

UNIT3

Structures (Records)

Array of structures (records)

Database implementation using array of records.

Dynamic memory allocation and deallocation.

Dynamically allocated single and
multi-dimensional arrays.

polynomial representation.

STRUCTURES

INTRODUCTION

Memory is allocated for the structure where structure is same as that of records. It stores related information about an entity.

Structure is basically a user defined data type that can store related information (even of different data types) together.

A structure is declared using the keyword struct followed by a structure name. All the variables of the structures are declared within the structure. A structure type is defined by using the given syntax.

struct struct-name

```
{      data_type var-name;
      data_type var-name;
      ...
};
```

struct student

```
{      int r_no;
      char name[20];
      char course[20];
      float fees;
} stud2;
```

The structure definition does not allocate any memory. It just gives a template that conveys to the C compiler how the structure is laid out in memory and gives details of the member names. When we declare a variable of the structure. For ex, we can define a variable of student by writing

```
struct student stud1;
```

```
Typedef int int1;  
int1 a;  
a=15;
```

```
Typedef float fl;  
fl b;
```

TYPDEF DECLARATIONS

When we precede a struct name with typedef keyword, then the struct becomes a new type. It is used to make the construct shorter with more meaningful names for types already defined by C or for types that you have declared. With a typedef declaration, becomes a synonym for the type.

For example, writing

typedef struct student

```
{  
    int r_no;  
    char name[20];  
    char course[20];  
    float fees;  
};
```

Now that you have preceded the structure's name with the keyword typedef, the student becomes a new data type. Therefore, now you can straight away declare variables of this new data type as you declare variables of type int, float, char, double, etc. to declare a variable of structure student you will just write,

student stud1;

INITIALIZATION OF STRUCTURES

Initializing a structure means assigning some constants to the members of the structure.

When the user does not explicitly initialize the structure then C automatically does that. For int and float members, the values are initialized to zero and char and string members are initialized to the '\0' by default.

The initializers are enclosed in braces and are separated by commas. Note that initializers match their corresponding types in the structure definition.

The general syntax to initialize a structure variable is given as follows.

struct struct_name

```
{      data_type member_name1;  
      data_type member_name2;  
      data_type member_name3;
```

.....

```
}struct_var = {constant1, constant2, constant 3,...};
```

OR

struct struct_name

```
{      data_type member_name1;  
      data_type member_name2;  
      data_type member_name3;}
```

.....

```
struct struct_name struct_var = {constant1, constant2, ....};
```

```
struct student stud1 = {01, "Rahul", "BCA", 45000};
```

ACCESSING THE MEMBERS OF A STRUCTURE

Each member of a structure can be used just like a normal variable, but its name will be a bit longer. A structure member variable is generally accessed using a '.' (dot operator).

The syntax of accessing a structure a member of a structure is:

struct_var.member_name

For ex, to assign value to the individual data members of the structure variable Rahul, we may write,

```
stud1.r_no = 01;  
strcpy(stud1.name, "Rahul");  
stud1.course = "BCA";  
stud1.fees = 45000;
```

We can assign a structure to another structure of the same type. For ex, if we have two structure variables stu1 and stud2 of type struct student given as

```
struct student stud1 = {01, "Rahul", "BCA", 45000};  
struct student stud2;
```

Then to assign one structure variable to another we will write,

```
stud2 = stud1;
```

Write a program using structures to read and display the information about a student

```
#include<stdio.h>

int main()
{
    struct student
    {
        int roll_no;
        char name[80];
        float fees;
        char DOB[80];
    };
    struct student stud1;
    printf("\n Enter the roll number : ");
    scanf("%d", &stud1.roll_no);
    printf("\n Enter the name : ");
    scanf("%s", stud1.name);
    printf("\n Enter the fees : ");
    scanf("%f", &stud1.fees);
    printf("\n Enter the DOB : ");
    scanf("%s", stud1.DOB);
    printf("\n *****STUDENT'S DETAILS *****");
    printf("\n ROLL No. = %d", stud1.roll_no);
    printf("\n NAME. = %s", stud1.name);
    printf("\n ROLL No. = %f", stud1.fees);
    printf("\n ROLL No. = %s", stud1.DOB);
}
```


NESTED STRUCTURES

A structure can be placed within another structure. That is, a structure may contain another structure as its member. Such a structure that contains another structure as its member is called a nested structure.

