REVA UNIVERSITY
Bengaluru, India

Data Structures(M23DE0102)

Dr.S.Manju Priya
Professor
School of CSA

Introduction

LECTURE 1

OUTLINE

- COURSE OBJECTIVES
- COURSE CONTENTS
- LEARNING RESOURCES
  - TEXT BOOKS
  - REFERENCE BOOKS
- COURSE OUTCOMES
- COMPUTER BASICS

COURSE OBJECTIVES

1. Introduce the concept of algorithms and flow charts to understand and analyze the problem to write optimized algorithm for given problem statements.
2. Provide detailed understanding of basic concepts of C and Data Structure.
3. Provide detailed understanding of control statements, function and arrays.
4. Provide the knowledge of structures and unions,
5. Introduce the concepts of Files for application data maintenance
6. Introduce the concepts of Linear and Non-Linear Data Structure

COURSE CONTENTS:

UNIT -1:
13 Hours

Arrays -Insertion and deletion operations-Functions-Pointers- Declaring and Initializing Pointers-Pointer Arithmetic- Function and Pointer Parameters-Pointer and Arrays-Dynamic Memory Allocation Structures- Defining and using a Structure-Passing Structures to Functions-Structure and Pointers

Basics of Data Structures- Classifications (Primitive & Non-Primitive)- Data Structure Operations- Linear Data Structures- Stack: DefinitionArray representation- Operations- Recursion, Towers of HanoiApplications of stack (Infix to postfix conversion, evaluation of expression)

BASICS OF DATA STRUCTURE

Data structures are the fundamental building blocks of computer programming. They define how data is organized, stored, and manipulated within a program. Understanding data structures is very important for developing efficient and effective algorithms.

A **data structure** is a storage that is used to store and organize data. It is a way of arranging data on a computer so that it can be accessed and updated efficiently.

A data structure is not only used for organizing the data. It is also used for processing, retrieving, and storing data

UNDERSTANDING THE NEED FOR DATA STRUCTURES

As applications are becoming more complex and the amount of data is increasing every day, which may lead to problems with data searching, processing speed, multiple requests handling, and many more.

Data Structures support different methods to organize, manage, and store data efficiently.

With the help of Data Structures, we can easily traverse the data items. Data Structures provide Efficiency, Reusability, and Abstraction.

WHY SHOULD WE LEARN DATA STRUCTURES?

1. Data Structures and Algorithms are two of the key aspects of Computer Science.
2. Data Structures allow us to organize and store data, whereas Algorithms allow us to process that data meaningfully.
3. Learning Data Structures and Algorithms will help us become better Programmers.
4. We will be able to write code that is more effective and reliable.
5. We will also be able to solve problems more quickly and efficiently.

HOW ARE DATA STRUCTURES USED?

Data structures can be implemented using different programming languages. Some languages, such as C++ and Java, provide built-in support for common data structures. Other languages, such as Python and Ruby, do not have built-in support for data structures but offer libraries that implement them.

Common use cases for data structures:

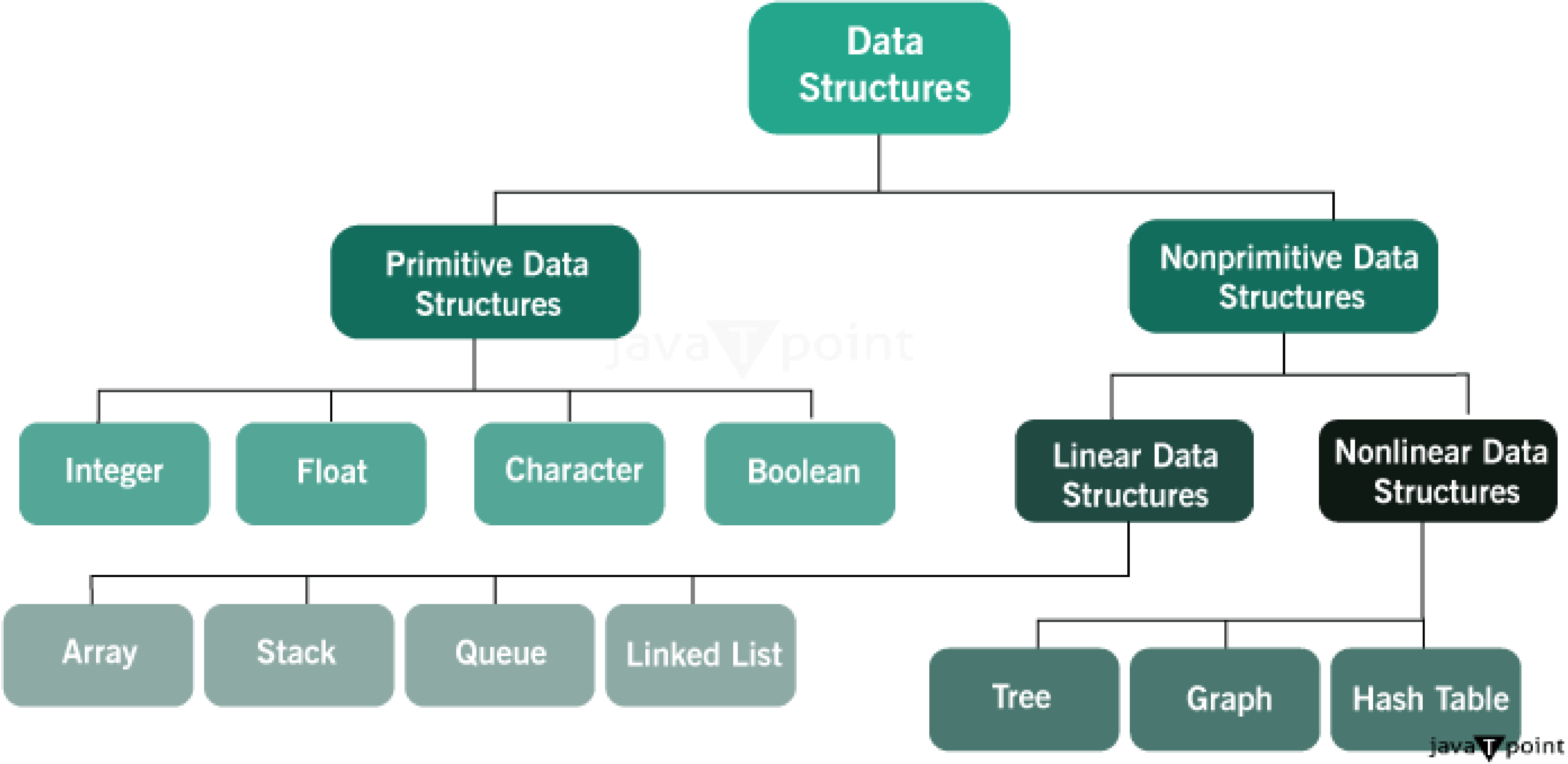
Managing resources and services: Data structures can be used to keep track of resources and services in a computer system. For example, hash tables can use the IP addresses of all the computers on a network to store data.

Maintaining information about users: Data structures can store information about a software application's users. For example, you can use a linked list to store the names and contact information of all the users of a social networking site.

Storing data from sensors: Data structures can store data like temperature sensors or motion detectors. For example, you can use an array to store the temperatures recorded by a thermometer over time.

Indexing: Indexing is a way of accessing data. Indexing is often used to speed up searches. For example, you can use a binary search tree to index the words in a dictionary so that you can quickly find the definition of a word.

Searching: Data structures can be used to search for data. For example, a binary search tree can quickly find a person's contact information in an extensive database


Data Structures
Primitive Data Structures
Nonprimitive Data Structures
Integer
Float
Character
Boolean
Linear Data Structures
Nonlinear Data Structures
Array
Stack
Queue
Linked List
Tree
Graph
Hash Table
javaTpoint

PRIMITIVE VS NON-PRIMITIVE DATA STRUCTURE

Data structure means organizing the data in the memory. The data can be organized in two ways either linear or non-linear way.

There are two types of data structure available for the programming purpose:

1. Primitive data structure
2. Non-primitive data structure

Primitive data structure is a fundamental type of data structure that stores the data of only one type whereas the non-primitive data structure is a type of data structure which is a user-defined that stores the data of different types in a single entity.

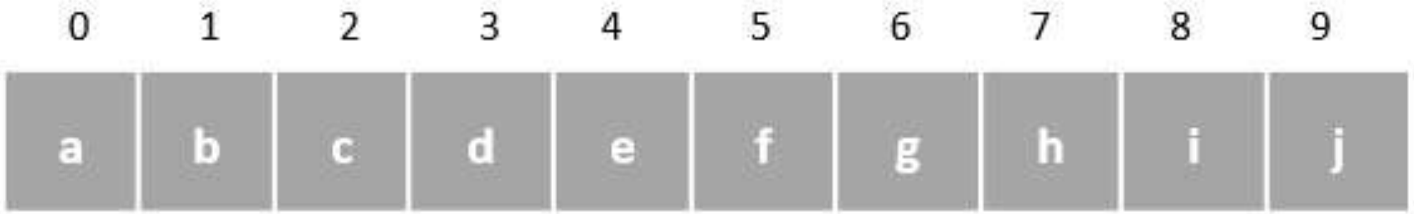
In the case of primitive data structure, it contains fundamental data types such as integer, float, character, pointer, and these fundamental data types can hold a single type of value.

For example, integer variable can hold integer type of value, float variable can hold floating type of value, character variable can hold character type of value whereas the pointer variable can hold pointer type of value.

In the case of non-primitive data structure, it is categorized into two parts such as linear data structure and non-linear data structure.

LINEAR DATA STRUCTURES

The data is stored in linear data structures sequentially. These are rudimentary structures since the elements are stored one after the other without applying any mathematical operations.



0	1	2	3	4	5	6	7	8	9
a	b	c	d	e	f	g	h	i	j

Linear data structures are usually easy to implement but since the memory allocation might become complicated, time and space complexities increase. Few examples of linear data structures include −

1. Arrays
2. Linked Lists
3. Stacks
4. Queues

Based on the data storage methods, these linear data structures are divided into two sub-types. They are − **static** and **dynamic** data structures.

STATIC LINEAR DATA STRUCTURES

In Static Linear Data Structures, the memory allocation is not scalable. Once the entire memory is used, no more space can be retrieved to store more data. Hence, the memory is required to be reserved based on the size of the program.

This will also act as a drawback since reserving more memory than required can cause a wastage of memory blocks.

The best example for static linear data structures is an array.

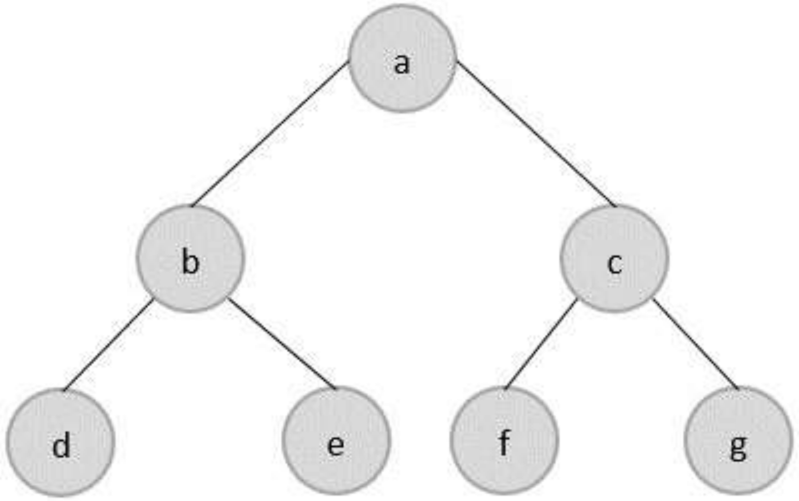
DYNAMIC LINEAR DATA STRUCTURES

In Dynamic linear data structures, the memory allocation can be done dynamically when required. These data structures are efficient considering the space complexity of the program.

