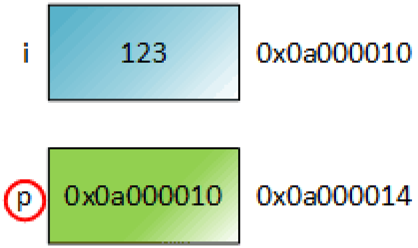
DATA STRUCTURES AND APPLICATIONS

Module 1_2
Introduction

Dr. Pushpalatha K
Associate Professor
Sahyadri College of Engineering & Management

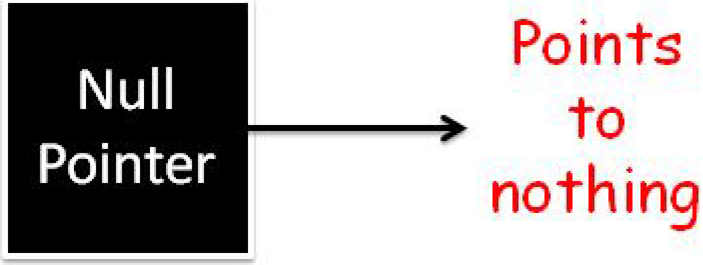
Pointers and Dynamic Memory Allocation

- **Pointers**
  - Fundamentals of C language
  - For any type in T in C, there is a corresponding type pointer to T
  - The actual value of a pointer type is an address of memory
  - Operators used with pointer type
    - & the addressing operator
    - * the dereferencing (indirection) operator
  - Eg:
    - int i, *p;
      - i is an integer values
      - p is a pointer to an integer
    - p=&i;
      - The address of i is stored in p
    - Then,
      - i=123 is same as *p=123;
  - Size of a pointer is different for different data types.



➢ Null Pointer

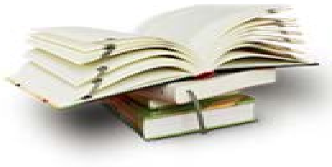
- The null pointer points to no object or function.
- The null pointer is represented by the integer 0.
- The null pointer can be used in relational expression, where it is interpreted as false.
  - Ex: if (p = = NULL) or if (!p)



Pointers Can Be Dangerous

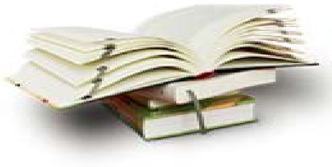
- **Pointer can be dangerous**
  - when an attempt is made to access an area of memory that is either out of range of program or that does not contain a pointer reference to a legitimate object.

```
main ()
{
	int *p;
	int pa = 10;
	p = &pa;
	printf(“%d”, *p); //output = 10;
	printf(“%d”, *(p+1));  //accessing memory which is out of range
}
```



- Set all pointers to NULL when they are not actually pointing to an object.
  - This prevents the access to an area of memory that is either out of or that does not contain a pointer reference to a legitimate object.
  - on some computer it may return null and permitting execution to continue
  - On other computers it may return the result stored in location zero, so it may produce a serious error
- use explicit type casts when converting between pointer types.

```
iptr= malloc (sizeof (int) ) ;
fptr = (float * ) iptr;
        / * casts an int pointer to a float pointer * /
```



- In some system, pointers have the same size as type int, since int is the default type specifier, some programmers omit the return type when defining a function.
- The return type defaults to int which can later be interpreted as a pointer.
  - Avoided by defining explicit types for functions.

➢ Dynamic memory allocation

- C provides **heap** to allocate the memory dynamically
  - The function **malloc** is used to allocate the required amount of memory
    - If there is insufficient memory to make the allocation, the returned value is NULL.

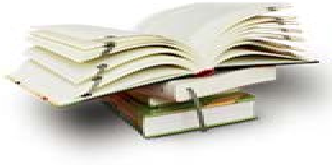
- Syntax:

```
data_type *x;

x= (data_type *) malloc(size);
```

Where, x is a pointer variable of data_type , size is the number of bytes

- Ex:

```
int *ptr;
ptr = (int *) malloc(100*sizeof(int));
```



- The function free is used to return the memory to system
  - Ex:

```
int i, *iptr;
float f, *fptr;
iptr=(int *) malloc(sizeof(int));
pptr=(float *) malloc(sizeof(float));
*iptr=100;
*fptr=3.1417;
printf(“i=%d, f=%f”,*iptr,*fptr);
free (iptr);
free (fptr);
```

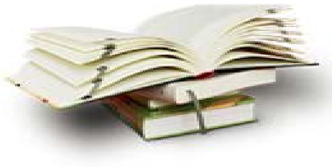
(int *) and (float *) are type cast expressions

- **malloc may fail for lack of sufficient memory**
- **Robust version of Program**

```
if ((iptr = (int *) malloc(sizeof (int) ))==NULL ||
     fptr=(float *) malloc (sizeof (float))) ==NULL)
      { printf ("Insufficient memory") ;
      exit (EXIT_FAILURE) ; }
```