typedef struct

```
{          char first_name[20];  
          char mid_name[20];  
          char last_name[20];
```

```
}NAME;
```

typedef struct

```
{      int dd;  
      int mm;  
      int yy;
```

```
}DATE;
```

struct student stud1;

```
    stud1.name.first_name = "joy";
```

```
    stud1.name.mid_name = "Raj";
```

```
stud1.name.last_name = "Thareja";
```

```
stud1.course = "BCA";
```

```
stud1.Date.dd = 15;
```

```
stud1.DOB.mm = 09;
```

```
stud1.DOB.yy = 2000;
```

```
stud1.fees = 75000;
```

Write a program to read and display information of a student using structure within a structure

```
#include<stdio.h>
int main()
{
    struct DOB
    {
        int day;
        int month;
        int year;
    };
    struct student
    {
        int roll_no;
        char name[100];
        float fees;
        struct DOB date;
    };
    struct student stud1;
    printf("\n Enter the roll number : ");
    scanf("%d", &stud1.roll_no);
    printf("\n Enter the name : ");
    scanf("%s", stud1.name);
    printf("\n Enter the fees : ");
    scanf("%f", &stud1.fees);
    printf("\n Enter the DOB : ");
    scanf("%d %d %d", &stud1.date.day, &stud1.date.month, &stud1.date.year);
    printf("\n *****STUDENT'S DETAILS *****");
    printf("\n ROLL No. = %d", stud1.roll_no);
    printf("\n NAME. = %s", stud1.name);
    printf("\n FEES. = %f", stud1.fees);
    printf("\n DOB = %d - %d - %d", stud1.date.day, stud1.date.month, stud1.date.year);
}
```

ARRAYS OF STRUCTURES

The general syntax for declaring an array of structure can be given as,

```
struct struct_name struct_var[index];
```

```
struct student stud[30];
```

Now, to assign values to the i^{th} student of the class, we will write,

```
stud[i].r_no = 09;
```

```
stud[i].name = "RASHI";
```

```
stud[i].course = "MCA";
```

```
stud[i].fees = 60000;
```

Write a program to read and display information of all the students in the class.

```
#include<stdio.h>
int main()
{
    struct student
    {
        int roll_no;
        char name[80];
        float fees;
        char DOB[80];
    };
    struct student stud[50];
    int n, i;
    printf("\n Enter the number of students : ");
    scanf("%d", &n);
    for(i=0;i<n;i++)
    {
        printf("\n Enter the roll number : ");
        scanf("%d", &stud[i].roll_no);
        printf("\n Enter the name : ");
        scanf("%s", stud[i].name);
        printf("\n Enter the fees : ");
        scanf("%f", stud[i].fees);
        printf("\n Enter the DOB : ");
        scanf("%s", stud[i].DOB);
    }
    for(i=0;i<n;i++)
    {
        printf("\n *****DETAILS OF %dth STUDENT*****", i+1);
        printf("\n ROLL No. = %d", stud[i].roll_no);
        printf("\n NAME. = %s", stud[i].name);
        printf("\n ROLL No. = %f", stud[i].fees);
        printf("\n ROLL No. = %s", stud[i].DOB);
    }
}
```

```
Struct Book
{ int issnno;
  Bookname
  Authorname
  Pages
  Type
  Category } bookdata[50];
```

```
Struct book bookdata2[50];
```

Passing Individual Structure Members to a Function

To pass any individual member of the structure to a function we must use the direct selection operator to refer to the individual members for the actual parameters. The called program does not know if the two variables are ordinary variables or structure members.

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

```
typedef struct
```

```
{
```

```
    int x;
```

```
    int y;
```

```
}POINT;
```

```
void display(int, int);
```

```
main()
```

```
{
```

```
    POINT p1 = {2, 3};
```

```
    display(p1.x, p1.y);
```

```
    return 0;
```

```
}
```

```
void display( int a, int b)
```

```
{
```

```
    printf("%d %d", a, b);
```

```
}
```

Add(int a, int b)

add(10,14);

PASSING A STRUCTURE TO A FUNCTION

When a structure is passed as an argument, it is passed using call by value method. That is a copy of each member of the structure is made. No doubt, this is a very inefficient method especially when the structure is very big or the function is called frequently. Therefore, in such a situation passing and working with pointers may be more efficient.

