MODULE 1

INTRODUCTION TO DATA STRUCTURES: Data Structures, Classifications (Primitive & Non-Primitive), Data structure Operations Review of pointers and dynamic Memory Allocation

ARRAYS and STRUCTURES: Arrays, Dynamic Allocated Arrays, Structures and Unions, Polynomials, Sparse Matrices, representation of Multidimensional Arrays, Strings

STACKS: Stacks, Stacks Using Dynamic Arrays, Evaluation and conversion of Expressions

DATA STRUCTURE

Data structure is a representation of the logical relationships existing between individual elements of data. A data structure is a way of organizing all data items that considers not only the elements stored but also their relationship to each other.

The logical or mathematical model of a particular organization of data is called a **data structure.**

The choice of a particular data model depends on the two considerations:

1. It must be rich enough in structure to mirror the actual relationships of the data in the real world.
2. The structure should be simple enough that one can effectively process the data whenever necessary.

BASIC TERMINOLOGY

Elementary Data Organization

Data: Data are simply values or sets of values.

Data items: Data items refers to a single unit of values. Data items that are divided into sub-items are called Group items. Ex: An Employee Name may be divided into three subitems- first name, middle name, and last name. Data items that are not able to divide into sub-items are called Elementary items. Ex: SSN

Entity: An entity is something that has certain attributes or properties which may be assigned values. The values may be either numeric or non-numeric. Ex: Attributes- Names, Age, Sex, SSN Values- Rohland Gail, 34, F, 134-34-5533 Entities with similar attributes form an entity set. Each attribute of an entity set has a range of values, the set of all possible values that could be assigned to the particular attribute. The term “information” is sometimes used for data with given attributes, of, in other words meaningful or processed data.

Field: is a single elementary unit of information representing an attribute of an entity. Record is the collection of field values of a given entity.

File: is the collection of records of the entities in a given entity set.

Each record in a file may contain many field items but the value in a certain field may uniquely determine the record in the file. Such a field K is called a primary key and the values k1, k2, ….. in such a field are called keys or key values.

Records may also be classified according to length.

A file can have fixed-length records or variable-length records.

• In fixed-length records, all the records contain the same data items with the same amount of space assigned to each data item.

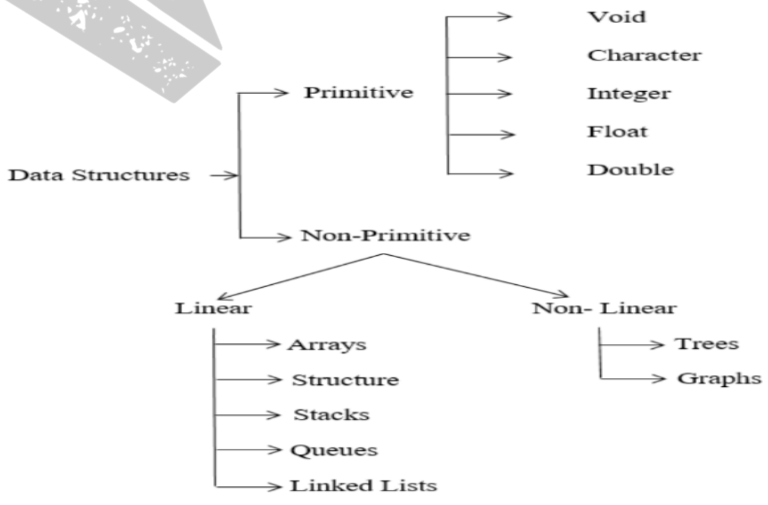
• In variable-length records file records may contain different lengths. Example: Student records have variable lengths, since different students take different numbers of courses. Variable-length records have a minimum and a maximum length. The above organization of data into fields, records and files may not be complex enough to maintain and efficiently process certain collections of data. For this reason, data are also organized into more complex types of structures.

CLASSIFICATION OF DATA STRUCTURES

Data Structures can be divided into two categories,

i) Primitive Data Structures

ii) Non-Primitive Data Structures



Primitive Data Structures

These are basic data structures and are directly operated upon by the machine instructions. These data types consists of characters that cannot be divided and hence they also called simple data types.

Example: Integers, Floating Point Numbers, Characters and Pointers etc.

Non-Primitive Data Structures

These are derived from the primitive data structures. The non-primitive data structures emphasizeon structuring of a group of homogeneous or heterogeneous data items.

Example: Arrays, Lists and Files, Graphs, trees etc.

Based on the structure and arrangement of data, non-primitive data structures is furtherclassified into

1. Linear Data Structure
2. Non-linear Data Structure

1. Linear Data Structure:

A data structure is said to be linear if its elements form a sequence or a linear list. There are basically two ways of representing such linear structure in memory.

1. One way is to have the linear relationships between the elements represented by means of sequential memory location. These linear structures are called arrays.

2. The other way is to have the linear relationship between the elements represented by means of pointers or links. These linear structures are called linked lists.

The common examples of linear data structure are Arrays, Queues, Stacks, Linked lists

2. Non-linear Data Structure:

A data structure is said to be non-linear if the data are not arranged in sequence or a linear. The insertion and deletion of data is not possible in linear fashion. This structure is mainly used to represent data containing a hierarchical relationship between elements. Trees and graphs are the examples of non-linear data structure.

OPERATIONS ON DATA STRUCTURES

The commonly used operations on data structures are as follows,

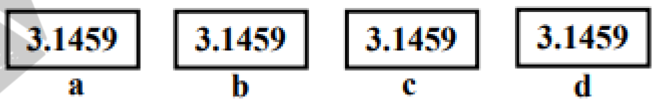
1. **Create:** The Create operation results in reserving memory for the program elements. The creation of data structures may take place either during compile time or during run time.

2. **Destroy:** The Destroy operation destroys the memory space allocated for the specified data structure.
3. **Selection:** The Selection operation deals with accessing a particular data within a data structure.
4. **Updating**: The Update operation updates or modifies the data in the data structure.
5. **Searching:** The Searching operation finds the presence of the desired data item in the list of data items.
6. **Sorting:** Sorting is the process of arranging all the data items in the data structure in a particular order, say for example, either in ascending order or in descending order.
7. **Merging:** Merging is a process of combing the data items of two different sorted list into a single list.

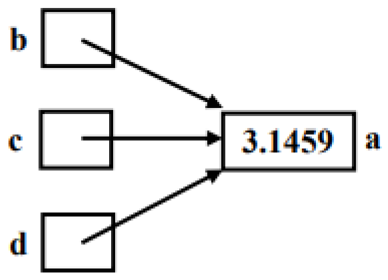
REVIEW OF POINTERS AND DYNAMIC MEMORY ALLOCATION

Pointers to data significantly improve performance for repetitive operations such as traversing strings, lookup tables, control tables and tree structures. In particular, it is often much cheaper in time and space to copy and dereference pointers than it is to copy and access the data to which the pointers point. Pointers are also used to hold the addresses of entry points for called subroutines in procedural programming and for run-time linking to dynamic link libraries (DLLs).

Pointer: A pointer is a special variable which contains address of a memory location. Using this pointer, the data can be accessed. For example, assume that a program contains four occurrences of a constant 3.1459. During the compilation process, four copies of 3.1459 can be created as shown below:



However, it is more efficient to use one copy of 3.1459 and three pointers referencing a single copy, since less space is required for a pointer when compared to floating point number. This can be represented pictorially as shown below:



General form of pointer declaration is –

type* name;

where **type** represent the type to which pointer thinks it is pointing to.

Pointers to machine defined as well as user-defined types can be made Pointer Intialization:

```
     variable_type *pointer_name = 0;
     or variable_type *pointer_name = NULL;
     char *pointer_name = "string value here";
```

DYNAMIC MEMORY ALLOCATION

This is process of allocating memory-space during execution-time (or run-time).

• This is used if there is an unpredictable storage requirement.

• Memory-allocation is done on a heap.

• Memory management functions include:

→ malloc (memory allocate)

→ calloc (contiguous memory allocate)

→ realloc (resize memory)

→ free (deallocate memory)

• **malloc function** is used to allocate required amount of memory-space during run-time.

• If memory allocation succeeds, then address of first byte of allocated space is returned. If memory allocation fails, then NULL is returned.

• **free()** function is used to deallocate(or free) an area of memory previously allocated by malloc() or calloc().

```
#include void main()
     {
                 int i,*pi;
                 pi=(int*)malloc(sizeof(int));
                *pi=1024;
                 printf("an integer =%d",pi);
                 free(pi);
     }
```

Prg: Allocation and deallocation of memory

• If we frequently allocate the memory space, then it is better to define a macro as shown below:

```
      #define MALLOC(p,s)
       if(!((p)==malloc(s)))
        }         printf("insufficient memory");
                  exit(0);
        }
```

• Now memory can be initialized using following:

```
MALLOC(pi,sizeof(int));
MALLOC(pf,sizeof(float))
```

DANGLING REFERENCE

• Whenever all pointers to a dynamically allocated area of storage are lost, the storage is lost to the program. This is called a dangling reference.

POINTERS CAN BE DANGEROUS

1) Set all pointers to NULL when they are not actually pointing to an object. This makes sure that you will not attempt to access an area of memory that is either

   → out of range of your program or

   → that does not contain a pointer reference to a legitimate object

2) Use explicit type casts when converting between pointer types.

   pi=malloc(sizeof(int)); //assign to pi a pointer to int

   pf=(float*)pi; //casts an 'int' pointer to a 'float' pointer

3) Pointers have same size as data type 'int'. Since int is the default type specifier, some programmers omit return type when defining a function. The return type defaults to 'int' which can later be interpreted as a pointer. Therefore, programmer has to define explicit return types for functions.

```
void swap(int *p,int *q)   //both parameters are pointers to ints
{
     int temp=*p;       //declares temp as an int and assigns to it the contents
                        of what p points to
     *p=*q;             //stores what q points to into the location where p
                        points
     *q=temp;           //places the contents temp in location pointed to by q
}
```

Prg: Swap Function

ALGORITHM SPECIFICATION

An algorithm is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms must satisfy the following criteria:

1. **Input:** There are zero or more quantities that are externally supplied.
2. **Output:** At least one quantity is produced.
3. **Definiteness:** Each instruction is clear and unambiguous
4. **Finiteness:** If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
5. **Effectiveness:** Every instruction must be basic enough to be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in (3); it also must be feasible.

Algorithm can be described in following ways:

1) We can use natural language consisting of some mathematical equations.
2) We can use graphic representations such as flowcharts.
3) We can use combination of C and English language constructs.

• Algorithm 1.1: Selection sort algorithm.

```
for(i=0;i<n;i++)
{
     Examine list[i] to list[n-1] and suppose that the
     smallest integer is at list[min]; Interchange list[i]
     and list[min];
}
```

Algorithm 1.2: finding the smallest integer.

```
assume that minimum is list[i]
compare current minimum with list[i+1] to list[n-1] and find
smaller number and make it the new minimum
```

• Algorithm 1.3: Binary search.

