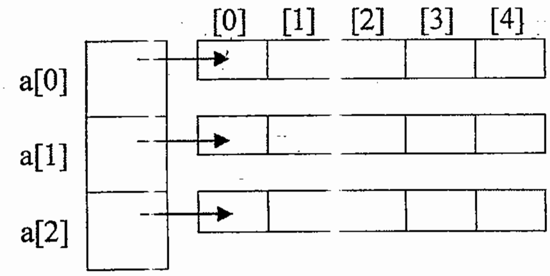
Program to print the elements of an array along with their address using pointers

```
#include<stdio.h>
void main( )
{
        int n, i, a[20] ;
        printf("Enter the number of elements\n");
        scanf("%d", &n);
        printf("Enter the elements\n");
        for( i = 0; i < n ; i + + )
        {
                scanf("%d", a+i );          // same as scanf("%d", &a[i] );
        }
        printf("The entered elements value and their corresponding address are\n");

        for( i = 0; i < n ; i + + )
        {
                printf("%u\t", a+i );       //address
                printf("%d\n", *(a+i) );    //value
        }

}
```

Program to find the sum of all elements in an array using pointers

```
#include<stdio.h>
void main( )
{
        int n, i, a[20], sum ;
        printf("Enter the number of elements\n");
        scanf("%d", &n);
        printf("Enter the elements\n");
        for( i = 0; i < n ; i + + )
        {
                scanf("%d", a+i );
        }
        sum = 0;
        for( i = 0; i < n ; i + + )
        {
                sum = sum + *(a+i);
        }
        printf("Sum of all elements = %d\n", sum);
}
```

For multidimensional arrays the concept of array-of-arrays is used. A 2-dimensional array is represented as a 1-dimensional array in which each element is itself, a 1-dimensional array. EX: To represent a 2-dimensional array int a[3][5]; we create a 1-dimensional array 'a' whose length is 3; each element of 'a' is a 1-dimensional array whose length is 5. The following shows the memory structure.



The element a[i][j] is found by first accessing the pointer in a[i]. This gives the address of the zeroth element of row 'i' of the array. Then by adding 'j' to this pointer the address of the j^{th} element of row 'i' is found

This means,

- ✓ The expression *(a + i) + j points to the j^{th} element in the i^{th} row i.e. *(a + i) + j = &a[i][j]
- ✓ The expression *(*(a + i) + j) gives the value of the element in i^{th} row and j^{th} column i.e. *(*(a + i) + j) = a[i][j].

Program to read and print an array of 'm X n' matrix using pointers

```
#include<stdio.h>

void main( )
{
        int m, n, i, j, a[20] [20];
        printf("Enter the number of rows in the matrix\n");
        scanf("%d", &m);
```

Dr.Mahesh G	Mr.Harish G
Assoc. Prof	Assoc. Prof.
BMSIT & M	Dr. AIT

```
        printf("Enter the number of columns in the matrix\n");
        scanf("%d", &n);
        printf("Enter the matrix elements\n");
        for( i = 0; i < m ; i + + )
        {
                for( j = 0; j < m ; j + + )
                {
                        scanf("%d", *(a + i) + j );
                }
        }
        printf("The entered the elements are\n");
        for( i = 0; i < m ; i + + )
        {
                for( j = 0; j < m ; j + + )
                {
                        printf("%d \t", *( *(a + i) + j ) );
                }
                printf("\n");
        }
}
```

Dr.Mahesh G Mr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

4.2 Operations

For all the operations, we use the following global declaration,

```
int *a;          // Pointer to hold the memory allocated for array
int arraysize;   // Maximum number of elements that the array can hold
int n = 0;       //Number of elements in the array at any given instant
```

25|8|18

4.2.1 Create

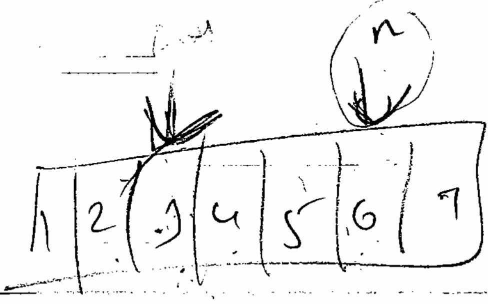
```
void create_array( )
{
        int i;
        a = (int*)malloc(arraysize*sizeof(int));
        if(a = = NULL)
        {
                printf("array creation failed\n");
                exit(0);
        }
        printf("array created successfully\n");
        printf("Enter %d elements\n", n);       // n is the current no. of elements
        for( i = 0; i < n ; i ++ )
        {
                scanf("%d", &a[i] );
        }
}
```

4.2.2 Traversing / Displaying

```
void display_array( )
{
        int i;
        if(n = = 0)
        {
                printf("No elements in the array\n");
                return;
        }
        printf("The array elements are\n");
         for(i = 0; i < n; i++)
        {
                printf("%d\t", a[i]);
        }
}
```

4.2.3 Insert

```
void insert_array(int item, int pos)
{
        int i;
        if(n = = arraysize)
        {
                printf("Array is full. Cannot insert \n");
                return ;
        }
        if(pos > n || pos < 0)
        {
                printf("Invalid position \n");
                return ;
        }

        for(i = n-1; i >= pos; i--)
        {
                a[i+1] = a[i];
        }

        a[pos] = item;

        n = n+1;
}
```



for (i =n-1; i>=pos; i-

Dr.Mahesh G Mr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

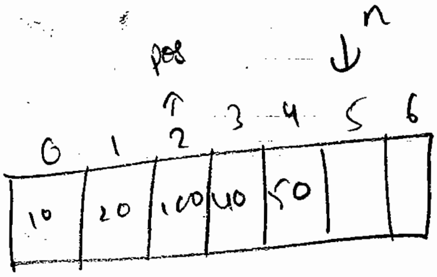
4.2.4 Delete

```
void delete_array(int pos)
{
        int i;
        if (n=0)
        { printf(" No items");
        return;
        }
        if(pos >= n || pos < 0)
        {
                printf("Invalid position \n");
                return ;
        }
        printf("Item deleted = %d \n",a[pos]);
        for(i = pos+1; i<n; i++)
        {
                a[i-1] = a[i];
        }

        n = n-1;    n--;
}
```

valid pos

4.2.5 Update

```
void update_array(int item, int pos)
{                                      if(n==0)
                                       printf("No ele in array");
        if(pos ≥ n || pos < 0)          return;
        {
                printf("Invalid position \n");
                return ;
        }
        a[pos] = item;
        printf("Updated Successfully \n");
}
```



Lab Program

Design, Develop and Implement a menu driven Program in C for the following Array operations

a. Creating an Array of N Integer Elements

b. Display of Array Elements with Suitable Headings

c. Inserting an Element at a given valid Position

d. Deleting an Element at a given valid Position

e. Exit.

Dr.Mahesh G Assoc. Prof. BMSIT & M
Mr.Harish G Assoc. Prof. Dr. AIT

```
#include<stdio.h>

int *a, arraysize, n = 0;

void main( )
{
        int  item, pos, choice;

        printf("To Create an Array\n");
        printf("Enter the maximum size of array\n");
        scanf("%d", &arraysize);
        printf("Enter the number of elements to be read initially \n");
        scanf("%d", &n);
        create_array( );
        for(;;)
        {
                printf(“1.Display  2.Insert  3.Delete 4.Exit\n”);
                printf(“Enter your choice\n”);
                scanf(“%d”, &choice);
                switch(choice)
                {
                        case 1: display_array( );
                                break;

                        case 2: printf(“Enter the item to insert\n”);
                                scanf(“%d”, &item);
                                printf(“Enter the index position to insert\n”);
                                scanf(“%d”, &pos);
                                insert_array(item, pos);
                                break;

                        case 3: printf(“Enter the index position to delete\n”);
                                scanf(“%d”, &pos);
                                delete_array(pos);
                                break;

                        case 4: free(a);
                                exit(0);
                }
        }
}
```

Dr.Mahesh G
Assoc. Prof.
BMSIT & M

Mr.Harish G
Assoc. Prof.
Dr. AIT

4.2.6 Search

Linear Search

Write a ‘C’ program to search for a given ‘key’ element in an array of ‘n’ elements using linear search technique

```
#include<stdio.h>
void main( )
{
        int n, i, a[20], key ;

        printf(“Enter the number of elements\n”);
        scanf(“%d”, &n);

        printf(“Enter the elements\n”);
        for( i = 0; i < n ; i + + )
        {
                scanf(“%d”, &a[i] );
        }

        printf(“Enter the key element to be searched\n”);
        scanf(“%d”, &key);
        for( i = 0; i < n ; i + + )
        {
                if(key = = a[i])
                {
                        printf(“SUCCESSFUL SEARCH\n” );
                        printf(“Element found at %d location\n”, i + 1 );
                        exit(0);
                }
        }
        printf(“UNSUCCESSFUL SEARCH\n” );
}
```

Binary Search

Write a ‘C’ program to search for a given ‘key’ element in an array of ‘n’ elements using binary search technique

```
#include<stdio.h>
void main( )
{
        int n, a[20], key, low, high, mid ;
        printf(“Enter the number of elements\n”);
        scanf(“%d”, &n);
        printf(“Enter the elements\n”);
        for( i = 0; i < n ; i + + )
        {
                scanf(“%d”, &a[i] );
        }

        printf(“Enter the key element to be searched\n”);
        scanf(“%d”, &key);

        low = 0;
        high = n-1;
        while( low < = high )
        {
                mid = (low + high) / 2;
                if(key = = a[mid])
                {
                        printf(“SUCCESSFUL SEARCH\n” );
                        printf(“Element found at %d location\n”, mid + 1 );
                        exit(0);
                }
                if(key < a[mid])
                        high = mid – 1;
                if(key > a[mid])
                        low = mid +1;
        }
        printf(“UNSUCCESSFUL SEARCH\n” );
}
```

4.2.7 Sort

Bubble Sort

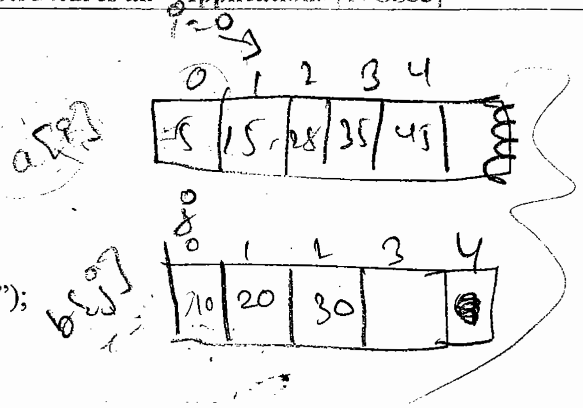
Write a 'C' program to sort 'n' elements of an array using bubble sort technique