The general syntax for passing a structure to a function and returning a structure can be given as,
`struct struct_name func_name(struct struct_name struct_var);`

The code given below passes a structure to the function using call-by-value method.

```
#include<stdio.h>
typedef struct
{
    int x;
    int y;
}POINT;
void display(POINT);
main()
{
    POINT p1 = {2, 3};
    display(p1);
    return 0;
}
void display( POINT p)
{
    printf("%d %d", p.x, p.y);
}
```


PASSING STRUCTURES THROUGH POINTERS

C allows to create a pointer to a structure. Like in other cases, a pointer to a structure is never itself a structure, but merely a variable that holds the address of a structure. The syntax to declare a pointer to a structure can be given as

```
struct struct_name
{
    data_type member_name1;
    data_type member_name2;
    .....
}*ptr;
```

OR

```
struct struct_name *ptr;
```

For our student structure we can declare a pointer variable by writing

```
struct student *ptr_stud, stud;
```

The next step is to assign the address of stud to the pointer using the address operator (&). So to assign the address, we will write

```
ptr_stud = &stud;
```

To access the members of the structure, one way is to write

```
/* get the structure, then select a member */
```

```
(*ptr_stud).roll_no;
```

An alternative to the above statement can be used by using 'pointing-to' operator (->) as shown below.

```
/* the roll_no in the structure ptr_stud points to */
```

```
ptr_stud->roll_no = 01;
```

```
int a
```

```
int *a
```

```
struct student s1;
```

```
struct student *s1;
```

Write a program using pointer to structure to initialize the members in the structure.

```
#include<stdio.h>
struct student
{
    int r_no;
    char name[20];
    char course[20];
    float fees;
};
main()
{
    struct student stud1, *ptr_stud1;
    ptr_stud1 = &stud1;
    ptr_stud1->r_no = 01;
    strcpy(ptr_stud1->name, "Rahul");
    strcpy(ptr_stud1->course, "BCA");
    ptr_stud1->fees = 45000;
    printf("\n DETAILS OF STUDENT");
    printf("\n -----");
    printf("\n ROLL NUMBER = %d", ptr_stud1->r_no);
    printf("\n NAME = ", puts(ptr_stud1->name));
    printf("\n COURSE = ", puts(ptr_stud1->course));
    printf("\n FEES = %f", ptr_stud1->fees);
}
```

SELF REFERENTIAL STRUCTURES

Self referential structures are those structures that contain a reference to data of its same type. That is, a self referential structure in addition to other data contains a pointer to a data that is of the same type as that of the structure. For example, consider the structure node given below.

```
struct node
{
    int val;
    struct node *next;
};
```

Here the structure node will contain two types of data- an integer val and next that is a pointer to a node. You must be wondering why do we need such a structure? Actually, self-referential structure is the foundation of other data structures.

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

```
int main()
```

```
{          struct employees
```

```
{
```

```
    int id;
```

```
    char ename[80];
```

```
    float salary;
```

```
    int age;
```

```
};
```

```
struct employees e[10];
```

```
int i;
```

```
printf("\n Enter the details of employees : ");
```

```
    for(i=0;i<10;i++)
```

```
{          printf("\n Enter the id : ");
```

```
    scanf("%d", &e[i].id);
```

```
    printf("\n Enter the name : ");
```

```
    scanf("%s", stud[i].name);
```

```
    printf("\n Enter the fees : ");
```

```
    scanf("%f", stud[i].fees);
```

```
    printf("\n Enter the DOB : ");
```

```
    scanf("%s", stud[i].DOB);
```

```
}
```

```
for(i=0;i<10;i++)
```

```
{          printf("\n *****DETAILS OF %dth STUDENT*****", i+1);
```

```
    printf("\n ROLL No. = %d", stud[i].roll_no);
```

```
    printf("\n NAME. = %s", stud[i].name);
```

```
    printf("\n ROLL No. = %f", stud[i].fees);
```

```
    printf("\n ROLL No. = %s", stud[i].DOB);
```

```
}
```

Write a c prog to : a) create a structure keeping details of employees of a company.

Structure members are:

empid int, Ename ,Salary, Age

b) array of structures to store information of 10 employees

c) Display information of all these employees

d) Compute total salary paid to all employees and display it

e) Compute average age of employees and display it.

```
totsal =0;
for(i=0;i<10;i++)
{
Totsal= totsalsal+e[i].salary;
avgage= avgage+e[i].age;
}
```

create one structure variable of employee -
emp1 and one pointer variable - emp2 to
structure employee

access data of emp1 using emp2


```
main(){
Struct employee{
    int eid;
    char ename[20];
    float sal;
    int age;
}
Struct employee emp1, *emp2;
Printf(" enter details of employee");
Scanf("%d %s%f %d", &emp1.eid,emp1.ename, emp1.sal,&emp1.age);
Emp2= &emp1;
Printf("details of employee are:");
Printf("(“%d %s%f %d”, emp2->eid,emp2->ename, emp2-> sal,emp2-
>age);
}
```

Dynamic Memory Allocation

Problem with Arrays

- Sometimes
 - Amount of data cannot be predicted beforehand
 - Number of data items keeps changing during program execution
- Example: Search for an element in an array of N elements
- One solution: find the maximum possible value of N and allocate an array of N elements
 - Wasteful of memory space, as N may be much smaller in some executions
 - Example: maximum value of N may be 10,000, but a particular run may need to search only among 100 elements
 - Using array of size 10,000 always wastes memory in most cases