Assume that we have n > 1 distinct integers that are already sorted and stored in the array list.

That is, list[0]<= list[1]…list[n]

We must figure out if an integer searchnum is in this list.

If it is we should return an index, i, such that list[i] = searchnum.

Ifsearchnum is not present, we should return -1.

Since the list is sorted we may use the following method to search for the value. Let left and right, respectively, denote the left and right ends of the list to be searched. Initially, left = 0 and right = n-l. Let middle = (left+right)/2 be the middle position in the list. If we compare list [middle] with searchnum, we obtain one of three results:

```
assumption :sorted n(≥1) distinct integers stored in the array list
return i if  list[i] = searchnum;
-1 if no such index exists
denote left and right as left and right ends of the list to
be searched (left=0 & right=n-1) let middle=(left+right)/2
middle position in the list
compare list[middle] with searchnum and adjust
left or right compare list[middle] with
searchnum
   1) searchnum <
      list[middle] set
      right to middle-1
   2) searchnum =
      list[middle] return
      middle
   3) searchnum >
      list[middle] set
      left to middle+1
if searchnum has not been found and there are more integers to
check recalculate middle and continue search


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

• Algorithm 1.4: Permutations

```
given a set of n(≥1)
elements print out all
possible permutations of
this set
e.g. if set {a,b,c} is given,
      then set of permutations is {(a,b,c), (a,c,b), (b,a,c),
      (b,c,a), (c,a,b), (c,b,a)}
```

RECURSIVE ALGORITHMS

• A function calls itself either directly or indirectly during execution.

• Recursive-algorithms when compared to iterative-algorithms are normally compact and easy to understand.

• Various types of recursion:

1) **Direct recursion:** where a recursive-function invokes itself.
2) **Indirect recursion:** A function which contains a call to another function which in turn calls another function and so on and eventually calls the first function.

```
void f1()              void f2()              void f3()
{                      {                      {
....                   ....                   ....
f2();                  f3();                  f1();
}                      }                      }
```

For example:

1. FACTORIAL

“The product of the positive integers from 1 to n, is called "n factorial" and is denoted by n!” $n! = 1*2*3 \ldots (n-2)*(n-1)*n$

It is also convenient to define $0! = 1$, so that the function is defined for all nonnegative integers.

Definition: (Factorial Function)

a) If $n = 0$, then $n! = 1$.
b) If $n > 0$, then $n! = n*(n-1)!$

Observe that this definition of n! is recursive, since it refers to itself when it uses $(n-1)!$

(a) The value of n! is explicitly given when $n = 0$ (thus 0 is the base value)

(b) The value of n! for arbitrary *n* is defined in terms of a smaller value of *n* which is closer to the base value 0.

The following are two procedures that each calculate n factorial .

Using for loop: This procedure evaluates N! using an iterative loop process

Procedure: FACTORIAL (FACT, N)

This procedure calculates N! and returns the value in the variable FACT.

If N = 0, then: Set FACT: = 1, and Return.

Set FACT: = 1. [Initializes FACT for loop.]

Repeat for K = 1 to N.

Set FACT: = K*FACT.

[End of loop.]

Return.

Using recursive function: This is a recursive procedure, since it contains a call to itself

Procedure: FACTORIAL (FACT, N)

This procedure calculates N! and returns the value in the variable FACT.

If N = 0, then: Set FACT: = 1, and Return.

Call FACTORIAL (FACT, N - 1).

Set FACT: = N*FACT.

Return.

```
 int fact(int n)//to find factorial of a number
 {
      if(n==0)
          return 1;

return n*fact(n-1);
}
```

2. <u>BINARY SEARCH:</u>

To transform function into a recursive one, we must

(1) establish boundary conditions that terminate the recursive calls, and

(2) implement the recursive calls so that each call brings us one step closer to a solution

```
 int binsearch(int list[], int searchnum, int left, int right)
 {     // search list[0]<= list[1]<=...<=list[n-1] for searchnum
       int middle;
       if (left<= right)
       {
             middle= (left+ right)/2;
             switch(compare(list[middle], searchnum))
             {
              case -1:return binsearch(list, searchnum, middle+1, right);
              case 0: return middle;
              case 1: return binsearch(list, searchnum, left, middle- 1);
             }
        }
       return -1;
 }

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

Recursive Implementation of Binary Search

3. PERMUTATIONS:

Given a set of n > 1 elements, print out all possible permutations of this set. For example, if the set is (a, b. c), then the set of permutations is {(a, b, c), (a, c, b), (b, a, c), (b, c, d), (c, a, b),(c, b, a)}.

It is easy to see that, given n elements, there are n! permutations. We can obtain a simple algorithm for generating the permutations if we look at the set (a, b, c, d). We can construct the set of permutations by printing:

1. a followed by all permutations of (b, c, d)
2. b followed by all permutations of (a, c, d)
3. c followed by all permutations of (a, b, d)
4. d followed by all permutations of (a, b, c)

The clue to the recursive solution is the phrase "followed by all permutations." It implies that we can solve the problem for a set with n elements if we have an algorithm that works on n - 1 elements. We assume that list is a character array. Notice that it recursively generates permutations until i = n. The initial function call is perm(list. 0, n-1);

```
 void perm(char *list,int i,int n)
 {
       int j,temp; if(i==n)
       {
             for(j=0;j<=n;j++) printf("%c", list[j]); printf("
             ");
       }
       else
       {
             for(j=i;j<=n;j++)
             {
               SWAP(list[i],list[j],temp); perm(list,i+1,n);
               SWAP(list[i],list[j],temp);
             }
       }
}
```

Recursive permutations generator

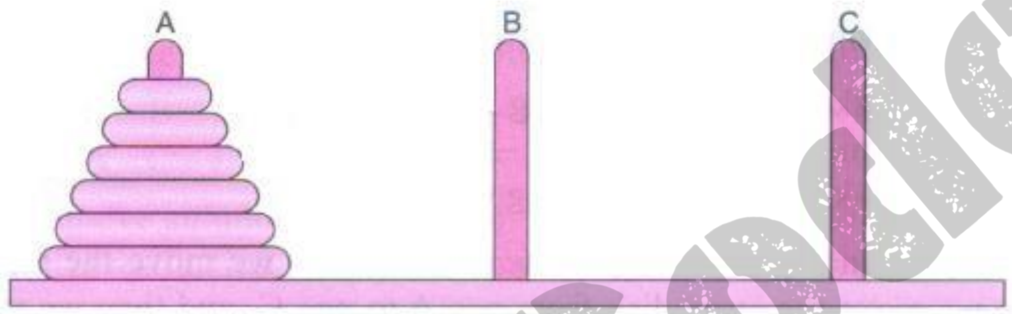
4.TOWER OF HANOI

Problem description

Suppose three pegs, labeled A, Band C, are given, and suppose on peg A a finite number ***n*** of disks with decreasing size are placed.

The objective of the game is to move the disks from peg A to peg C using peg B as an auxiliary. The rules of the game are as follows:

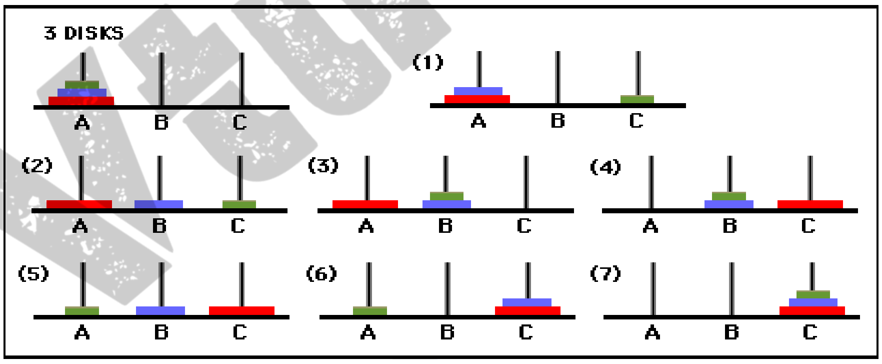
1. Only one disk may be moved at a time. Only the top disk on any peg may be moved to
   any other peg.
2. At no time can a larger disk be placed on a smaller disk.



Initial Setup of Towers of Hanoi with n = 6

We write A→B to denote the instruction "Move top disk from peg A to peg B"

Example: Towers of Hanoi problem for n = 3.



Solution: Observe that it consists of the following seven moves

1. Move top disk from peg A to peg C.
2. Move top disk from peg A to peg B.
3. Move top disk from peg C to peg B.
4. Move top disk from peg A to peg C.
5. Move top disk from peg B to peg A.
6. Move top disk from peg B to peg C.
7. Move top disk from peg A to peg C.

In other words,

n=3: A→C, A→B, C→B, A→C, B→A, B→C, A→C

For completeness, the solution to the Towers of Hanoi problem for n = 1 and n = 2 n=l: A→C

n=2: A→B, A→C, B→C

The Towers of Hanoi problem for n > 1 disks may be reduced to the following sub-problems:

(1) Move the top n - 1 disks from peg A to peg B

(2) Move the top disk from peg A to peg C: A→C.

(3) Move the top n - 1 disks from peg B to peg C.

