

DATA STRUCTURES



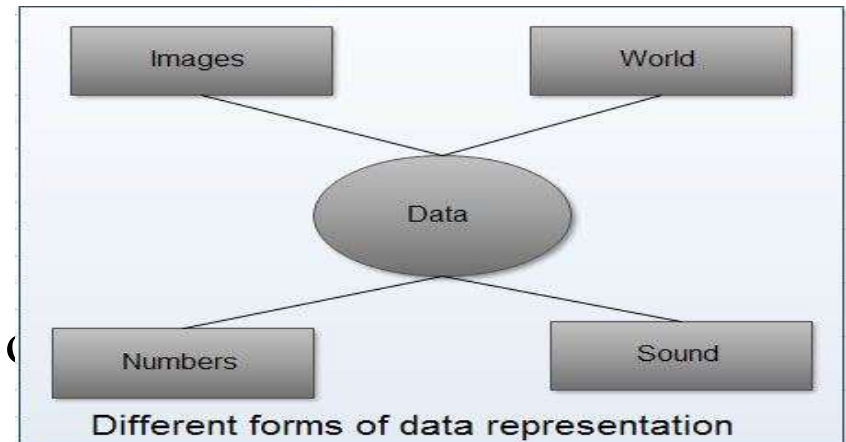
Introduction and basic terminology

DATA

- **raw, unorganized facts and figures** that can be collected, recorded, and used for analysis.
- It is the foundation upon which meaningful information is built.

Data can take many forms, including:

1. **Text** - Words, sentences, or documents (e.g., a book, a message log).
2. **Numbers** - Quantitative measurements (e.g., temperatures, scores, sales figures).
3. **Images** - Photographs, scanned documents, etc.
4. **Audio** - Sound recordings or voice data.
5. **Video** – Motion pictures or visual recordings.
6. **Sensors or Logs** - Data collected from devices or systems (e.g., IoT sensor data, website logs).



- **Types of Data:**

1. **Structured Data** - Organized and stored in a predefined format (e.g., rows and columns in a database).
 2. **Unstructured Data** - Not organized into a specific format (e.g., emails, videos, or social media posts).
 3. **Semi-Structured Data** - Partially organized (e.g., JSON, XML).
- When processed and interpreted, data becomes information, which can be used to make decisions, draw conclusions, or support research.

INFORMATION

Information is **processed, organized, or structured data** that has meaning, context, and relevance.

It is the result of analyzing or interpreting raw data to make it useful for decision-making, problem-solving, or gaining knowledge.

- Eg: Data: 25, 30, 35

Information: The temperatures recorded in the last three days were 25°C, 30°C, and 35°C.

- **Key Characteristics of Information:**

1. **Meaningful:** Information provides insight or understanding, unlike raw data, which lacks context.
2. **Organized:** It is structured in a way that makes it easy to interpret (e.g., a report or chart).
3. **Contextual:** Information is derived with a specific purpose or context in mind.

Algorithm

- An algorithm is a set of instructions to be done sequentially. Any work to be done can be thought as
 1. Set up the required apparatus
 2. Do the process required
 3. Note any observations
 4. Summarize the results series of steps

Necessary Criteria For An Algorithm

- Input: - The input of an algorithm can either be given by the user or generated internally.
- Output: - An algorithm should have at least one output.
- Finiteness: - An algorithm should end in a finite number of steps.
- Definiteness: - Every step of an algorithm should be clear and unambiguously defined.
- Effectiveness: - Each step must be simple enough to execute in a finite time.

LIFE CYCLE OF AN ALGORITHM

The life cycle of an Algorithm consists of four phases:

1.Design:-Develop a clear, step-by-step plan for solving the problem.

2.Write(Implementation):- Translate the algorithm into code using an appropriate programming language.

3. Test:- Verify the correctness of the algorithm by running it with different inputs and analyzing outputs.

4.Analyze :- Evaluate its efficiency by estimating the amount of time/space (which are considered to be prime resources) required while executing the algorithm.

DATA STRUCTURE

- **Data Structure** is a way of organizing, managing, and storing data in a computer so that it can be accessed and modified efficiently.
- It provides a systematic way to handle data for various operations such as searching, sorting, inserting, and deleting.