Better Solution

- Dynamic memory allocation
 - Know how much memory is needed after the program is run
 - Example: ask the user to enter from keyboard
 - Dynamically allocate only the amount of memory needed
- C provides functions to dynamically allocate memory
 - malloc, calloc, realloc

Memory Allocation Functions

- `malloc`
 - Allocates requested number of bytes and returns a pointer to the first byte of the allocated space
- `calloc`
 - Allocates space for an array of elements, initializes them to zero and then returns a pointer to the memory.
- `free`
 - Frees previously allocated space.
- `realloc`
 - Modifies the size of previously allocated space.

Allocating a Block of Memory

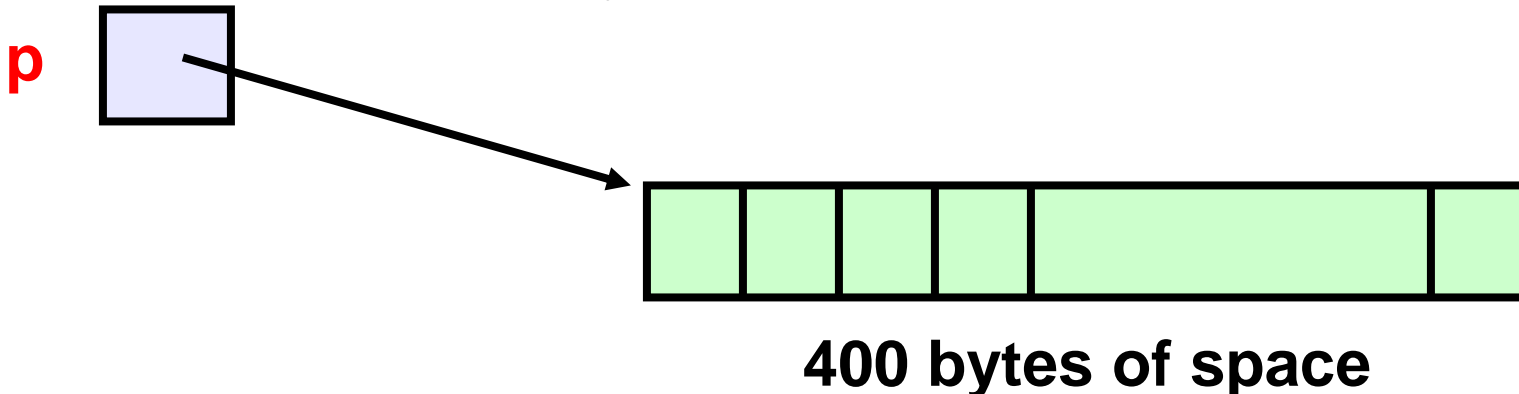
- A block of memory can be allocated using the function `malloc`
 - Reserves a block of memory of specified size and returns a pointer of type `void`
 - The return pointer can be type-casted to any pointer type
- General format:

```
type *p;  
p = (type *) malloc (byte_size);
```

Example

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

- A memory space equivalent to **100 times the size of an int** bytes is reserved
- The address of the first byte of the allocated memory is assigned to the pointer **p** of type **int**



Contd.

- `cptr = (char *) malloc (20);`

Allocates 20 bytes of space for the pointer `cptr` of type `char`

- `sptr = (struct stud *) malloc(10*sizeof(struct stud));`

Allocates space for a structure array of 10 elements. `sptr` points to a structure element of type `struct stud`

Always use sizeof operator to find number of bytes for a data type, as it can vary from machine to machine

Points to Note

- `malloc` always allocates a block of contiguous bytes
 - The allocation can fail if sufficient contiguous memory space is not available
 - If it fails, `malloc` returns `NULL`

```
if ((p = (int *) malloc(100 * sizeof(int))) == NULL)
{
    printf ("\n Memory cannot be allocated");
    exit();
}
```

Can we allocate only arrays?