OR

```
if  (iptr = malloc(sizeof(int) )) || (fptr= malloc (sizeof (float))))
{
      printf ("Insufficient memory") ; exit (EXIT_FAILURE) ;}
```



calloc

- calloc allocates user specified amount of memory and initializes the allocated memory to 0
- Returns a pointer to the start of the allocated memory.
  - If there is insufficient memory to make the allocation, the returned value is NULL.
- Syntax:

  data_type *x;

  x= (data_type *) calloc(n, size);

    - Where x is a pointer variable of type int , n is the number of block to be allocated , size is the number of bytes in each block

  - Eg:

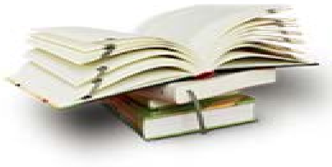
    int *x, n=10;

    x = (int *)calloc (n, sizeof (int)) ;

      - define a one-dimensional array of integers.
      - The capacity of this array is n=10 and x [0: n-1] (x [0, 9]) are initially 0

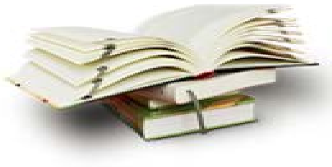
realloc

- function realloc resizes memory previously allocated by either malloc or calloc.
- Syntax: x= (data_type *) realloc(p, s);
  - the size of the memory block pointed at by p is changed to s as
- realloc doesnot change the contents of the first min{s, oldSize} bytes of the block.
  - When s>oldSize the additional s-oldSize have an unspecified value
  - When s<oldSize, the rightmost oldSize-s bytes of the old block are freed.
  - When realloc is able to do the resizing, it returns a pointer to the start of the new block
  - when it is unable to do the resizing, the old block is unchanged and the function returns the value NULL.



free( )

- Dynamically allocated memory with either malloc() or calloc() does not return on its own.
- The programmer must use free() explicitly to release space.
- Syntax:
  - free(ptr);
    - This statement cause the space in memory pointer by ptr to be deallocated

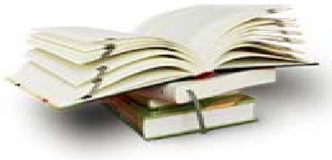


- Instead of invoking malloc in several places, it is convenient to define a macro that invokes malloc and exits when malloc fails.
- Macro Definition

```
#define  MALLOC(p,s) \
if ( ! ( (p) = malloc(s) ) ) { \
        printf ("Insufficient memory") ; \
        exit (EXIT_FAILURE) ; \
}
```

- Malloc can be invoked by

```
MALLOC (iptr, sizeof (int) ) ;
MALLOC(fptr, sizeof(float) ) ;
```



CALLOC Macro

```
#define CALLOC (p, n, s) \
if ( ! ((p) = calloc(n,s) ) ){\
        printf ("Insufficient memory") ;
        exit (EXIT_FAILURE) ; \
}
```

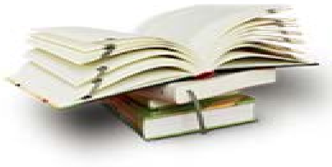
Macro REALLOC

```
define REALLOC (p, s) \
if (!((p)=realloc(p,s))) {\
        printf ("Insufficient memory") ; \
        exit (EXIT_FAILURE) ; \
}
```

DYNAMICALLY ALLOCATED ARRAYS

```
#define MAX_SZ 100
void main()
{
        int i, A[MAX_SZ], result=0;
        for(i=0;i< MAX_SZ; i++)
        {
                scanf("%d",&A[i]);
                result=result+A[i];
        }
        printf("The total=%d",result);
}
```

- **MAX_SZ is a constant**
  - Used to define array size i.e. 100



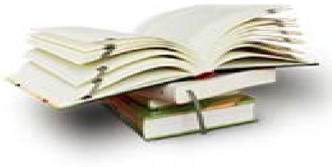
- When the size of an array cannot be determined, the array size can be allocated at runtime.

```
int i, n, *list;
printf ( "inter the number of numbers to generate .”);
scanf ("%d", &n);
if( n < 1 )
    {printf ( "Improper value of n\n") ;
     exit (EXIT_FAILURE) ; }
list= (int *)malloc( n * sizeof (int) ) ;
```

Note: the program fails only when n<1 or no sufficient memory

Example for malloc

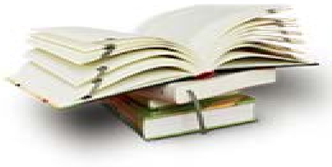
```
#include <stdio.h>
#include <stdlib.h>
int main()
{
        int n, i, *ptr, sum = 0;
        printf("Enter number of elements: ");
        scanf("%d", &n);
    //memory allocated using malloc
        ptr = (int*) malloc(num * sizeof(int));
        if(ptr == NULL)
            {
            printf("Error!         memory          not
           allocated.");
             exit(0);
             }
printf("Enter elements of array: ");
        for(i = 0; i < n; ++i)
        {
       scanf("%d", ptr + i);
      sum += *(ptr + i);
      }
        printf("Sum = %d", sum);
        free(ptr);
        return 0;
 }
```