Few examples of dynamic linear data structures include: linked lists, stacks and queues.

NON-LINEAR DATA STRUCTURES

Non-Linear data structures store the data in the form of a hierarchy. Therefore, in contrast to the linear data structures, the data can be found in multiple levels and are difficult to traverse through.

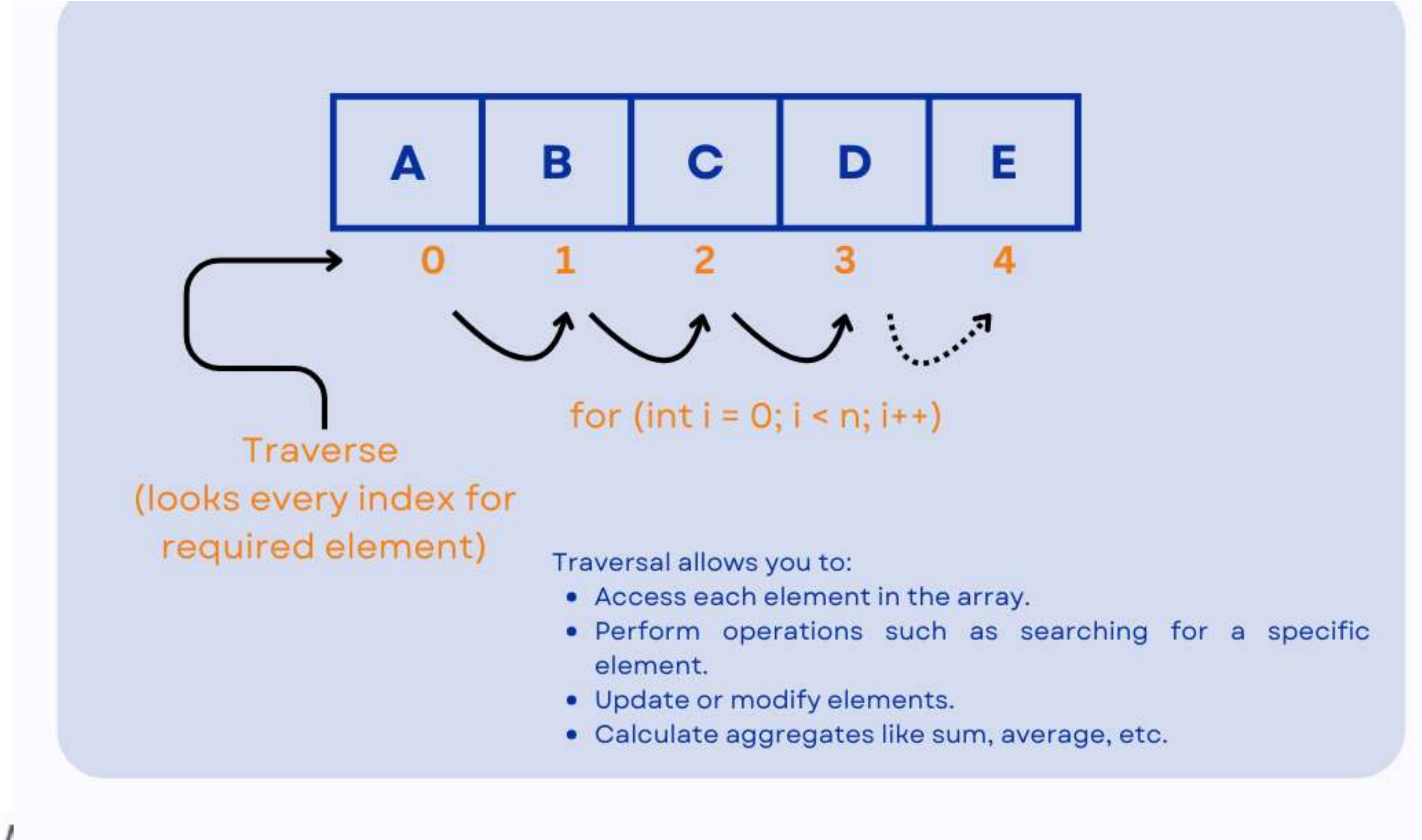


Few types of non-linear data structures are −
Graphs
Trees
Tries
Maps

BASIC OPERATIONS OF DATA STRUCTURES

1. Traversal

Traversal operations are used to visit each node in a data structure in a specific order. This technique is typically employed for printing, searching, displaying, and reading the data stored in a data structure.

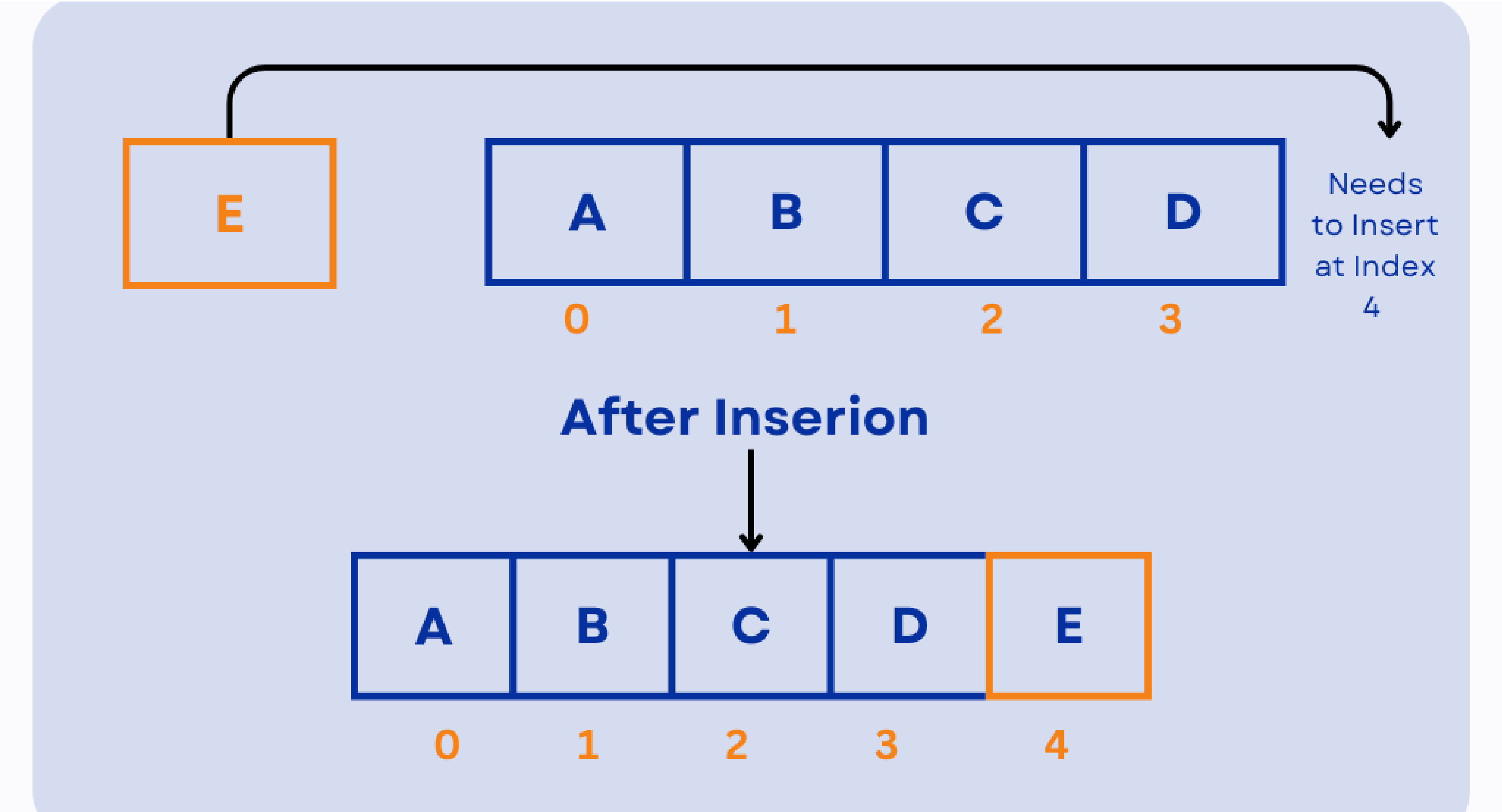


Traversal allows you to:

- Access each element in the array.
- Perform operations such as searching for a specific element.
- Update or modify elements.
- Calculate aggregates like sum, average, etc.

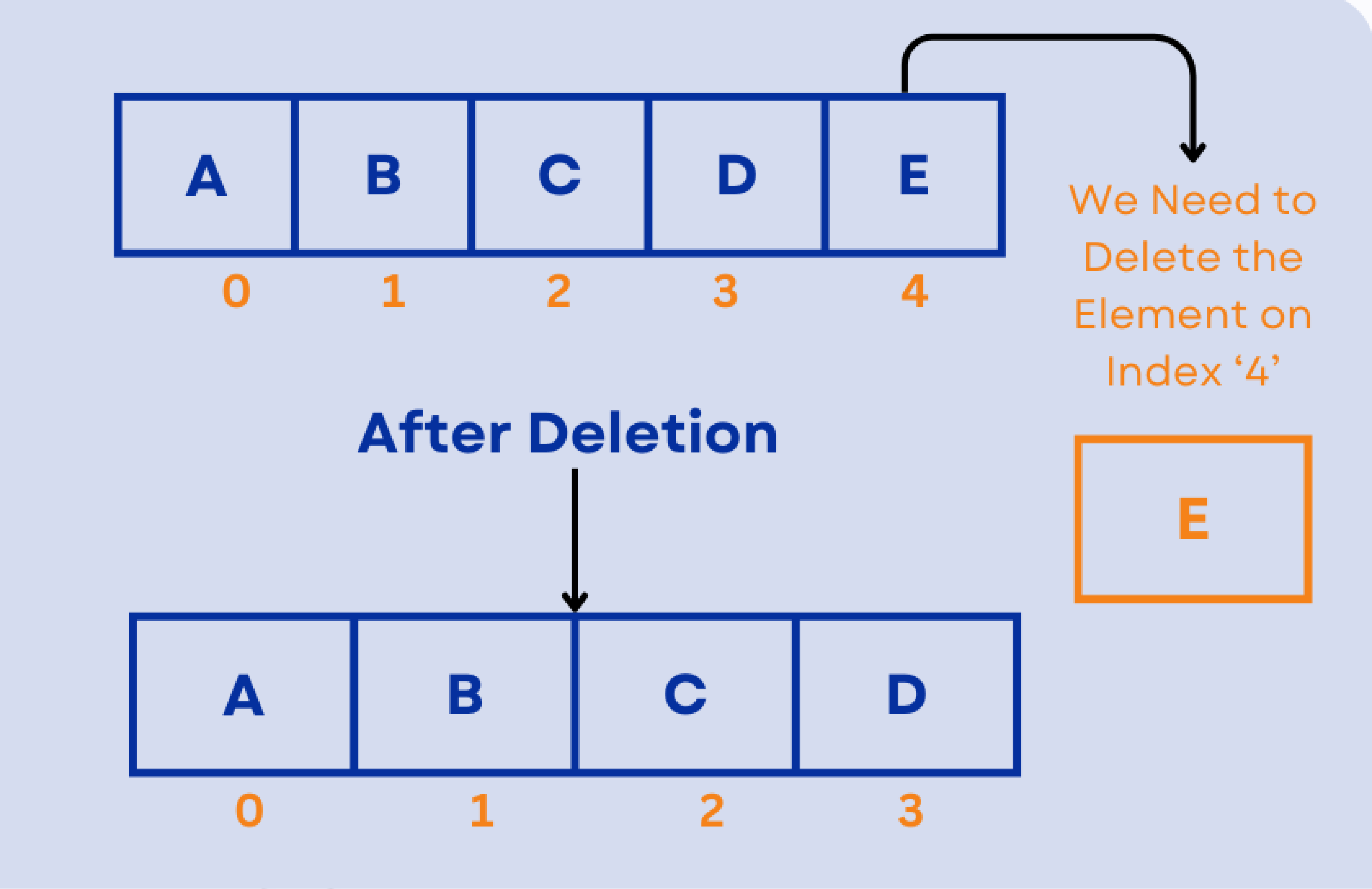
Insertion

Insertion operations add new data elements to a data structure. You can do this at the data structure's beginning, middle, or end.