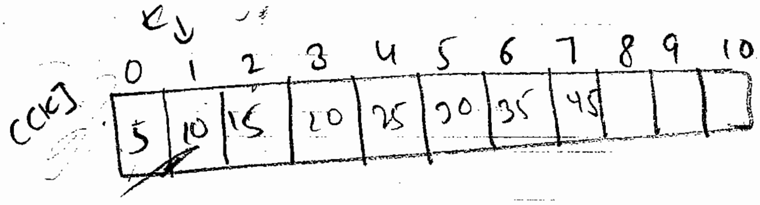
```
#include<stdio.h>
void main( )
{
        int n, i, a[20], j, temp ;

        printf("Enter the number of elements\n");
        scanf("%d", &n);

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

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

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

4.2.8 Simple Merge

Write a 'C' program to merge two sorted array elements into a single sorted array

```
#include<stdio.h>
void main( )
{
        int n1, n2, i, j, k, a[20], b[20], c[40];

        printf("Enter the number of elements for array1\n");
        scanf("%d", &n1);
printf("Enter the elements in sorted order\n");
for( i = 0; i < n1 ; i + + )
{
        scanf("%d", &a[i] );
}

printf("Enter the number of elements for array2\n");
scanf("%d", &n2);
printf("Enter the elements in sorted order\n");
for( i = 0; i < n2 ; i + + )
{
        scanf("%d", &b[i] );
}

i = 0;
j = 0;
k = 0;

/* Merging starts */
while (i < n1 && j < n2)
{
        if (a[i] < = b[j])
        {
                c[k] = a[i];
                i++;
                k++;
        }
        else
        {
                c[k] = b[j];
                j++;
                k++;
        }
}

/* Some elements in array 'a' are still remaining where as the array 'b' is exhausted */
while (i < n1)          For left values in a Ci]
{
        c[k] = a[i];
        i++;
        k++;
}

/* Some elements in array 'b' are still remaining where as the array 'a' is exhausted */
while (j < n2)
{
        c[k] = b[j];
        j++;
        k++;
}
```





```
        /* Displaying the elements of the merged array */
        printf("The merged array is \n");
        for (i = 0; i < n1 + n2; i++)
        {
                printf("%d ", c[i]);
        }
}
```

4.3 Dynamic Arrays

Arrays are simplistic yet extremely powerful data structures. Their basic limitation is their size normally has to be determined at compile time, not run time. A standard array also cannot change its size while an application is running. To overcome these drawbacks we use dynamic arrays

Dynamic array is an array whose size can be determined at run time. In addition, it is also possible to change the size of a dynamic array at run time.

Program to read and print the 'n' elements using dynamic arrays

```
#include<stdio.h>
void main( )
{
        int n, i, *a;
        printf("Enter the number of elements\n");
        scanf("%d", &n);
        a = (int*) malloc(n*sizeof(int));
        printf("Enter the elements\n");
        for( i = 0; i < n ; i + + )
        {
                scanf("%d", a + i );            // scanf("%d", &a[i] ); can also be used
        }
        printf("The entered elements are\n");

        for( i = 0; i < n ; i + + )
        {
                printf("%d\n", *(a+i) );        // printf("%d\n", a[i] ); can also be used
        }
        free(a);
}
```

Program to find maximum of 'n' elements using dynamic arrays

```
#include<stdio.h>
void main( )
{
        int n, i, *a, max;
        printf("Enter the number of elements\n");
        scanf("%d", &n);
        a = (int*) malloc(n*sizeof(int));
        printf("Enter the elements\n");
        for( i = 0; i < n ; i + + )
        {
                scanf("%d", a + i );                // scanf("%d", &a[i] ); can also be used
        }

        max = *(a+0);           // max = a[0]; can also be used
        for( i = 1; i < n ; i + + )
        {
                if(*(a+i) > max)
                {
                        max = *(a+i);
                }
        }
        printf("Maximum element = %d\n", max);
        free(a);
}
```

max a[0]
max *(a+0)

Program to find the sum of positive and negative numbers out of 'n' elements using dynamic arrays

```
#include<stdio.h>
void main( )
{
        int n, i, *a, psum, nsum;
        printf("Enter the number of elements\n");
        scanf("%d", &n);
        a = (int*) malloc(n*sizeof(int));
        printf("Enter the elements\n");
        for( i = 0; i < n ; i + + )
        {
                scanf("%d", a + i );                // scanf("%d", &a[i] ); can also be used
        }
        psum = 0;
        nsum = 0;
        for( i = 0; i < n ; i + + )
        {               a[i]
                if(*(a+i) > 0)
                {
                        psum = psum + *(a+i);
                }
                else
                {
                        nsum = nsum + *(a+i);
                }
        }
        printf("Sum of positive elements = %d\n", psum);
        printf("Sum of negative elements = %d\n", nsum
        free(a);
}
```

a[i] > 0

Dr.Mahesh G Mr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

Program to read and print an of 'm X n' matrix using dynamic arrays

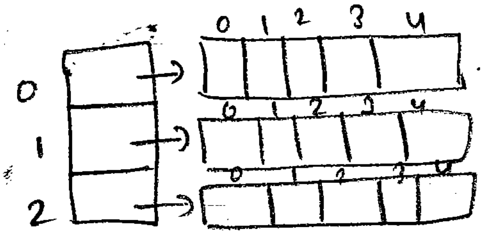
```
#include<stdio.h>
void main( )
{
        int rows, columns, i, j;
        int **a;

        printf("Enter the number of rows in the matrix\n");
        scanf("%d", &rows);
        printf("Enter the number of columns in the matrix\n");
        scanf("%d", &columns);

        // ALLOCATE MEMORY FOR ROWS
        a = (int **) malloc(rows * sizeof(int *));

        // FOR EACH ROW ALLOCATE MEMORY FOR COLUMNS
        for(i = 0; i < rows; i++)
        {
                a[i] = (int *) malloc(columns * sizeof(int));
        }

        printf("Enter the matrix elements\n");
        for( i = 0; i < rows ; i + + )
        {
                for( j = 0; j < columns ; j + + )
                {
                        scanf("%d", *(a + i) +j);    // scanf("%d", &a[i][j] );

                }
        }
        printf("The entered the elements are\n");
        for( i = 0; i < rows ; i + + )
        {
                for( j = 0; j < columns ; j + + )
                {
                        printf("%d \t", *( *(a + i) +j ) );      // printf("%d \t", a[i][j] );
                }
                printf("\n");
        }
free(a);
}
```

m n

code

& a[i][j] = *(a+i)+j

a[i][j] = *(*(a+i)+j)



rows = 3
columns = 5

if int then int *
if int* then int **

Dr.Mahesh G
Assoc. Prof.
BMSIT & M

Mr.Harish G
Assoc. Prof.
Dr. AIT

4.4 Multidimensional Arrays

Multidimensional arrays can be represented either by using array-of-arrays representation or by using a linear list with consecutive memory location as in a one dimensional array.

A multidimensional array can be declared as $a[u_0][u_1][u_2]$ ....$[u_{n-1}]$ and the number of elements in it is u_0 X u_1 X u_2 X .... X u_{n-1}.

EX: If we declare an array [10][10][10] then we require 10X10X10 = 1000 units of storage to hold the array.

There are two major ways to represent multidimensional arrays, row major order and column major order. We consider the row major order, where the rows of the multidimensional arrays are stored in contiguous memory locations.

For a one-dimensional array $a[u_0]$, if we assume 'α' as the starting address, then

Address of a[0] is α

Address of a[1] is α + 1

Address of a[2] is α + 2

…………………………

…………………………

…………………………

Address of $a[u_0 - 1]$ is $\alpha + (u_0 - 1)$

In general, Address of a[i] is α + i

Dr.Mahesh G Mr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

Similarly,

For a 2-dimensional array $a[u_0][u_1]$, the address of $a[i][j] = \alpha + i * u_1 + j$

For a 3-dimensional array $a[u_0][u_1][u_2]$, the address of $a[i][j][k] = \underline{\alpha + i * u_1 * u_2} + \underline{j * u_2} + \underline{k}$

In general, for a 'n' dimensional array, $a[u_0][u_1][u_2]$ ……..$[u_{n-1}]$, if we assume 'α' as the starting address i.e. address of a[0][0][0] ……..[0] then

Address of $a[i_0][0][0]$ ……. $[0] = \alpha + i_0 * u_1 * u_2 \ldots\ldots * u_{n-1}$

Address of $a[i_0][i_1][0]$ ……..$[0] = \alpha + i_0 * u_1 * u_2 \ldots\ldots * u_{n-1} + i_1 * u_2 * u_3 \ldots\ldots * u_{n-1}$

Repeating in this way the address of $a[i_0][i_1][i_2]$ ……..$[i_{n-1}]$ is

$$\begin{aligned}
&\alpha + i_0 * u_1 * u_2 \ldots\ldots * u_{n-1}\\
&\quad + i_1 * u_2 * u_3 \ldots\ldots * u_{n-1}\\
&\quad + i_2 * u_3 * u_4 \ldots\ldots * u_{n-1}\\
&\quad \ldots\ldots\\
&\quad \ldots\ldots\\
&\quad + i_{n-2} * u_{n-1}\\
&\quad + i_{n-1}
\end{aligned}$$

$$= \alpha + \sum_{j=0}^{n-1} i_j a_j \qquad \text{Where, } \begin{cases} a_j = \overset{\text{product of } U_k}{\prod_{k=j+1}^{n-1}} u_k \quad 0 \le j < n-1 \\ a_{n-1} = 1 \end{cases}$$

1/9/18

5.1 Sparse Matrices

A "m x n" matrix is said to be a sparse matrix if many of its elements are zero. Although it is difficult to determine exactly whether a matrix is sparse or not, intuitively we can recognize a sparse matrix when we see one.

EX: Consider the following 6x6 matrix,

$$\begin{bmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{bmatrix}$$

Here only 8 of 36 elements are nonzero, and hence the matrix is sparse.

5.1.1 Sparse Matrix Representation

Just like any other matrix, a 2-dimensional array can be used to represent a sparse matrix; however, a lot of space is wasted in this representation.

EX: Consider the space necessary to store a 1000 x 1000 matrix that has only 2000 nonzero elements. The corresponding 2-dimensional array requires space for 1,000,000 elements. To save space and running time it is critical to only store the nonzero elements.

Representation

- ✓ Any element within a matrix can be represented using a triple <row, col, value>.
- ✓ Triples are organized so that the row indices are in ascending order, and for any given row, the column indices are also in ascending order.
- ✓ To perform any operation on a sparse the following needs to be known
  - Number of rows
  - Number of columns
  - Number of nonzero elements

We use the following structure to represent a sparse matrix.