Arrays using malloc macro

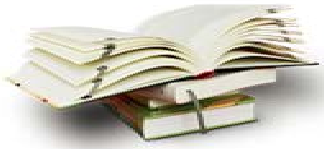
```
# include <stdlib.h>
 #define MALLOC(p,s) \
if (!((p)=malloc(s))) {\
printf("Insufficient memory");\
exit(0);\
}
```

```
main()
{
    int *ar1;
    int n=2,i;
    MALLOC(ar1,n*sizeof(int));
    for(i=0; i<n; i++)
        scanf("%d",ar1+i);
    for(i=0; i<n; i++)
        printf("%d",*(ar1+i));
    getch();
    return 0;
}
```



Example for calloc

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

int main()
{
    int num, i, *ptr, sum = 0;
    printf("Enter number of elements: ");
    scanf("%d", &num);
    ptr = (int*) calloc(num, sizeof(int));
if(ptr == NULL)
    {
        printf("Error! memory not allocated.");
         exit(0);
    }
    printf("Enter elements of array: ");
    for(i = 0; i < num; ++i)
    {
     scanf("%d", ptr + i);
     sum += *(ptr + i);
 }
    printf("Sum = %d", sum);
    free(ptr);
    return 0;
}
```

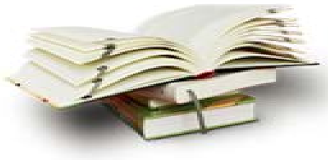


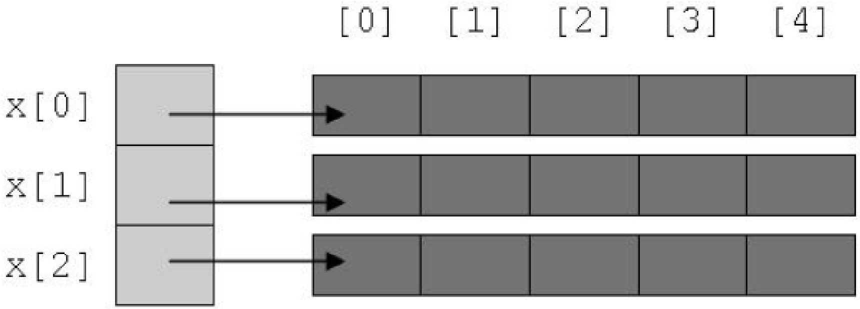
Example for realloc

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
   int *ptr, i , n1, n2;
   printf("Enter size of array: ");
   scanf("%d", &n1);
   ptr = (int*) malloc(n1 * sizeof(int));
   printf("Address of previously allocated memory: ");
   for(i = 0; i < n1; ++i)
        printf("%u\t",ptr + i);
   printf("\nEnter new size of array: ");
   scanf("%d", &n2);
   ptr = realloc(ptr, n2);
   for(i = 0; i < n2; ++i)
        printf("%u\t", ptr + i);
   return 0;
}
```

TWO DIMENSIONAL ARRAYS

- C uses the **array of arrays** representation to represent a multidirnensional array.
- A two dimcnsional array is represented as a one-dimensional array in which each elerment is itself a one-dimensional array
- Eg: a two dimension array **int x[3][5]**
  - Here a one dimensional array **x** of length **3** is created
  - Each element of **x** is a one-dimensional array whose length is **5**.





- The first element of array x[i[j] is found by first accessing the pointer in x [i].
- This pointer gives the address, in memory, of the zeroth element of row i of the array.
- Then by adding j*sign(int) to this pointer, the address of the [j]th element of row i is determined.

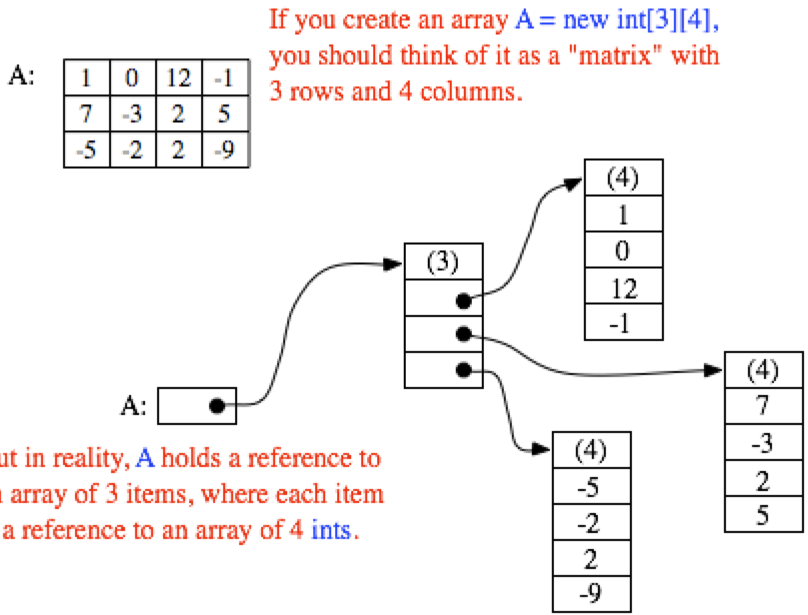
9	6	1	***(aiData+ 0)**
144	70	50	***(aiData+ 1)**
10	12	78	***(aiData+ 2)**