Operations on Data Structures

- **Insertion:** Adding an element to the structure.
- **Deletion:** Removing an element.
- **Traversal:** Accessing each element of the structure.
- **Searching:** Finding a particular element.
- **Sorting:** Arranging elements in a specific order.

Aspect	<u>Linear Data Structure</u>	<u>Non-Linear Data Structure</u>
<u>Structure</u>	Data elements are arranged sequentially (one after another).	Data elements are arranged hierarchically or in an interconnected manner.
<u>Traversal</u>	Traversed in a single run (e.g., from start to end).	May require multiple runs to traverse completely (e.g., depth-first, breadth-first).
<u>Complexity</u>	Simple to implement and understand.	More complex due to hierarchical or interconnected relationships.
<u>Memory Utilization</u>	May waste memory due to fixed sizes (e.g., arrays).	Efficient in terms of memory allocation based on requirements.
<u>Examples</u>	Array, Linked List, Stack, Queue	Tree, Graph, Heap
<u>Relationship Between Elements</u>	Every element has a direct successor and/or predecessor (except at the ends).	Each element can have multiple relationships (e.g., parent-child, connected nodes).
<u>Use Cases</u>	Ideal for tasks like searching, sorting, or sequential data processing.	Suitable for complex tasks like hierarchical representation, network routing, and shortest path finding.

Why Data Structure is important?

- ✓ Data structures study how data are stored in a computer so that operations can be implemented efficiently
- ✓ Data structures are especially important when you have a large amount of information
- ✓ Efficiently organize and manage data for efficient storage and manipulation.
- ✓ Optimize application performance by reducing time complexity.
- ✓ Enable scalability for handling large datasets or growing systems.
- ✓ support real-world applications like databases, social networks, and AI.
- ✓ Form the foundation for advanced topics like machine learning, cryptography, and block chain