```
struct sparsematrix
{
        int row;
        int col;
        int value;
};
struct sparsematrix a[100];
```

Dr.Mahesh G
Assoc. Prof.
BMSIT & M

Mr.Harish G
Assoc. Prof.
Dr. AIT

EX: The above sparse matrix can be represented as

row no =0 non zero val = 15
col no = 0

rol = 0
col = 3 NZV = 22

rol:0
col = 5 NZV = -15

	row	col	value
a[0]	**6**	**6**	**8**
a[1]	0	0	15
a[2]	0	3	22
a[3]	0	5	-15
a[4]	1	1	11
a[5]	1	2	3
a[6]	2	3	-6
a[7]	4	0	91
a[8]	5	2	28

Here,

- ✓ a[0]. row contains the number of rows.
- ✓ a[0]. col contains the number of columns.
- ✓ a[0]. value contains the total number of nonzero elements.
- ✓ The triples are ordered by rows and within rows by columns.

5.1.2.Transposing a Sparse Matrix

Transpose of a matrix can be done by interchanging the rows and columns. This means each element a[i][j] in the original matrix will become b[j][i] in the transposed matrix.
Direct interchanging of rows and columns is not correct in case of sparse matrices because the ordering of rows and columns in the representation will be lost.

Algorithm

```
for all elements in column j
        place < i, j, value > from original matrix to < j, i, value > in the transposed matrix
```

This indicates that we should find all the elements in columns 0 and store them in row 0 of the transpose matrix, then find all elements in column 1 and store them in row 1 etc.

EX: Matrix 'a' and its transpose 'b' is as shown below

	row	col	value
a[0]	**6**	**6**	**8**
a[1]	0	0	15
a[2]	0	3	22
a[3]	0	5	-15
a[4]	1	1	11
a[5]	1	2	3
a[6]	2	3	-6
a[7]	4	0	91
a[8]	5	2	28

	row	col	value
b[0]	**6**	**6**	**8**
b[1]	0	0	15
b[2]	0	4	91
b[3]	1	1	11
b[4]	2	1	3
b[5]	2	5	28
b[6]	3	0	22
b[7]	3	2	-6
b[8]	5	0	-15

k=1

The following is the function for finding the transpose of a sparse matrix. The array 'a' holds the original matrix and the array 'b' holds the transposed matrix. The structure used in the function is given by

Dr.Mahesh G Assoc. Prof. BMSIT & M
Mr.Harish G Assoc. Prof. Dr. AIT

```
struct sparsematrix
{
        int row;
        int col;
        int value;
};
typedef struct sparsematrix MATRIX;

// Function to find the transpose of a sparse matrix
void transpose(MATRIX a[ ], MATRIX b[ ])
{
        int n;
        int i, j, k;
```

Transpose

```
    b[0].row = a[0].col;          // rows in b = cols in a
    b[0].col = a[0].row;          // cols in b = rows in a
    b[0].value = a[0].value;      // number of elements in b = number of elements in a

    n = a[0].value;               // total number of elements

    if(n>0)                       // nonzero matrix
    {
            k = 1;                // position to put the next transposed triple in b

            for(i=0; i<a[0].col; i++)
            {
                    for(j=1; j<= n; j++)
                    {
                            if(a[j].col = = i)
                            {
                                    b[k].row = a[j].col;
                                    b[k].col = a[j].row;
                                    b[k].value = a[j].value;
                                    k++;
                            }
                    }
            }
    }
}
```

Dr.Mahesh G	Mr.Harish G
Assoc. Prof.	Assoc. Prof.
BMSIT & M	Dr. AIT

5.2 Polynomials

Polynomial Description

A polynomial is a sum of terms, where each term has a form '$a\,x^{b}$', where 'x' is a variable, 'a' is a coefficient and 'b' is the power or exponent. The largest power or exponent of a polynomial is called as its degree.

EX: $A(x) = 3x^{20} + 2x^5 + 4$ (degree = 20)
$B(x) = x^4 + 10x^3 + 3x^2 + 1$ (degree = 4)

Note:

- ✓ Coefficients that are zero are not displayed.
- ✓ Term with exponent / power equal to zero does not show the variable 'x' since $x^0 = 1$.
- ✓ If $A(x) = \sum a_i x^i$ and $B(x) = \sum b_i x^i$ are two polynomials then $A(x) + B(x) = \sum (a_i+b_i)\, x^i$.

5.2.1Representation

Assumption

- ✓ Exponents / powers are arranged in decreasing order.

Representation – 1

We use the following structure to represent a polynomial.

```
#define MAX_DEGREE 101          // Max degree of polynomial + 1
```

```
struct polynomial
{
        int degree;
        float coefficient[MAX_DEGREE];
};
```

Let ‘a’ be a variable of this structure defined as struct polynomial a;

The polynomial $A(x) = \sum_{i=0}^{n} a_i x^i$ would be represented as

a.degree = n // n is the max no. of terms = degree + 1
a.coefficent[i] = a_{n-i}

In this representation, we store the coefficients in order of decreasing exponents, such that a.coefficient[i] is the coefficient of x^{n-i} provided the term with exponent n-i exists; otherwise, a.coefficient[i] = 0.

EX: The representation for the polynomial $A(x) = 7x^{99} + 5x^{50} + 3x^{2} + 1$ is as shown below.

Index / Exponent	0	1	2	3	.	.	.	24	25	.	.	.	49	50	.	.	.	98	99	100
a.coefficient[ ]	0	7	0	0	.	.	.	0	0	.	.	.	0	5	.	.	.	3	0	1

a.coefficent[1] = $a_{100-1} = a_{99} = 7$. // n = 99 +1
a.coefficent[98] = $a_{100-98} = a_{2} = 3$.

The drawback of this representation is that it wastes lot of space. This is because,

- ✓ If a.degree is very much less than MAX_DEGREE then most of the positions in a.coefficient[MAX_DEGREE] will be wasted.
- ✓ If the polynomial is sparse i.e. the number of terms with non-zero coefficient is very less.

Representation – 2

To overcome the drawbacks of Representation – 1, we use only one global array to store all the polynomials. We use the following structure to represent a polynomial.

```
#define MAX_TERMS 100
struct polynomial
{
        float coefficient;
        int exponent;
};
struct polynomial a[MAX_TERMS];
int avail = 0;
```

Dr.Mahesh G
Assoc. Prof.
BMSIT & M

Mr.Harish G
Assoc. Prof.
Dr. AIT

EX: The representation for the polynomial $A(x) = 7x^{99} + 1$ and $B(x) = 5x^{4} + 3x^{3} + 7x^{2} + 1$ is as shown below.

	startA ↓	finishA ↓	startB ↓				finishB ↓	avail ↓							
coefficient	7	1	5	3	7		1								
Exponent	99	0	4	3	2		0								
	0	1	2	3	4		5	6	7	8	9	.	.	.	. .

powers

⊕ avail is index of next free location where we can store array.

The index of the first term of A . id B is given by startA and startB. The index of the last term of A and B is given by finis A and finishB. The index of the next free location in the array is given by avail.

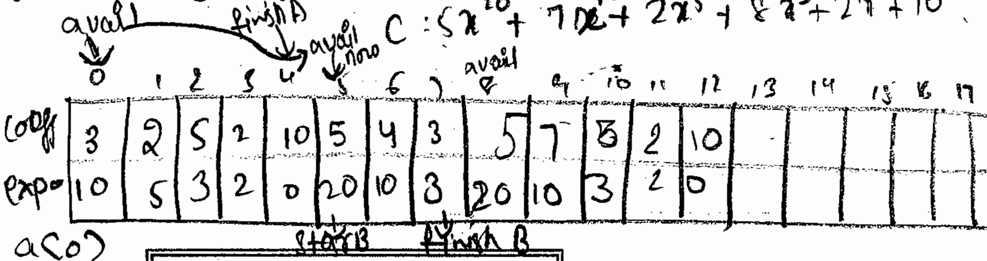
The drawback of this representati 1 is that when all the terms are non-zero, it requires about twice as much space as the first or .

$A = 3x^{10} + 2x^5 + 5x^3 + 2x^2 + 10$

$B = 5x^{20} + 4x^{10} + 3x^3$

$C : 5x^{20} + 7x^{10} + 2x^5 + 8x^3 + 2x^2 + 10$

5.2.3 Program to add two Polyn nials using Arrays



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
coeff	3	2	5	2	10	5	4	3	5	7	8	2	10					
expo	10	5	3	2	0	20	10	3	20	10	3	2	0					

Dr.Mahesh G Mr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

```
#include<stdio.h>
#define MAX_TERMS 100

struct polynomial
{
        int coefficient;
        int exponent;
};

struct polynomial a[MAX_TERM!.];
int avail = 0;

void main( )
{
        int na, nb, i, startA, finishA . startB, finishB,  startC, finishC, nc;

        printf("Enter the number oi terms in Polynomial A\n");
        scanf("%d", &na);
        startA = avail;  A will point to avail which is at '0' in beginnig.
        printf("Enter the coefficien: and exponents of the Polynomial A\n");
        for(i = 0; i<na; i++)
        {
                printf("Enter the co:efficient \n");
                scanf("%d", &a[av:il].coefficient);  avail giving coeff
                printf("Enter the co:responding exponent \n");
                ~~scanf("%d", &a[av:il].exponent);  avail giving expo~~
                avail++;
        }
        finishA = avail – 1;  Here avail is at 5 ∴ finishA is -1

        printf("Enter the number oi terms in Polynomial B\n");
        scanf("%d", &nb);
        startB = avail;
        printf("Enter the coefficien: and exponents of the Polynomial B\n");
        for(i = 0; i<nb; i++)
        {
                printf("Enter the co:efficient \n");
                scanf("%d", &a[av:il].coefficient);
                printf("Enter the co:responding exponent \n");
                scanf("%d", &a[av:il].exponent);
                avail++;
        }
        finishB = avail – 1;
```

now 8 becomes avail.

for 'c'

```
        nc = padd(startA, finishA, startB, finishB, &startC, &finishC);

        printf("Polynomial A is ");
        display(startA, na);

        printf("Polynomial B is ");
        display(startB, nb);

        printf("Sum of Polynomial A and Polynomial B is ");
        display(startC, nc);
}


int padd(int startA, int finishA, int startB, int finishB, int* startC, int* finishC)
{
        int nc = 0;

        *startC = avail;   at 8, as avail is at 8 so 'c' starts from avail
        while(startA<=finishA && startB<=finishB)
        {
                switch( compare(a[startA].exponent, a[startB].exponent ))   here both x^10, x^20 expo are compared.
                {
                        case -1: // a's exponent < b's exponent
                                a[avail].coefficient = a[startB].coefficient;   as comparision is done
                                a[avail].exponent = a[startB].exponent;         5x^20 is large
                                startB++;                                       at avail we put
                                avail++;                                        5x^20
                                nc++;
                                break;

