Basic Data Structures

Unit 2 : Introduction to Arrays, Structures and Pointers

Prepared by :
Prof. Khushbu Chauhan
Computer Engg. Dept.

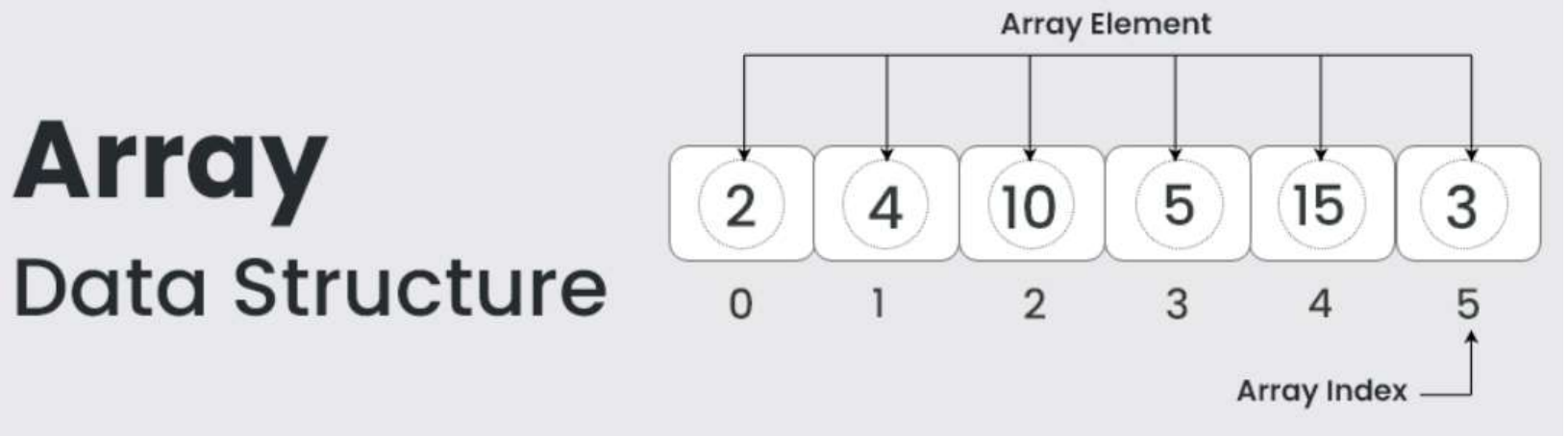
MPSTME, NMIMS

Outlines

- Arrays
- Structures
- Pointers

One Dimensional Array

- Simplest data structure that makes use of computed address to locate its elements is the one- dimensional array or vector; number of memory locations is sequentially allocated to the vector.
- A vector size is fixed and therefore requires a fixed number of memory locations.



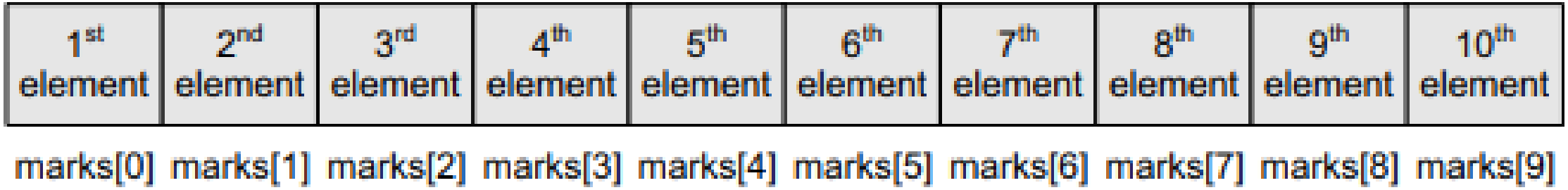
Declaration of Array

- An array must be declared before being used. Declaring an array means specifying the following:
- Data type — The kind of values it can store, for example, int, char, float, double.
- Name — To identify the array.
- Size — The maximum number of values that the array can hold
- Arrays are declared using the following syntax:

type name[size];

- The type can be either int, float, double, char, or any other valid data type. The number within brackets indicates the size of the array, i.e., the maximum number of elements that can be stored in the array. For example, if we write

int marks[10];