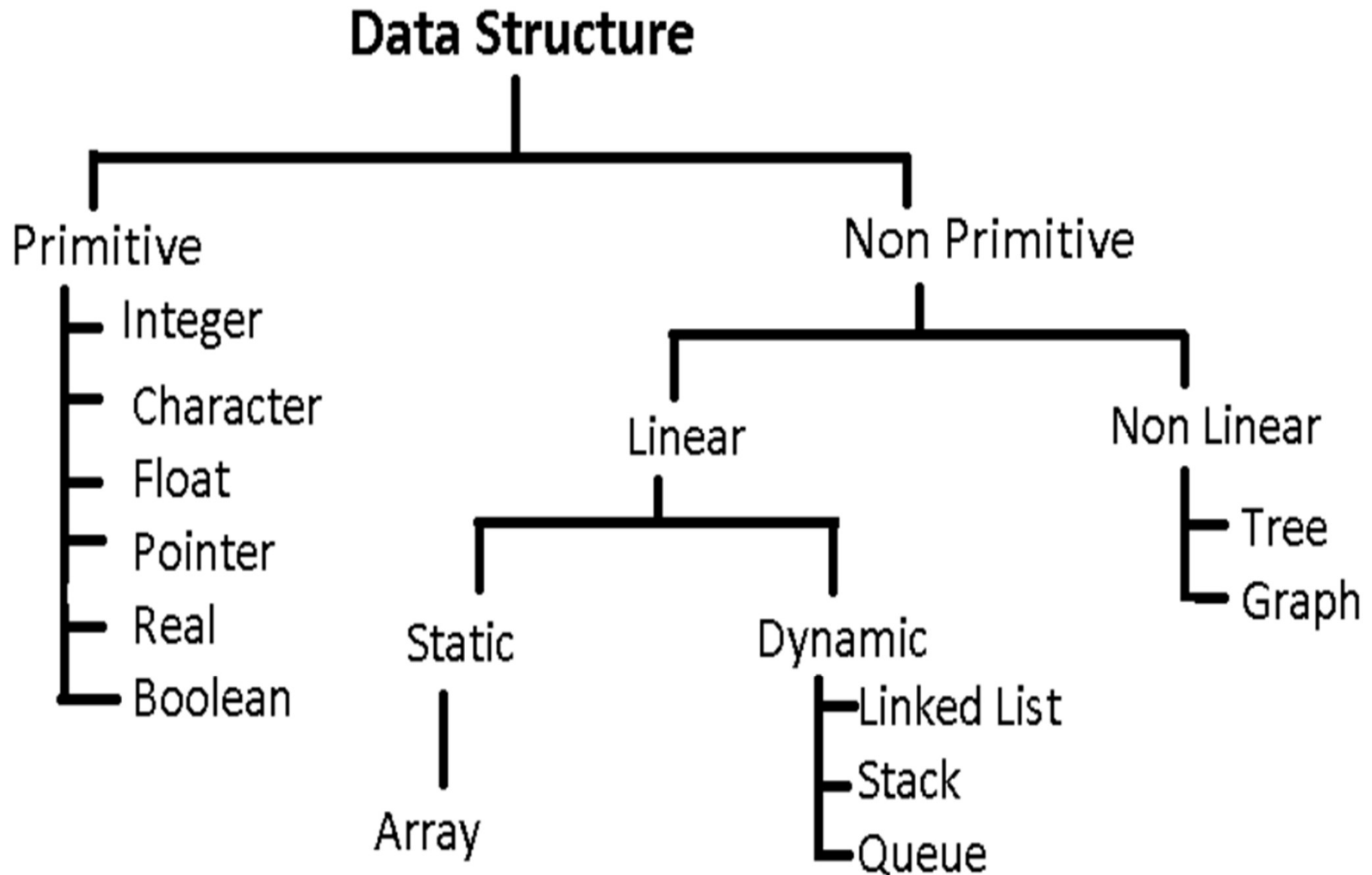
Need of Data Structures

- Processor speed: To handle very large amount of data, high speed processing is required, but as the data is growing day by day to the billions of files per entity, processor may fail to deal with that much amount of data.
- Data Search: Consider an inventory size of 10^6 items in a store, If our application needs to search for a particular item, it needs to traverse 10^6 items every time, results in slowing down the search process.
- Multiple requests: If thousands of users are searching the data simultaneously on a web server, then there are the chances that a very large server can be failed during that process in order to solve the above problems, data structures are used.

Advantages of Data Structures

- **Efficiency:** Efficiency of a program depends upon the choice of data structures. For example: suppose, we have some data and we need to perform the search for a particular record. In that case, if we organize our data in an array, we will have to search sequentially element by element. hence, using array may not be very efficient here. There are better data structures which can make the search process efficient like ordered array, binary search tree or hash tables.
- **Reusability:** Data structures are reusable, i.e. once we have implemented a particular data structure, we can use it at any other place. Implementation of data structures can be compiled into libraries which can be used by different clients.
- **Abstraction:** Data structure is specified by the ADT which provides a level of abstraction. The client program uses the data structure through interface only, without getting into the implementation details

CLASSIFICATION OF DATA STRUCTURE



Linear Data Structures

- If a data structure organizes the data in sequential order, then that data structure is called a Linear Data Structure.
- **Arrays:** An array is a collection of similar type of data items and each data item is called an element of the array. The data type of the element may be any valid data type like char, int, float or double. The elements of array share the same variable name but each one carries a different index number known as subscript. The array can be one dimensional, two dimensional or multidimensional. The individual elements of the array age are: age[0], age[1], age[2], age[3],..... age[98], age[99].

- **Linked List:** Linked list is a linear data structure which is used to maintain a list in the memory. It can be seen as the collection of nodes stored at non-contiguous memory locations. Each node of the list contains a pointer to its adjacent node.
- **Stack:** Stack is a linear list in which insertion and deletions are allowed only at one end, called top. A stack is an abstract data type (ADT), can be implemented in most of the programming languages. Follow LIFO methodology.
- **Queue:** Queue is a linear list in which elements can be inserted only at one end called rear and deleted only at the other end called front. It is an abstract data structure, similar to stack. Queue is opened at both end therefore it follows First-In-First-Out (FIFO) methodology for storing the data items.

Non Linear Data Structures

- If a data structure organizes the data in random order, then that data structure is called as Non-Linear Data Structure.
- Trees: Trees are multilevel data structures with a hierarchical relationship among its elements known as nodes. The bottommost nodes in the hierarchy are called leaf node while the topmost node is called root node. Tree data structure is based on the parent-child relationship among the nodes. Each node in the tree can have more than one children except the leaf nodes whereas each node can have atmost one parent except the root node.
- Graphs: Graphs can be defined as the pictorial representation of the set of elements (represented by vertices) connected by the links known as edges. A graph is different from tree in the sense that a graph can have cycle while the tree can not have the one.