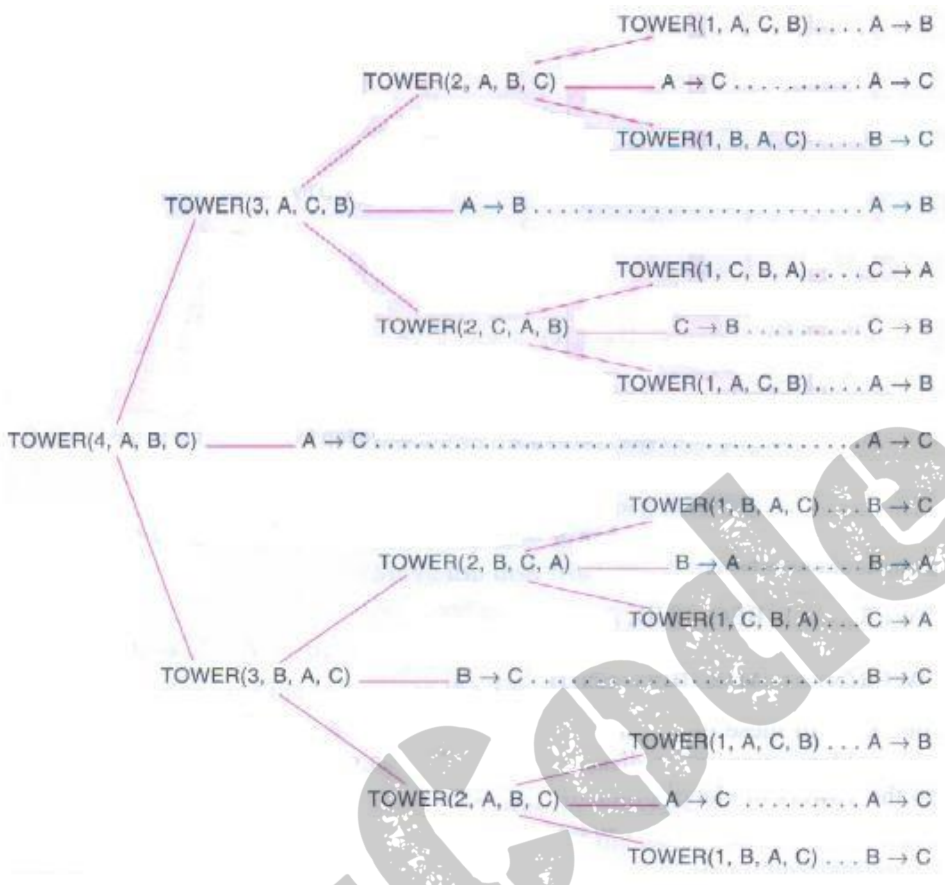
The general notation

- TOWER (N, BEG, AUX, END) to denote a procedure which moves the top n disks from the initial peg BEG to the final peg END using the peg AUX as an auxiliary.
- When n = 1, the solution:
  TOWER (1, BEG, AUX, END) consists of the single instruction BEG→END
- When n > 1, the solution may be reduced to the solution of the following three sub- problems:
  (a) TOWER (N - I, BEG, END, AUX)
  (b) TOWER (l, BEG, AUX, END) or BEG → END
  (c) TOWER (N - I, AUX, BEG, END)

```
void Hanoi(int n, char x, char y, char z)
{
      if (n > 1)
       {
          Hanoi(n-1,x,z,y);
          printf("Move disk %d from %c to %c.\n",n,x,z);
          Hanoi(n-1,y,x,z);
      }
      else
      {
          printf("Move disk %d from %c to %c.\n",n,x,z);
      }
 }
```

Recursive Implementation of tower of Hanoi

Example: Towers of Hanoi problem for n = 4



TOWER(4, A, B, C) — A → C .... A → C
TOWER(3, A, C, B) — A → B .... A → B
TOWER(2, A, B, C) — A → C .... A → C
TOWER(1, A, C, B) .... A → B
TOWER(1, B, A, C) .... B → C
TOWER(2, C, A, B) — C → B .... C → B
TOWER(1, C, B, A) .... C → A
TOWER(1, A, C, B) .... A → B
TOWER(3, B, A, C) — B → C .... B → C
TOWER(2, B, C, A) — B → A .... B → A
TOWER(1, B, A, C) .... B → C
TOWER(1, C, B, A) .... C → A
TOWER(2, A, B, C) — A → C .... A → C
TOWER(1, A, C, B) ... A → B
TOWER(1, B, A, C) ... B → C

ARRAYS

- An Array is defined as, an ordered set of similar data items. All the data items of an array are stored in consecutive memory locations.
- The data items of an array are of same type and each data items can be accessed using the same name but different index value.
- An array is a set of pairs, such that each index has a value associated with it. It can be called as corresponding or a mapping

Ex:	<index, value>	
	< 0 , 25 >	list[0]=25
	< 1 , 15 >	list[1]=15
	< 2 , 20 >	list[2]=20
	< 3 , 17 >	list[3]=17
	< 4 , 35 >	list[4]=35

Here, list is the name of array. By using, list [0] to list [4] the data items in list can be accessed.

```
Structure Array is
objects: A set of pairs <index, value> where for each value of
     index there is a value from the set item. Index is a finite
     ordered set of one or more dimensions, for example,
     {0, … , n-1} for one dimension,
     {(0,0),(0,1),(0,2),(1,0),(1,1),(1,2),(2,0),(2,1),(2,2)} for
     two dimensions, etc.

 Functions:
 for all A ∈ Array, i ∈index, x ∈ item, j, size ∈ integer
 Array Create(j, list)  ::= return an array of j dimensions where
                            list is a j-tuple whose ith element is
                            the size of the ith dimension. Items are
                            undefined.
 Item Retrieve(A, i)  ::=   if (i ∈index)
                                 return the item associated with
                                 index value i in array A
                            else
                                 return error
 Array Store(A, i, x)   ::= if (i in index)
                                 return an array that is identical
                                 to array A except the new pair
                                 <i, x> has been inserted
                            else
                                 return error
end array
```

Abstract data type Array

Example: Program to find sum of n numbers

```
#define MAX_SIZE 100
float sum(float [], int);
float input[MAX_SIZE], answer;
int i;
void main (void)
{
      for (i = 0; i < MAX_SIZE; i++)
         input[i] = i;
      answer = sum(input, MAX_SIZE);
      printf("The sum is: %f\n", answer);
}
float sum(float list[], int n)
{
      int i;
      float tempsum = 0;
      for (i = 0; i < n; i++)
          tempsum += list[i];
      return tempsum;
}
```

Program to find sum of n numbers

ARRAYS IN C

• A one-dimensional array can be declared as follows:

int list[5]; //array of 5 integers

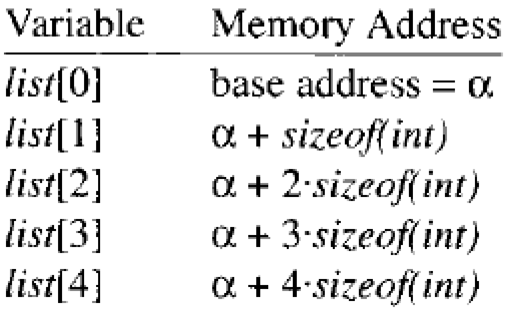
int *plist[5]; //array of 5 pointers to integers

• Compiler allocates 5 consecutive memory-locations for each of the variables 'list' and 'plist'.

• Address of first element list[0] is called base-address.

• Memory-address of list[i] can be computed by compiler as

α+i*sizeof(int) **where α=base address**



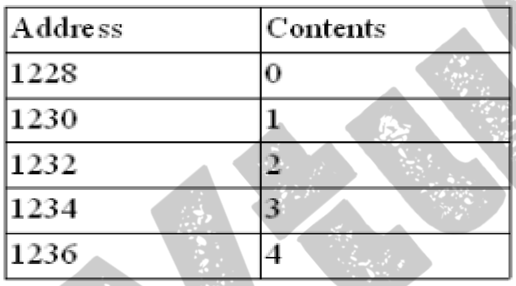
Variable	Memory Address
list[0]	base address = α
list[1]	α + *sizeof(int)*
list[2]	α + 2·*sizeof(int)*
list[3]	α + 3·*sizeof(int)*
list[4]	α + 4·*sizeof(int)*

Program to print both address of ith element of given array & the value found at that address:

```
void print1(int *ptr, int rows)

{

      /* print out a one-dimensional array using a pointer
      */ int i;

      printf("Address Contents\n");

      for (i=0; i < rows; i++)

            printf("%8u%5d\n", ptr+i, *(ptr+i));
      printf("\n");

}


void main()

{

   int one[] = {0, 1, 2, 3, 4};

   print1(&one[0], 5)

}
```

Program to print both address of ith element of given array

Output: one dimensional array addressing



Address	Contents
1228	0
1230	1
1232	2
1234	3
1236	4

DYNAMICALLY ALLOCATED ARRAYS ONE-DIMENSIONAL ARRAYS

• When writing programs, sometimes we cannot reliably determine how large an array must be.

• A good solution to this problem is to

→ defer this decision to run-time &

→ allocate the array when we have a good estimate of required array-size

• Dynamic memory allocation can be performed as follows:

```
int i,n,*list;
printf("enter the number of numbers to generate");
scanf("%d",&n);
if(n<1)
{
      printf("improper value");
      exit(0);
}
MALLOC(list, n*sizeof(int));
```

• The above code would allocate an array of exactly the required size and hence would not result in any wastage.

TWO DIMENSIONAL ARRAYS

• These are created by using the concept of array of arrays.

• A 2-dimensional array is represented as a 1-dimensional array in which each element has a pointer to a 1-dimensional array as shown below

int x[5][7]; //we create a 1-dimensional array x whose length is 5;

//each element of x is a 1-dimensional array whose length is 7.

• Address of x[i][j] = x[i]+j*sizeof(int)

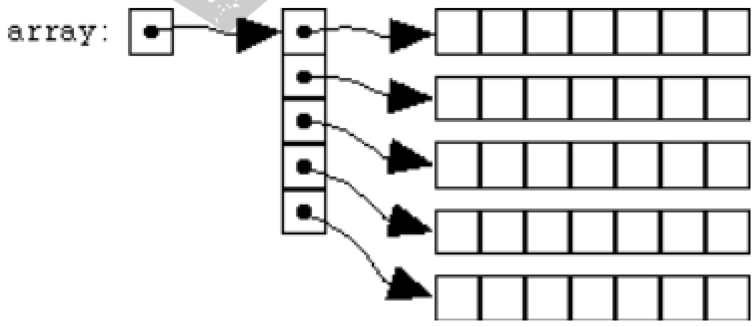


FIG: Array-of-arrays representation

```
 #include <stdlib.h> int
 **array;
 array = malloc(nrows * sizeof(int *));
 if(array == NULL)
 {
       printf("out of memory\n"); exit
       or return
 }
 for(i = 0; i < nrows; i++)
 {
       array[i] = malloc(ncolumns * sizeof(int));
       if(array[i] == NULL)
       {
             printf("out of memory\n"); exit
             or return
       }
}
```

Prg: Dynamically create a two-dimensional array

CALLOC

• These functions → allocate user-specified amount of memory & → initialize the allocated memory to 0.

• On successful memory-allocation, it returns a pointer to the start of the new block. On failure, it returns the value NULL.

• Memory can be allocated using calloc as shown below:

```
int *p;
p=calloc(n, sizeof(int)); //where n=array size
```

• To create clean and readable programs, a CALLOC macro can be created as shown below:

```
#define CALLOC(p,n,s)
if((p=calloc(n,s))==NULL)
{

printf("insufficient memory");

exit(1);

}
```

REALLOC

• These functions resize memory previously allocated by either malloc or calloc. For example,

realloc(p,s); //this changes the size of memory-block pointed at by p to s < oldSize, the rightmost oldSize-s bytes of old block are freed..

• When s>oldSize, the additional s-oldSize have an unspecified value and when s

• On successful resizing, it returns a pointer to the start of the new block. On failure, it returns the value NULL.

• To create clean and readable programs, the REALLOC macro can be created as shown below:

```
#define REALLOC(p,s)
if((p=realloc(p,s))==NULL)
{
      printf("insufficient memory");
      exit(0);
}
```

STRUCTURES AND UNIONS

Structures

Arrays are collections of data of the same type. In C there is an alternate way of grouping data that permits the data to vary in type. This mechanism is called the **struct,** short for structure. A structure (called a record in many other programming languages) is a collection of data items, where each item is identified as to its type and name.