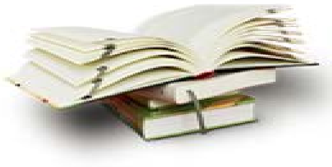
aiData[3][3]

9	***(*(aiData+ 0) + 0)**
6	***(*(aiData+ 0) + 1)**
1	***(*(aiData+ 0) + 2)**
144	***(*(aiData+ 1) + 0)**
70	***(*(aiData+ 1) + 1)**
50	***(*(aiData+ 1) + 2)**
10	***(*(aiData+ 2) + 0)**
12	***(*(aiData+ 2) + 1)**
78	***(*(aiData+ 2) + 2)**

Some points about pointers

- Array name (without an index) represents a pointer to the first object of the array
  - A is a pointer to the element A[0] in one-dimensional array
  - In two-dimensional array A[0] points to first object of first row
    - i.e. A[0] points to A[0][0]
- For any k points to the beginning of the kth row, i.e. A[k] is the address of A[k][0]
  - *A[k] accesses A[k][0]
- Adding 1 to array results in a pointer that points to the next array or row, i.e. A+1 is same as A[1]
- Adding 1 to row results in a pointer to the next element of that row i.e. A[0][1]





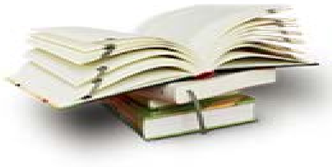
- Dereferencing a pointer, *(A + k), accesses A[k] or &A[k][0].
- Dereferencing *(*(A + k) + j) accesses the location A[k][j]

Pointer Equivalence for two dimensional arrays

Pointer	≈	≈
*A	A[0]	&A[0][0]
*A+1	A[0]+1	&A[0][1]
*A+J	A[0]+J	&A[0][J]
*(A+1)	A[1]	&A[1][0]
*(A+K)	A[K]	&A[K][0]
((A))	*A[0]	A[0][0]
((A+1))	*A[1]	A[1][0]
((A+K)+J)	*(A[K]+J)	A[K][J]

Two dimensional array creation in C

```
int  **A;
myArray = make2dArray (5, 10) ;
myArray[2][4]=6;

int** make2dArray(int rows, int cols)
{
int **x,i;
x=(int **)malloc(rows*sizeof(*x) ) ;
      for (i = 0; i < rows; i++)
      x[i]=(int *)malloc(cols* sizeof(**x) ) ;
      return x;
}
```

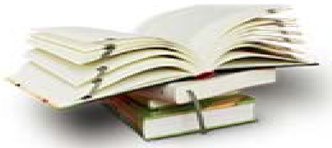


```
void main()
    {
    int i,j,rows=3,cols=4;
    int  **TA;
    TA=(int **)malloc(rows*sizeof(int)) ;
    for (i = 0; i < rows; i++)
        TA[i]=(int *)malloc(cols* sizeof(int) ) ;
    for (i = 0; i < rows; i++)
            for (j = 0; j < cols; j++)
                    TA[i][j]=i*10+j;        //OR *(*(TA+i)+j)= i*10+j;
    for (i = 0; i < rows; i++)
            for (j = 0; j < cols; j++)
                    printf(“%d”,TA[i][j]);
    }
```

STRUCTURES AND UNIONS

- **Structures**
  - Structures are an alternative way to group data
  - permits the data to vary in type
  - A structure is a collection of data items, where each item is identified as to its type and name.
  - Syntax:

```
struct struct_name
        { data_type member1;
        data_type member2;
        ………………………
        ………………………
        data_type membern;
} variable_name;
```



- **Ex:**

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

- **Creating objects**
  - **struct employee emp, emp1;**
- **Dot(.)operator to select a particular member of the structure**
  - **Eg:**

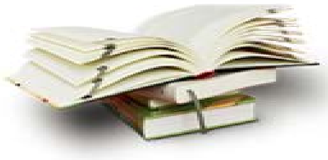
```
strcpy (emp. name, " james" ) ;
emp.age= 10;
emp.salary=35000;
```

- Using typedef statement our own structure data types can be created

```
typedef struct {
 char fname [10] ;
 float price;
 int qty;
} food;
```

- Then, the structure variables are created
  - food f1,f2;
- Usage:

```
it (strcmp(f1.name, f2.name))
        printf ("Same food items\n") .
else
        print f ("Different food items\n") ;
```



Structure Operations

- Function to check equality of structures

```
int Comparefood(food f1,food f2
)
{
    if (strcmp (f1.name, f2.name) )
            return FALSE;
     if (f1.price ! = f2.price)
              return FALSE;
    if (f1.qty= f2.qty)
              return FALSE;
     return TRUE;
    }
```

```
#define FALSE 0;
#define TRUE 1

strcpy (f1.name, f2.name) ;
f1.price= f2.price;
f1.qty= f2.qty;
If (Comparefood(f1,f2))
   printf( “Both food items are equal);
else
printf(“not equal);
```