Memory Allocation

- Memory Allocation: Memory allocation is a process by which computer programs and services are assigned with physical or virtual memory space. The memory allocation is done either before or at the time of program execution.

There are two types of memory allocations:

- Compile-time or Static Memory Allocation
- Run-time or Dynamic Memory Allocation

Static Memory Allocation	Dynamic Memory Allocation
In static memory allocation, memory is allocated before the execution of the program begins.	In Dynamic memory allocation, memory is allocated during the execution of the program.
Memory allocation and deallocation actions are not performed during the execution.	Memory allocation and deallocation actions are performed during the execution.
It uses stack for managing the static allocation of memory	It uses heap for managing the dynamic allocation of memory
The data in static memory is allocated permanently.	The data in dynamic memory is allocated only when program unit is active.
It is less efficient	It is more efficient

UNIT 2

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 in RAM. The elements of an array are of same data type and each item can be accessed using the same name.

- -is a kind of data structure.
 - -is a finite ordered collection of homogeneous data elements.
 - i.e arrays is a collection of variables of same type.
- All the elements are stored one by one in contiguous location of memory in a linear ordered fashion.
- -An array is used to represent a list of numbers
 - or a list of names.

Why do we need arrays?

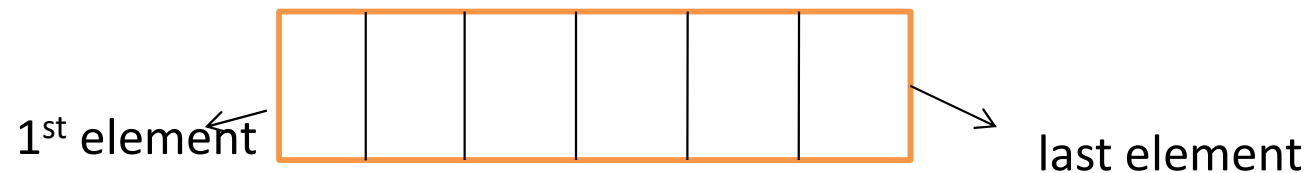
- *Arrays are best for storing multiple values in a single variable.
- *Arrays are better at processing many values easily and quickly
- *Sorting and searching the values is easier in arrays.

Array declaration.

Data_type var_name[Expression];

`int a[5] = { 5, 10, 15, 20, 25} ;`

`char word[10] = { 'h', 'a', 'i' } ;`



* Element :