- malloc can be used to allocate memory for single variables also
 - `p = (int *) malloc (sizeof(int));`
 - Allocates space for a single int, which can be accessed as `*p`
- Single variable allocations are just special case of array allocations
 - Array with only one element

write c program to allocate memory to one integer variable dynamically , print its value and then release memory occupied by that variable .

```
#include<stdlib.h>
main()
{
    int *p;
    p=(int *) malloc( sizeof(int));
    *p=10;
    printf("%d", *p); //    *(&p)
    free(p);
}
```

Using the malloc'd Array

- Once the memory is allocated, it can be used with pointers, or with array notation
- Example:

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

The n integers allocated can be accessed as *p, *(p+1), *(p+2), ..., *(p+n-1) or just as p[0], p[1], p[2], ..., p[n-1]

Example

```
int main()
{
    int i,N;
    float *height;
    float sum=0,avg;

    printf("Input no. of students\n");
    scanf("%d", &N);

    height = (float *)
        malloc(N * sizeof(float));
```

```
    printf("Input heights for %d
students \n",N);
    for (i=0; i<N; i++)
        scanf ("%f", &height[i]);

    for(i=0;i<N;i++)
        sum += height[i];

    avg = sum / (float) N;

    printf("Average height = %f \n",
        avg);

    free (height);
    return 0;
}
```

Releasing the Allocated Space:

`free`

- An allocated block can be returned to the system for future use by using the `free` function
- General syntax:
`free (ptr);`
where `ptr` is a pointer to a memory block which has been previously created using `malloc`
- Note that no size needs to be mentioned for the allocated block, the system remembers it for each pointer returned

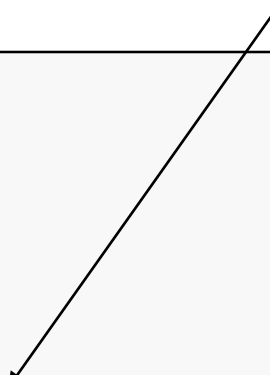
malloc()-ing array of structures

```
typedef struct{
    char name[20];
    int roll;
    float SGPA[8], CGPA;
} person;

void main()
{
    person *student;
    int i,j,n;
    scanf("%d", &n);
    student = (person *)malloc(n*sizeof(person));
    for (i=0; i<n; i++) {
        scanf("%s", student[i].name);
        scanf("%d", &student[i].roll);
        for(j=0;j<8;j++) scanf("%f", &student[i].SGPA[j]);
        scanf("%f", &student[i].CGPA);
    }
}
```

Static array of pointers

```
#define N 20
#define M 10
int main()
{
    char word[N], *w[M];
    int i, n;
    scanf("%d",&n);
    for (i=0; i<n; ++i) {
        scanf("%s", word);
        w[i] = (char *) malloc ((strlen(word)+1)*sizeof(char));
        strcpy (w[i], word) ;
    }
    for (i=0; i<n; i++) printf("w[%d] = %s \n",i,w[i]);
    return 0;
}
```



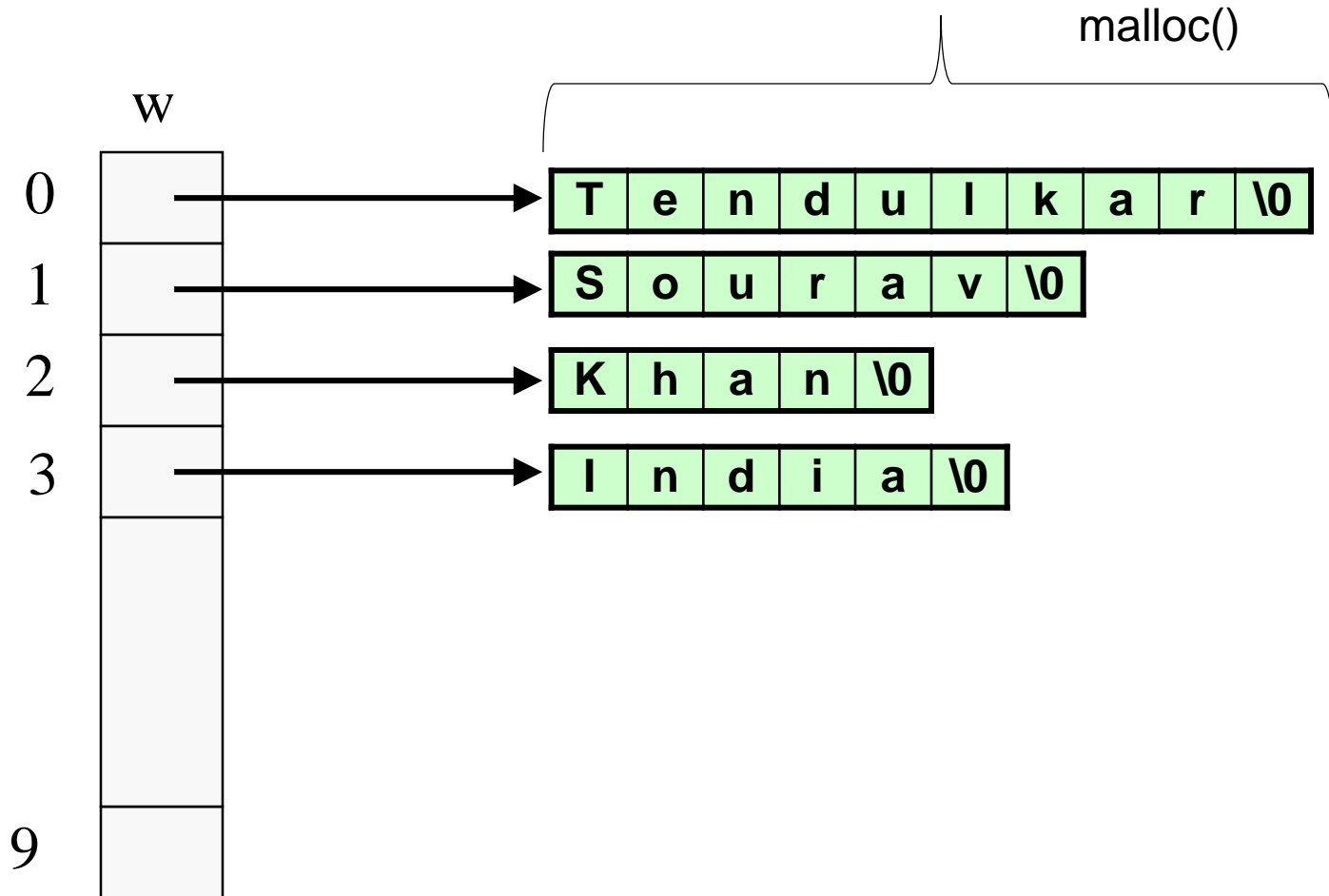
Static array of pointers

```
#define N 20
#define M 10
int main()
{
    char word[N], *w[M];
    int i, n;
    scanf("%d",&n);
    for (i=0; i<n; ++i) {
        scanf("%s", word);
        w[i] = (char *) malloc ((strlen(word)+1)*sizeof(char));
        strcpy (w[i], word) ;
    }
    for (i=0; i<n; i++) printf("w[%d] = %s \n",i,w[i]);
    return 0;
}
```