Accessing the elements of an array

- To access all the elements loop must be use.
- To access all the elements of an array by varying the value of the subscript into the array. But note that the subscript must be an integral value or an expression that evaluates to an integral value.

```
// Set each element of the array to –1
 int i, marks[10];
for(i=0;i<10;i++)
marks[i] = -1;
```

Calculating the Address of Array Elements

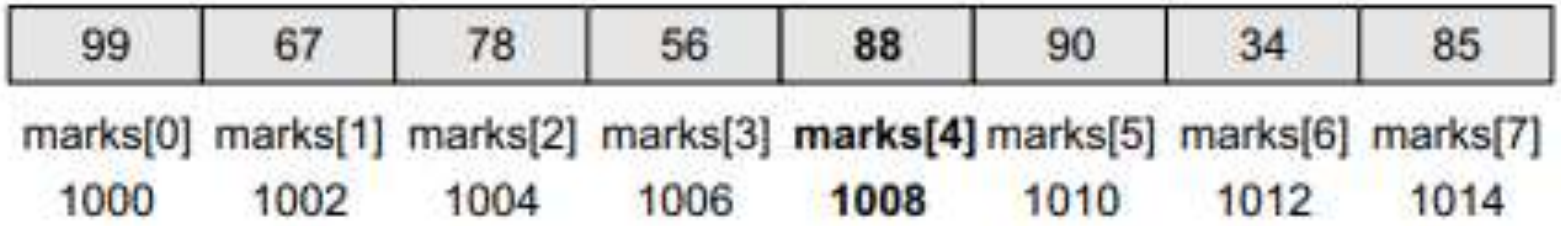
- The array name is a symbolic reference to the address of the first byte of the array. When user use the array name, it is actually referring to the first byte of the array. The subscript or the index represents the offset from the beginning of the array to the element being referenced. That is, with just the array name and the index, C can calculate the address of any element in the array.
- An array stores all its data elements in consecutive memory locations, storing just the base address, that is the address of the first element in the array, is sufficient. The address of other data elements can simply be calculated using the base address. The formula to perform this calculation is, Address of data element,

$$A[k] = BA(A) + w(k - lower\_bound)$$

Example:

- Given an array int marks[]={99,67,78,56,88,90,34,85}, calculate the address of marks[4] if the base address = 1000.

Solution



storing an integer value requires 2 bytes, therefore, its size is 2 bytes.

marks[4] = 1000 + 2(4 – 0)

= 1000 + 2(4) = 1008

Calculating the Length of an Array

- The length of an array is given by the number of elements stored in it.
- The general formula to calculate the length of an array is

Length = upper_bound – lower_bound + 1

- Where, upper_bound is the index of the last element and lower_bound is the index of the first element in the array.

Example

Let Age[5] be an array of integers such that Age[0] = 2, Age[1] = 5, Age[2] = 3, Age[3] = 1, Age[4] = 7 Show the memory representation of the array and calculate its length.

Solution

Length = upper_bound – lower_bound + 1

Here, lower_bound = 0, upper_bound = 4

Therefore, length = 4 – 0 + 1 = 5

Traversing of an array

- Traversing an array means accessing each and every element of the array for a specific purpose.
- Traversing the data elements of an array A can include printing every element, counting the total number of elements, or performing any process on these elements. Since, array is a linear data structure (because all its elements form a sequence).

- The algorithm for array traversal is given below:

  Step 1: [INITIALIZATION] SET I= lower_bound

  Step 2: Repeat Steps 3 to 4 while I<= upper_bound

  Step 3: Apply Process to A[I]

  Step 4: SET I=I+1 [END OF LOOP]

  Step 5: EXIT

Inserting an Element in an Array

- If an element has to be inserted at the end of an existing array, then the task of insertion is quite simple. Just have to add 1 to the upper_ bound and assign the value

  Step 1: Set upper_bound = upper_bound+1

  Step 2: Set A[upper_bound] = VAL

  Step 3: EXIT

Example

Data[] is an array that is declared as int Data[20]; and contains the following values: Data[] = {12, 23, 34, 45, 56, 67, 78, 89, 90, 100};

(a) Calculate the length of the array.

(b) Find the upper_bound and lower_bound.

(c) Show the memory representation of the array.