                        case  0: // a's exponent = b's exponent
                                if((a[startA].coefficient  + a[startB].coefficient) != 0)
                                {
                                    a[avail].coefficient = a[startA].coefficient  + a[startB].coefficient;
                                    a[avail].exponent = a[startA].exponent;
                                    startA++;
                                    startB++;
                                    avail++;
                                    nc++;
                                }

                                else
                                {
                                    startA++;
                                    startB++;
                                }
                                break;

                        case  1: // a's exponent > b's exponent
                                a[avail].coefficient = a[startA].coefficient;
```

```
                    a[avail].exponent = a[startA].exponent;
                    startA++;
                    avail++;
                    nc++;
                    break;

            }
        }

        // Add remaining terms of Polynomial A
        while(startA < = finishA)
        {
            a[avail].coefficient = a[startA].coefficient;
            a[avail].exponent = a[startA].exponent;
            startA++;
            avail++;
            nc++;
        }

        // Add remaining terms of Polynomial B
        while(startB <= finishB)
        {
            a[avail].coefficient = a[startB].coefficient;
            a[avail].exponent = a[startB].exponent;
            startB++;
            avail++;
            nc++;
        }

        *finishC = avail – 1;
        return nc;
}
```

```
void display(int start, int n)
{
        int i;                                  (3x^10)
        for(i=1; i<=n; i++)
        {
            if(a[start].coefficient > 0) To check coeffs of +ue
                printf("+");
            printf("%d",a[start].coefficient); 3

            printf("x ^");
            printf("%d",a[start].exponent); x^10
            start++;
        }
}
```

```
int compare(int x, int y)
{
        if(x<y)
            return -1;
        else if(x = =y)
            return 0;
          else
            return 1;
}
```

6. Strings

A string is any finite sequence of characters (i.e., letters, numerals, symbols and punctuation marks).

6.1 Representation

In C strings are represented by arrays of characters. The end of the string is marked with a special character, the null character (\0).

The following declaration and initialization create a string consisting of the word "bmsit". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "bmsit"

char str[6] = {'b', 'm', 's', 'i', 't' '\0'};

If you follow the rule of array initialization then you can write the above statement as

char str[ ] = "bmsit";

The following is the memory representation of the above string

Index	0	1	2	3	4	5
String	b	M	s	i	t	\0
Address	1000	1001	1002	1003	1004	1005

The NULL character at the end of the string is automatically placed by the compiler when it initializes the array.

Dr.Mahesh G Mr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

6.2 Storing Strings

Strings are stored in three types of structures

1) Fixed Length Structures - Here each line of print is viewed as a record, where all records have the same length.

Example: Consider a the following FORTRAN program,

```
C PROGRAM PRINTING TWO INTEGERS IN INCREASING ORDER
        READ *, J, K
        IF(J.LE.K) THEN
                PRINT *, J, K
        ELSE
                PRINT *, K, J
        ENDIF
        STOP
        END
```

Assuming 200 is the address of the first character of the program, the following is the fixed length structure.

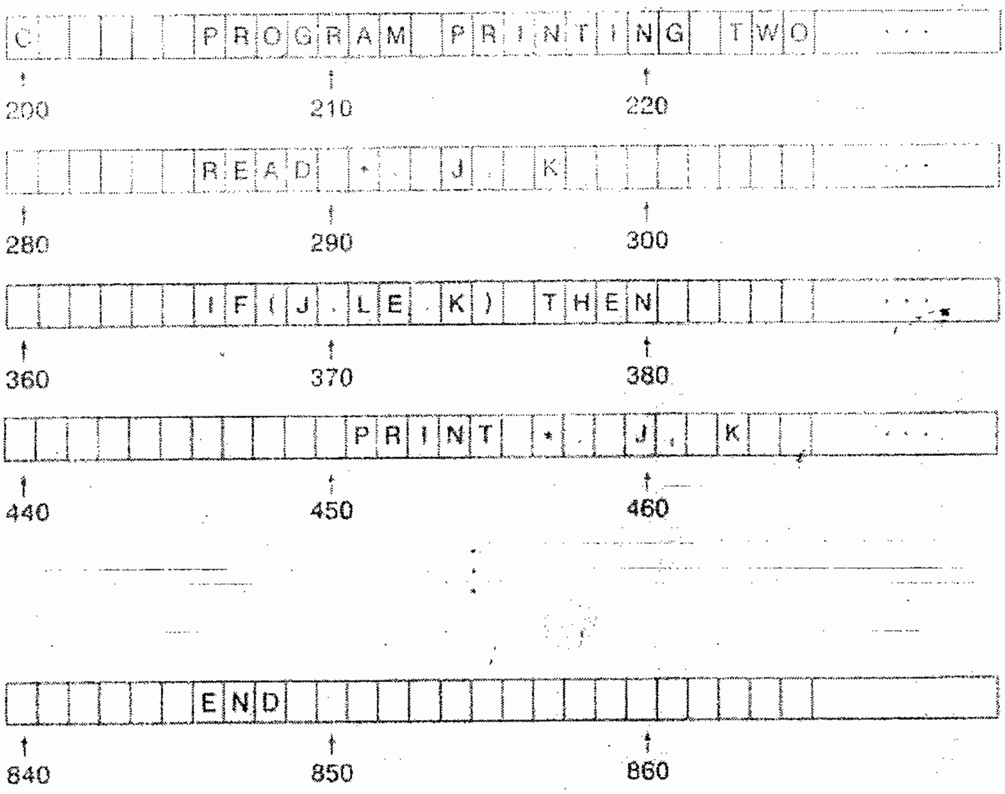


Fig. 3.2 *Records Stored Sequentially in the Computer*

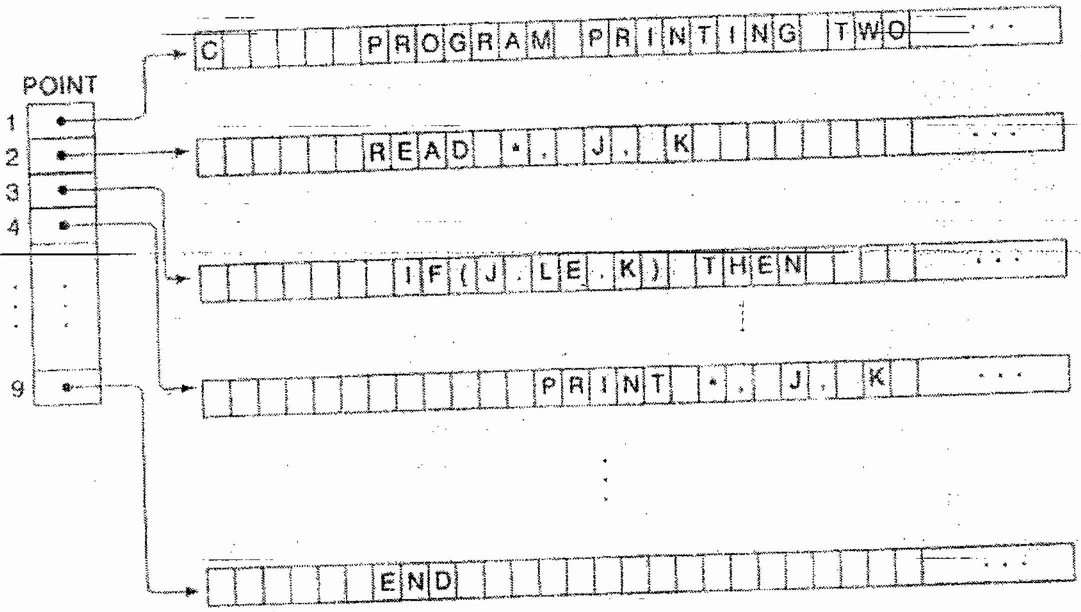


Fig. 3.3 *Records Stored Using Pointers*

Advantages

- The ease of accessing data from any given record
- The ease of updating data in any given record (as long as the length of the new data does not exceed the record length)

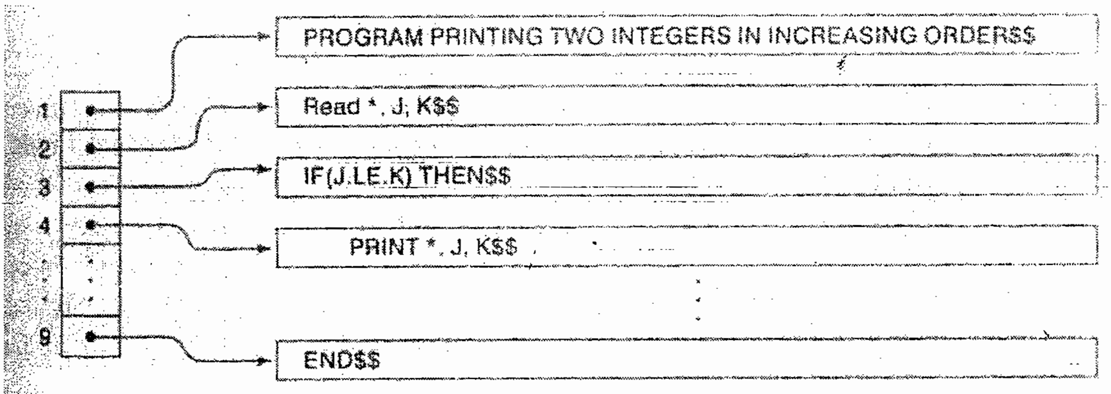
Disadvantages

- Time is wasted reading an entire record if most of the storage consists of inessential blank spaces.
- Certain records may require more spaces than available.
- When the correction consists of more or fewer characters than the original te , changing a misspelled word requires the entire record to be changed.

2) Variable Length Structures with fixed maximums

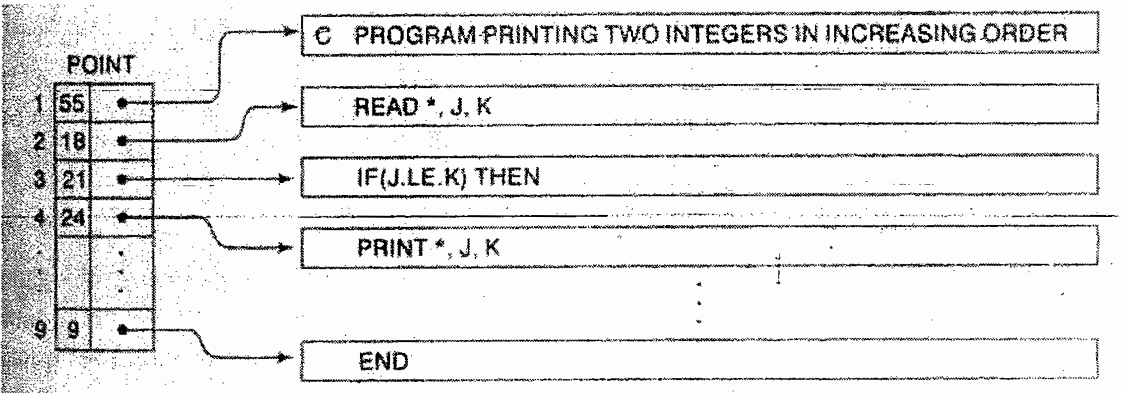
In this method, the strings are stored using one the following ways

a) Using a end marker such as two dollar signs ($$), to signal the end of the string.



(a) Records with sentinels.

b) List the length of the string as an additional item in the pointer array.



(b) Record whose lengths are listed.

Advantage

- We need not have to read the entire record when the string occupies only the beginning part of the memory location.

Disadvantage

- This method of storage is usually inefficient when the strings and their lengths are frequently being changed.

3) Linked Structures

Computers are frequently used for word processing applications. The ability to correct and modify the printed matter (deleting, changing and inserting words, phrases, sentences and paragraphs) by the computer must be good. Hence for extensive word processing applications, strings are stored by means of linked lists.

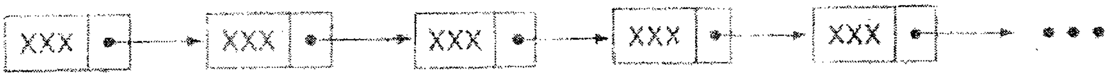
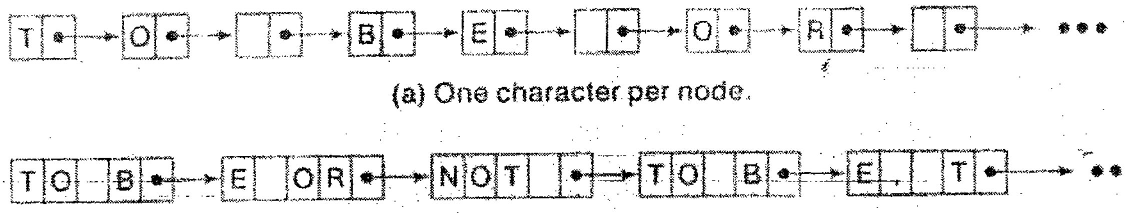


Fig. 3.6 *Linked List*



(a) One character per node.

(b) Four characters per node.

Fig. 3.7

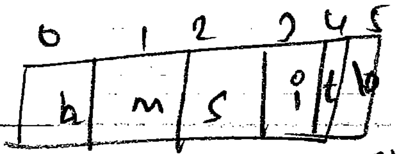
Each memory cell is assigned one character or a fixed number of characters, and a link contained in the cell gives the address of the cell containing the next character or group of characters in the string.

6.3 String Operations

6.3.1 String Length

Program to find the length of a string