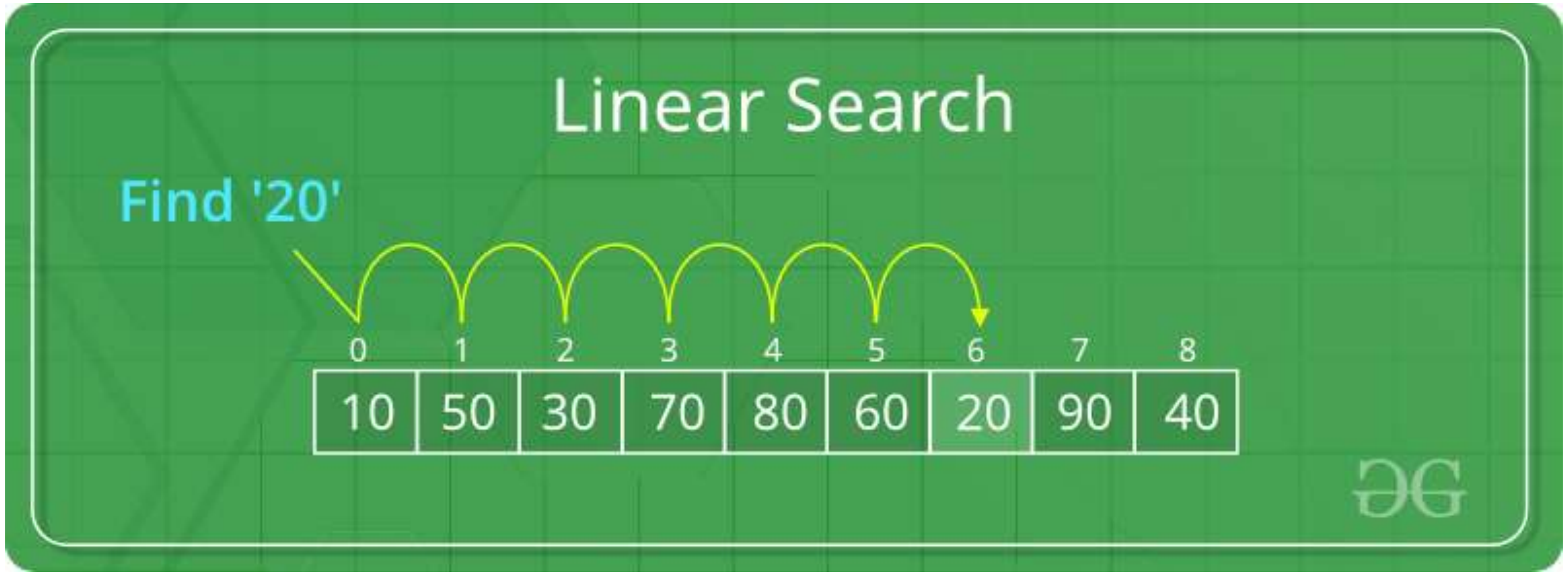
Deletion

Deletion operations remove data elements from a data structure. These operations are typically performed on nodes that are no longer needed.



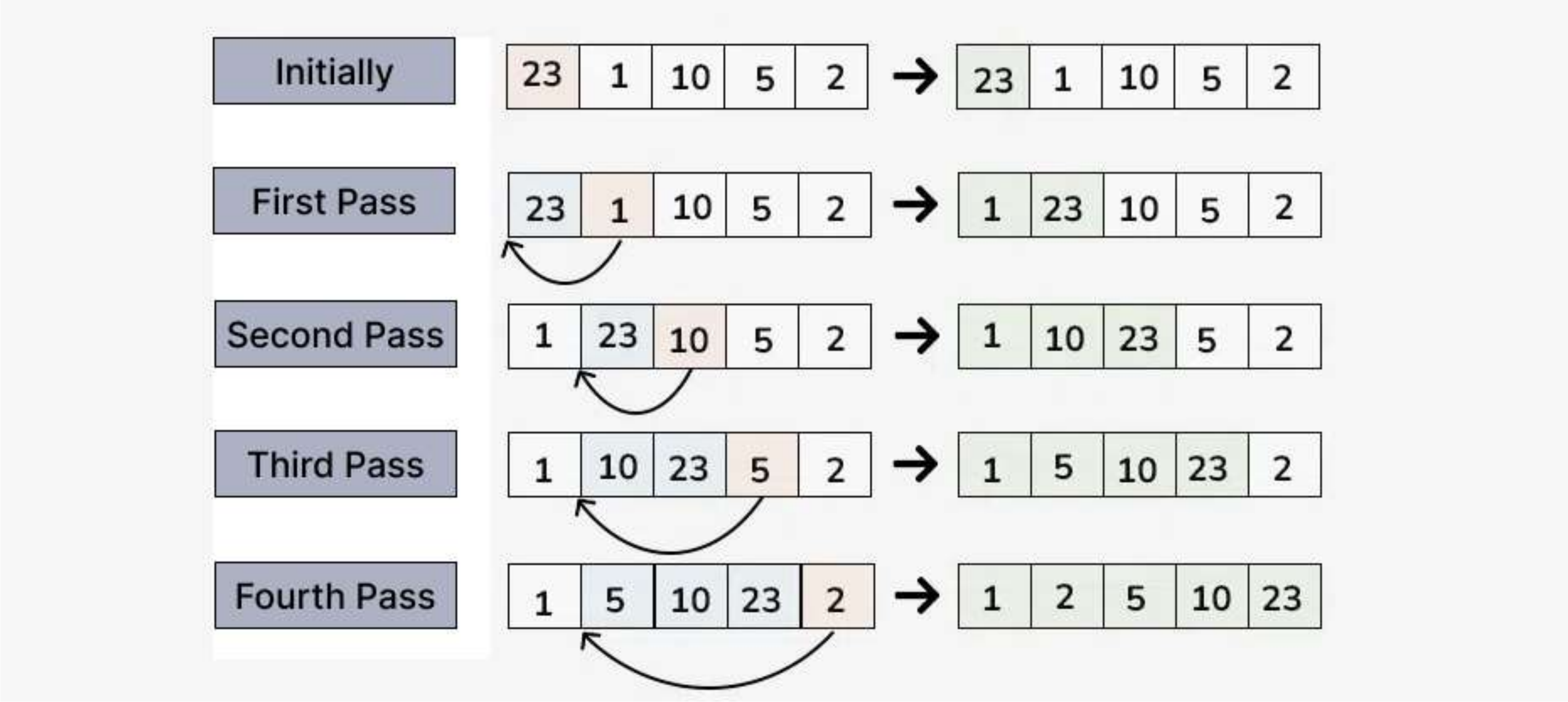
Search

Search operations are used to find a specific data element in a data structure. These operations typically employ a compare function to determine if two data elements are equal.



Sort

Sort operations are used to arrange the data elements in a data structure in a specific order. This can be done using various sorting algorithms, such as insertion sort, bubble sort, merge sort, and quick sort.



Initially 23 1 10 5 2 → 23 1 10 5 2

First Pass 23 1 10 5 2 → 1 23 10 5 2

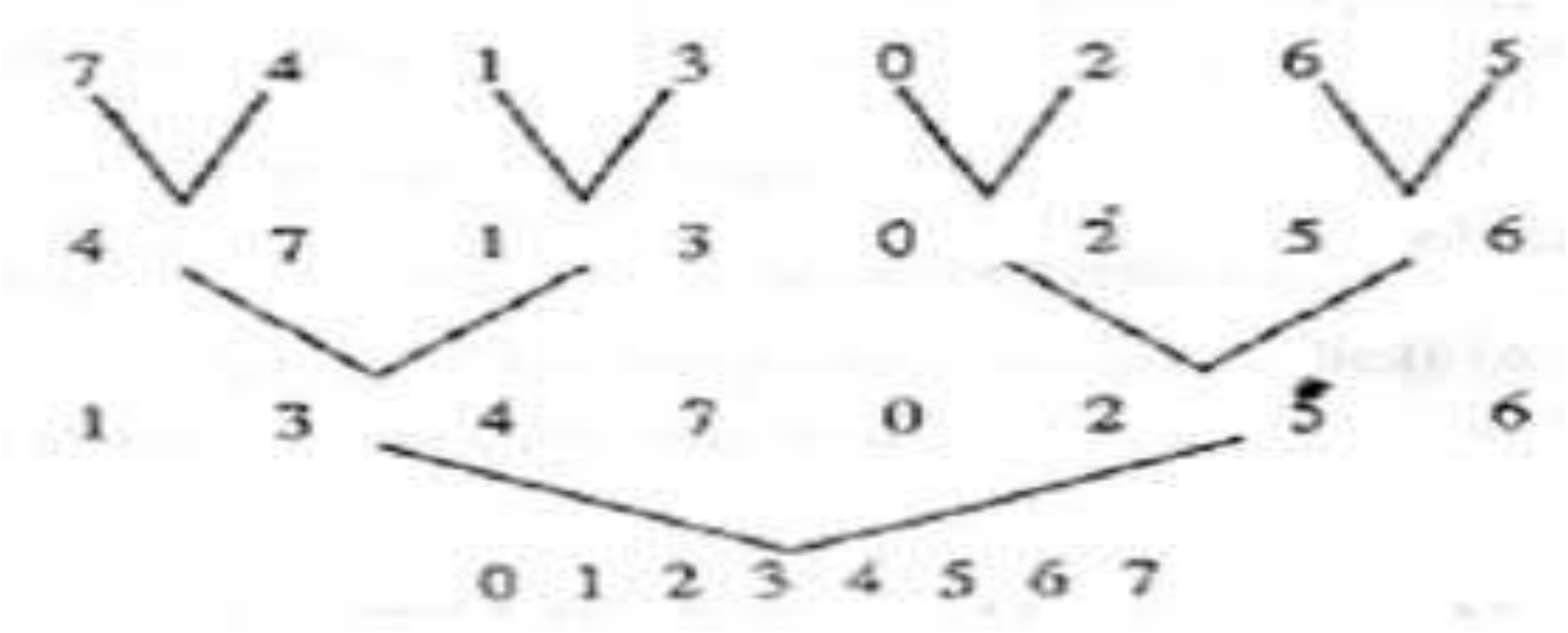
Second Pass 1 23 10 5 2 → 1 10 23 5 2

Third Pass 1 10 23 5 2 → 1 5 10 23 2

Fourth Pass 1 5 10 23 2 → 1 2 5 10 23

Merge

Merge operations are used to combine two data structures into one. This operation is typically used when two data structures need to be combined into a single structure.