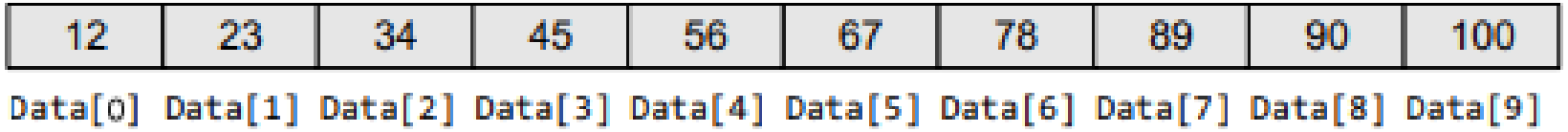
(d) If a new data element with the value 75 has to be inserted, find its position.

(e) Insert a new data element 75 and show the memory representation after the insertion.

(a) Length of the array = number of elements Therefore, length of the array = 10

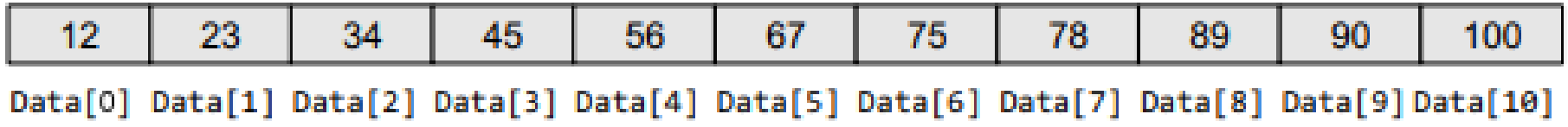
(b) By default, lower_bound = 0 and upper_bound = 9

(c)



(d) Since the elements of the array are stored in ascending order, the new data element will be stored after 67, i.e., at the 6th location. So, all the array elements from the 6th position will be moved one position towards the right to accommodate the new value

(e)



Insert an Element in the Middle of an Array

- The algorithm INSERT will be declared as INSERT (A, N, POS, VAL). The arguments are

(a) A, the array in which the element has to be inserted

(b) N, the number of elements in the array

(c) POS, the position at which the element has to be inserted

(d) VAL, the value that has to be inserted

- Step 1: [INITIALIZATION] SET I=N
- Step 2: Repeat Steps 3 and 4 while I>= POS
- Step 3: SET A[I + 1] = A[I]
- Step 4: SET I=I–1 [END OF LOOP]
- Step 5: SET N=N+1
- Step 6: SET A[POS] = VAL
- Step 7: EXIT

```
// shift elements to the right
// which are on the right side of pos
for (int i = n - 1; i >= pos; i--)
    arr[i + 1] = arr[i];

arr[pos] = x;
```

Deleting an Element from an Array

- Deleting an element from an array means removing a data element from an already existing array. If the element has to be deleted from the end of the existing array, then the task of deletion is quite simple. We just have to subtract 1 from the upper_bound.
- An algorithm to delete an element from the end of an array.
- Step 1: SET upper_bound = upper_bound-1
- Step 2: EXIT

Example

Data[] is an array that is declared as int Data[10]; and contains the following values: Data[] = {12, 23, 34, 45, 56, 67, 78, 89, 90, 100};

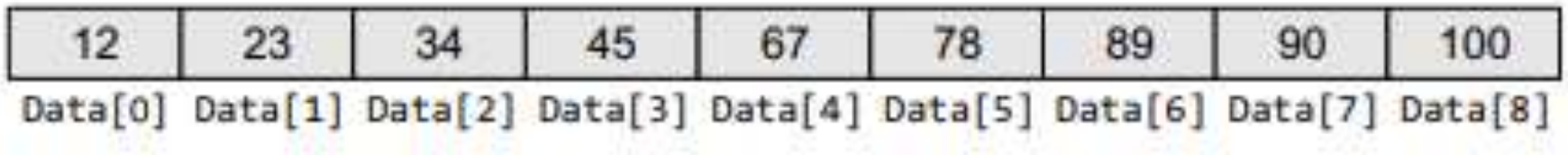
(a) If a data element with value 56 has to be deleted, find its position.

(b) Delete the data element 56 and show the memory representation after the deletion.