```
#include<stdio.h>
int my_strlen(char str[ ])
{
        int i = 0;
        while( str[i] != '\0')
        {
                i + +;
        }
        return i;
}
void main( )
{
        char str[20];
        int i;

        printf("Enter the string\n");
        gets(str);

        i = my_strlen(str);

        printf("Length of the string = %d\n", i);
}
```



stop when it comes to \0

Dr.Mahesh G Mr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

6.3.2 String Copy

Program to copy string src to string dest

```
#include<stdio.h>
void my_strcpy(char dest[ ], char src[ ])
{
        int i = 0;
        while( src[i] != '\0')
        {
                dest[i] = src[i];
                i + +;
        }
        dest[i] = '\0';
}
void main( )
{
        char src[20], dest[20];

        printf("Enter the string\n");
        gets(src);

        my_strcpy(dest, src);

        printf("Destination string = %s\n", dest);
}
```

Dr.Mahesh G Mr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

6.3.3 String NCopy

Program to copy 'n' characters from string src to string dest

```
#include<stdio.h>
void my_strncpy(char dest[ ], char src[ ], int n)
{
        int i;
        if( n < 0 || n > my_strlen(src))
        {
                printf(" n value out of bounds\n");
                exit(0);
        }

        for(i=0; i<n; i++)
        {
                dest[i] = src[i];
        }
        dest[i] = '\0';
}
void main( )
{
        char src[20], dest[20];
        int n;

        printf("Enter the string\n");
        gets(src);
        printf("Enter the number of characters to be copied from source string\n");
        scanf("%d",&n);

        my_strncpy(dest, src, n);

        printf("Destination string = %s\n", dest);
}
```

6.3.4 String Concatenation

Program to concatenate two strings

```
#include<stdio.h>
void my_strcat(char str1[ ], char str2[ ])
{
        int i, j;

        i = 0;
        while( str1[i] != '\0')
        {
                i++;
        }

        j=0;
        while( str2[j] != '\0')
        {
                str1[i] = str2[j];
                i++;
                j++;
        }
        str1[i] = '\0';
}

void main( )
{
        char str1[20], str2[20];

        printf("Enter the string1\n");
        gets(str1);

        printf("Enter the string2\n");
        gets(str2);


        my_strcat(str1, str2);

        printf("Concatenated string = %s\n", str1);
}
```

Dr.Mahesh G Mr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

* Study

6.3.5 String Compare

Program to compare two strings

```
#include<stdio.h>
int my_strcmp(char str1[ ], char str2[ ])
{
        int i = 0;

        while( str1[i] != '\0' && str2[i] != '\0')
        {
                if(str1[i] > str2[i])
                        return 1;
                else if(str1[i] < str2[i])
                        return -1;
                else
                        i++;
        }
        if(str1[i] != '\0' && str2[i] = = '\0')
                return 1;

        if(str1[i] = = '\0' && str2[i] != '\0')
                return -1;

        return 0;
}

void main( )
{
        char str1[20], str2[20];
        int difference;

        printf("Enter the string1\n");
        gets(str1);

        printf("Enter the string2\n");
        gets(str2);


        difference = my_strcmp(str1, str2);

        if(difference = = 0)
                printf("%s = %s\n", str1, str2);
        else if (difference = = 1)
                printf("%s > %s\n", str1, str2);
        else if (difference = = -1)
                printf("%s < %s\n", str1, str2);

}
```

6.3.6 String Reverse

Program to reverse a string

```
#include<stdio.h>
void my_strrev(char str[ ], char revstr[ ])
{
        int i, j;

        i = 0;
        while( str1[i] != '\0')
        {
                i + +;
        }

        i - -;
        j=0;
        while( i > = 0)
        {
                revstr[j] = str[i];
                i - -;
                j + +;
        }
        revstr[j] = '\0';
}

void main( )
{
        char str[20], revstr[20];

        printf("Enter the string1\n");
        gets(str);

        my_strrev(str, revstr);

        printf("Reversed string = %s\n", revstr);
}
```

7 Applications of Strings

Pattern Matching

Pattern matching is the problem of deciding whether or not a given string of pattern 'P' appears in the text 'T'

First Pattern Matching Algorithm (Brute Force Pattern Matching Algorithm)

Here we compare a given pattern 'P' with each of the substrings of 'T', moving from left to right, until we get a match or the pattern is not found.

Let W_k denote the substring of 'T' having the same length as 'P' and beginning with the K^{th} character of T.

First we compare 'P', character by character, with the first substring W_0. If all the characters are the same, the $P = W_0$ and so P appears in T. Its INDEX (starting position where the substring is found) is 0.

On the other hand suppose we find that some character of P is not the same as the corresponding character of W_2. Then $P \neq W_2$ and we can immediately move onto the next substring W_3.

That is we next compare P with W_3. If $P \neq W_3$, then we compare P with W_4, and so on. This process ends when

- ✓ we find a match of P with some substring W_K and so P appears in T with INDEX = K OR
- ✓ we exhaust all the W_K's with no match and hence P does not appear in T.

Illustration

Let P be a 4 character string and T be a 20 character string.

$P = P[0]P[1]P[2]P[3]$

$T = T[0]T[1]T[2]T[3]$..................$T[18]T[19]$

Now P is compared with each of the following 4-character substring of T,

$W_0 = T[0]T[1]T[2]T[3]$

$W_1 = T[1]T[2]T[3]T[4]$

$W_2 = T[2]T[3]T[4]T[5]$

.

.

.

$W_{16} = T[16]T[17]T[18]T[19]$

Note: Maximum number of substring is given by LENGTH(T) - LENGTH(P) + 1

Ex: 20 - 4 + 1 = 17 i.e. W range from 0 to 16

Algorithm First Pattern Matching

Let

'T' denote the text string

'P' denote the pattern string

'N' denote LENGTH(T)

'M' denote LENGTH(P)

```
Repeat for I = 0 to N - M
{
        Repeat for J = 0 to M - 1
        {
                if(T[I+J] ≠ P[J] )
                        break;
        }
        if J = = M
                return  I                  // Success
}
Return -1       // Failure
```

Function to search for a Pattern in Text
OR
Function to search for a substring in a string

```
int search(char pattern[ ], char text[ ])
{
        int m,n, i,j;
        m = strlen(pattern);
        n = strlen(text);
        for( i = 0; i < = n - m; i++ )
        {
                for( j = 0; j < m; j++ )
                {
                        if(text[i+j] != pattern[j])
                                break;
                }
                if( j= = m)
                        return i;          // Success
        }
        return - 1;                        // Failure
}
```

Design, Develop and Implement a Program in C for the following operations on Strings

a. Read a main String (STR), a Pattern String (PAT) and a Replace String (REP)

b. Perform Pattern Matching Operation: Find and Replace all occurrences of PAT in STR with REP if PAT exists in STR.

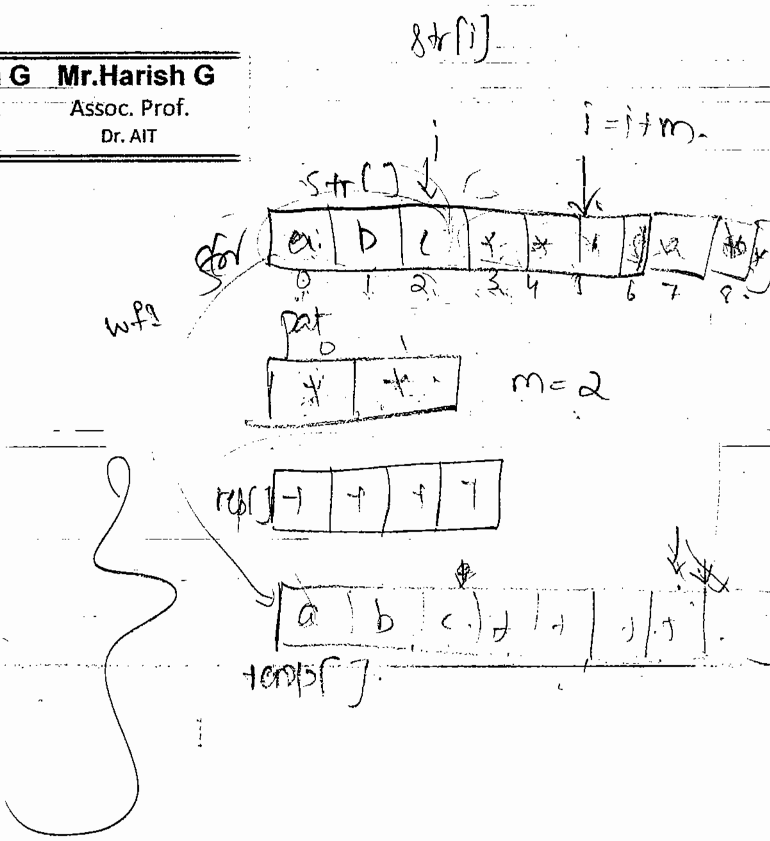
Report suitable messages in case PAT does not exist in STR
Support the program with functions for each of the above operations.
Don't use Built-in functions.