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

➢ Creates a variable whose name is person and that has three fields:

- a **name** that is a **character array**
- an **integer** value representing the **age** of the person
- a **float value** representing the **salary** of the individual

➢ Dot operator(.) is used to access a particular member of the structure.

```
strcpy(person.name,"james") ;
person.age
person.salary = 35000;
```

➢ We can create our own structure data types by using the typedef statement as below:

```
typedef struct human—
being {
char name[10];
int age;
float salary;
 };
```

-OR-

```
typedef struct {
char name[10];
int age;
float salary;
      } human-being;
```

- Variables can be declared as follows:

```
humanBeing person1,person2;
```

- Structures cannot be directly checked for equality or inequality. So, we can write a function to do this.

```
int humans—equal(human—being personl,
human—being person2)
{ /* being otherwise return FALSE
if (strcmp(personl.name, person2.name)) return FALSE;
if (personl.age != person2.age) return FALSE;
if (personl.salary 1= person2.salary) return FALSE;
return TRUE;
return TRUE if personl and person2 are the same human
*/
}




if (humans—equal(personl,person2))
printf("The two human beings are the same\n"); else
printf{"The two human beings are not the same\n");
```

PRG: Function to check equality of structures

- We can embed a structure within a structure.

```
typedef struct {
int month;
int day;
int year; } date;

typedef struct human—being {
char name[10];
int age;
float salary;
date dob;
};
```

- A person born on February 11, 1944, would have the values for the *date* **struct** set as:

```
personl.dob.month = 2;
personl.dob.day = 11; personl.dob.year = 1944;
```

Unions

- This is similar to a structure, but the fields of a union must share their memory space. This means that only one field of the union is "active" at any given time.

```
typedef struct sex—type {
enum tag—field {female, male
} sex;

union {
int children;
int beard ;
} u;
};

typedef struct human—being {
char name[10];
int age;
float salary;
date dob;
sex—type sex—info;
};
human—being personl, person2;
```

- We could assign values to *person!* and *person2* as:

```
personl.sex—info.sex = male;
personl.sex—info.u.beard = FALSE;

and

person2.sex—info.sex = female;
person2.sex—info.u.children - 4;
```

- we first place a value in the tag field. This allows us to determine which field in the **union** is active. We then place a value in the appropriate field of the **union.**

Internal Implementation Of Structures

- The size of an object of a struct or union type is the amount of storage necessary to represent the largest component, including any padding that may be required.
- Structures must begin and end on the same type of memory boundary, for example, an even byte boundary or an address that is a multiple of 4, 8, or 16.

Self-Referential Structures

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

• These require dynamic storage management routines (malloc & free) to explicitly obtain and release memory.

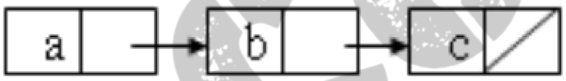
```
typedef struct list
{
char data;
list *link;  //list is a pointer to a list structure
} ;
```

- Consider three structures and values assigned to their respective fields:

```
List item1,item2,item3;
item1.data='a';
item2.data='b';
item3.data='c';
item1.link=item2.link=item3.link=NULL;
```

- We can attach these structures together as follows

```
item1.link=&item2;
item2.link=&item3;
```



POLYNOMIALS ABSTRACT DATA TYPE

- A polynomial is a sum of terms, where each term has a form ax^e , where x=variable, a=coefficient and e=exponent.
- For ex, $A(x)=3x^{20}+2x^5+4$ and $B(x)=x^4+10x^3+3x^2+1$
- The largest(or leading) exponent of a polynomial is called its degree.
- Assume that we have 2 polynomials,

```
Structure Polynomial is
 objects: p(x)=a1xe + .  . anxe ; a set of ordered pairs of <ei,ai> where
 ai in Coefficients and ei in Exponents,  ei are integers >= 0
functions:
for all poly, poly1, poly2 ∈ Polynomial, coef ∈Coefficients,
expon ∈ Exponents Polynomial Zero( ) ::= return the
polynomial, p(x) = 0
Boolean IsZero(poly) ::=     if (poly)
                                   return FALSE
                             else
                                   return TRUE

Coefficient Coef(poly, expon) ::= if (expon ∈ poly)
                                   return its coefficient
                               else
                                   return Zero
Exponent Lead_Exp(poly) ::= return the largest exponent in poly
Polynomial Attach(poly,coef, expon) ::= if (expon ∈ poly)
                                           return error
                                       else
                                         return the polynomial poly
                                         with the term <coef, expon>
                                         inserted
Polynomial Remove(poly, expon)::= if (expon ∈ poly)
                                    return the polynomial poly with
                                    the term whose exponent is expon
                                    deleted
                                  else
                                       return error
Polynomial SingleMult(poly, coef, expon) ::=
                              return the polynomial  poly•coef•xexpon
Polynomial Add(poly1, poly2) ::=
                              return   the   polynomial   poly1   +poly2
Polynomial Mult(poly1, poly2)::=
                              return the polynomial  poly1 • poly2
End Polynomia
```

$A(x)= \sum a_i x^i$ & $B(x)= \sum b_i x^i$ then $A(x)+B(x)= \sum(a_i + b_i)x^i$

POLYNOMIAL REPRESENTATION: FIRST METHOD

```
#define MAX_DEGREE 100
typedef struct
{
      int degree;
      float coef[MAX_DEGREE];
}polynomial;

polynomial a;
```

```
/* d =a + b, where a, b, and d are polynomials */
d = Zero( )
while (! IsZero(a) && ! IsZero(b))
do
{
      switch COMPARE (Lead_Exp(a), Lead_Exp(b))
      {
            case -1: d = Attach(d, Coef (b, Lead_Exp(b)), Lead_Exp(b));
                           b = Remove(b, Lead_Exp(b));
                           break;

            case  0: sum = Coef (a, Lead_Exp (a)) + Coef ( b,Lead_Exp(b));
                  if (sum)
                  {
                        Attach (d, sum, Lead_Exp(a));
                        a = Remove(a , Lead_Exp(a));
                        b = Remove(b , Lead_Exp(b));
                  }
                  break;

            case 1: d = Attach(d, Coef (a, Lead_Exp(a)), Lead_Exp(a));
                           a = Remove(a, Lead_Exp(a));
      }
}
insert any remaining terms of a or b into d
```

Initial version of padd function

• If a is of type 'polynomial' then $A(x)= \sum a_i x^i$ can be represented as:

a.degree=n

a.coeff[i]=a_{n-i}

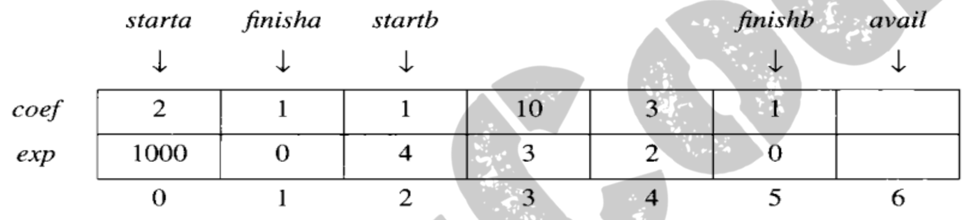
• In this representation, we store coefficients in order of decreasing exponents, such that a.coef[i] is the coefficient of x^{n-i} provided a term with exponent n-i exists; otherwise, a.coeff[i]=0

• Disadvantage: This representation wastes a lot of space. For instance, if a.degree<<MAX_DEGREE and polynomial is sparse, then we will not need most of the positions in a.coef[MAX_DEGREE] (sparse means number of terms with non-zero coefficient is small relative to degree of the polynomial).

POLYNOMIAL REPRESENTATION: SECOND METHOD

```
#define MAX_TERMS 100
typedef struct polynomial
{
      float coef;
      int expon;
}polynomial;

polynomial terms[MAX_TERMS];

int avail=0;
```

• A(x)=$2x^{1000}+1$ and B(x)=$x^4+10x^3+3x^2+1$ can be represented as shown below.



	starta ↓	finisha ↓	startb ↓			finishb ↓	avail ↓
coef	2	1	1	10	3	1	
exp	1000	0	4	3	2	0	
	0	1	2	3	4	5	6

Array representation of two polynomials

• startA & startB give the index of first term of A and B respectively . finishA & finishB give the index of the last term of A & B respectively avail gives the index of next free location in the array.

• Any polynomial A that has ‘n’ non-zero terms has startA & finishA such that finishA=startA+n-1

• Advantage: This representation solves the problem of many 0 terms since A(x)-2x1000+1 uses only 6 units of storage (one for startA, one for finishA, 2 for the coefficients and 2 for the exponents)

• Disadvantage: However, when all the terms are non-zero, the current representation requires about twice as much space as the first one.

POLYNOMIAL ADDITION:

```
void padd (int starta, int finisha, int startb, int finishb,int *
startd, int *finishd)
{
    /* add A(x) and B(x) to obtain D(x) */
    float coefficient;
   *startd = avail;
   while (starta <= finisha && startb <= finishb)
   {
     switch (COMPARE(terms[starta].expon, terms[startb].expon))
     {
      case -1: /* a expon < b expon */
           attach(terms[startb].coef,
           terms[startb].expon); startb++
           break;

      case 0: /* equal exponents */
           coefficient = terms[starta].coef +
           terms[startb].coef; if (coefficient)
             attach (coefficient, terms[starta].expon);
           starta++;
           startb++; break;

      case 1: /* a expon > b expon */
           attach(terms[starta].coef,
           terms[starta].expon); starta++;
     }
       /* add in remaining terms of A(x) */
     for( ; starta <= finisha; starta++)
       attach(terms[starta].coef,
       terms[starta].expon);
       /* add in remaining terms of B(x) */
     for( ; startb <= finishb; startb++)
       attach(terms[startb].coef,
       terms[startb].expon);
     *finishd =avail -1;
   }
}
```

Function to add two polynomials

```
void attach(float coefficient, int exponent)
{
   /* add a new term to the polynomial */
   if (avail >= MAX_TERMS)
   {
      fprintf(stderr, “Too many terms in the polynomial\n”);
      exit(1);
   }
   terms[avail].coef = coefficient;
   terms[avail++].expon = exponent;
}
```

Function to add a new term

ANALYSIS

• Let m and n be the number of non-zero terms in A and B respectively.

• If m>0 and n>0, the while loop is entered. At each iteration, we increment the value of startA or startB or both.

• Since the iteration terminates when either startA or startB exceeds finishA or finishB respectively, the number of iterations is bounded by m+n-1. This worst case occurs when $A(x)=\sum x^{2i}$ and $B(x)=\sum^{x2i}+1$

• The asymptotic computing time of this algorithm is O(n+m)

SPARSE MATRIX REPRESENTATION

• We can classify uniquely any element within a matrix by using the triple . Therefore, we can use an array of triples to represent a sparse matrix.

• Sparse matrix contains many zero entries.

• When a sparse matrix is represented as a 2-dimensional array, we waste space For ex, if 100*100 matrix contains only 100 entries then we waste 9900 out of 10000 memory spaces.