Solution

(a) Since the elements of the array are stored in ascending order, we will compare the value that has to be deleted with the value of every element in the array. As soon as VAL = Data[I], where I is the index or subscript of the array, we will get the position from which the element has to be deleted. For example, if we see this array, here VAL = 56. Data[0] = 12 which is not equal to 56. We will continue to compare and finally get the value of POS = 4

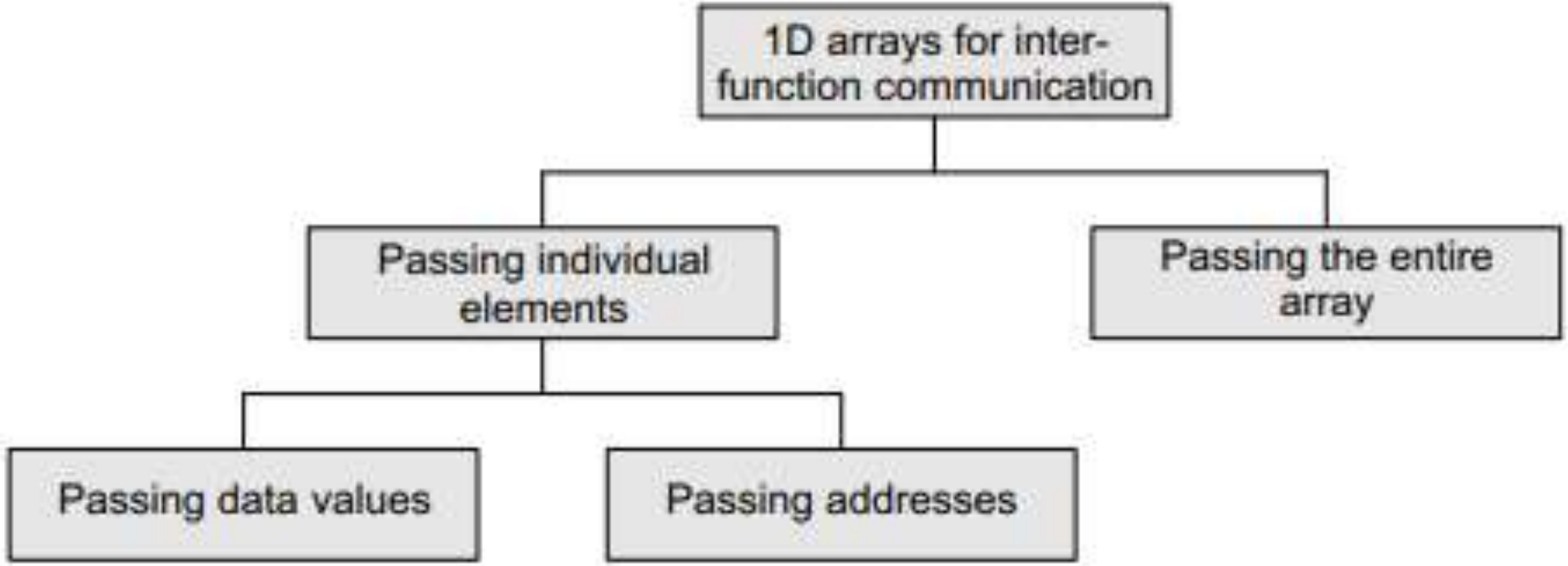
(b) Ans.



```c
for(i = 0; i < size; i++)
{
    if(arr[i] == key)
    {
        index = i;
        break;
    }
}

if(index != -1)
{
    //shift all the element from index+1 by one position to the left
    for(i = index; i < size - 1; i++)
        arr[i] = arr[i+1];

    printf("New Array : ");
    for(i = 0; i < size - 1; i++)
        printf("%d ",arr[i]);
}
else
    printf("Element Not Found\n");

return 0;
```

Passing Arrays to Functions

- Array can be pass in different situations.



Passing Individual Elements

- Passing Data Values:
- Individual elements can be passed in the same manner as we pass variables of any other data type. The condition is just that the data type of the array element must match with the type of the function parameter

Calling function

```
main()
{
        int arr[5] ={1, 2, 3, 4, 5};
        func(arr[3]);
}
```

Called function

```
void func(int num)
{
        printf("%d", num);
}
```

Passing Individual Elements

- Passing Addresses:
- Like ordinary variables, we can pass the address of an individual array element by preceding the indexed array element with the address operator. Therefore, to pass the address of the fourth element of the array to the called function, we will write &arr[3]

Calling function

```
main()
{
        int arr[5] ={1, 2, 3, 4, 5};
        func(&arr[3]);
}
```