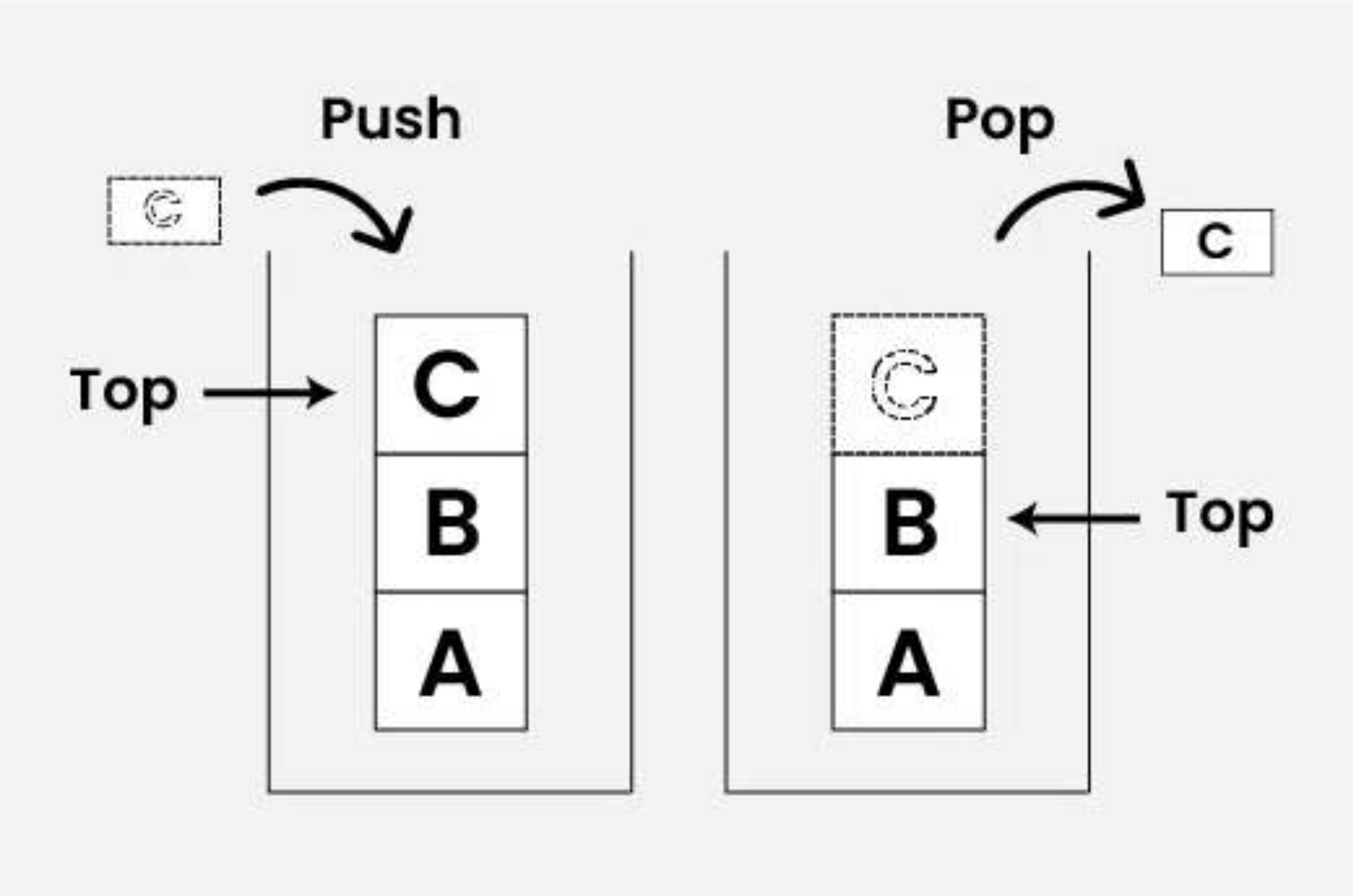


Copy

Copy operations are used to create a duplicate of a data structure. This can be done by copying each element in the original data structure to the new one.

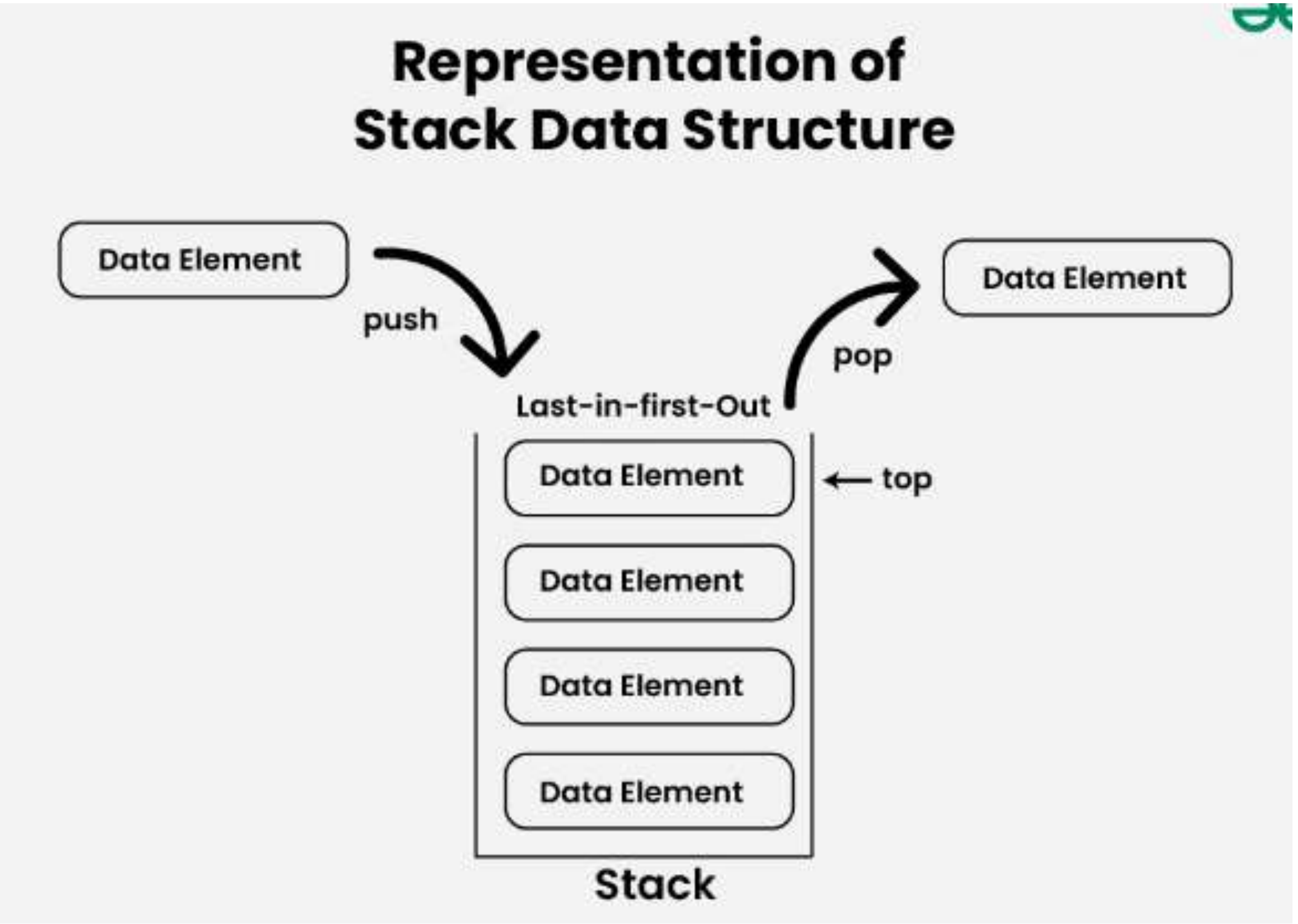
LINEAR DATA STRUCTURE : STACK

Stack is a linear data structure that follows **LIFO (Last In First Out) Principle**, the last element inserted is the first to be popped out. It means both insertion and deletion operations happen at one end only.



LIFO(LAST IN FIRST OUT) PRINCIPLE

Consider a stack of plates. When we add a plate, we add at the top. When we remove, we remove from the top.



TYPES OF STACK:

1. **Fixed Size Stack** : As the name suggests, a fixed size stack has a fixed size and cannot grow or shrink dynamically. If the stack is full and an attempt is made to add an element to it, an overflow error occurs. If the stack is empty and an attempt is made to remove an element from it, an underflow error occurs.
2. **Dynamic Size Stack** : A dynamic size stack can grow or shrink dynamically. When the stack is full, it automatically increases its size to accommodate the new element, and when the stack is empty, it decreases its size. This type of stack is implemented using a linked list, as it allows for easy resizing of the stack.

BASIC OPERATIONS ON STACK:

In order to make manipulations in a stack, there are certain operations provided to us.

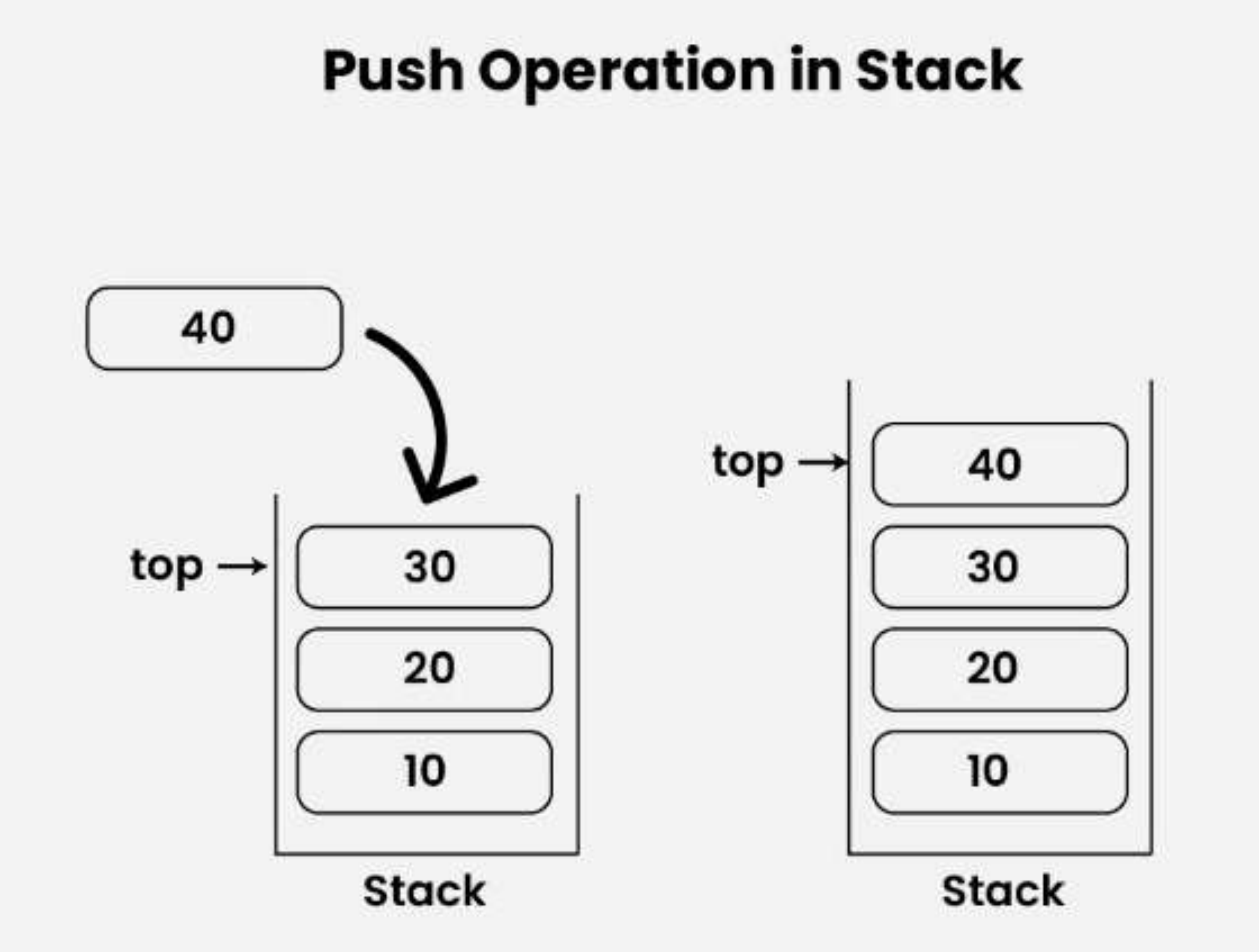
1. **push()** to insert an element into the stack
2. **pop()** to remove an element from the stack
3. **top()** Returns the top element of the stack.
4. **isEmpty()** returns true if stack is empty else false.
5. **isFull()** returns true if the stack is full else false.