• Solution: Store only the non-zero elements.

$$\begin{array}{c|ccc|} & \text{col 1} & \text{col 2} & \text{col 3} \\ \text{row 1} & -27 & 3 & 4 \\ \text{row 2} & 6 & 82 & -2 \\ \text{row 3} & 109 & -64 & 11 \\ \text{row 4} & 12 & 8 & 9 \\ \text{row 5} & 48 & 27 & 47 \end{array} \; 5*3$$

(a)

$$\begin{array}{c|cccccc|} & \text{col1} & \text{col2} & \text{col3} & \text{col4} & \text{col5} & \text{col6} \\ \text{row0} & 15 & 0 & 0 & 22 & 0 & -15 \\ \text{row1} & 0 & 11 & 3 & 0 & 0 & 0 \\ \text{row2} & 0 & 0 & 0 & -6 & 0 & 0 \\ \text{row3} & 0 & 0 & 0 & 0 & 0 & 0 \\ \text{row4} & 91 & 0 & 0 & 0 & 0 & 0 \\ \text{row5} & 0 & 0 & 28 & 0 & 0 & 0 \end{array} \; 6*6$$

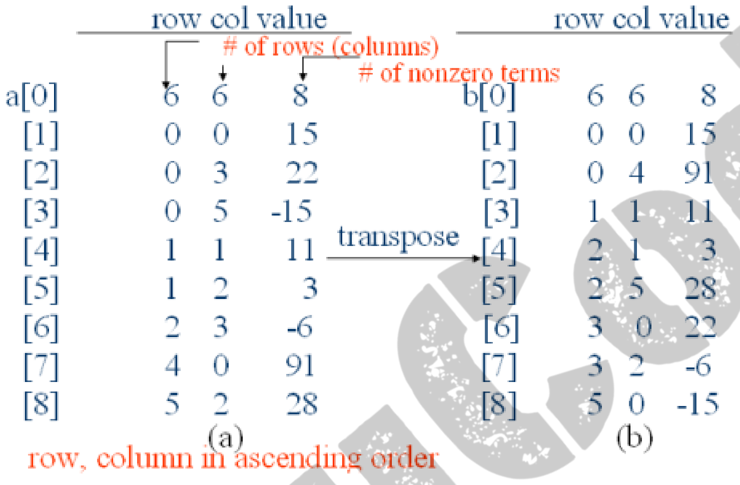
(b)

```
Structure Sparse_Matrix is

 objects: a set of triples, <row, column, value>, where row and column are
         integers and form a unique combination, andvalue comes from the set
         item.

 functions:

  for all a, b ∊ Sparse_Matrix, x ∊ item, i, j, max_col, max_row ∊ index
 Sparse_Marix Create(max_row, max_col) ::=

                  return a Sparse_matrix that can hold up to max_items = max
                  _row * max_col and  whose maximum row size is max_row and
                  whose maximum  column size is max_col.

Sparse_Matrix Transpose(a) ::=

                  return the matrix produced by interchanging the row and
                  column value of every triple.

Sparse_Matrix Add(a, b) ::=

                if the dimensions of a and b are the same

                   return the matrix produced by adding  corresponding items,
                   namely those with  identical row and column values.

               else

                    return error

Sparse_Matrix Multiply(a, b) ::=

               if number of columns in a equals number of  rows in b

                  return the matrix d produced by multiplying a by b according
                  to the formula:d [i] [j]=□(a[i][k]•b[k][j]) where d (i, j)
                  is the (i,j)th element

               else

                  return error.

End Sparse_Matrix
```

SPARSE MATRIX REPRESENTATION

- We can classify uniquely any element within a matrix by using the triple <row,col,value>. Therefore, we can use an array of triples to represent a sparse matrix

```
    SpareMatrix Create(maxRow,maxCol) ::=
    #define MAX_TERMS 101
    typedef struct term
           {
             int col;
             int row;
             int value;
           } term;
  term  a[MAX_TERMS];
```



# of rows (columns)
# of nonzero terms
transpose
(a)
(b)
row, column in ascending order

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

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

Sparse matrix and its transpose stored as triples

- a[0].row contains the number of rows;

  a[0].col contains number of columns and

  a[0].value contains the total number of nonzero entries.

TRANSPOSING A MATRIX

• To transpose a matrix ,we must interchange the rows and columns.

• Each element a[i][j] in the original matrix becomes element b[j][i] in the transpose matrix.

• Algorithm To transpose a matrix:

```
for each row i
          take element<i,j,value> and store it
          as element<i,j,value>  of the transpose;

for all elements in column j
          place element<i,j,value> in
          element<i,j,value>
```

```
 void transpose (term a[], term b[])
 {
/* b is set to the transpose of a */

      int n, i, j, currentb;
      n = a[0].value;       /* total number of elements */
      b[0].row = a[0].col;  /* rows in b = columns in a */
      b[0].col = a[0].row;  /*columns in b = rows in a */
      b[0].value = n;
      if (n > 0)
      {        /*non zero matrix */
            currentb = 1;
            for (i = 0; i < a[0].col; i++)   /* transpose by columns
                                                 in a */
              for( j = 1; j <= n; j++)  /* find elements from the
                                             current column */
                  if (a[j].col == i)
                  {/* element is in current column, add it to b */
                    b[currentb].row = a[j].col;
                    b[currentb].col  = a[j].row;
                    b[currentb].value = a[j].value;
                    currentb++
                  }
      }
 }
```

Transpose of a sparse matrix

```
  void mmult (term a[ ], term b[ ], term d[ ] )
  {   /* multiply two sparse matrices */

      int i, j, column, totalb = b[].value, totald = 0;
      int rows_a = a[0].row, cols_a = a[0].col, totala = a[0].value;
      int cols_b = b[0].col,
      int row_begin = 1, row = a[1].row, sum =0;
      int new_b[MAX_TERMS][3];
      if (cols_a != b[0].row)
      {
         fprintf (stderr, "Incompatible matrices\n");
         exit (1);
      }

      fast_transpose(b, new_b);

      a[totala+1].row = rows_a;  /* set boundary condition */
      new_b[totalb+1].row = cols_b;
      new_b[totalb+1].col = 0;

      for (i = 1; i <= totala; )
      {
            column = new_b[1].row;
            for (j = 1; j <= totalb+1;)
            {     /* mutiply row of a by column of b */
                  if (a[i].row != row)
                  {
                        storesum(d, &totald, row, column, &sum);
                        i = row_begin;
                        for (; new_b[j].row == column; j++);
                              column =new_b[j].row
                  }
                  else
                     switch (COMPARE (a[i].col, new_b[j].col))
                     {
                        case -1: i++; break; /* go to next term in a */
                        case 0: /* add terms, go to next term in a
                                                            and b */
                               sum += (a[i++].value * new_b[j++].value);
                        case 1: j++  /* advance to next term in b*/
                     }
            }     /* end of for j <= totalb+1 */
            for (; a[i].row == row; i++);
                row_begin = i;
            row = a[i].row;
      }     /* end of for i <=totala */
      d[0].row = rows_a;
      d[0].col = cols_b;
      d[0].value = totald;
}
```

Sparse matrix multiplication

```
void storesum(term d[ ], int *totald, int row, int column, int *sum)

{

      /* if *sum != 0, then it along with its row and column position is
      stored as the *totald+1 entry in d */

       if (*sum)

          if (*totald < MAX_TERMS)

          {

            d[++*totald].row      =      row;
            d[*totald].col      =      column;
            d[*totald].value = *sum;

           }

           else

          {

           fprintf(stderr, "Numbers of terms in product exceed %d\n",
           MAX_TERMS); exit(1);

          }

      }

}
```

storesum function

THE STRING ABSTRACT DATA TYPE

The string, whose component elements are characters. As an ADT, we define a string to have the form, S = So, .. . , where Si are characters taken from the character set of the programming language. If n = 0, then S is an empty or null string.There are several useful operations we could specify for strings.

```
structure String is
  objects: a finite set of zero or more characters.
  functions:
    for all s, t ∈ String, i, j, m ∈ non-negative integers

    String Null(m)          ::=   return a string whose maximum length is
                                  m characters, but is initially set to NULL
                                  We write NULL as "".
    Integer Compare(s, t)   ::=   if s equals t
                                  return 0
                                  else if s precedes t
                                          return −1
                                          else return +1
    Boolean IsNull(s)       ::=   if (Compare(s, NULL))
                                  return FALSE
                                  else return TRUE
    Integer Length(s)       ::=   if (Compare(s, NULL))
                                  return the number of characters in s
                                  else return 0.
    String Concat(s, t)     ::=   if (Compare(t, NULL))
                                  return a string whose elements are those
                                  of s followed by those of t
                                  else return s.
    String Substr(s, i, j)  ::=   if ((j > 0) && (i + j−1) < Length(s))
                                  return the string containing the characters
                                  of s at positions i, i + 1, · · · , i + j−1.
                                  else return NULL.
```

we represent strings as character arrays terminated with the null character \0.

```
#define MAX_SIZE 100 /*maximum size of string */
char s[MAX_SIZE] = {"dog"};
char t[MAX_SIZE] = {"house"};
```

Function	Description
*char *strcat(char *dest, char *src)*	concatenate *dest* and *src* strings; return result in *dest*
*char *strncat(char *dest, char *src, int n)*	concatenate *dest* and *n* characters from *src*; return result in *dest*
*char *strcmp(char *str1, char *str2)*	compare two strings; return < 0 if *str1* < *str2*; 0 if *str1* = *str2*; > 0 if *str1* > *str2*
*char *strncmp(char *str1, char *str2, int n)*	compare first *n* characters return < 0 if *str1* < *str2*; 0 if *str1* = *str2*; > 1 if *str1* > *str2*
*char *strcpy(char *dest, char *src)*	copy *src* into *dest*; return *dest*
*char *strncpy(char *dest, char *src, int n)*	copy *n* characters from *src* string into *dest*; return *dest*;
*size_t strlen(char *s)*	return the length of a *s*
*char *strchr(char *s, int c)*	return pointer to the first occurrence of *c* in *s*; return *NULL* if not present
*char *strrchr(char *s, int c)*	return pointer to last occurrence of *c* in *s*; return *NULL* if not present
*char *strtok(char *s, char *delimiters)*	return a token from *s*; token is surrounded by *delimiters*
*char *strstr(char *s, char *pat)*	return pointer to start of *pat* in *s*
*size_t strspn(char *s, char *spanset)*	scan *s* for characters in *spanset*; return length of span
*size_t strcspn(char *s, char *spanset)*	scan *s* for characters not in *spanset*; return length of span
*char *strpbrk(char *s, char *spanset)*	scan *s* for characters in *spanset*; return pointer to first occurrence of a character from *spanset*