Output

```
4
Tendulkar
Sourav
Khan
India
w[0] = Tendulkar
w[1] = Sourav
w[2] = Khan
w[3] = India
```

How it will look like



Pointers to Pointers

- Pointers are also variables (storing addresses), so they have a memory location, so they also have an address
- Pointer to pointer – stores the address of a pointer variable

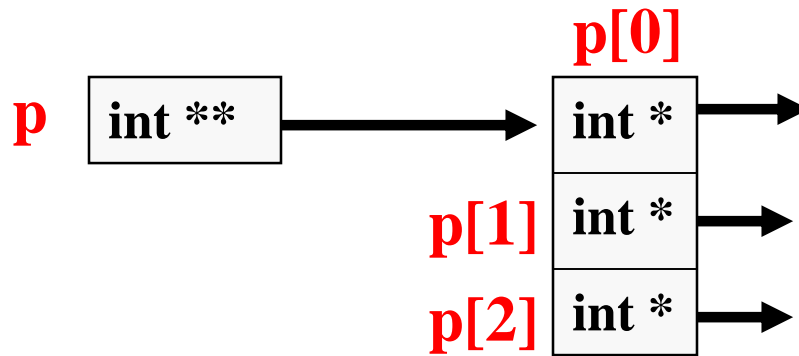
```
int x = 10, *p, **q;  
p = &x;  
q = &p;  
printf("%d %d %d", x, *p, *(*q));
```

will print 10 10 10 (since *q = p)

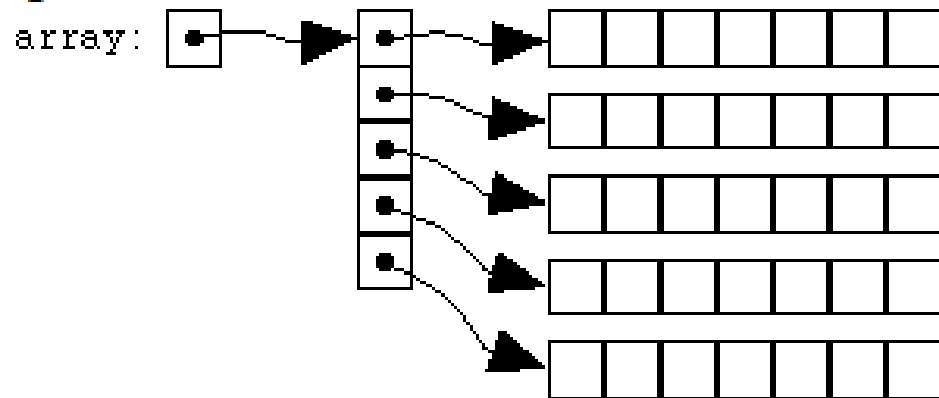
Allocating Pointer to Pointer

```
int **p;
```

```
p = (int **) malloc(3 * sizeof(int *));
```