PUSH OPERATION ON STACK

Adds an item to the stack. If the stack is full, then it is said to be an **Overflow condition.**

Algorithm for Push Operation:

1. Before pushing the element to the stack, we check if the stack is **full** .
2. If the stack is full **(top == capacity-1)** , then **Stack Overflows** and we cannot insert the element to the stack.
3. Otherwise, we increment the value of top by 1 **(top = top + 1)** and the new value is inserted at **top position** .
4. The elements can be pushed into the stack till we reach the **capacity** of the stack.


Push Operation in Stack
40
top →
30
20
10
Stack
top →
40
30
20
10
Stack

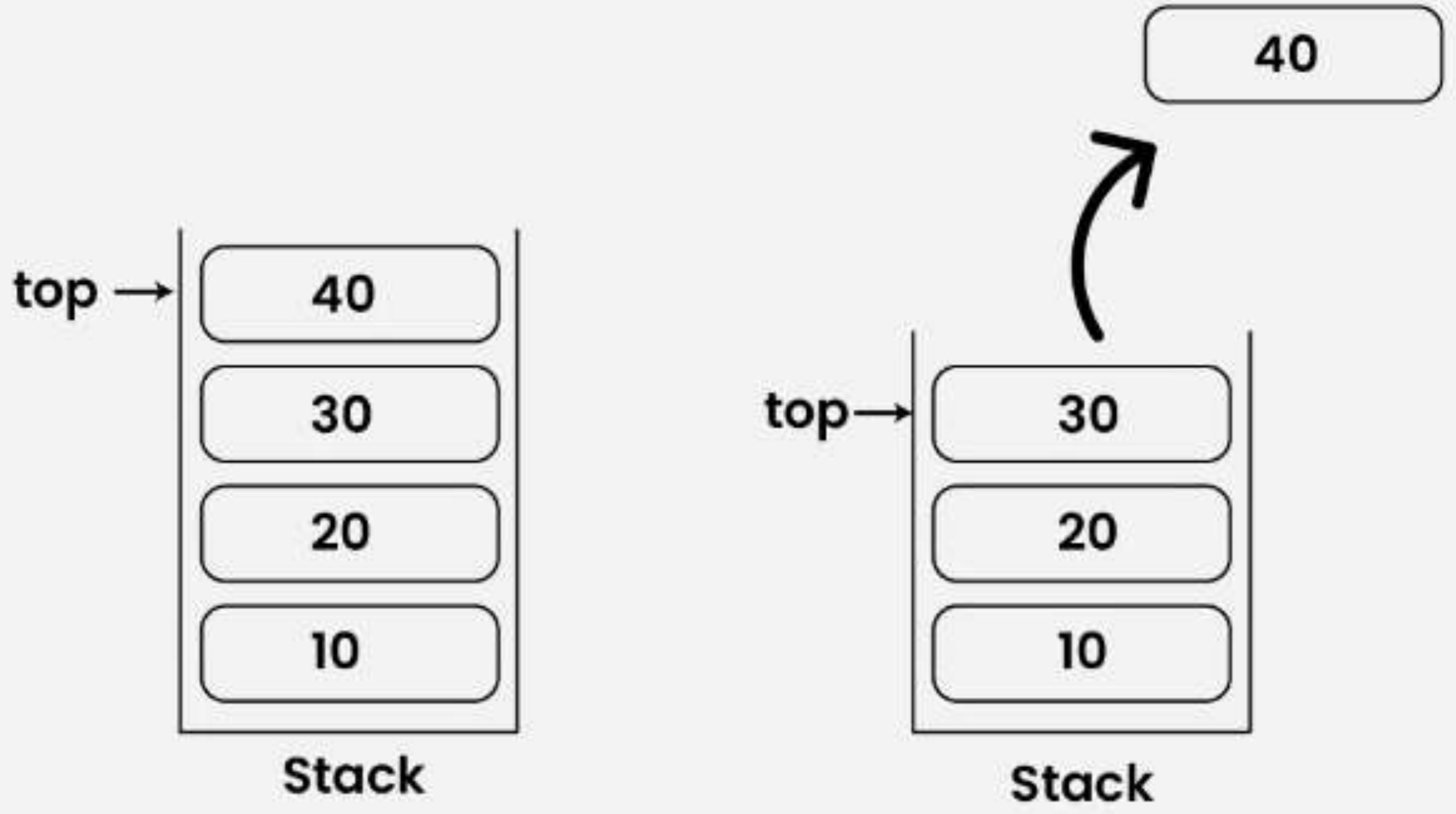
POP OPERATION IN STACK

Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an **Underflow condition.**

Algorithm for Pop Operation:

1. Before popping the element from the stack, we check if the stack is **empty** .
2. If the stack is empty (top == -1), then **Stack Underflows** and we cannot remove any element from the stack.
3. Otherwise, we store the value at top, decrement the value of top by 1 **(top = top – 1)** and return the stored top value.

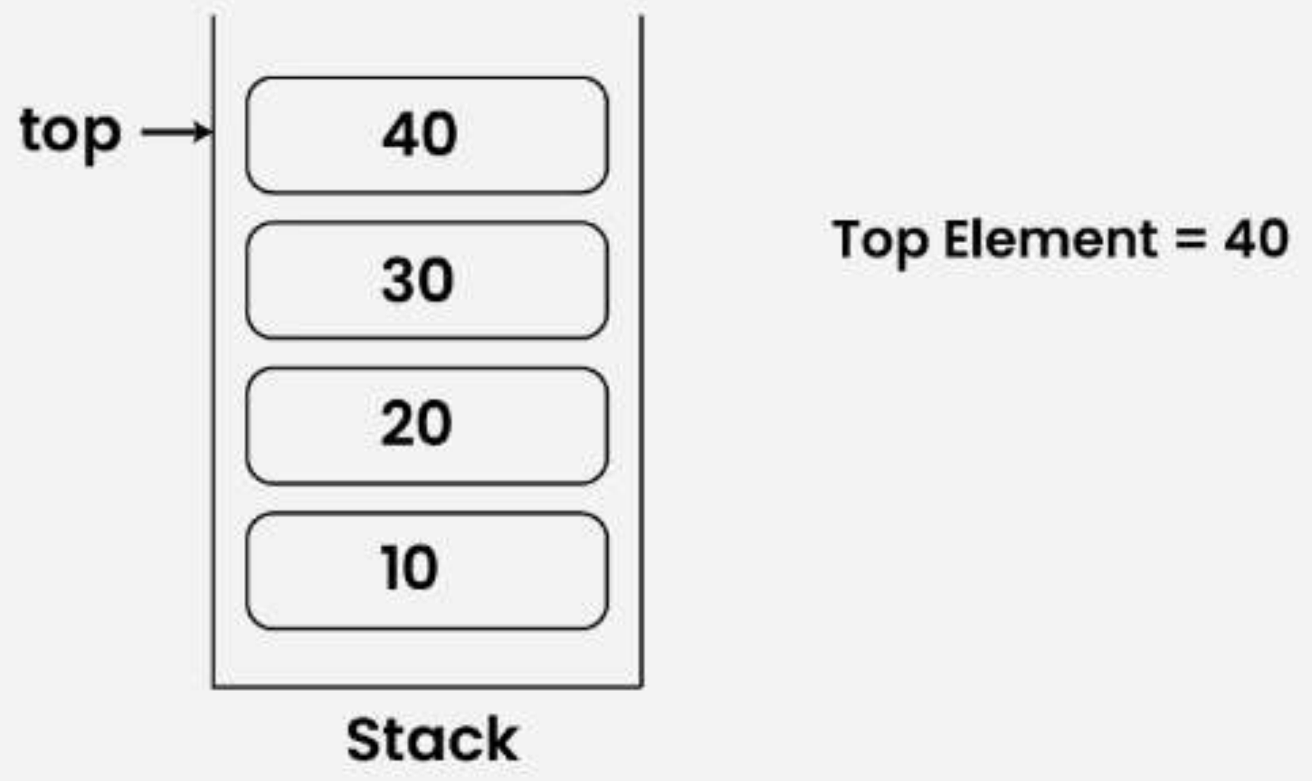
Pop Operation in Stack



TOP OR PEEK OPERATION ON STACK

1. Returns the top element of the stack.
2. **Algorithm for Top Operation:**
3. Before returning the top element from the stack, we check if the stack is empty.
4. If the stack is empty (top == -1), we simply print “Stack is empty”.
5. Otherwise, we return the element stored at **index = top** .

Top or Peek Operation in Stack

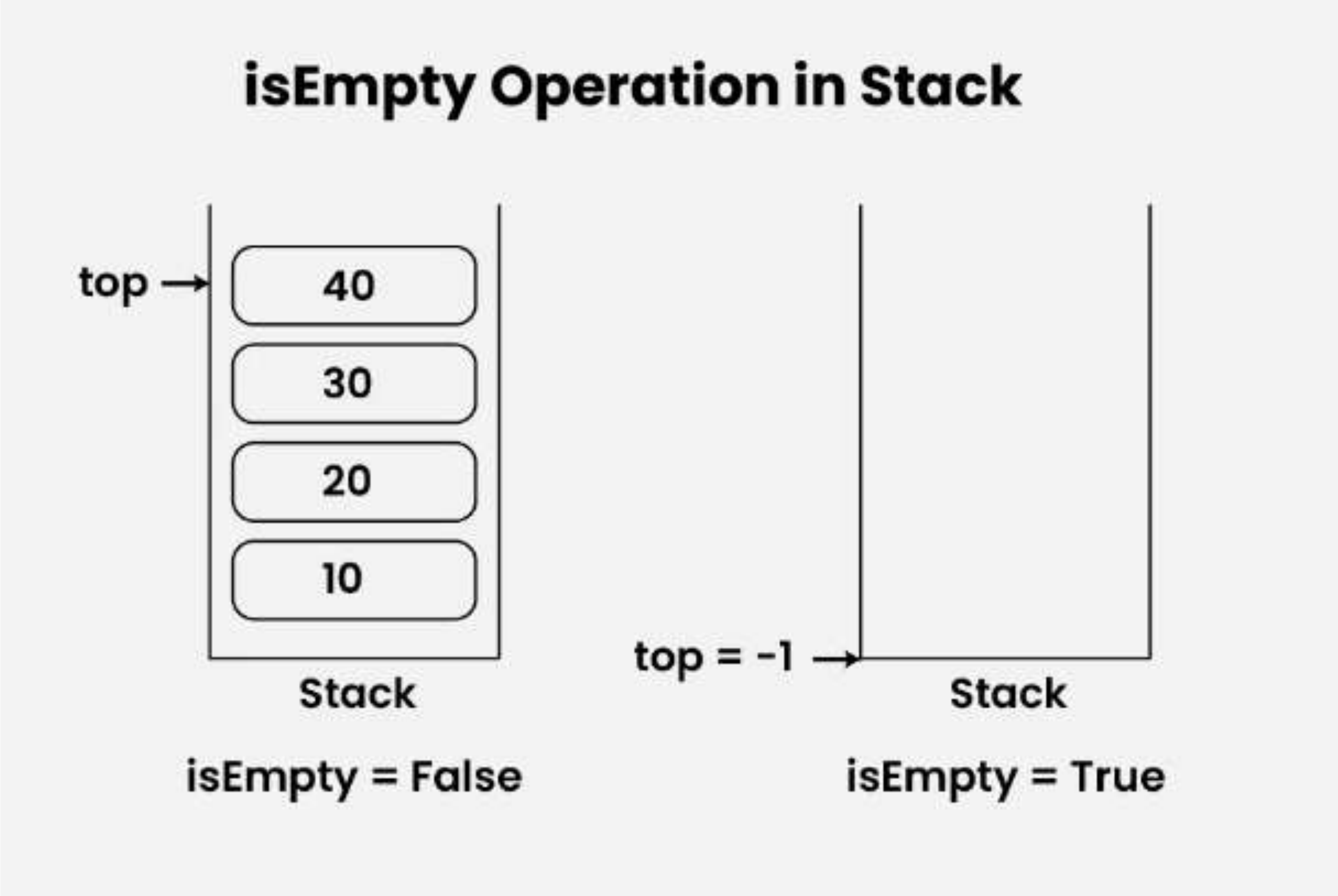


ISEMPTY OPERATION IN STACK DATA STRUCTURE:

Returns true if the stack is empty, else false.