Figure 2.7: C string functions

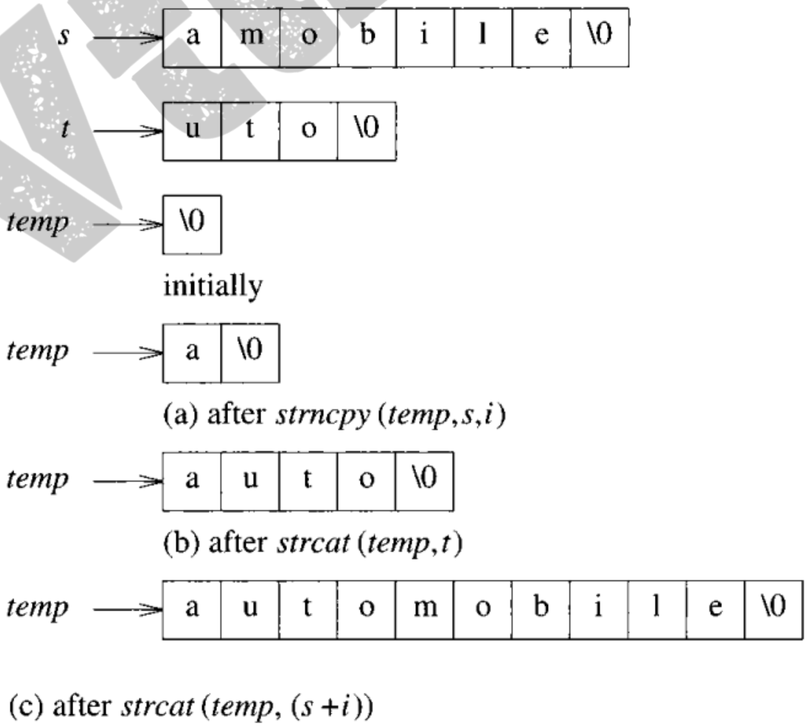
String insertion:

Assume that we have two strings, say string 1 and string 2, and that we want to insert string 2 into string 1 starting at the ith position of string 1. We begin with the declarations:

```
#include <string.h>
#define MAX_SIZE 100 /*size of largest string*/
char string1[MAX_SIZE], *s = string1;
char string2[MAX_SIZE],  *t = string2;
```

```
void strnins(char *s, char *t, int i)
{
/* insert string t into string s at position i */
   char string[MAX_SIZE], *temp = string;

   if (i < 0 && i > strlen(s)) {
      fprintf(stderr,"Position is out of bounds \n");
      exit(1);
   }
   if (!strlen(s))
      strcpy(s,t);
   else if (strlen(t)) {
      strncpy(temp, s,i);
      strcat(temp,t);
      strcat(temp, (s+i));
      strcpy(s, temp);
   }
}
```

Program 2.11: String insertion function



Pattern Matching

Assume that we have two strings, string and pat, where pat is a pattern to be searched for in string. The easiest way to determine if pat is in string is to use the built-in function strstr. If we have the following declarations:

```
char pat[MAX_SIZE], string[MAX_SIZE], *t;
```

then we use the following statements to determine if *pat* is in *string*:

```
if (t = strstr(string,pat))
  printf("The string from strstr is: %s\n",t);
else
  printf("The pattern was not found with strstr\n");
```

The call (t = strstr(string,pat)) returns a null pointer if pat is not in string.

If pat is in string, t holds a pointer to the start of pat in string. The entire string beginning at position t is printed out.

Although strstr seems ideally suited to pattern matching, there are two reasons why we may want to develop our own pattern matching function:

- The function strstr is new to ANSI C. Therefore, it may not be available with the compiler we are using.
- There are several different methods for implementing a pattern matching function. The easiest but least efficient method sequentially examines each character of the string until it finds the pattern or it reaches the end of the string. If pat is not in string, this method has a computing time of O(n . m) where n is the length of pat and w is the length of string. We can do much better than this, if we create our own pattern matching function.

Knuth, Morris, Pratt Pattern Matching algorithm.

Definition: If $p = p_0p_1 \cdots p_{n-1}$ is a pattern, then its *failure function, f,* is defined as:

$$f(j) = \begin{cases} \text{largest } i < j \text{ such that } p_0p_1 \cdots p_i = p_{j-i}p_{j-i+2} \cdots p_j & \text{if such an } i \geq 0 \text{ exists} \\ -1 & \text{otherwise} \end{cases} \quad \square$$

For the example pattern, *pat* = *abcabcacab*, we have:

j	0	1	2	3	4	5	6	7	8	9
pat	*a*	*b*	*c*	*a*	*b*	*c*	*a*	*c*	*a*	*b*
f	−1	−1	−1	0	1	2	3	−1	0	1

```
int pmatch(char *string, char *pat)
{/* Knuth, Morris, Pratt string matching algorithm */
    int i = 0, j = 0;
    int lens = strlen(string);
    int lenp = strlen(pat);
    while ( i < lens && j < lenp ) {
        if (string[i] == pat[j]) {
            i++; j++; }
        else if (j == 0) i++;
             else j = failure[j-1]+1;
    }
    return ( (j == lenp) ? (i-lenp) : -1);
}
```

Program 2.14: Knuth, Morris, Pratt pattern matching algorithm

```
void fail(char *pat)
{/* compute the pattern's failure function */
   int n = strlen(pat);
   failure[0] = -1;
   for (j=1; j < n; j++) {
       i = failure[j-1];
       while ((pat[j] != pat[i+1]) && (i >= 0))
          i = failure[i];
       if (pat[j] == pat[i+1])
          failure[j] = i+1;
       else failure[j] = -1;
   }
}
```

Program 2.15: Computing the failure function

THE STACK ABSTRACT DATA TYPE

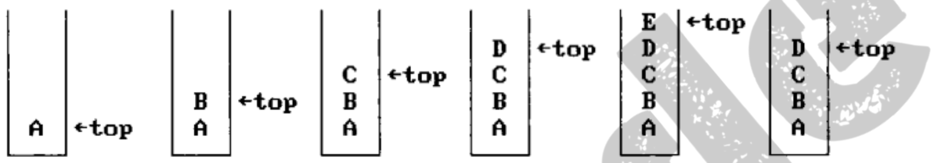
STACK

• This is an ordered-list in which insertions(called push) and deletions(called pop) are made at one end called the top

• Since last element inserted into a stack is first element removed, a stack is also known as a LIFO list(Last In First Out).

When an element is inserted in a stack, the concept is called push, and when an element is removed from the stack, the concept is called pop.

Trying to pop out an empty stack is called underflow and trying to push an element in a full stack is called overflow.



: **Inserting and deleting elements in a stack**

As shown in above figure, the elements are added in the stack in the order A, B, C, D, E, then E is the first element that is deleted from the stack and the last element is deleted from stack is A. Figure illustrates this sequence of operations.

Since the last element inserted into a stack is the first element removed, a stack is also known as a Last-In-First-Out (LIFO) list.

SYSTEM STACK

A stack used by a program at run-time to process function-calls is called system-stack.

• When functions are invoked, programs

→ create a stack-frame (or activation-record) &

→ place the stack-frame on top of system-stack

• Initially, stack-frame for invoked-function contains only

→ pointer to previous stack-frame &

→ return-address

• The previous stack-frame pointer points to the stack-frame of the invoking-function while return-address contains the location of the statement to be executed after the function terminates.

• If one function invokes another function, local variables and parameters of the invoking-function are added to its stack-frame.

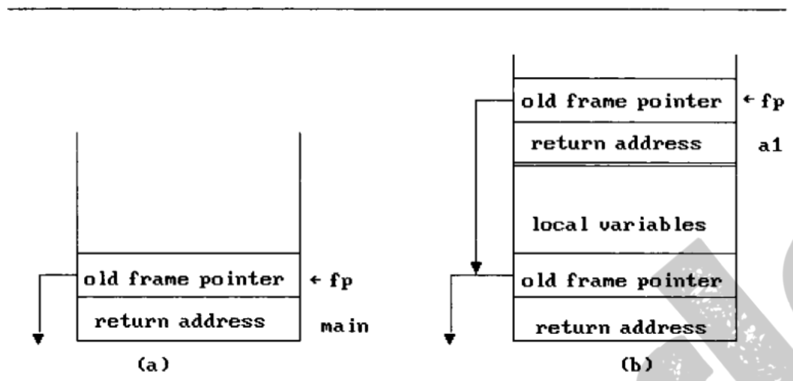
• A new stack-frame is then

→ created for the invoked-function &

→ placed on top of the system-stack

• When this function terminates, its stack-frame is removed (and processing of the invoking-function, which is again on top of the stack, continues).

• Frame-pointer(fp) is a pointer to the current stack-frame.