2D array



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

```
int main()
```

```
{
```

```
    int **array;
```

```
    array = (int**) malloc(nrows * sizeof(int *));
```

```
    for(i = 0; i < nrows; i++)
```

```
    {
```

```
        array[i] = (int*)malloc(ncolumns * sizeof(int));
```

```
    }
```

```
{
```

```

#define ROW    3
#define COLUMN 4

int main()
{
    int i, j, count=0;

    // Allocating two dimensional dynamic array
    int **arr = (int **)malloc(ROW * sizeof(int *));

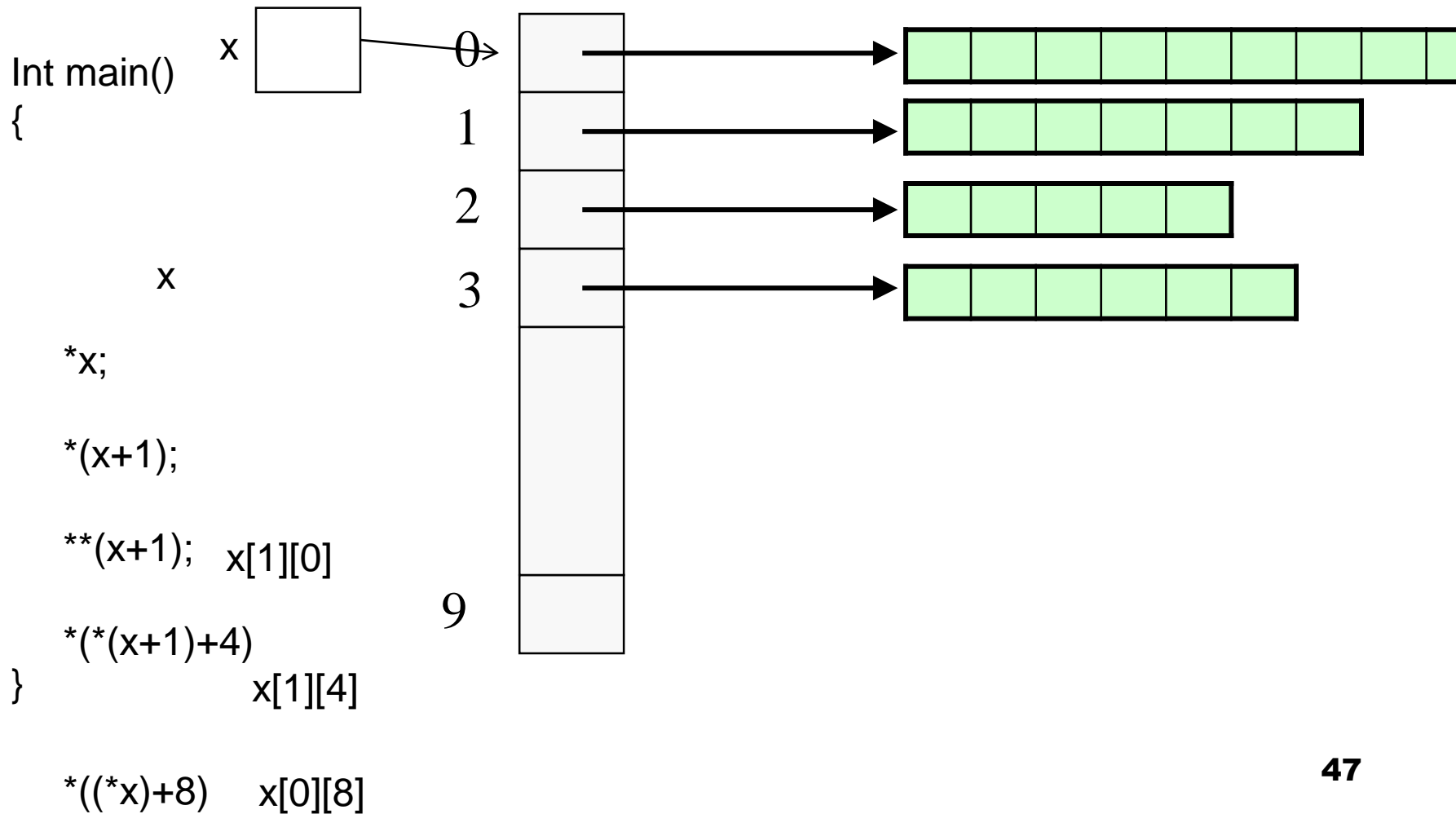
    for (i=0; i<ROW ; i++)
        arr[i] = (int *)malloc(COLUMN * sizeof(int));

    // Accessing two dimensional array
    for (i = 0; i <ROW; i++)
    {
        for (j = 0; j <COLUMN; j++)
        {
            arr[i][j] = ++count; // OR (*(arr+i)+j) = ++count
            printf("%d ", arr[i][j]);
        }
    }

    return 0;
}