Algorithm for isEmpty Operation:

1. Check for the value of **top** in stack.
2. If **(top == -1)**, then the stack is **empty** so return **true** .
3. Otherwise, the stack is not empty so return **false** .



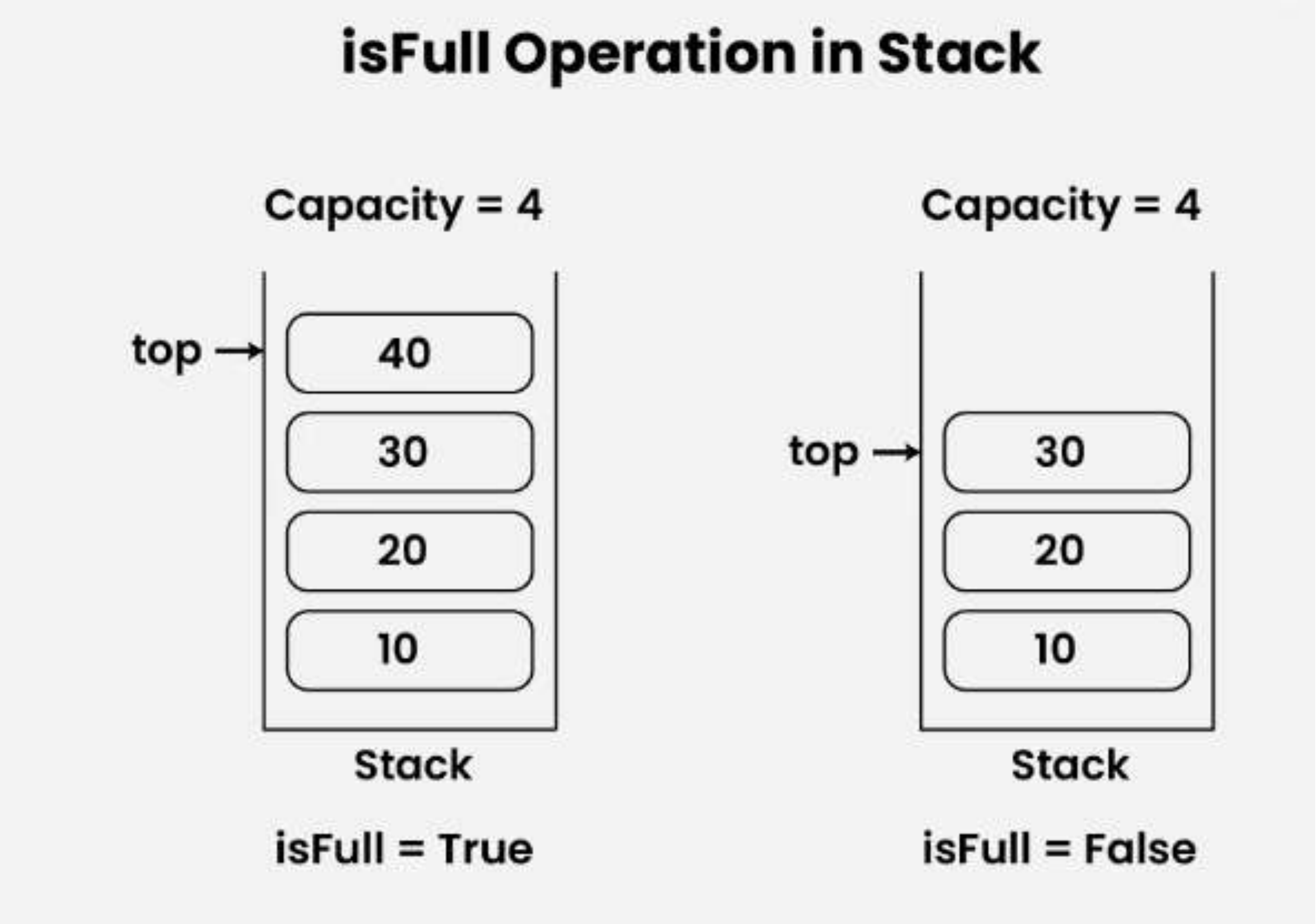
isEmpty Operation in Stack

ISFULL OPERATION IN STACK DATA STRUCTURE:

Returns true if the stack is full, else false.

Algorithm for isFull Operation:

1. Check for the value of **top** in stack.
2. If **(top == capacity-1),** then the stack is **full** so return **true**.
3. Otherwise, the stack is not full so return **false**.


isFull Operation in Stack
Capacity = 4
top →
40
30
20
10
Stack
isFull = True
Capacity = 4
top →
30
20
10
Stack
isFull = False

TIME COMPLEXITY OF STACK OPERATIONS

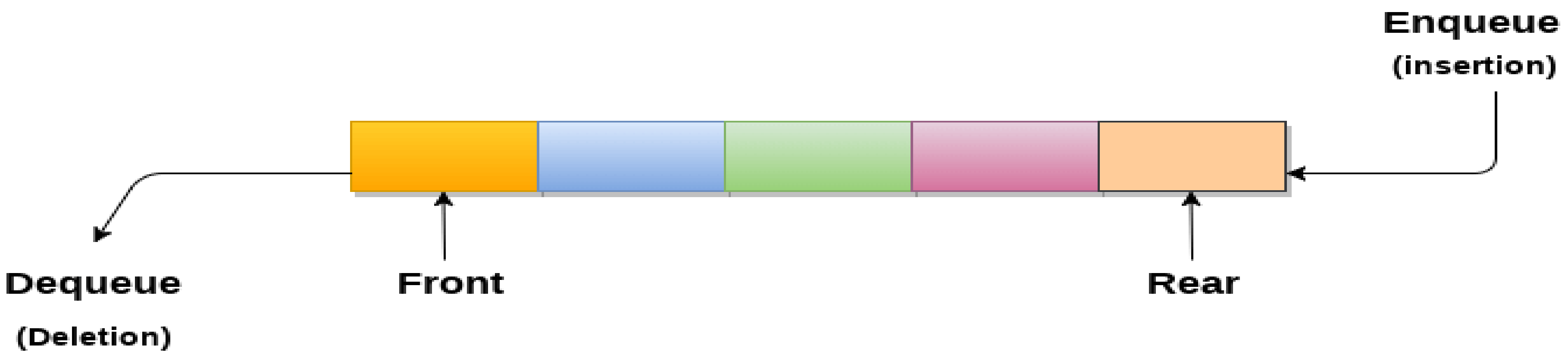
1. Only a single element can be accessed at a time in stacks.
2. While performing push() and pop() operations on the stack, it takes O(1) time.

QUEUE

1. A queue can be defined as an ordered list which enables insert operations to be performed at one end called **REAR** and delete operations to be performed at another end called **FRONT**.

2. Queue is referred to be as First In First Out list.

3. For example, people waiting in line for a rail ticket form a queue.

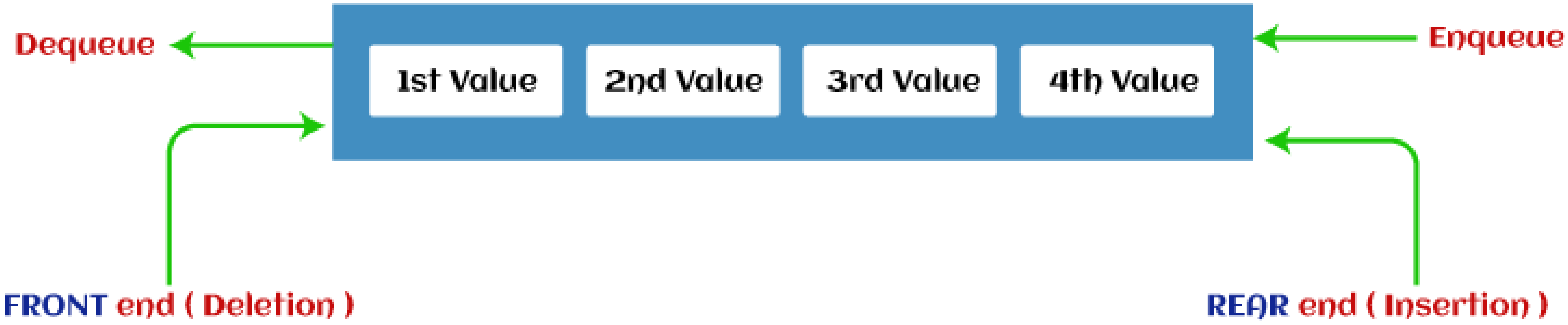


APPLICATIONS OF QUEUE

Due to the fact that queue performs actions on first in first out basis which is quite fair for the ordering of actions. There are various applications of queues discussed as below.

1. Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
2. Queues are used in asynchronous transfer of data (where data is not being transferred at the same rate between two processes) for eg. pipes, file IO, sockets.
3. Queues are used as buffers in most of the applications like MP3 media player, CD player, etc.
4. Queue are used to maintain the play list in media players in order to add and remove the songs from the play-list.
5. Queues are used in operating systems for handling interrupts.

THE REPRESENTATION OF THE QUEUE IS SHOWN IN THE BELOW IMAGE -



BASIC OPERATIONS WE CAN DO ON A QUEUE ARE:

1. **Enqueue:** Adds a new element to the queue.
2. **Dequeue:** Removes and returns the first (front) element from the queue.
3. **Peek:** Returns the first element in the queue.
4. **isEmpty:** Checks if the queue is empty.
5. **Size:** Finds the number of elements in the queue.