- **Assigning one structure variable to another**
  - Ex: f1=f2 ->Invalid statement
  - Requires assignment functions
  - Includes following statements
    - strcpy (f1.name, f2.name) ;
    - f1.price= f2.price;
    - f1.qty= f2.qty;

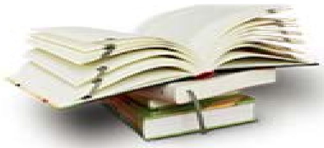
Structure within structure

Structure can be used as member of another structure.

```
typedef struct {
      int month;
      int day;
      int year;
      } date;
typedef struct {
      int rollno;
      char name[10];
      date dob;
      } student;
```

Usage:

```
student  s1,s2;
s1.dob.month=3;
s1.dob.day=13;
s1.dob.year=2003;
```



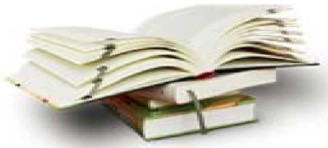
Nested Structure

- Structure written inside another structure is called as nesting of two structures.

```
struct Employee
{
  char ename[20];
  int ssn;
  float salary;
       struct date
          {
          int date;
          int month;
          int year;
          }doj;
}emp1;
```

Accessing Nested Members:

```
emp1.doj.month
emp1.doj.day
emp1.doj.year
```



Array of structures

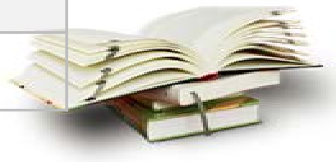
- In array of structure variables, each element of the array will represent a structure variable.

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

  – struct employee emp[4];

    - Each element of emp is of type employee

	Name	Age	Salary
emp[0]	Rama	18	55000
emp[1]	Bhima	21	50000
emp[2]	Soma	23	40000
emp[3]	Suma	25	60000



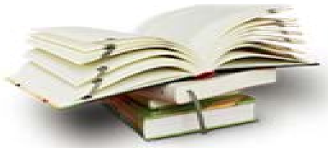
Structure as Function Arguments

- Structure can be passed to function through its object
- Like normal variable, structure variable(structure object) can be pass by value or by references / addresses
- Passing Structure by Value
  - Ex: void Display(struct Employee); //Function Definition

```
void Display(struct Employee E)
    {
            printf("\n\nEmployee Id : %d",E.Id);
            printf("\nEmployee Name : %s",E.Name);
            ...
    }
```

Passing Structure by Reference

```
 void Display(struct Employee*);   //Function Definition
void Display(struct Employee *E)
      {
                printf("\n\nEmployee Id : %d",E->Id);
                printf("\nEmployee Name : %s",E->Name);
                printf("\nEmployee Age : %d",E->Age);
                printf("\nEmployee Salary : %ld",E->Salary);
      }
```



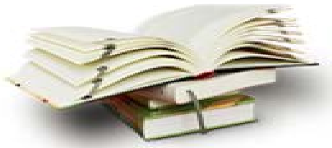
Function Returning Structure

```
Employee Input(); //Function Definition


Employee Input()
    {
        struct Employee E;
            printf("\nEnter Employee Details : ");
            scanf("%d",&E.Id);
             scanf("%s",&E.Name);
             return E;
    }
```

int arr[] = {1, 2, 3, 4, 5, 6, 7, 8};
unsigned int n = sizeof(arr)/sizeof(arr[0]);
fun(arr, n);
arr[]
return 0;

Passing array to function using call by value

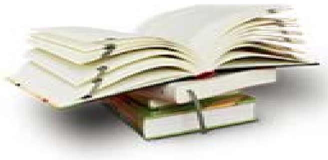
```
void fun(int arr[], unsigned int n) // SAME AS void fun(int *arr)
{
   int i;
   for (i=0; i<n; i++)
     printf("%d  ", arr[i]);
}
int main()
{
         int arr[] = {1, 2, 3, 4, 5, 6, 7, 8};
         unsigned int n = sizeof(arr)/sizeof(arr[0]);
         printf("Array size inside main() is %d", n);
         fun(arr);
         return 0;
}
```



return single dimensional array from function

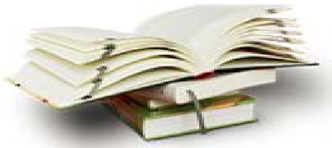
```c
#include <stdio.h>
int * getArray()
{
    int num[] = {1, 2, 3, 4, 5};
    int i;
    printf("Array inside function: ");
for (i = 0; i < 5; ++i)
    {       printf("%d\n", num[i]);     }
    return num;
}
```

```c
int main()
{
    int i;
    int * num;
    num = getArray();
  for (i = 0; i < 5; ++i)
    {
        printf("%d\n", num[i]);
    }

   return 0;
}
```



pass a single element of an array to function

```
#include <stdio.h>
void display(int age)
{
    printf("%d", age);
}

int main()
{
    int ageArray[] = { 2, 3, 4 };
    display(ageArray[2]); //Passing array element ageArray[2] only.
    return 0;
}
```