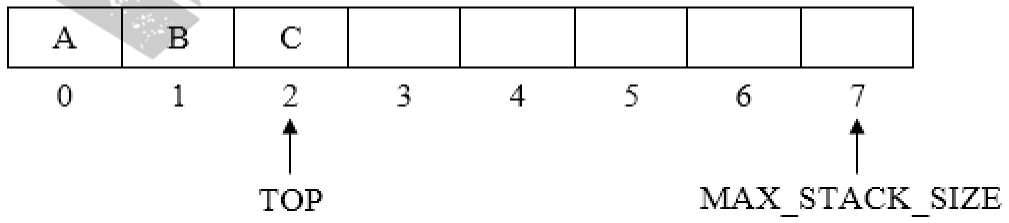


System stack after function call

ARRAY REPRESENTATION OF STACKS

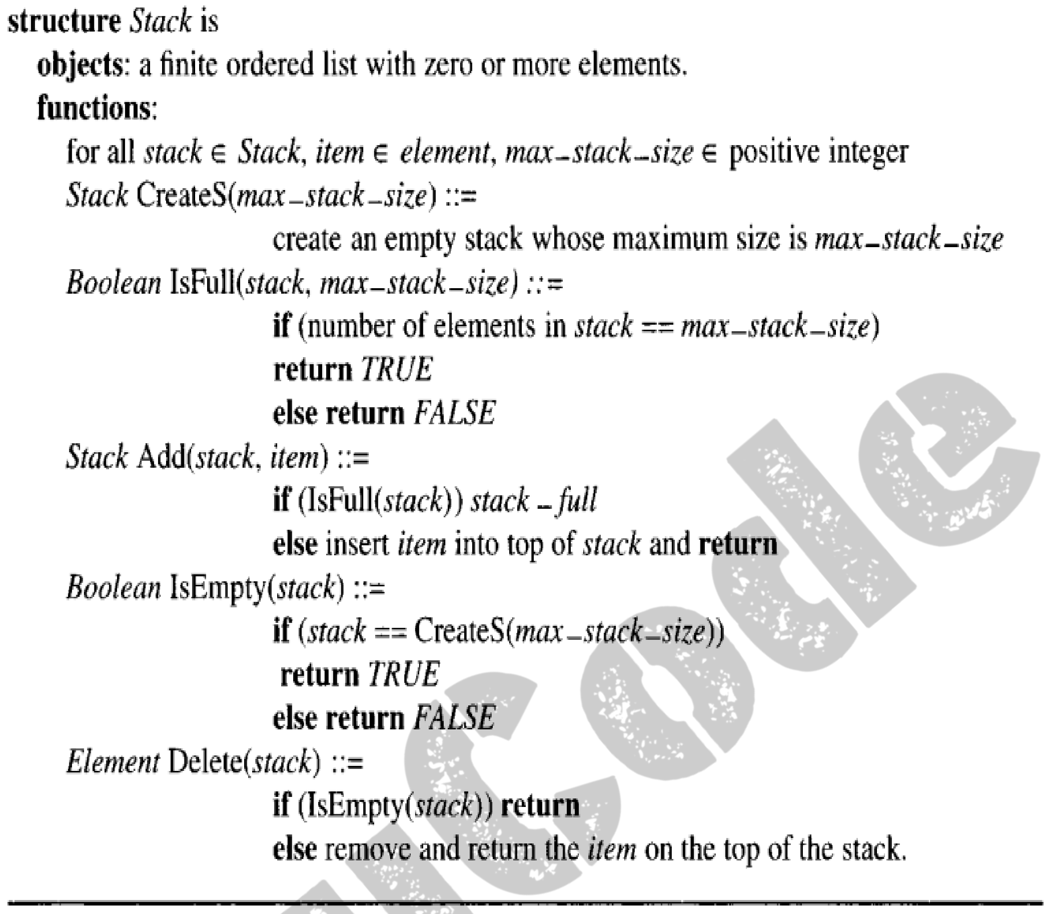
- Stacks may be represented in the computer in various ways such as one-way linked list (Singly linked list) or linear array.
- Stacks are maintained by the two variables such as TOP and MAX_STACK_SIZE.
- TOP which contains the location of the top element in the stack. If TOP= -1, then it indicates stack is empty.
- MAX_STACK_SIZE which gives maximum number of elements that can be stored in stack.

Stack can represented using linear array as shown below



<u>Stack ADT</u>

• The following operations make a stack an ADT. For simplicity, assume the data is an integer type.



```
structure Stack is
  objects: a finite ordered list with zero or more elements.
  functions:
    for all stack ∈ Stack, item ∈ element, max_stack_size ∈ positive integer
    Stack CreateS(max_stack_size) ::=
                    create an empty stack whose maximum size is max_stack_size
    Boolean IsFull(stack, max_stack_size) ::=
                    if (number of elements in stack == max_stack_size)
                    return TRUE
                    else return FALSE
    Stack Add(stack, item) ::=
                    if (IsFull(stack)) stack_full
                    else insert item into top of stack and return
    Boolean IsEmpty(stack) ::=
                    if (stack == CreateS(max_stack_size))
                     return TRUE
                    else return FALSE
    Element Delete(stack) ::=
                    if (IsEmpty(stack)) return
                    else remove and return the item on the top of the stack.
```

Structure 3.1: Abstract data type *Stack*

• Main stack operations

– Push (int data): Inserts data onto stack.

– int Pop(): Removes and returns the last inserted element from the stack.

• Auxiliary stack operations

– int Top(): Returns the last inserted element without removing it.

– int Size(): Returns the number of elements stored in the stack.

– int IsEmptyStack(): Indicates whether any elements are stored in the stack or not.

– int IsFullStack(): Indicates whether the stack is full or not.

• The easiest way to implement this ADT is by using a one-dimensional array, say, stack [MAX-STACK-SIZE], where MAX STACK SIZE is the maximum number of entries.

• The first, or bottom, element of the stack is stored in stack[0], the second in stack[1] and the ith in stack [i-1].

- Associated with the array is a variable, top, which points to the top element in the stack. Initially, top is set to -1 to denote an empty stack.
- we have specified that element is a structure that consists of only a key field.

1. CREATE STACK:

```
Stack CreateS(maxStackSize ):::=
        #define MAX_STACK_SIZE 100 /* maximum stack size*/
        typedef struct
                {
                        int key;
                                        /* other fields */
                } element;
        element stack[MAX_STACK_SIZE];
        int top = -1;
```

The **element** which is used to insert or delete is specified as a structure that consists of only a **key** field.

1. Boolean IsEmpty(Stack)::= top < 0;

2. Boolean IsFull(Stack)::= top >= MAX_STACK_SIZE-1;

The **IsEmpty** and **IsFull** operations are simple, and is implemented directly in the program push and pop functions. Each of these functions assumes that the variables **stack** and **top** are global.

<u>Add an item to a stack</u>

```
void add(int *top, element item)
{
/* add an item to the global stack */
   if (*top >= MAX_STACK_SIZE-1) {
      stack_full();
      return;
   }
   stack[++*top] = item;
}
```

• Function push() checks to see if the stack is full. If it is, it calls stackFull, which prints an error message and terminates execution.

• When the stack is not full, we increment top and assign item to stack[top].

Delete an item in a stack

```
element delete(int *top)
{
/* return the top element from the stack */
   if (*top == -1)
      return stack_empty(); /* returns an error key */
   return stack[(*top)--];
}
```

For deletion, the stack-empty function should print an error message and return an item of type element with a key field that contains an error code.

STACK USING DYNAMIC ARRAYS

• Shortcoming of static stack implementation: is the need to know at compile-time, a good bound(MAX_STACK_SIZE) on how large the stack will become.

• This shortcoming can be overcome by

→ using a dynamically allocated array for the elements &

→ then increasing the size of the array as needed

• Initially, capacity=1 where capacity=maximum no. of stack-elements that may be stored in array.

• The CreateS() function can be implemented as follows

```
Stack CreateS(max-stack-size') ::=
 #define MAX—STACK—SIZE 100 /*maximum stack size */
typedef struct
{
      int key;
       /* other fields */
} element;
element stack[MAX—STACK—SIZE];
int top - -1;
Boolean IsEmpty(Stack) ::= top <0;
Boolean IsFulI(Stack) ::= top >= MAX-STACK-SIZE-1;
```

• Once the stack is full, realloc() function is used to increase the size of array.

• In array-doubling, we double array-capacity whenever it becomes necessary to increase the capacity of an array.

ANALYSIS

• In worst case, the realloc function needs to

→ allocate 2*capacity*sizeof(*stack) bytes of memory and

→ copy capacity*sizeof(*stack) bytes of memory from the old array into the new one.

• The total time spent over all array doublings = O(2k) where capacity=2k

• Since the total number of pushes is more than 2k-1 , the total time spend in array doubling is O(n) where n=total number of pushes.

STACK APPLICATIONS: POLISH NOTATION

Expressions: It is sequence of operators and operands that reduces to a single value after evaluation is called an expression.

X = a / b – c + d * e – a * c

In above expression contains operators (+, –, /, *) operands (a, b, c, d, e).

Expression can be represented in in different format such as

- Prefix Expression or Polish notation
- Infix Expression
- Postfix Expression or Reverse Polish notation

- **Infix Expression:** In this expression, the binary operator is placed in-between the operand. The expression can be parenthesized or un- parenthesized.

  Example: A + B

  Here, **A** & **B** are operands and **+** is operand

- **Prefix or Polish Expression:** In this expression, the operator appears before its operand.

  Example: + A B

  Here, **A** & **B** are operands and **+** is operand

- **Postfix or Reverse Polish Expression:** In this expression, the operator appears after its operand.

  Example: A B +

  Here, **A** & **B** are operands and **+** is operand

Precedence of the operators

The first problem with understanding the meaning of expressions and statements is finding out the order in which the operations are performed.

Example: assume that a =4, b =c =2, d =e =3 in below expression

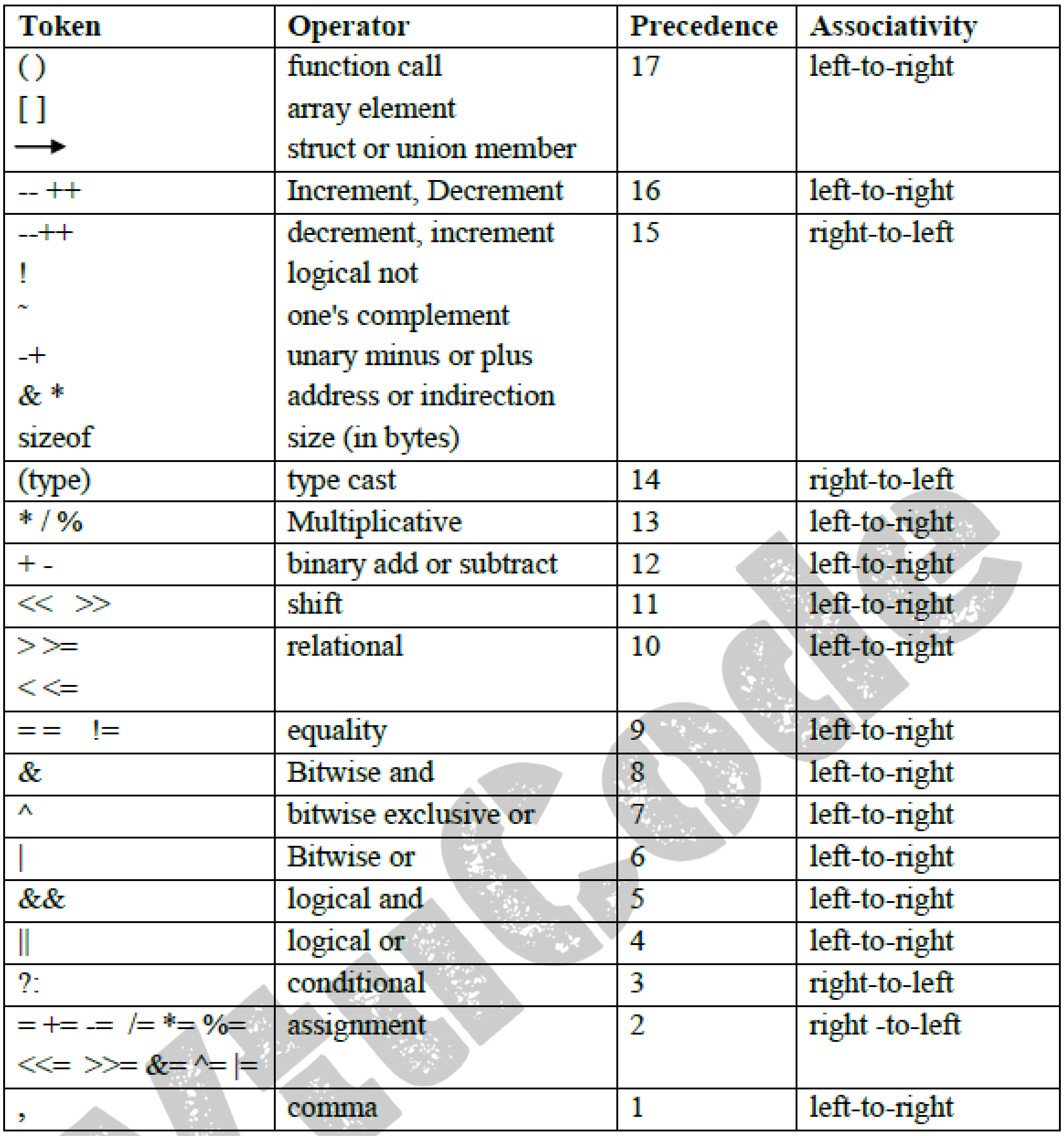
X = a / b – c + d * e – a * c

((4/2)-2) + (3*3)-(4*2)		(4/ (2-2 +3)) *(3-4)*2
=0+9-8 =1	**OR**	= (4/3) * (-1) * 2 = -2.66666

The first answer is picked most because division is carried out before subtraction, and multiplication before addition. If we wanted the second answer, write expression differently using parentheses to change the order of evaluation

X= ((a / (b – c + d)) * (e – a) * c

In C, there is a precedence hierarchy that determines the order in which operators are evaluated. Below figure contains the precedence hierarchy for C.