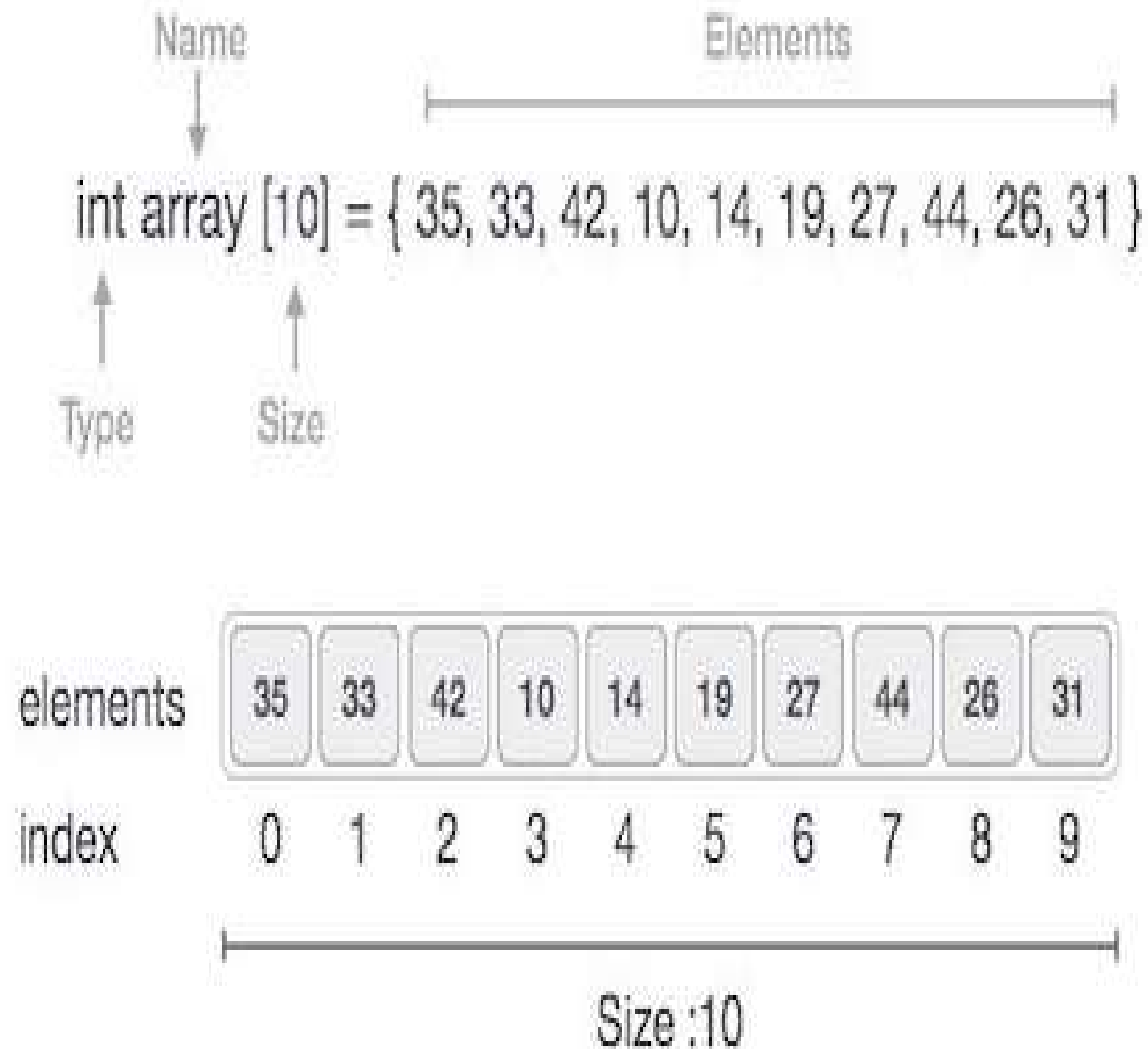
- each item stored in an array is called an element.

• Index :

- each location of an element in an array has a numerical index which is used to identify the element.

- index also called subscript

Array Representation:(Storage structure)



As per the above illustration, following are the important points to be considered.

- Index starts with 0.
- Array length is 10 which means it can store 10 elements.
- Each element can be accessed via its index.
ie `array[3]=10`

- An array consisting of two subscripts is known as two-dimensional array.
- These are often known as array of the array.
- In two dimensional arrays the array is divided into rows and columns.
- These are well suited to handle a table of data.

- Declaration Syntax :-

```
data_type array_name[row_size][column_size];
```

- Ex:- `int arr[3][3];`

where first index value shows the number of the rows
and second index value shows the number of the
columns in the array.

- Multidimensional arrays are often known as array of the arrays.
- In multidimensional arrays the array is divided into rows and columns, mainly while considering multidimensional arrays we will be discussing mainly about two dimensional arrays and a bit about three dimensional arrays.

Syntax:

```
data_type array_name[size1][size2][size3]-----[sizeN];
```

In 2-D array we can declare an array as :

```
int arr[3][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

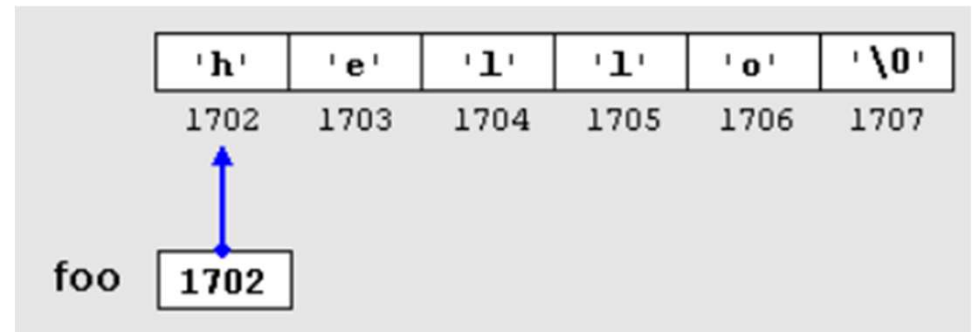
where first index value shows the number of the rows and second index value shows the number of the columns in the array.

Pointers and Arrays

- When an array is declared:
- The compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory locations.
- The base address is the location of the first element (index 0) of the array.
- The compiler also defines the array name as a constant pointer to the first element.

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

- Declaration: `int *ptr;`
- Initialization:
- `int a = 10;`
- `int *ptr = &a;`
- `printf("%d", *ptr); // Prints 10`