Unions

- A union is a special data type available in C that allows to store different data types in the same memory location.
- A union declaration 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.
- Syntax:

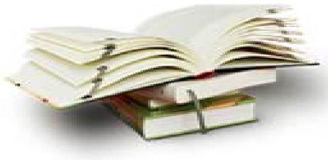
```
union union_name
 {
        data_type member1;
        data_type member2;
        ………………………
        ………………………
        data_type membern;
} variable_name;
```

- **Ex:**

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

- **Dot(.)operator to select a particular member of the union**
  - **Eg:**

```
strcpy (emp. name, “ james” ) ;
emp.age= 10;
emp.salary=35000;
```



Structures vs Union

Characteristics	Structure	Union
Memory Allocation	• Members of structure do not share memory. • A structure need separate memory space for all its members i.e. all the members have unique storage.	• A union shares the memory space among its members so no need to allocate memory to all the members. • Shared memory space is allocated i.e. equivalent to the size of member having largest memory.
Member Access	• Members of structure can be accessed individually at any time.	• At a time, only one member of union can be accessed.
Size	• Size of the structure is > to the sum of the each member's size.	• Size of union is equivalent to the size of the member having largest size.
Change in Value	• Change in the value of one member can not affect the other in structure.	• Change in the value of one member may affect the value of other member.

Structure vs union

Example:

Struct std

{

char c;

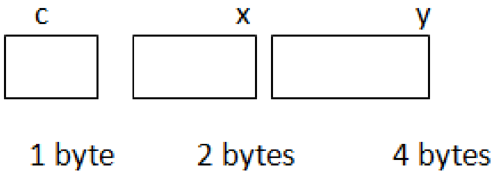
int x;

float y;

}s;

Memory location for s is:

1+2+4= 7 bytes



Example:

union std

{

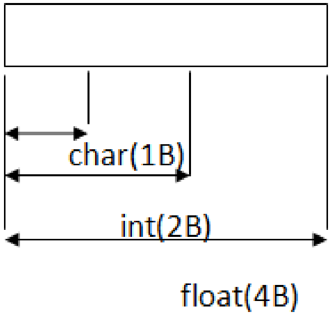
char c;

int x;

float y;}s;

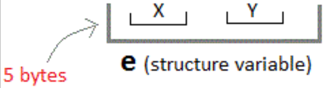
Memory location for s is 4 bytes (largest size among members, commonly used for all members)

4 bytes(float)



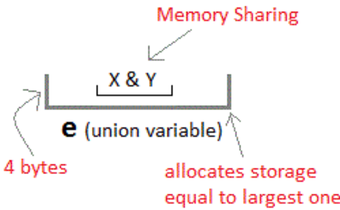
Structure

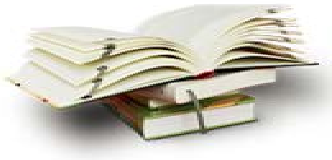
```
struct Emp
{
 char X ;     // size 1 byte
 float Y ;    // size 4 byte
} e ;
```



Unions

```
union Emp
{
 char X ;
 float Y ;
} e ;
```

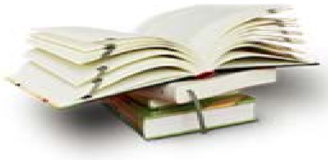




- **In the case of structure, all of its members can be accessed at any time.**
- **But, in the case of union, only one of its members can be accessed at a time**
  - **all other members will contain garbage values.**
  - **string member may have empty value**

Internal Implementation of structures

- **The structure variables are using the increasing address locations in the order specified in the structure definition**
- **But padding occurs within a structure to permit two consecutive components to be properly aligned within memory**
- **The size of an object of a struct or union is the amount of storage necessary to represent the largest component including the padding.**
- **Structures must begin and end of on same type of memory boundary**
  - **An even byte boundary is a multiple of 4,8,or 16**



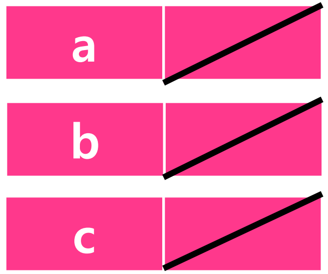
SELF-REFERENTIAL STRUCTURES

- **A self-referential structure is one in which one or more of its components is a pointer to itself.**
- **Self-referential structures require dynamic storage management routines (malloc and free) to explicitly obtain and release memory**
- **Ex:**

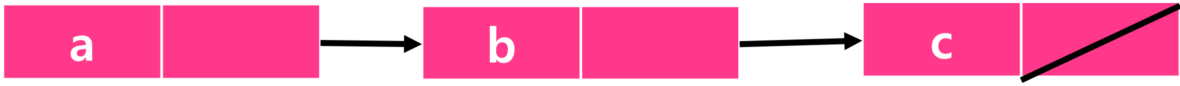
```
typedef struct {
        char data;
        struct list *link ;
} list;
```