Called function

```
void func(int *num)
{
        printf("%d", *num);
}
```

Passing the Entire Array

- In C the array name refers to the first byte of the array in the memory. The address of the remaining elements in the array can be calculated using the array name and the index value of the element. Therefore, when we need to pass an entire array to a function, we can simply pass the name of the array

Calling function

```
main()
{
        int arr[5] ={1, 2, 3, 4, 5};
        func(arr);
}
```

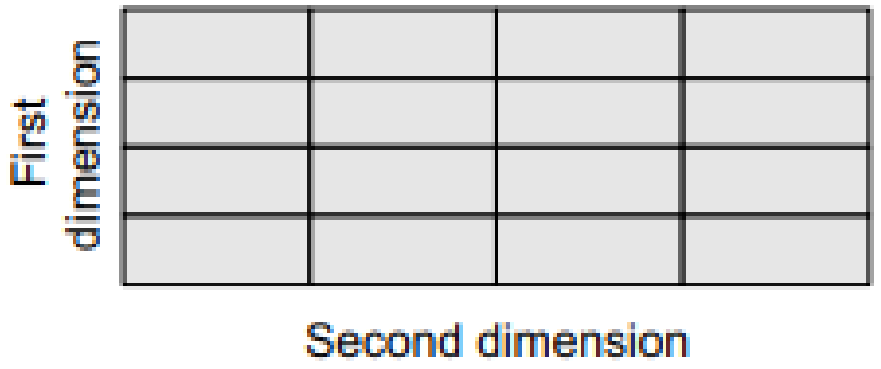
Called function

```
void func(int arr[5])
{
        int i;
        for(i=0;i<5;i++)
                printf("%d", arr[i]);
}
```

Two Dimensional Array

- A two-dimensional array is specified using two subscripts where the first subscript denotes the row and the second denotes the column. The C compiler treats a two-dimensional array as an array of one-dimensional arrays

data_type array_name[row_size][column_size];



Accessing the Elements of Two-dimensional Array

```
int arr[2][2],int i, j;
for(i=0;i<2;i++)
        For(j=0;j<2;j++)
arr[i][j] = data;
```

Rows / Columns	Col 0	Col 1	Col 2	Col 3	Col 4
Row 0	marks[0][0]	marks[0][1]	marks[0][2]	marks[0][3]	marks[0][4]
Row 1	marks[1][0]	marks[1][1]	marks[1][2]	marks[1][3]	marks[1][4]
Row 2	marks[2][0]	marks[2][1]	marks[2][2]	marks[2][3]	marks[2][4]

Operations on Two-dimensional Array

- Transpose: Transpose of an mxn matrix A is given as a nxm matrix B, where $B_{i,j} = {}_{Aj,I}$
- Sum: $C_{i,j} = A_{i,j} + B_{i,j}$
- Difference: $C_{i,j} = A_{i,j} - B_{i,j}$
- Product: $C_{i,j} = \sum A_{i,k}B_{k,j}$ for k=1 to n

Structures

- A structure is in many ways similar to a record. 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.
- The major difference between a structure and an array is that an array can store only information of same data type.
- A structure is therefore a collection of variables under a single name. The variables within a structure are of different data types and each has a name that is used to select it from the structure

Structure Declaration

- A structure is declared using the keyword struct followed by the structure name. All the variables of the structure are declared within the structure. A structure type is generally declared by using the following syntax:

```
struct struct–name
{
        data_type var–name;
        data_type var–name;
        ..............
};
```

```
struct student
{
        int r_no;
        char name[20];
        float weight;
};
```

Initialization of Structure

- A structure can be initialized in the same way as other data types are initialized. 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 it. For int and float members, the values are initialized to zero, and char and string members are initialized to '\0' by default.
- The initializers are enclosed in braces and are separated by commas. However, care must be taken to ensure that the initializers match their corresponding types in the structure definition.