QUEUE IMPLEMENTATION USING ARRAYS

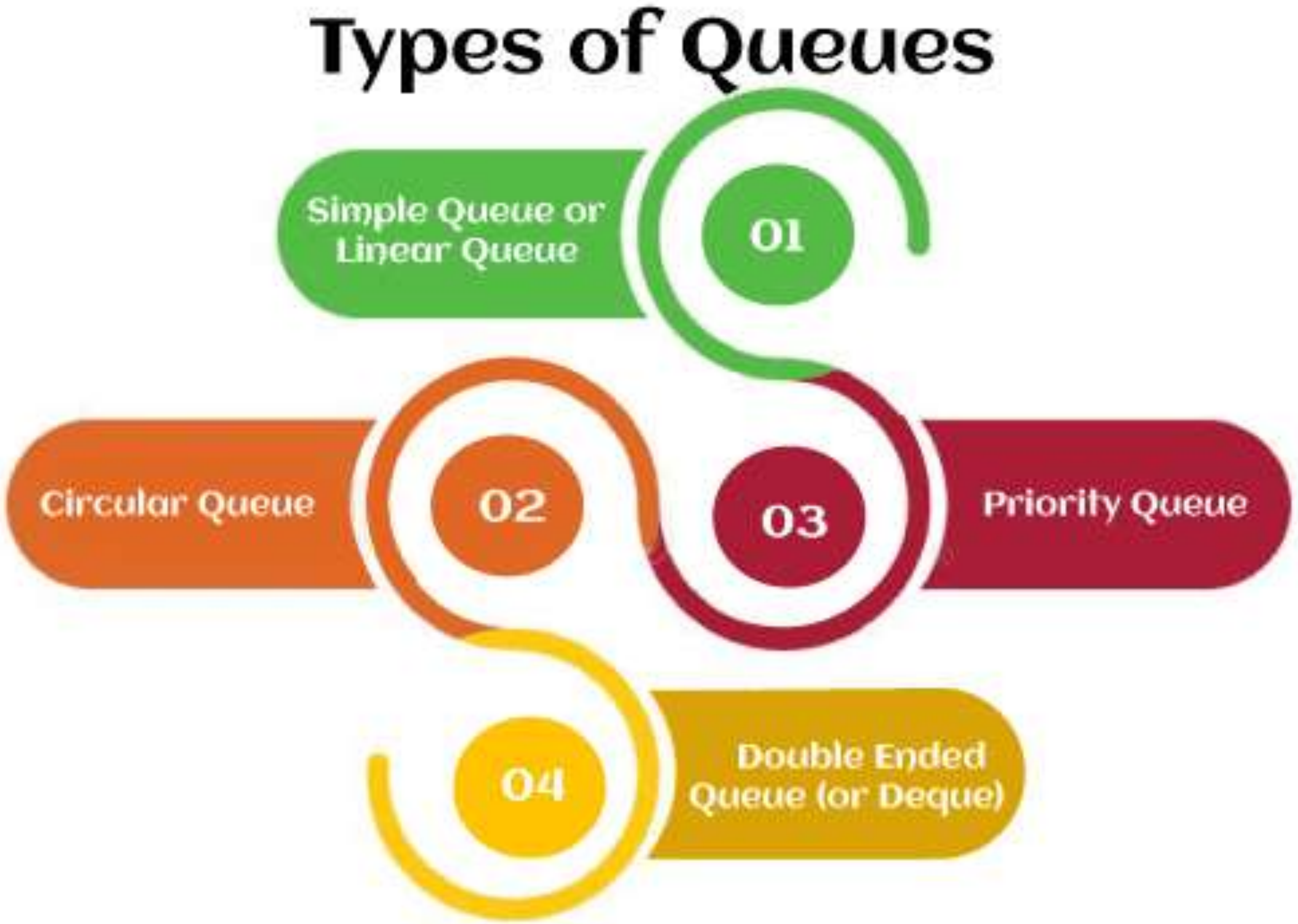
Reasons to implement queues using arrays:

1. **Memory Efficient:** Array elements do not hold the next elements address like linked list nodes do.
2. **Easier to implement and understand:** Using arrays to implement queues require less code than using linked lists, and for this reason it is typically easier to understand as well.

Reasons for **not** using arrays to implement queues:

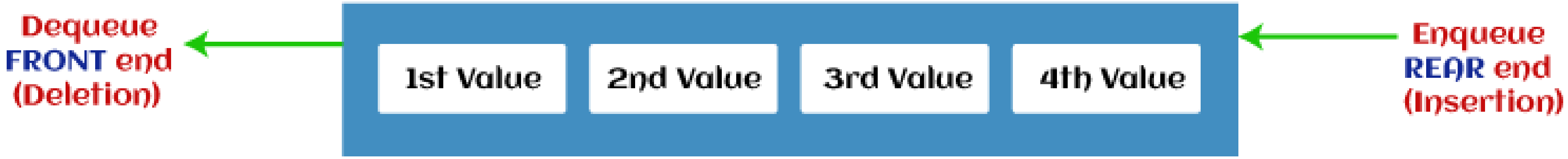
1. **Fixed size:** An array occupies a fixed part of the memory. This means that it could take up more memory than needed, or if the array fills up, it cannot hold more elements. And resizing an array can be costly.
2. **Shifting cost:** Dequeue causes the first element in a queue to be removed, and the other elements must be shifted to take the removed elements' place. This is inefficient and can cause problems, especially if the queue is long.
3. **Alternatives:** Some programming languages have built-in data structures optimized for queue operations that are better than using arrays.

TYPES OF QUEUE



SIMPLE QUEUE OR LINEAR QUEUE

In Linear Queue, an insertion takes place from one end while the deletion occurs from another end. The end at which the insertion takes place is known as the rear end, and the end at which the deletion takes place is known as front end. It strictly follows the FIFO rule.



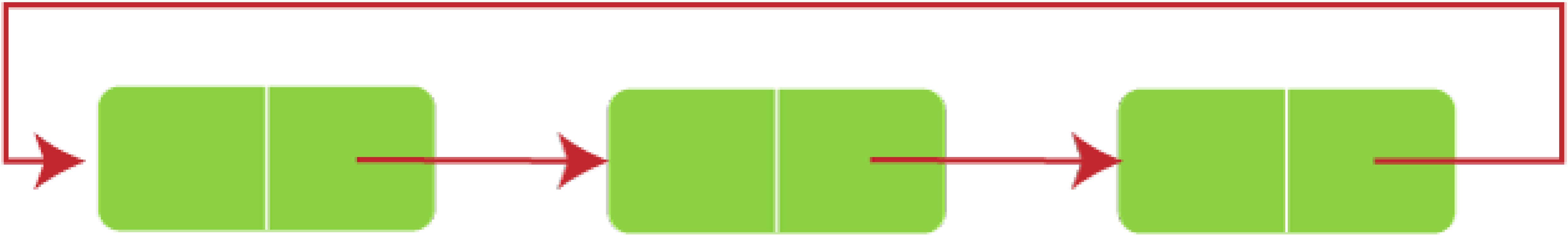
The major drawback of using a linear Queue is that insertion is done only from the rear end.

If the first three elements are deleted from the Queue, we cannot insert more elements even though the space is available in a Linear Queue.

In this case, the linear Queue shows the overflow condition as the rear is pointing to the last element of the Queue.

CIRCULAR QUEUE

In Circular Queue, all the nodes are represented as circular. It is similar to the linear Queue except that the last element of the queue is connected to the first element. It is also known as Ring Buffer, as all the ends are connected to another end. The representation of circular queue is shown in the below image



Circular Queue

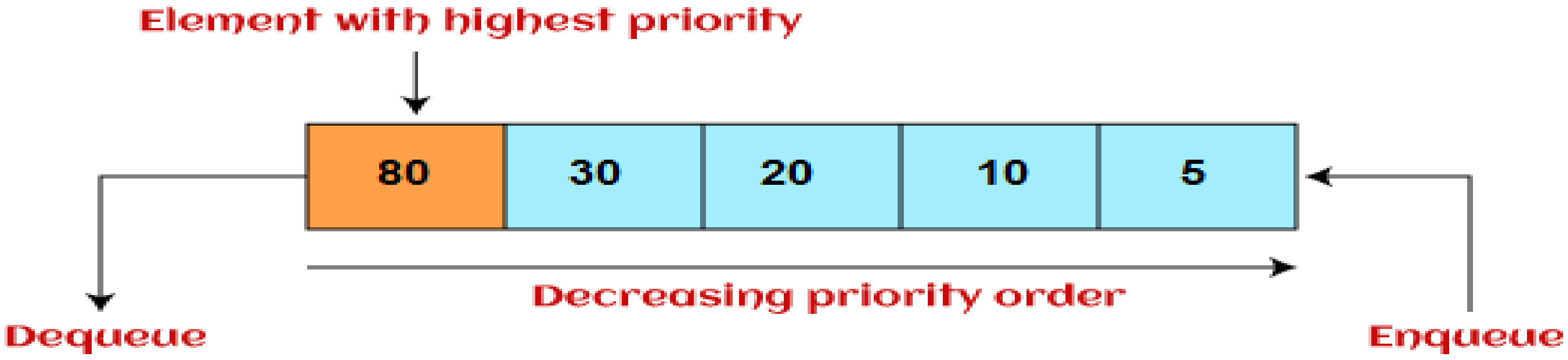
The drawback that occurs in a linear queue is overcome by using the circular queue.

If the empty space is available in a circular queue, the new element can be added in an empty space by simply incrementing the value of rear.

The main advantage of using the circular queue is better memory utilization.

PRIORITY QUEUE

It is a special type of queue in which the elements are arranged based on the priority. It is a special type of queue data structure in which every element has a priority associated with it. Suppose some elements occur with the same priority, they will be arranged according to the FIFO principle. The representation of priority queue is shown in the below image -



Insertion in priority queue takes place based on the arrival, while deletion in the priority queue occurs based on the priority. Priority queue is mainly used to implement the CPU scheduling algorithms.

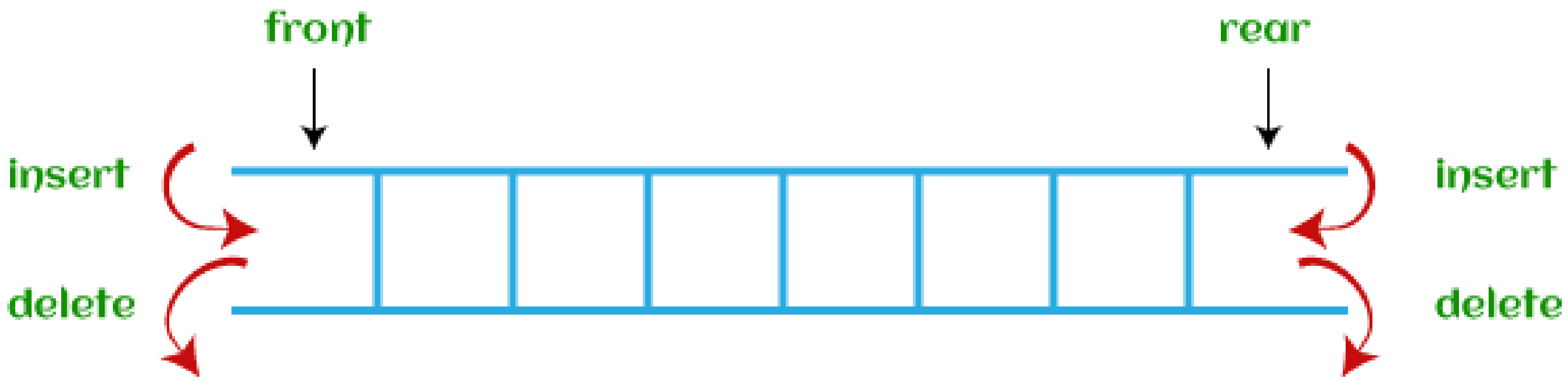
There are two types of priority queue that are discussed as follows -

1. **Ascending priority queue -** In ascending priority queue, elements can be inserted in arbitrary order, but only smallest can be deleted first. Suppose an array with elements 7, 5, and 3 in the same order, so, insertion can be done with the same sequence, but the order of deleting the elements is 3, 5, 7.

2. **Descending priority queue -** In descending priority queue, elements can be inserted in arbitrary order, but only the largest element can be deleted first. Suppose an array with elements 7, 3, and 5 in the same order, so, insertion can be done with the same sequence, but the order of deleting the elements is 7, 5, 3.

DEQUE (OR, DOUBLE ENDED QUEUE)

In Deque or Double Ended Queue, insertion and deletion can be done from both ends of the queue either from the front or rear. It means that we can insert and delete elements from both front and rear ends of the queue. Deque can be used as a palindrome checker means that if we read the string from both ends, then the string would be the same.



double ended queue

WAYS TO IMPLEMENT THE QUEUE

Implementation using array: The sequential allocation in a Queue can be implemented using an array.