```
#include<stdio.h>
// Function to find the length of a string
int my_strlen(char str[ ])
{
        int i = 0;
        while( str[i] != '\0')
        {
                i + +;
        }
        return i;
}
```

```
// Function to copy string src to string dest
void my_strcpy(char dest[ ], char src[ ])
{
        int i = 0;
        while( src[i] != '\0')
        {
                dest[i] = src[i];
                i + +;
        }
        dest[i] = '\0';
}

// Function to copy 'n' characters from string src to string dest
void my_strncpy(char dest[ ], char src[ ], int n)
{
        int i;
        if( n < 0 || n > my_strlen(src))
        {
                printf(" n value out of bounds\n");
                exit(0);
        }

        for(i=0; i<n; i++)
        {
                dest[i] = src[i];
        }
        dest[i] = '\0';
}
// Function to concatenate two strings
void my_strcat(char str1[ ], char str2[ ])
{
        int i, j;

        i = 0;
        while( str1[i] != '\0')
        {
                i + +;
        }

        j=0;
        while( str2[j] != '\0')
        {
                str1[i] = str2[j];
                i + +;
                j + +;
        }
        str1[i] = '\0';
}
```

```
void main( )
{
        char str[100], )at[100], repstr[100], temp[100];
        int  m, n, i, j,   ag, len;

        printf("Enter  ie main string\n");
        gets(str);

        printf("Enter  ie pattern string\n");
        gets(pat);

        printf("Enter  ie replace string\n");
        gets(repstr);

        m = my_strle: (pat);

        flag = 0;
        i = 0;
        while( str[i] ! = '\0')
        {
                for( j = 0; j < m; j++ )
                {
                        if(str[i+j] != pat[j])
                                break;
                }
                if( j= = m)
                {
                        flag ++;
                        my_strncpy(temp, str, i);
                        my_strcat(temp, repstr);
                        i = i + m;
                        len = my_strlen(temp);
                        ~~my_strcat(temp, str + i);~~
                        my_strcpy(str, temp);
                        i = len;
                }
                else
                {
                        i = i + 1;
                }
        }

        if(flag = = 0)
                printf("Pattern not found in main string\n");
        else
        {
                printf("Pattern found %d times in main string\n", fla ;);
                printf("The resultant string after replacing is %s\n", ;tr);
        }
}
```

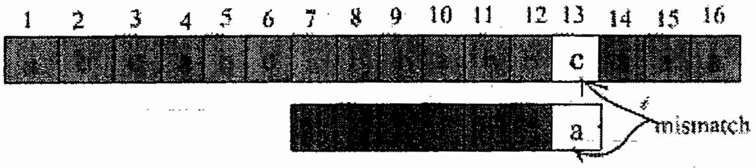
Dr.Mahesh G Mr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT



Knuth-Morris-Pratt(KMP) string matching algorithm

In the previous method the first element of the pattern to be searched 'P' is compared with the first element of the string 'S'. If it matches, then the second element of 'P' is matched with the second element of 'S'. If match found proceed likewise until entire ‘P’ is found. If a mismatch is found at any position, shift ‘P’ one position to the right and repeat comparison beginning from first element of ‘P’ i.e. irrespective of the structure of the pattern *P*, if there is a mismatch, we restart the matching process with shift *S* = *S*+1. But if we know the structure of the pattern, we can intelligently avoid some number of shifts.

Example:



Suppose 6 characters are matched successfully and there is a mismatch at the 7 th position. The shift *S* =*S*+1 is obviously invalid as the first pattern character ‘a' would be aligned to the text character ‘b', which is known to match the second pattern character. Also if we observe, the shift *S* = *S*+2 is also invalid. But the shift S = S+3 may be valid as first three characters of the pattern will be nicely aligned to the last three characters of the portion of the text which are already matched.

The basic idea behind KMP’s algorithm is: whenever we detect a mismatch (after some matches), we already know some of the characters in the string (since they matched the pattern characters prior to the mismatch). We take advantage of this information to avoid matching the characters that we know will anyway match.

Proper Prefix and Proper Suffix

Proper Prefix - All the characters in a string, with one or more cut off the end.

Proper Suffix - All the characters in a string, with one or more cut off the beginning.

Example: For the word "India"

Proper Prefix = {"Indi","Ind","In","I"}

Proper Suffix = {"ndia","dia","ia","a"}

Dr.Mahesh G Mr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

Step 1: Constructing Table for a given Pattern [Failure Function]

Note: The length of the longest proper prefix in the pattern that matches a proper suffix in the same pattern needs to be found.

Example 1: Pattern "ABCDABD"

J	0	1	2	3	4	5	6
Substring 0 to J	A	AB	ABC	ABCD	ABCDA	ABCDAB	ABCDABD
Longest Prefix Suffix Match	None	None	None	None	A	AB	None
Length of Prefix Suffix	0	0	0	0	1	2	0

Example 2: Pattern "AABAACAABAA"

J	0	1	2	3	4	5	6	7	8	9	10
Substring 0 to J	A	AA	AAB	AABA	AABAA	AABAAC	AABAACA	AABAACAA	AABAACAAB	AABAACAABA	AABAACAABAA
Longest Prefix Suffix Match	-	A	-	A	AA	-	A	AA	AAB	AABA	AABAA
Length of Prefix Suffix	0	1	0	1	2	0	1	2	3	4	5

Example 3: Pattern "AAABAAA"

J	0	1	2	3	4	5	6
Substring 0 to J	A	AA	AAA	AAAB	AAABA	AAABAA	AAABAAA
Longest Prefix Suffix Match	-	A	AA	-	A	AA	AAA
Length of Prefix Suffix	0	1	2	0	1	2	3

Example 4: Pattern "ABABABCA"

J	0	1	2	3	4	5	6	7
Substring 0 to J	A	AB	ABA	ABAB	ABABA	ABABAB	ABABABC	ABABABCA
Longest Prefix Suffix Match	-	-	A	AB	ABA	ABAB	-	A
Length of Prefix Suffix	0	0	1	2	3	4	0	1

Step 2: Obtaining skip lengths and matching pattern [KMP]

Note:

Rule 1: If Mismatch occurs at the first position of pattern, then shift pattern by one position.
Rule 2: If a partial match of length **len** is found and **table[len - 1] = 0**, we may skip ahead **len** characters and continue matching.
Rule 3: If a partial match of length **len** is found and **table[len - 1] > = 1**, we may skip ahead **len - table[len - 1]** characters and continue matching after **table[len - 1]** characters.

Example 1: Consider the table for the pattern "ABCDAD"

J	0	1	2	3	4	5	6
	0	0	0	0	1	2	0

Let the string be, "ABC ABCDAB ABCDABCDABDE"

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
A	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E
A	B	C	**D**	A	B	D																

We get a partial match of length = 3, and table[3-1] = 0, so skip 3 characters

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
A	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E
			A	B	C	D	A	B	D													

The first character in the pattern mismatches, no partial match found, just shift pattern by 1

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
A	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E
				A	B	C	D	A	B	**D**												

We get a partial match of length = 6, and table[6-1] = 2, so we skip by 6-2 = 4 characters and continue matching after 2 characters

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
A	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E
								A	B	C	D	A	B	D								

We get a partial match of length = 2, and table[2-1] = 0, so skip 2 characters

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
A	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E
										A	B	C	D	A	B	D						

The first character in the pattern mismatches, no partial match found, just shift pattern by 1

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
A	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E
											A	B	C	D	A	B	**D**					

We get a partial match of length = 6, and table[6-1] = 2, so we skip by 6-2 = 4 characters and continue matching after 2 characters.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
A	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E
															A	**B**	**C**	**D**	**A**	**B**	**D**	

Pattern successfully found.

Dr.Mahesh G Mr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

Example 2: Consider the table for the pattern " ABABABCA "

J	0	1	2	3	4	5	6	7
	0	0	1	2	3	4	0	1

Let the string be, "BACBABABAABCBAB"

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
B	A	C	B	A	B	A	B	A	A	B	C	B	A	B
A	B	A	B	A	B	C	A							

The first character in the pattern mismatches, no partial match found, just shift pattern by 1

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
B	A	C	B	A	B	A	B	A	A	B	C	B	A	B
	A	**B**	A	B	A	B	C	A						

We get a partial match of length = 1, and table[1-1] = 0, so skip 1 character

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
B	A	C	B	A	B	A	B	A	A	B	C	B	A	B
		A	B	A	B	A	B	C	A					

The first character in the pattern mismatches, no partial match found, just shift pattern by 1

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
B	A	C	B	A	B	A	B	A	A	B	C	B	A	B
			A	B	A	B	A	B	C	A				

Dr.Mahesh G Assoc. Prof. BMSIT & M
Mr.Harish G Assoc. Prof. Dr. AIT

The first character in the pattern mismatches, no partial match found, just shift pattern by 1

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
B	A	C	B	A	B	A	B	A	A	B	C	B	A	B
				A	B	A	B	A	**B**	C	A			

We get a partial match of length = 5, and table[5-1] = 3, so we skip by 5-3 = 2 characters and continue matching after 3 characters

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
B	A	C	B	A	B	A	B	A	A	B	C	B	A	B
						A	B	A	B	A	B	C	A	

We get a partial match of length = 3, and table[3-1] = 1, so we skip by 3-1 = 2 characters and continue matching after 1 character

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	
B	A	C	B	A	B	A	B	A	A	B	C	B	A	B	
								A	B	A	B	A	B	C	A

As the pattern is longer than the remaining characters in the text, we stop and announce "pattern not found"

Example 3: Consider the table for the pattern " AABAACAABAA "

J	0	1	2	3	4	5	6	7	8	9	10
	0	1	0	1	2	0	1	2	3	4	5

Let the string be, " AAAABAACAACAABAACAABAAB"

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	2	22
A	A	A	A	B	A	A	C	A	A	C	A	A	B	A	A	C	A	A	B	A	A	B
A	A	**B**	A	A	C	A	A	B	A	A												

We get a partial match of length = 2, and table[2-1] = 1, so we skip by 2-1 = 1 character and continue matching after 1 character

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
A	A	A	A	B	A	A	C	A	A	C	A	A	B	A	A	C	A	A	B	A	A	B
	A	A	**B**	A	A	C	A	A	B	A	A											

We get a partial match of length = 2, and table[2-1] = 1, so we skip by 2-1 = 1 character and continue matching after 1 character

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
A	A	A	A	B	A	A	C	A	A	C	A	A	B	A	A	C	A	A	B	A	A	B
		A	A	B	A	A	C	A	A	**B**	A	A										

We get a partial match of length = 8, and table[8-1] = 2, so we skip by 8-2 = 6 characters and continue matching after 2 character

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
A	A	A	A	B	A	A	C	A	A	C	A	A	B	A	A	C	A	A	B	A	A	B
								A	A	B	A	A	C	A	A	B	A	A				

We get a partial match of length = 2, and table[2-1] = 1, so we skip by 2-1 = 1 character and continue matching after 1 character

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
A	A	A	A	B	A	A	C	A	A	C	A	A	B	A	A	C	A	A	B	A	A	B
									A	A	B	A	A	C	A	A	B	A	A			

We get a partial match of length = 1, and table[1-1] = 0, so we skip 1 character and continue

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
A	A	A	A	B	A	A	C	A	A	C	A	A	B	A	A	C	A	A	B	A	A	B
										A	A	B	A	A	C	A	A	B	A	A		

The first character in the pattern mismatches, no partial match found, just shift pattern by 1

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
A	A	A	A	B	A	A	C	A	A	C	A	A	B	A	A	C	A	A	B	A	A	B
											A	**A**	**B**	**A**	**A**	**C**	**A**	**A**	**B**	**A**	**A**	

Pattern successfully found.

Note: Rules to compute failure function

- ✓ Initialize f[0] = 0, i=0, and j=1
- ✓ If p[i] is same as p[j] then assign i+1 to f[j], increment i and j
- ✓ If p[i] not equal to p[j] and if i is 0, then assign i to f[j] and increment j
- ✓ If p[i] not equal to p[j] and if i is not 0, then assign f[i-1] to i

Program to search for a pattern in a given string using KMP Method

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

//Failure function for pattern string
void failure(int f[ ], char p[])
{
        int i=0, j=1;

        f[0]=0;
        while(j < strlen(p))
        {
                if(p[i]==p[j])
                {
                        f[j]=i+1;
                        i++;
                        j++;
                }
                elseif (i==0)
                {
                        f[j]=i;
                        j++;
                }
                else
                        i=f[i-1];
        }
}
```

```
// Knuth, Moris and Pratt Function for pattern matching
int KMP(char p[], char t[], int f[])
{
        int i=0, j=0;

        while(i<strlen(t) && j<strlen(p))
        {
                if(t[i]==p[j])
                {
                        i++;
                        j++;
                }
                elseif(j==0)
                        j++;
```

```
            else
                  j=f[j-1];
      }

      if(j==strlen(p)
            return i - strlen(p);
      else
            return -1;
}
void main( )
{
      int i, pos;
      char t[100], p[100];
      int f[50];

      printf("Enter the main string\n");
      gets(t);

      printf("Enter the pattern string\n");
      gets(p);

      failure(f,p);

      pos = KMP(p, t, f);

      if(pos = = -1)
            printf("Pattern string not found\n");
      else
            printf("Pattern string found at position %d\n", pos);
}
```