Token	Operator	Precedence	Associativity
( ) [ ] →	function call array element struct or union member	17	left-to-right
-- ++	Increment, Decrement	16	left-to-right
--++ ! ~ -+ & * sizeof	decrement, increment logical not one's complement unary minus or plus address or indirection size (in bytes)	15	right-to-left
(type)	type cast	14	right-to-left
* / %	Multiplicative	13	left-to-right
+ -	binary add or subtract	12	left-to-right
<< >>	shift	11	left-to-right
> >= < <=	relational	10	left-to-right
= = !=	equality	9	left-to-right
&	Bitwise and	8	left-to-right
^	bitwise exclusive or	7	left-to-right
|	Bitwise or	6	left-to-right
&&	logical and	5	left-to-right
||	logical or	4	left-to-right
?:	conditional	3	right-to-left
= += -= /= *= %= <<= >>= &= ^= |=	assignment	2	right -to-left
,	comma	1	left-to-right

- The operators are arranged from highest precedence to lowest. Operators with highest precedence are evaluated first.
- The associativity column indicates how to evaluate operators with the same precedence. For example, the multiplicative operators have left-to-right associativity. This means that the expression **a * b / c % d / e** is equivalent to **((((a * b) / c) % d) / e)**
- Parentheses are used to override precedence, and expressions are always evaluated from the innermost parenthesized expression first

INFIX TO POSTFIX CONVERSION

An algorithm to convert infix to a postfix expression as follows:

1. Fully parenthesize the expression.
2. Move all binary operators so that they replace their corresponding right parentheses.
3. Delete all parentheses.

Example: Infix expression: a/b -c
+d*e -a*c Fully parenthesized :
((((a/b)-c) + (d*e))-a*c))
: a b / e – d e * + a c *

Example [Parenthesized expression]: Parentheses make the translation process more difficult because the equivalent postfix expression will be parenthesis-free. The expression **a*(b +c)*d** which results **abc +*d*** in postfix. Figure shows the translation process.

Token	Stack			Top	Output
	[0]	[1]	[2]		
a				-1	a
*	*			0	a
(	*	(		1	a
b	*	(		1	ab
+	*	(	+	2	ab
c	*	(	+	2	abc
)	*			0	abc+
*	*			0	abc +*
d	*			0	abc +*d
eos	*			0	abc +*d*

- The analysis of the examples suggests a precedence-based scheme for stacking and unstacking operators.

- The left parenthesis complicates matters because it behaves like a low-precedence operator when it is on the stack and a high-precedence one when it is not. It is placed in the stack whenever it is found in the expression, but it is unstacked only when its matching right parenthesis is found.
- There are two types of precedence, **in-stack precedence (isp)** and **incoming precedence (icp)**.

The declarations that establish the precedence's are:

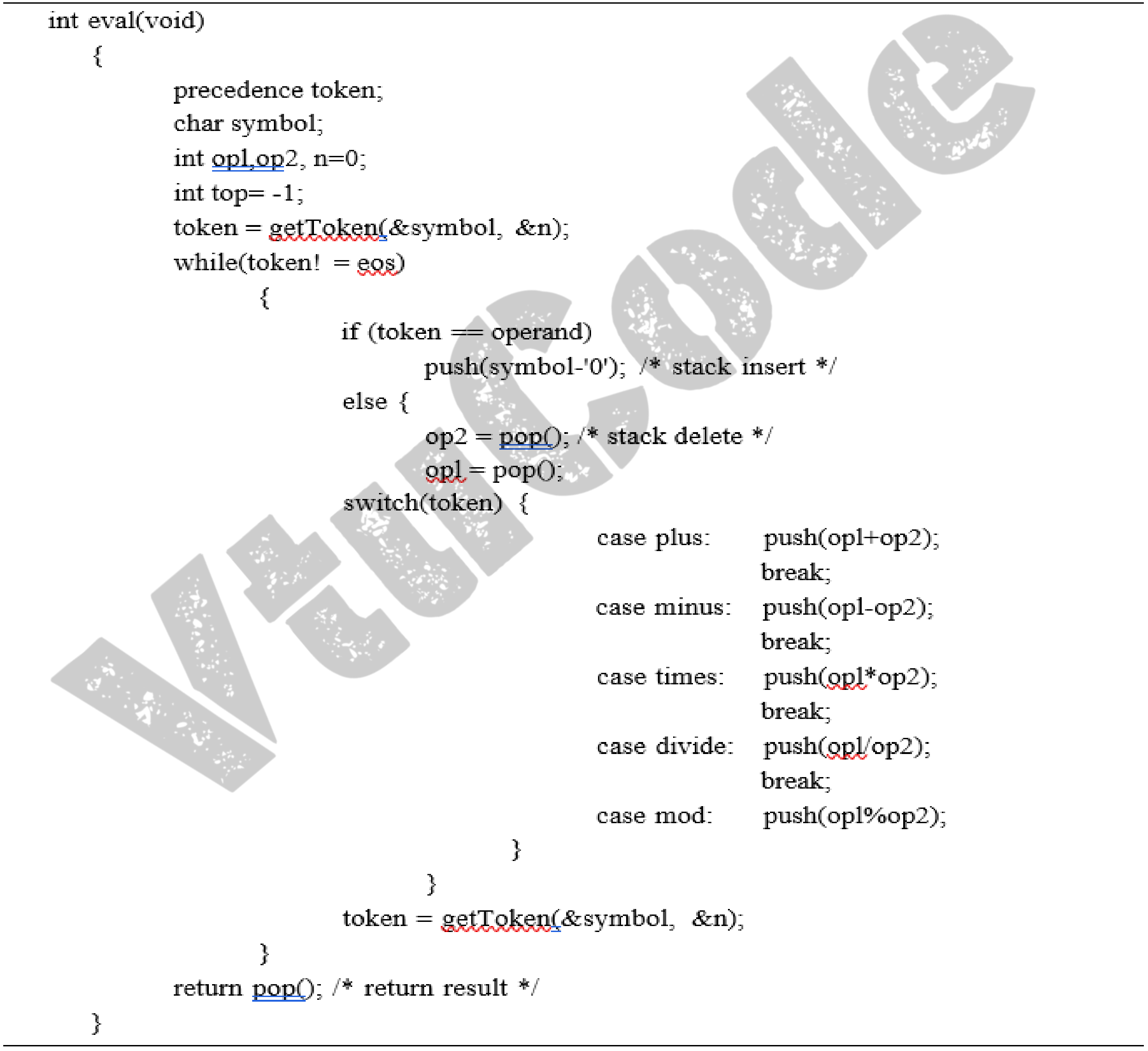
```
/* isp and icp arrays-index is value of precedence lparen rparen, plus, minus, times,
divide, mod, eos */
int isp[] = {0,19,12,12,13,13,13,0};
int icp[] = {20,19,12,12,13,13,13,0};
```

```
void postfix(void)
{
        char
        symbol;
        precede
        nce
        token;
        int n = 0,top = 0; /* place eos on
        stack */ stack[0] = eos;
        for (token = getToken(&symbol, &n); token != eos; token =
              getToken(&symbol,& n ))
                {
                        if (token == operand)
                        printf("%c",
                symbol); else if
                (token == rparen)
                        {
                                while (stack[top] !=
                                        lparen)
                                        printToken(p
                                        op( ));
                          pop( );
                  }
          else{
                        while(isp[stack[top]] >=
                                icp[token])
                                printToken(pop());
                        push(token);
                     }
        }
    while((token = pop ())!= eos)
            printToken(token);
    printf("\n");
}
```

Program: Function to convert from infix to postfix

EVALUATION OF POSTFIX EXPRESSION

- The evaluation process of postfix expression is simpler than the evaluation of infix expressions because there are no parentheses to consider.
- To evaluate an expression, make a single **left-to-right** scan of it. Place the operands on a stack until an operator is found. Then remove from the stack, the correct number of operands for the operator, perform the operation, and place the result back on the stack and continue this fashion until the end of the expression. We then remove the answer from the top of the stack.



```
int eval(void)
    {
            precedence token;
            char symbol;
            int opl,op2, n=0;
            int top= -1;
            token = getToken(&symbol, &n);
            while(token! = eos)
                    {
                            if (token == operand)
                                    push(symbol-'0'); /* stack insert */
                            else {
                                    op2 = pop(); /* stack delete */
                                    opl = pop();
                            switch(token) {
                                                    case plus:      push(opl+op2);
                                                                    break;
                                                    case minus:     push(opl-op2);
                                                                    break;
                                                    case times:     push(opl*op2);
                                                                    break;
                                                    case divide:    push(opl/op2);
                                                                    break;
                                                    case mod:       push(opl%op2);
                                            }
                                    }
                            token = getToken(&symbol, &n);
                    }
            return pop(); /* return result */
    }
```

Program: Function to evaluate a postfix expression

```
precedence getToken(char  *symbol, int *n)
        {
                *symbol =
                expr[(*n)++]; switch
                (*symbol)
                        {
                                case '(' : return lparen;
                                case ')' : return rparen;
                                case '+' : return plus;
                                case '-' : return minus;
                                case '/' : return divide;
                                case '*' : return times;
                                case '%' : return mod;
                                case ' ' : return eos;
                                default: return operand;
                        }
        }
```

Program: Function to get a token from the input string

- The function **eval ( )** contains the code to evaluate a postfix expression. Since an operand (symbol) is initially a character, convert it into a single digit integer.
- To convert use the statement, **symbol-'0'**. The statement takes the ASCII value of **symbol** and subtracts the ASCII value of '0', which is 48, from it. For example, suppose **symbol = '1.** The character '1' has an ASCII value of 49. Therefore, the statement **symbol-'0'** produces as result the number 1.
- The function **getToken( )**, obtain tokens from the expression string. If the token is an operand, convert it to a number and add it to the stack. Otherwise remove two operands from the stack, perform the specified operation, and place the result back on the stack. When the end of expression is reached, remove the result from the stack.