```
#int myNumbers[4] = {25, 50, 75, 100};
```

- ```
int myNumbers[4] = {25, 50, 75, 100};
int i;
```

```
for (i = 0; i < 4; i++) {
 printf("%d\n", myNumbers[i]);
}
```

Result:

```
25
50
75
100
```

- Instead of printing the value of each array element, let's print the memory address of each array element:

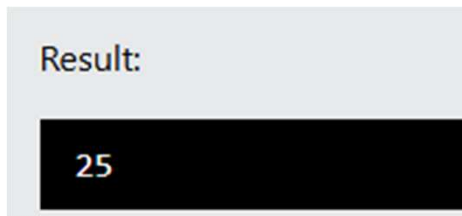
```
int myNumbers[4] = {25, 50, 75, 100};
int i;
for (i = 0; i < 4; i++) {
 printf("%p\n", &myNumbers[i]);
}
```

Result:

```
0x7ffe70f9d8f0
0x7ffe70f9d8f4
0x7ffe70f9d8f8
0x7ffe70f9d8fc
```

- `int myNumbers[4] = {25, 50, 75, 100};`

```
// Get the value of the first element in
myNumbers
printf("%d", *myNumbers);
```



Result:  
25

`myNumbers` is a pointer to the first element in `myNumbers`, you can use the `*` operator to access it

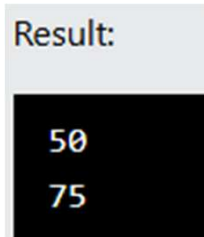
- `int myNumbers[4] = {25, 50, 75, 100};`

`// Get the value of the second element in  
myNumbers`

`printf("%d\n", *(myNumbers + 1));`

`// Get the value of the third element in myNumbers`

`printf("%d", *(myNumbers + 2));`



Result:  
50  
75

- Using Loop

```
int myNumbers[4] = {25, 50, 75, 100};
int *ptr = myNumbers;
int i;
for (i = 0; i < 4; i++) {
 printf("%d\n", *(ptr + i));
}
```

Result:

```
25
50
75
100
```

```
int myNumbers[4] = {25, 50, 75, 100};
```

```
// Change the value of the first element to 13
*myNumbers = 13;
```

```
// Change the value of the second element to 17
*(myNumbers + 1) = 17;
```

```
// Get the value of the first element
printf("%d\n", *myNumbers);
```

```
// Get the value of the second element
printf("%d\n", *(myNumbers + 1));
```

Result:

13  
17

**Q : C program for 1-D array operations:  
Insertion and Deletion**

```

#include <stdio.h>
#define MAX 100 // Maximum size of array
int main() {
 int arr[MAX], n, pos, val, i;
 // Input size
 printf("Enter number of elements: ");
 scanf("%d", &n);
 // Input elements
 printf("Enter %d elements:\n", n);
 for(i = 0; i < n; i++) {
 scanf("%d", &arr[i]);
 }
 // Input position and value for insertion
 printf("Enter position (1 to %d): ", n+1);
 scanf("%d", &pos);
 printf("Enter value to insert: ");
 scanf("%d", &val);
 // Check for valid position
 if(pos < 1 || pos > n+1) {
 printf("Invalid position!\n");
 } else {
 // Shift elements to right
 for(i = n; i >= pos; i--) {
 arr[i] = arr[i-1];
 }
 arr[pos-1] = val; // Insert element
 n++; // Increase size
 }
}

```

```

// Check for valid position
if(pos < 1 || pos > n+1) {
 printf("Invalid position!\n");
} else {
 // Shift elements to right
 for(i = n; i >= pos; i--) {
 arr[i] = arr[i-1];
 }
 arr[pos-1] = val; // Insert element
 n++; // Increase size
}