8. Structures

8.1 Why structures?

If we want to store a number of data items of same datatype, we can use arrays. However, if we want to store the information which is a collection of items of dissimilar datatype, we cannot use arrays. This is where we use structures.

8.2 Structures Definition and Syntax

A structure is a collection of related information of possibly different datatype under one name.

Syntax:

```
struct tagname
{
        datatype member1;
        datatype member2;
        ..
        ..
};
```

Dr.Mahesh G Mr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

Where

- struct is the keyword.
- tagname is user defined name for structure.
- datatype can be any of the basic datatype (int, float, char etc) or user defined datatype.
- member1, member2 are attributes of the structure (also called members or fields).

Example:

```
struct student
{
        char name[20];
        int marks1;
        int marks2;
        int marks3;
        char regno[10];
};
```

Just by defining a structure, memory will not be allocated for the various fields. It simply describes a format called template to represent information as shown

name	Array of 20 characters
marks1	Integer
marks2	Integer
marks3	Integer
regno	Array of 10 characters

If the structure is associated with a variable then the memory is allocated.

8.3 The Type Definition

The "typedef" is a keyword that allows the programmer to create a new datatype name for an existing datatype.

The general syntax for typedef is

new name

typedef datatype userprovidedname;

Where

- typedef is the keyword.
- datatype is any existing datatype.
- userprovidedname is the name for the existing datatype. Usually this is written in capital letters.

Example: Suppose if,

```
int m1,m2,m3;
```

represents the declaration of 3 variables m1, m2 and m3 to store marks scored in 3 subjects. The same thing can be written using typedef as shown

```
typedef int MARKS;
MARKS m1, m2, m3;
```

Dr.Mahesh G Assoc. Prof. BMSIT & M **Mr.Harish G** Assoc. Prof. Dr. AIT

Advantages of typedef

- It helps in self-documenting a program by allowing the use of more descriptive names for the standard datatypes.
- Complex and lengthy declarations can be reduced to short and meaningful declarations.
- Improves readability of the program.

8.4 Various structure variable declaration forms

Format 1:

```
struct student
{
        char name[20];
        int marks1;
        int marks2;
        int marks3;
        char regno[10];
};
struct student a, b, c;
```

Format 2:

```
struct student
{
        char name[20];
        int marks1;
        int marks2;
        int marks3;
        char regno[10];
}a, b, c;
```

amount memory allocated is sum of all datatypes in bytes.

Format 3: Using typedef

```
typedef struct
{
        char name[20];
        int marks1;
        int marks2;
        int marks3;
        char regno[20];
}STD;
```

Dr.Mahesh G	Mr.Harish G
Assoc. Prof.	Assoc. Prof.
BMSIT & M	Dr. AIT

OR

```
struct student
{
        char name[10];
        int marks1;
        int marks2;
        int marks3;
        char regno[10];
};
typedef struct student STD;
```

Here STD is a new datatype of type struct student, Therefore the statement

STD a, b;

creates 2 variables a, b of type STD.

Note:

1. The structure type declaration defines a template of the structure and does not tell the compiler to reserve any space in memory. Memory is allocated only when the structure is associated with a variable.

Example:

```
struct student
{
        char name[20];
        int marks1;
        int marks2;
        int marks3;
        char regno[10];
};
struct student a, b, c;
```

Here each of the variable (a. b, c) will have 5 members (name, marks1, marks2, marks3, regno). The statement **struct student a, b, c;** sets aside space in memory. It makes available space to hold all the members in the structure for all the structure variables declared.

2. Usually structure type declaration appears at the top of the source code i.e. before any variables or functions are defined. In such cases, the definition is global and can be used by other functions.

8.5 Accessing structure fields or members

The fields of the structure can be accessed by using a dot operator '.'(member selection operator).

Example: Consider the declaration

```
struct student
{
        char name[20];
        int marks1;
        int marks2;
        int marks3;
        char regno[10];
};
typedef struct student STD;
STD a;
```

Dr.Mahesh G Mr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

Here 'a' is a variable of type STD and the elements of struct can be accessed as a.name, a.marks1, a.marks2, a.marks3 and a.regno.

Note – Before the dot operator there must always be a structure variable and after the dot operator there must always be a structure member.

8.6 Assigning Values to Structure Members

We can assign values like

```
strcpy(a.name, "Mahesh");
a.marks1=97;
a.marks2=98;
a.marks3=99;
strcpy(a.regno, "1AY10IS001");
```

in the program OR

We can use scanf to read values through the keyboard

```
scanf("%s", a.name);
scanf("%d%d%d", &a.marks1, &a.marks2, &a.marks3);
scanf("%s", a.regno);
```

8.7 Structure Initialization

Like primary variables and arrays, structure variables can also be initialized where they are declared. The format used is quite similar to that used to initialize arrays.

Array example:

int num[6] = {2, 4, 12, 5, 45, 5};

int num[ ] = {2, 4, 12, 5, 45, 5};

If the array is initialized where it is declared mentioning the dimension of the array is optional.

Structure Example:

```
        struct book
        {
                char name[10];   } →no memory allocated
                float price;
                int pages;
        };
        struct book b1 = {"c++", 150.00, 550};
        struct book b2 = {"physics", 150.80, 800};
```

OR

```
        struct book
        {
                char name[10];
                float price;
                int pages;
        } b1 = {"c++", 150.00, 550}, b2 = {"physics", 150.80, 800};
```

Dr.Mahesh G Mr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

Note – C language does not permit the initialization of individual structure members within the template. The initialization must be done only in the declaration of the actual variables.

Example:

```
        struct person
        {
                int weight =60;
                float height = 170.60;
        }p1;
```

This is not permitted.

Program to demonstrate structures

```
struct student
{
        char name[20];
        int marks1;
        int marks2;
        int marks3;
        char regno[10];
};
```

```
void main( )
{
        struct student a, b;
        clrscr( );
        printf("enter the name, m1, m2, m3 and regno of student1\n");
        scanf("%s%d%d%d%s",a.name, &a.marks1, &a.marks2, &a.marks3, a.regno);
        printf("enter the name, m1, m2, m3 and regno of student2\n");
        scanf("%s%d%d%d%s",b.name, &b.marks1, &b.marks2, &b.marks3, b.regno);
        printf("the information of student 1 is\n");
        printf("%s%d%d%d%s",a.name, a.marks1, a.marks2, a.marks3, a.regno);
        printf("the information of student 2 is\n");
        printf("%s%d%d%d%s",b.name, b.marks1, b.marks2, b.marks3, b.regno);
        getch( );
}
```

Dr.Mahesh G Mr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

Comments:

1. The declaration at the beginning of the program combines different datatypes into a single entity called struct student. Here struct is a key word, student is the structure name and the different datatype elements are structure members.
2. a, b are structure variables of the type struct student.
3. The structure members are accessed using dot operator. So to refer name we use a.name, b.name and to refer marks1 we use a.marks1 and b.marks1. Before the dot we have a structure variable and after the dot we have a structure member.
4. Since a.name is a string, its base address can be obtained by just mentioning a.name. Hence the address operator (&) has been dropped while receiving the name in scanf.