- The general syntax to initialize a structure variable is as follows:

```
struct struct_name
 {
        data_type member_name1;
        data_type member_name2;
        data_type member_name3;
        ....................
 }struct_var = {constant1, constant2, constant3,...};
```

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 or a member of a structure can be given as:

  struct_var.member_name

- The dot operator is used to select a particular member of the structure. For example, to assign values to the individual data members of the structure variable studl, we may write

```
stud1.r_no = 01;
stud1.name = "Rahul";
stud1.course = "BCA";
stud1.fees = 45000;
```

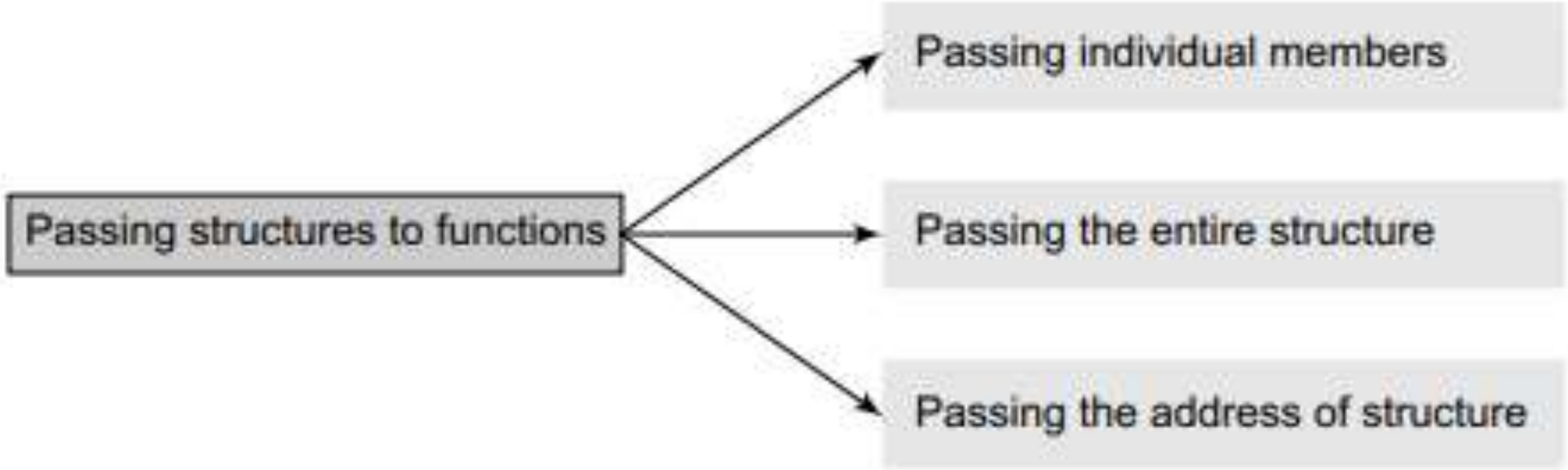
Arrays of Structures

- Defining a separate structure for multiple members is not a viable solution. So, here a common structure definition for all the accessing members. This can again be done by declaring an array of structure.
- The general syntax for declaring an array of structures can be given as,

```
struct struct_name
{
        data_type member_name1;
        data_type member_name2;
        data_type member_name3;
        .....................
}; struct struct_name struct_var[index];
```

Structures and Function

- For structures to be fully useful, there is a mechanism to pass them to functions and return them. A function may access the members of a structure in three ways:



Passing Individual Members

```
#include <stdio.h>
typedef struct
{
      int x;
      int y;
}POINT;
void display(int, int);
int main()
{
      POINT p1 = {2, 3};
      display(p1.x, p1.y);
      return 0;
}
void display(int a, int b)
{
      printf(" The coordinates of the point are: %d %d", a, b);
}
```

Passing the Entire Structure

- When a structure is passed as an argument, it is passed using the call by value method, i.e., a copy of each member of the structure is made.
- 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);
```

Pointer

- A pointer is a variable that represents the location of a data item, such as a variable or an array element. Pointers are frequently used in C, as they have a number of useful applications. These applications include:
- Pointers are used to pass information back and forth between functions.
- Pointers enable the programmers to return multiple data items from a function via function arguments.
- Pointers provide an alternate way to access the individual elements of an array.
- Pointers are used to pass arrays and strings as function arguments. We will discuss this in subsequent chapters.
- Pointers are used to create complex data structures, such as trees, linked lists, linked stacks, linked queues, and graphs

Declaring Pointer Variables

- The general syntax of declaring pointer variables can be given as below.

```
data_type *ptr_name;
```

- Here, data_type is the data type of the value that the pointer will point to.
- For example,

```
int *pnum;
char *pch;
float *pfnum;
```

Accessing values using pointer variable

- A pointer variable is declared to point to a variable of the specified data type.
- Although all these pointers point to different data types, they will occupy the same amount of space in the memory. But how much space they will occupy will depend on the platform where the code is going to run.