// Display updated array
printf("Array after insertion:\n");
for(i = 0; i < n; i++) {
 printf("%d ", arr[i]);
}
printf("\n");

return 0;
}

```



### Output

```
Enter number of elements: 3
Enter 3 elements:
23
234
56
Enter position (1 to 4): 4
Enter value to insert: 78
Array after insertion:
23 234 56 78
```

- A pointer is a variable that stores the address of another variable.
- `datatype *pointer_name;`
- `int *p; // pointer to int`

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

```
int main()
```

```
{
```

```
 int x=10;
```

```
 int *p=&x;
```

```
 printf("Value of x: %d\n",x);
```

```
 printf("Address of x: %p\n",&x);
```

```
 printf("Value using pointer: %d\n",*p);
```

```
 return 0;
```

```
}
```

- An array is a collection of elements stored in contiguous memory. Array name acts as pointer to first element.

```
datatype array_name[size];
```

```
datatype *p = array_name;
```

Program:

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

```
int main()
```

```
{
```

```
int arr[5]={10,20,30,40,50};
```

```
int *p=arr;
```

```
for(int i=0;i<5;i++)
```

```
{
```

```
 printf("arr[%d]=%d, via pointer=%d\n",i,arr[i],*(p+i));
```

```
}
```

```
return 0;
```

```
}
```

- Output:

arr[0]=10, via pointer=10

arr[1]=20, via pointer=20

arr[2]=30, via pointer=30

arr[3]=40, via pointer=40

arr[4]=50, via pointer=50

# Structure

- A structure is a user-defined datatype that groups variables of different types.

```
struct StructName {datatype member1;datatype
 member2;
};
```

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

```
// Define structure
```

```
struct Student
```

```
{
```

```
 int roll;
```

```
 char name[20];
```

```
 float marks;
```

```
};
```

Output:

Roll: 101

Name: Alice

Marks: 95.50

```
int main()
```

```
{
```

```
 struct Student s1 = {1, "Alice", 89.5};
```

```
 printf("Roll: %d\n", s1.roll);
```

```
 printf("Name: %s\n", s1.name);
```

```
 printf("Marks: %.2f\n", s1.marks);
```

```
 return 0;
```

```
}
```

- A pointer to structure stores address of a structure variable. Use '->' operator to access members.

```
struct StructName *ptr;
ptr = &struct_var;
ptr->member;
```

```
#include <stdio.h>
struct Student
{
 int roll; char name[20]; float marks;
};
int main()
{
 struct Student s1={102,"Bob",88.0};
 struct Student *ptr=&s1;
 printf("Roll: %d\n",ptr->roll);
 printf("Name: %s\n",ptr->name);
 printf("Marks: %.2f\n",ptr->marks);
 return 0;
}
```

# Sum of Array using Pointers

```
#include <stdio.h>
int main()
{
 int arr[5]={1,2,3,4,5};
 int *p=arr,sum=0;
 for(int i=0;i<5;i++)
 sum+=*(p+i);
 printf("Sum=%d\n",sum); return 0;
}
```



# Reverse Array using Pointers

- `#include <stdio.h>`
- `int main()`
- `{`
- `int arr[5]={10,20,30,40,50};`
- `int *p=arr;`
- `printf("Original: ");`
- `for(int i=0;i<5;i++)`
- `printf("%d ",*(p+i));`
- `printf("\nReversed: ");`
- `for(int i=4;i>=0;i--)`
- `printf("%d ",*(p+i));`
- `return 0;`
- 
- `}`

# Copy Array using Pointers

- `#include <stdio.h>`
- `int main()`
- `{`
- `int arr1[5]={5,10,15,20,25},arr2[5];`
- `int *p1=arr1,*p2=arr2;`
- `for(int i=0;i<5;i++) *(p2+i)=*(p1+i);`
- `printf("Copied: ");`
- `for(int i=0;i<5;i++)`
- `printf("%d ",arr2[i]); return 0;`
- 
- `}`

# Find Max/Min using Pointers

- `#include <stdio.h>`
- `int main()`
- `{`
- `int arr[6]={15,2,88,43,7,29};`
- `int *p=arr;`
- `int max=*p,min=*p;`
- `for(int i=1;i<6;i++)`
- `{`
- `if(*(p+i)>max) max=*(p+i);`
- `if(*(p+i)<min) min=*(p+i);`
- 
- 
- `}`
- `printf("Maximum=%d\n",max);`
- `printf("Minimum=%d\n",min);`
- `return 0;`
- 
- `}`