```

2D array



Dynamic Allocation of 2-d Arrays

- Recall that address of $[i][j]$ -th element is found by first finding the address of first element of i -th row, then adding j to it
- Now think of a 2-d array of dimension $[M][N]$ as M 1-d arrays, each with N elements, such that the starting address of the M arrays are contiguous (so the starting address of k -th row can be found by adding 1 to the starting address of $(k-1)$ -th row)
- This is done by allocating an array p of M pointers, the pointer $p[k]$ to store the starting address of the k -th row

Contd.

- Now, allocate the M arrays, each of N elements, with $p[k]$ holding the pointer for the k-th row array
- Now p can be subscripted and used as a 2-d array
- Address of $p[i][j] = *(p+i) + j$ (note that $*(p+i)$ is a pointer itself, and p is a pointer to a pointer)

Dynamic Allocation of 2-d Arrays

```
int **allocate (int h, int w)
```

```
{  
    int **p;  
    int i, j;  
  
    p = (int **) malloc(h*sizeof (int *) );  
    for (i=0;i<h;i++)  
        p[i] = (int *) malloc(w * sizeof (int));  
    return(p);  
}
```

**Allocate array
of pointers**




**Allocate array of
integers for each
row**



```
void read_data (int **p, int h, int w)
```

```
{  
    int i, j;  
    for (i=0;i<h;i++)  
        for (j=0;j<w;j++)  
            scanf ("%d", &p[i][j]);  
}
```

**Elements accessed
like 2-D array elements.**



Contd.

```
void print_data (int **p, int h, int w)  
{  
    int i, j;  
    for (i=0;i<h;i++)  
    {  
        for (j=0;j<w;j++)  
            printf ("%5d ", p[i][j]);  
            printf ("\n");  
        }  
    }
```

```
int main()  
{  
    int **p;  
    int M, N;  
    printf ("Give M and N \n");  
    scanf ("%d%d", &M, &N);  
    p = allocate (M, N);  
    read_data (p, M, N);  
    printf ("\nThe array read as \n");  
    print_data (p, M, N);  
    return 0;  
}
```

Contd.

```
void print_data (int **p, int h, int w)
{
    int i, j;
    for (i=0;i<h;i++)
    {
        for (j=0;j<w;j++)
            printf ("%5d ", p[i][j]);
        printf ("\n");
    }
}
```

Give M and N

3 3

1 2 3

4 5 6

7 8 9

**The array read
as**

1 2 3

4 5 6

```
int main()
{
    int **p;
    int M, N;
    printf ("Give M and N \n");
    scanf ("%d%d", &M, &N);
    p = allocate (M, N);
    read_data (p, M, N);
    printf ("\nThe array read as \n");
    print_data (p, M, N);
    return 0;
}
```

Memory Layout in Dynamic Allocation

```
int main()
{
    int **p;
    int M, N;
    printf ("Give M and N \n");
    scanf ("%d%d", &M, &N);
    p = allocate (M, N);
    for (i=0;i<M;i++) {
        for (j=0;j<N;j++)
            printf ("%10d", &p[i][j]);
        printf("\n");
    }
    return 0;
}
```

```
int **allocate (int h, int w)
{
    int **p;
    int i, j;

    p = (int **)malloc(h*sizeof (int *));
    for (i=0; i<h; i++)
        printf ("%10d", &p[i]);
    printf("\n\n");
    for (i=0;i<h;i++)
        p[i] = (int
        *)malloc(w*sizeof(int));
    return(p);
}
```

Output

3 3

31535120 31535128 31535136

31535152 31535156 31535160

31535184 31535188 31535192

31535216 31535220 31535224

Starting address of each row, contiguous (pointers are 8 bytes long)

Elements in each row are contiguous

Polynomial representation

Array representation assumes that the exponents of the given expression are arranged from 0 to the highest value (degree), which is represented by the subscript of the array beginning with 0. The coefficients of the respective exponent are placed at an appropriate index in the array. Consider The array representation for the below polynomial expression $P(x) = x^2 - 4x + 7$

polynomial can be represented in the various ways.

These are using :

- arrays
- Linked List