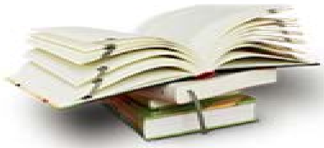
- Each instance of the structure list will have two components data and link.
  - Data: is a single character,
  - Link: link is a pointer to a list structure.
    - The value of link is either the address in memory of an instance of list or the null pointer

```
list item1, item2, item3;
item1.data = 'a';
item2.data = 'b';
item3.data = 'c';
item1.link = item2.1ink = item3.link = NULL;
```




```
item1.link = &item2;
item2.1ink = &item3;
```

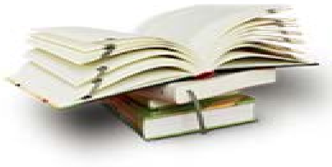




- Write a C program with an appropriate structure definition and variable declaration to store information about an employee using nested structure. Consider the following fields like Ename, Empid, DOJ(Date, month, year) and Salary (Basic, DA, HRA).

Polynomials

- ➢ Arrays are not only data structures in their own right.
- ➢ Most commonly found data structures: the ordered or linear list.
  - – Ex: Days of the week: (Sun,Mon,Tue....)
  - – Values in a deck of cards: (Ace,2,3,4,5,6,7)
  - – Years Switzerland fought in WorldWarII: ()
    - • An empty list is denoted as ().
- ➢ The other lists all contain items that are written in the form (item0,item1,.....item n-1)

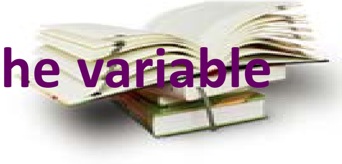


Operations on Ordered List

- **Finding** the length, n , of the list.
- **Reading** the items from left to right (or right to left).
- **Retrieving** the i'th element.
- **Storing** a new value into the i'th position.
- **Inserting** a new element at the position i , causing elements numbered i, $i+1$, …, n to become numbered $i+1$, $i+2$, …, $n+1$
- **Deleting** the element at position i , causing elements numbered $i+1$, …, n to become numbered i, $i+1$, …, $n-1$

Polynomials

- A polynomial is a sum of terms, where each term has a form axe, where x is the variable, a is the coefficient and e is the exponent.”
  - Ex:
    - $A(x) = 3x^{20} + 2x^{5} + 4$
    - $B(x) = x^{4} + 10x^{3} + 3x^{2} + 1$
- The largest (or leading) exponent of a polynomial is called its degree.
- Coefficients that are zero are not displayed.
  - The term with exponent equal to zero does not show the variable since x raised to a power of zero is 1.

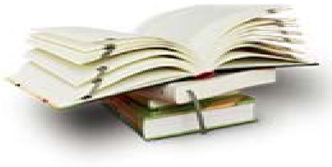


- **Assume there are two polynomials,**
  - $A(x) = \Sigma\, a_i x^i$ and $B(x) = \Sigma\, b_i x^i$ then:
    - $A(x) + B(x) = \Sigma\, (a_i + b_i)\, x^i$
    - $A(x).B(x) = \Sigma\, (a_i\, x^i.\ \Sigma\, (b_j\, x^j))$

- **Polynomial Representation**
  - Using typedef

```
#define MAX-DEGREE 101 /*Max degree of polynomial+1*/
typedef struct{
        int degree;
        float coef[MAX-DEGREE];
} polynomial;
```

- If a is type polynomial and $n <$ MAX-DEGREE, the polynomial $A(x) = \Sigma a_i x^i$ would be represented as
  - a.degree=n
  - $A.coef[i]=a_{n-l}$, $0 \leq i \leq n$
- In this representation, the coefficients are stored 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.coef [i] =0.
- This representation leads to very simple algorithms for most of the operations, so it wastes a lot of space.
  - If a.degree<< MAX-DEGREE or number of terms with nonzero coefficient is small, then most of the positions in a.coef[MAX_DEGREE] are not required



- To preserve space an alternate representation that uses only one global array, terms to store all polynomials.
- The C declarations needed are:

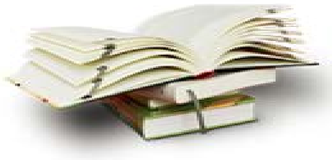
```
MAX_TERMS 100 /*size of terms array*/
typedef struct{
        float coef;
        int expon;
} polynomial;
polynomial terms[MAX-TERMS];
int avail = 0;
```

Representation of Polynomials

Two polynomials: $A(x) = 2x^{1000} + 1$ and $B(x) = x^4 + 10x^3 + 3x^2 + 1$

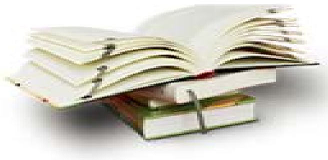
	startA	finishA	startB			finishB	avail
coef	2	1	1	10	3	1	
exp	1000	0	4	3	2	0	
	0	1	2	3	4	5	6

- The index of the first term of A and B is given by startA and startB
- finishA and finishB give the index of the last term of A and B.
- The index of the next free location in the array is given by avail.
- startA=0, finishA=1, startB=2, finishB=5, & avail=6.