8.8 Array of structures

Suppose if we want to store data about a book, then the following structure may be used.

```
struct book
{
        char name[10];
        float price;
        int pages;
};

struct book b1={"c++", 150.00, 570};
```

Now, if we want to store data of 100 books we would be required to use 100 different structure variables from b1 to b100 which is definitely not very convenient.

A better approach would be to use an array of structures.

Program to d monstrate array of structures

```
struct book
{
        char na ne[10];
        float pr ce;
        int pag s;
};

main( )
{
        struct b ook b[100];
        int i;
        clrscr(  ;
        for(i=0. i<100; i++)
        {
                printf("enter the name, price and  umber of pages of %d book\n",i);
                scanf("%s %f %d ",b[i].name, &  i].price, &b[i].pages);
        }
        printf("books information\n");
        printf("name   price    number of pages\  );
        for(i=0  i<100; i++)
        {
                printf("%s %f %d \n", b[i].name  [i].price, b[i].pages);
        }
        getch( );
}
```

Dr.M esh G Mr.Harish G
Ass Prof. Assoc. Prof.
BM & M Dr. AIT

Here the structure fields are still accessed using he dot operator and the array elements using the usual subscript notation.

8.9 Assignment or Copying of Structur Variables

The values of a structure variable can be assig ed to another structure variable of the same type using the assignment operator (=). It is n necessary to copy the structure elements or members piece-meal. Obviously programmers p efer assignment to piece-meal copying.

Program to demonstrate copying of structur variables

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

void main( )
{
        struct employee e1 = {"Harish", 30, 250 0.00};
        struct employee e2, e3;
```

```
    //piece-meal copying
    strcpy(e2.name, e1.name);
    e2.age = e1.age;
    e2.sal = e1.sal;

    //copying all elements at one go
    e3 = e2;

    printf("%s %d %f \n", e1.name, e1.age, e1.salary);
    printf("%s %d %f \n", e2.name, e2.age, e2.salary);
    printf("%s %d %f \n", e3.name, e3.age, e3.salary);
    getch( );
}
```

Output:

```
Harish  30 25000.00
Harish  30 25000.00
Harish  30 25000.00
```

Note:

1. Copying of all structure elements at one go is possible because the structure elements are stored in contiguous memory locations.
2. Assignment or copying of different structure variables is invalid.

Example:

```
struct
{
        char name[10];
        int age;
        float salary;
}a, b;

struct
{
        char name[10];
        int age;
        float salary;
}c, d;
```

Valid Statements (copying variables of same structure)	Invalid Statements (copying variables of different structure)	
a = b	a = c	c = a
b = a	a = d	c = b
c = d	b = c	d = a
d = c	b = d	d = b

8.10 Comparison of Structure Variables

Comparison of 2 structure variables of same type or dissimilar type is not allowed.

Example:

```
a = = b   //Invalid: Comparison is not allowed between structure variables.
a ! = b   //Invalid: Comparison is not allowed between structure variables.
```

However, the members of 2 structure variables of same type can be compared using relational operators.

Example:

```
a.member1 = = b.member2   //Return true if they are same, false otherwise
a.member1 ! = b.member2   //Return true if they are not same, false otherwise
```

Note – The arithmetic, relational, logical and other various operations can be performed on individual members of structures but not on structure variables.

8.11 Pointer to Structures

A variable which contains the address of a structure variable is called pointer to a structure. The members of a structure can be accessed very efficiently using pointers.

Example: Consider the following structure

```
struct employee
{
        char name[10];
        int age;
        float salary;
};
void main( )
{
        struct employee e1 = {"Harish", 30, 25000.00};
        struct employee *ptr;
        ptr = &e1;
//using de-referencing operator * and dot (.) operator
        printf("Name is %s \n", (*ptr).name)
        printf("Age is %d \n", (*ptr).age);
        printf("Salary is %f \n", (*ptr).salary
//using selection operator ->          (or)
        printf("Name is %s \n", ptr->name);
        printf("Age is %d \n", ptr ->age);
        printf("Salary is %f \n", ptr ->salary
}
```

Dr.Mahesh G Assoc. Prof. BMSIT & M | **Mr.Harish G** Assoc. Prof. Dr. AIT

Output:

```
Name is Harish
Age is 30
Salary is 25000.00
Name is Harish
Age is 30
Salary is 25000.00
```

8.12 Difference between structures and arrays

	Arrays	Structures
1	An array is a collection of related data elements of same datatype.	Structure is a collection of variables of similar or dissimilar datatype.
2	An array item can be accessed by using its subscript or by using dereferencing / indirection (*) operator for pointers.	Structure items can be accessed using '.'or by using –> for pointers.

8.13 Self Referential Structures

A structure which contains a reference (pointer) to itself is called a self referential structure. It is used to create data structures like linked lists, trees, etc.

General Format

```
struct tagname
{
        datatype member1;
        datatype member2;
                ..
                ..
        struct tagname * pointer_name;
};
```

Dr.Mahesh G Assoc. Prof. BMSIT & M
Mr.Harish G Assoc. Prof. Dr. AIT

Example

```
struct node
{
        int info;
        struct node* link;
};
```

9. Unions

9.1 Unions Definition and Synt x

Unions

A union is a collection of relate information of possibly di erent datatype under one name. It is a concept derived from stru tures. The main features t at distinguish it form structures are

* 1. The amount of memory alloc ted is equal to the size of t e member that occupies largest space.

* 2. The memory allocated is share d by individual members of union.

e[10] -

Syntax:

```
union tagname
{
        datatype member1;
        datatype member2;
        ..
        ..
};
```

Where

- union is the keyword.
- tagname is user defined name for union
- datatype can be any of the basic datatype (int, float, char etc) or user defined datatype.
- member1, member2 are attributes of the union (also call d members or fields).

Example:

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

1. The definition of a union does not allocate any me nory since the definition is not associated with any variable.
2. The different forms of variable declarations for struct ıres, holds good for unions.
3. The union members are accessed exactly the sa e way in which the structure elements are accessed i.e. using a '.' operator.
4. The major difference between structures and unions is the way in which memory is allocated.

Example: Consider the flowing declaration

```
struct a
{
        int i;
        char ch[2];
};
struct a key;
```

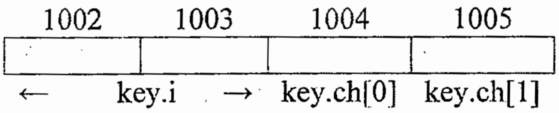
This datatype would occupy 4 bytes in memory

key.i requires 2 bytes.

key.ch[0] requires 1 byte.

key.ch[1] requires 1 byte.

This is as shown in the figure



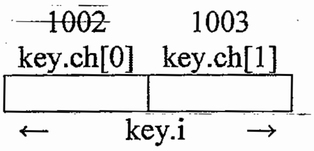
Here the amount of memory allocated is the sum of the storage specified for each of its members (for the above example 4 bytes). Each member in the structure is assigned its own unique storage area (for the above example int i – 1002 to 1003, char ch[2] – 1004 to 1005).

```
union a
{
        int i;
        char ch[2];
};
union a key;
```

This datatype would occupy 2 bytes in memory

key.i , key.ch[0] , key.ch[1] – all share this 2 bytes.

This is as shown in the figure



Here the amount of memory allocated is the size of the member that occupies largest space. (for the above example int i – 2bytes, char ch[2] – 2bytes, therefore larger = 2 bytes). The memory allocated is shared by individual members of union. (for the above example int i – 1002 to 1003, char ch[2] – 1002 to 1003).

Note:

1. The same memory locations which are used for key.i are also used by key.ch[0] and key.ch[1]. It means that the memory locations used by key.i can also be accessed using key.ch[0] and key.ch[1].

Example: Consider the program shown below

```
union a
{
        int i;
        char ch[2];
}
main()
{
        union a key;
        key.i = 512;
        printf("key.i = %d\n", key.i);
        printf("key.ch[0] = %d", key.ch[0]);
        printf("key.ch[1] = %d", key.ch[1]);
}
```

512 in binary is 0000 0010 0000 0000. This is stored in memory as shown.

```
        1002            1003
      key.ch[0]       key.ch[1]
    | 0000 0000     | 0000 0010  |
    ←             key.i          →
```

The output of the program will be

```
512
0
2
```

We cannot assign different values to the different elements at the same time i.e if we assign a value to key.i, it gets automatically assigned to key.ch[0] & key.ch[1] and if we assign a value to key.ch[0] and/or key.ch[1], it is bound to get assigned to key.i.

2. Consider the following statements

```
union item
{
        int a;
        float b;
}t;
t.a = 100;
t.b = 20.5;
```

printf("%d", t.a); will produce erroneous output because a union creates a storage location that can be used by anyone of its members at a time.

3. Only the first member of a union can be initialized.

Example: union item t = {100}; // valid
union item t = {20.5}; //invalid

9.2 Difference between structures and unions

slack bytes

	Structure	Union
1	The keyword 'struct' is used to start the declaration of a structure.	The keyword 'union' is used to start the declaration of a union.
2	The size of memory allocated by the compiler is equal to or greater than the sum of sizes of its members.	The size of memory allocated by the compiler is equal to the size of the member that occupies largest space.
3	Memory allocated is unique (separate) for all members declared within the structure.	Memory allocated is shared (common) for all the members declared within the union.
4	Altering the value of a member will not affect other members of the structure.	Altering the value of any of the member will alter other member values.
5	Several members of a structure can be initialized at once.	Only the first member of union can be initialized.
6	Individual members can be accessed at a time.	Only one member can be accessed at a time.

Q) Write C program with appropricate structure definition & variable declaration to store information about an employee using Nested structures. Consider following fields like Ename, Empid, DOJ (date, month, year) & Salary (Basic, DA, HRA).

Sol:
```
#include< stdio.h>
struct employee
{
char Ename [100];
 int Empid;
 struct doj DOJ;
struct Salary SAL;
};
main ()
{
struct employee e[100];
Printf (" Enter name ");
scanf ("%d",&n);
for ( i=0; i<n; i++)
 {
 printf(" Enter name\n");
 scanf(" %s",e[i].ename);
 printf ("entu empid");
 scanf("%d", & e[i]. empid);
 printf(" Entur date %n");
 scanf (" %d", & e[i]. DoJ. date);
 printf(" Enter month\n");
 scanf(" %d", & e[i]. DoJ. month);
```

```
struct doj
{
 int date;
 int month;
 int year;
};
struct salary;
{
 int basic;
 int DA;
 int HRA;
};
```

Dude what is she thinking

She is seeing ~~floor~~ again

and again.

Develop struct to represent planets in

solar ~~sys~~. Each planet has field for

planets name, distance from sun in miles

no of moons it has. Write program to

read data for each planet, also print

name of planet that has highest no of

moons.