```
int x= 10;
int *ptr;
ptr = &x;
```

```c
#include <stdio.h>
int main()
{
        int num, *pnum;
        pnum = &num;
        printf("\n Enter the number : ");
        scanf("%d", &num);
        printf("\n The number that was entered is : %d", *pnum);
        return 0;
}
```

Output

```
Enter the number : 10
The number that was entered is : 10
```

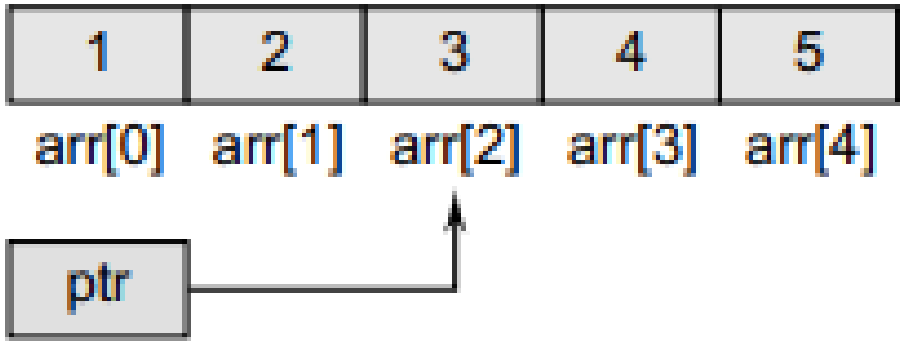
Pointer to arrays

- The concept of array is very much bound to the concept of pointer.

```
int arr[] = {1, 2, 3, 4, 5};
```

- Array notation is a form of pointer notation. The name of the array is the starting address of the array in memory. It is also known as the base address. In other words, base address is the address of the first element in the array or the address of arr[0].
- Now let us use a pointer variable as given in the statement below.

```
int *ptr;
ptr = &arr[0];
```



- writing ptr = &arr[2] makes ptr to point to the third element of the array that has index 2.
- If pointer variable ptr holds the address of the first element in the array, then the address of successive elements can be calculated by writing ptr++.
- int *ptr = &arr[0];
- ptr++;
- printf("\n The value of the second element of the array is %d", *ptr);

- An array of pointers can be declared as

```
int *ptr[10];
int p = 1, q = 2, r = 3, s = 4, t = 5;
ptr[0] = &p;
ptr[1] = &q;
ptr[2] = &r;
ptr[3] = &s;
ptr[4] = &t;
printf("\n %d", *ptr[3]);
```

Pointer to structures

- Pointers to structures are very important because you can use them to create complex and dynamic data structures such as linked lists, trees, graphs, etc. Such data structures use **self-referential structs**, where we define a struct type having one of its elements as a pointer to the same type.

```
struct mystruct
{
        int a;
        struct mystruct *b;
};
```

```c
struct person{
   char *name;
   int age;
   float weight;
};

int main(){

   struct person *personPtr, person1;

   strcpy(person1.name, "Meena");
   person1.age = 40;
   person1.weight = 60;

   personPtr = &person1;

   printf("Displaying the Data: \n");
   printf("Name: %s\n", personPtr -> name);
   printf("Age: %d\n", personPtr -> age);
   printf("Weight: %f", personPtr -> weight);

   return 0;
}
```

Output

```
Displaying the Data:
Name: Meena
Age: 40
weight: 60.000000
```

Passing pointer to the function

- A function in C can be called in two ways –
- Call by Value
- Call by Reference
- To call a function by reference, you need to define it to receive the pointer to a variable in the calling function.
- Here is the **syntax** that you would use to call a function by reference –

```
type function_name(type *var1, type *var2, ...)
```

- When a function is called by reference, the pointers of the actual argument variables are passed, instead of their values.

```c
/* function declaration */
int add(int *, int *);

int main(){

   int a = 10, b = 20;
   int c = add(&a, &b);
   printf("Addition: %d", c);
}

int add(int *x, int *y){
   int z = *x + *y;

   return z;
}
```

Output

```
Addition: 30
```

DISCUSSION…

SVKM'S NMIMS
Deemed to be UNIVERSITY

THANK YOU