Polynomial Addition

- D =A + B where A and B are two polynimials
- To produce D(x), padd( ) is used to add A(x) and B(x) term by term.
- Starting at position avail, attach( ) which places the terms of D into the array, *terms*.
- If there is not enough space in terms to accommodate D, an error message is printed to the standard error device & exits the program with an error condition

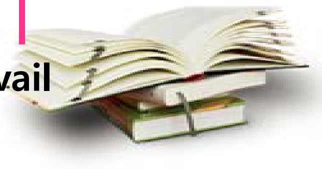


$A(x) = 7x^4 + 3x^3 + 4x + 5$ and $B(x) = 8x^4 + 4x^3 + 4x^2 + 6x + 8$

	0	1	2	3	4	5	6	7	8	9	10
Coefficient	7	3	4	5	8	4	4	6	8		
Exponent	4	3	1	0	4	3	2	1	0		
	StartA			EndA	StartB				EndB	Avail	

D=A+B

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Coef	7	3	4	5	8	4	4	6	8	15	3	4	10	13	
Exp	4	3	1	0	4	3	2	1	0	4	3	2	1	0	
	StartA			EndA	StartB				EndB	StartD				EndD	Avail



ALGORITHM

1. Declare structure polynomial

```
struct polynomial
{
    int expo
    int coef
}
```

2. Read number of terms of the first polynomial, no of terms
3. i=0, avail=0
4. if i<noofterms, repeat steps 5 and 6
5. Read terms[avail].coef and terms[avail].expo
6. i=i+1, avail++

7. Read number of terms of the second polynomial, noofterms
8. i=0
9. if i<noofterms, repeat steps 11 and 12
10. Read terms[avail].coef and terms[avail].expo
11. i=i+1, avail++
12. startD=avail
13. while(startA<=finishA)&&(startB<=finishB), repeat steps 14 to 17
14. if terms[startA].expo>terms[startB].expo

    Call attach(terms[startA].coeff,terms[startA].expo)

    startA++

```
15. if terms[startA].expo=terms[startB].expo
      coefficient= terms[startA].coeff+terms[startB].coeff
      if coefficient!=0
      Call attach(coefficient,terms[startB].expo)
      startA++;
      startB++;
16. if terms[startA].expo<terms[startB].expo
      Call attach(terms[startB].coeff,terms[startB].expo)
      startB++;
```

15. for(;startA<=finishA;startA++)

Call attach(terms[startA].coeff,terms[startA].expo)

for(;startB<=finishB;startB++)

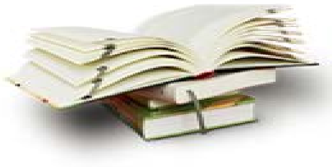
Call attach(terms[startB].coeff,terms[startB].expo)

16. *finishD=avail-1

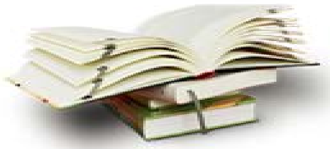
17. Stop

Function to add two polynomials

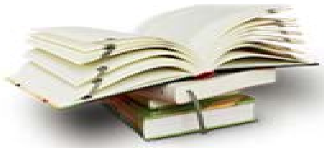
```
void padd(int startA, int finishA, int startB, int finishB, int
    *startD,int *finishD)
{
    float coefficient;
    *startD = avail;
    while (startA <= finishA && startB <= finishB)
    switch(COMPARE(terms[startA].expon, terms[startB].expon))
    {
    case -1: /* a expon < b expon */
    attach (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++;
}
```

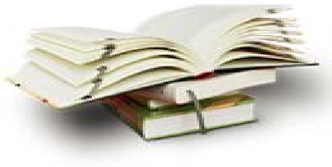


```
/* add in remaining terms of A(x) */
for(; startA <= finishA; startA++)
attach (terms[startA].coef, terms[startA].expon);
/* add in remaining terms of B(x) */
for( ; startB <= finishB; startB++)
attach (terms[startB].coef, terms[startB].expon);
*finishD = avail-1;
```



Function to add new term

```
void attach(float coefficient, int exponent)
{ /* add a new term to the polynomial */
    if (avail >= MAX-TERMS)
    {
        Printf(“Error…");
        exit(0);
    }
    terms[avail].coef = coefficient;
    terms[avail++].expon = exponent;
}
```



```
int compare(int exp1,int exp2)
{
    if(exp1>exp2)
        return 1;
    else if(exp1<exp2)
        return -1;
    else
        return 0;
}
```

Analysis of padd( )

- The number of non-zero terms in A and B is the most important factors in analyzing the time complexity.
- Let m and n be the number of non-zero terms in A and B
- If m >0 and n > 0, the while loop is entered.
  - Each iteration of the loop requires O(1) time.
  - At each iteration, the value of startA or startB or both is incremented. The iteration terminates when either startA or startB exceeds finishA or finishB.
- The number of iterations is bounded by m + n -1
- The time for the remaining two for loops is bounded by O(n + m) because we cannot iterate the first loop more than m times and the second more than n times. So, the asymptotic computing time of this algorithm is **O(n+m).**