Implementation using Linked list: The linked list allocation in a Queue can be implemented using a linked list.

ARRAY IMPLEMENTATION OF STACK

In array implementation, the stack is formed by using the array. All the operations regarding the stack are performed using arrays.

Adding an element

```
begin
    if top = n then stack full
    top = top + 1
    stack (top) : = item;
end
```

DELETION OF AN ELEMENT FROM A STACK (POP OPERATION)

```
begin
    if top = 0 then stack empty;
    item := stack(top);
    top = top - 1;
end;
```

RECURSION

Recursion is one of the most significant algorithms because if we can solve a smaller work, we can almost certainly solve the entire project utilizing the smaller jobs.

Recursion is a term that refers to the act of calling oneself.

It has a base case, which is the major case, in which it handles the smaller problem scenario before calling itself for the minor sections

It uses the fact that while we are standing in a certain state, we presume that our recursive function has done processing for smaller responses, which we can now combine to solve our present state.

When a recursive function is invoked, its parameters are stored in a data structure called activation records.

Every time the recursive function is invoked, a new activation record is generated and stored in the memory.

These activation records are stored in the special stack called the recursive stack. These activation records are deleted when the function execution is completed.

So, when a recursive function is invoked, it generates the activation records for different values of the recursive function hence, extending the recursion stack size.

When the recursive function execution is completed one by one its activation records get deleted hence, contracting the recursive stack size. The recursive stack size depends on the number of activation records created and deleted.

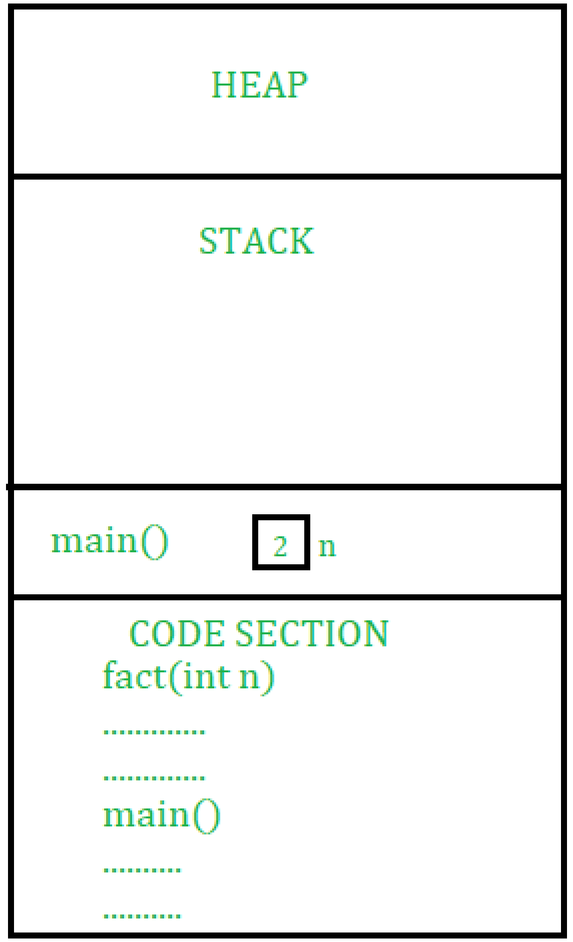
```
 int fact(int n)
{
   if (n == 1)
       return 1;
   return fact(n - 1);
}
 int main()
{
   int p;
   p = fact(2);
   return 0;
}
```

Time complexity: O(n)
Auxiliary space: O(n) // because we are using stack.

ILLUSTRATION:

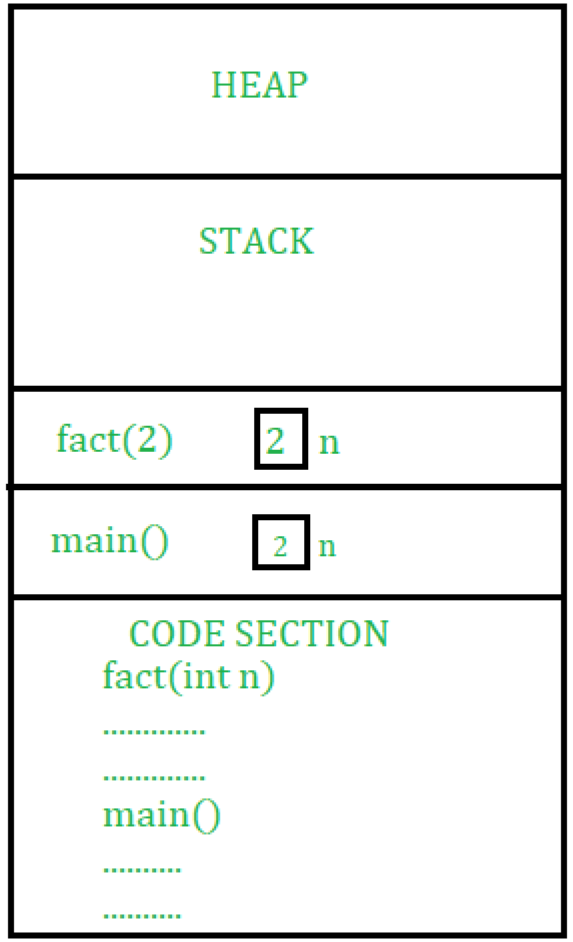
FOR THE ABOVE PROGRAM. FIRSTLY, THE ACTIVATION RECORD FOR MAIN STACK IS GENERATED AND STORED IN THE STACK.

In the above program, there is a recursive function fact that has n as the local parameter. In the above example program, n=2 is passed in the recursive function call.



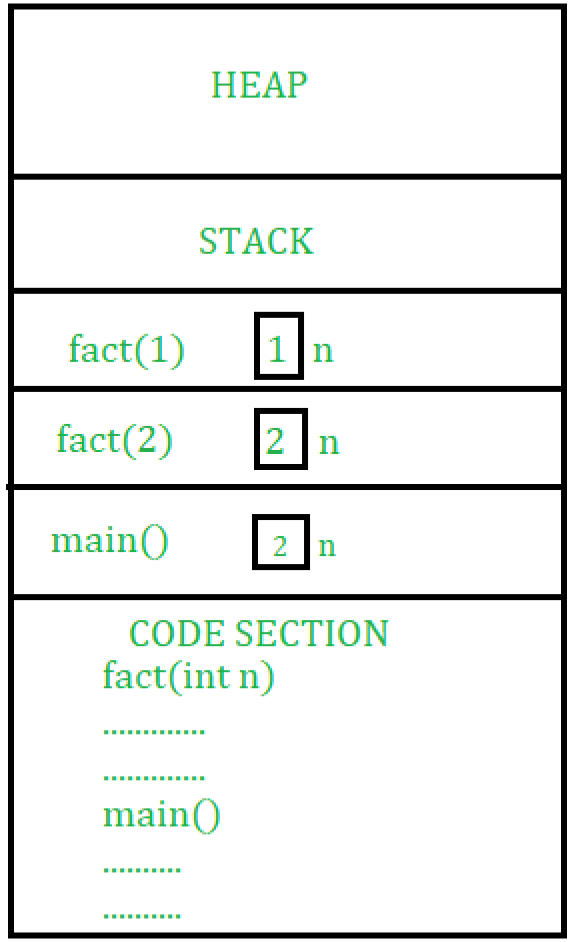
Initial state

FIRST STEP: FIRST, THE FUNCTION IS INVOKED FOR N =2 AND ITS ACTIVATION RECORD ARE CREATED IN THE RECURSIVE STACK.



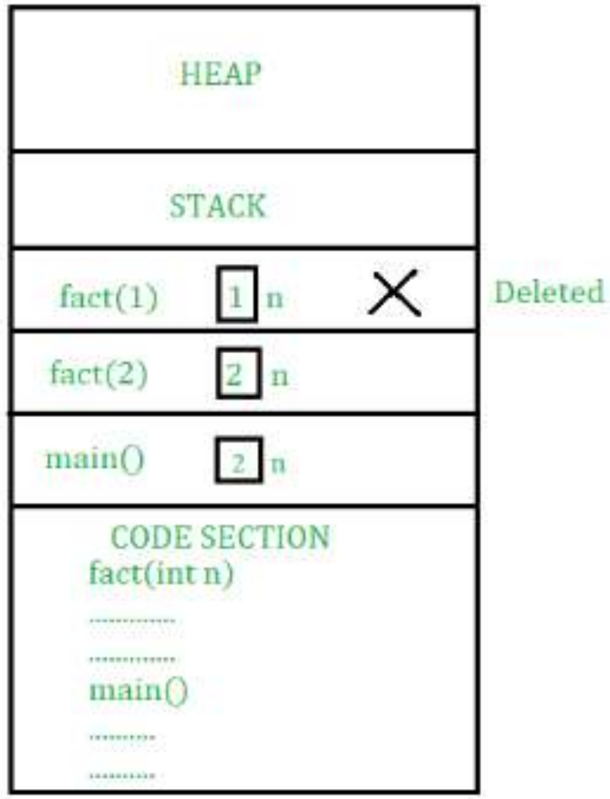
1st step

2ND STEP: THEN ACCORDING TO THE RECURSIVE FUNCTION, IT IS INVOKED FOR N=1 AND ITS ACTIVATION RECORD IS CREATED IN THE RECURSIVE STACK.



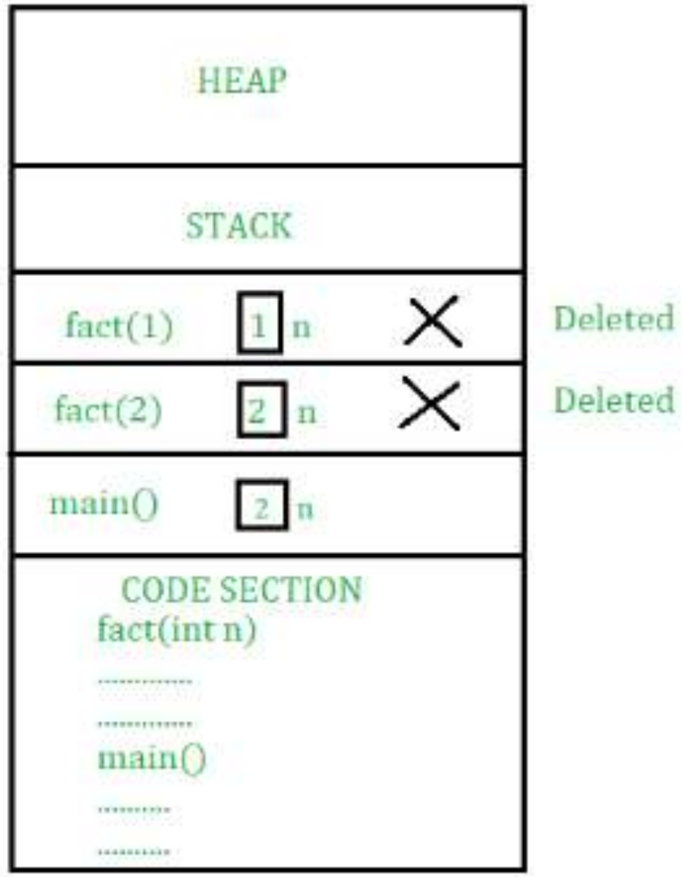
2nd step

3RD STEP: AFTER THE EXECUTION OF THE FUNCTION FOR VALUE N=1 AS IT IS A BASE CONDITION, ITS EXECUTION GETS COMPLETED AND ITS ACTIVATION RECORD GETS DELETED.



3rd step

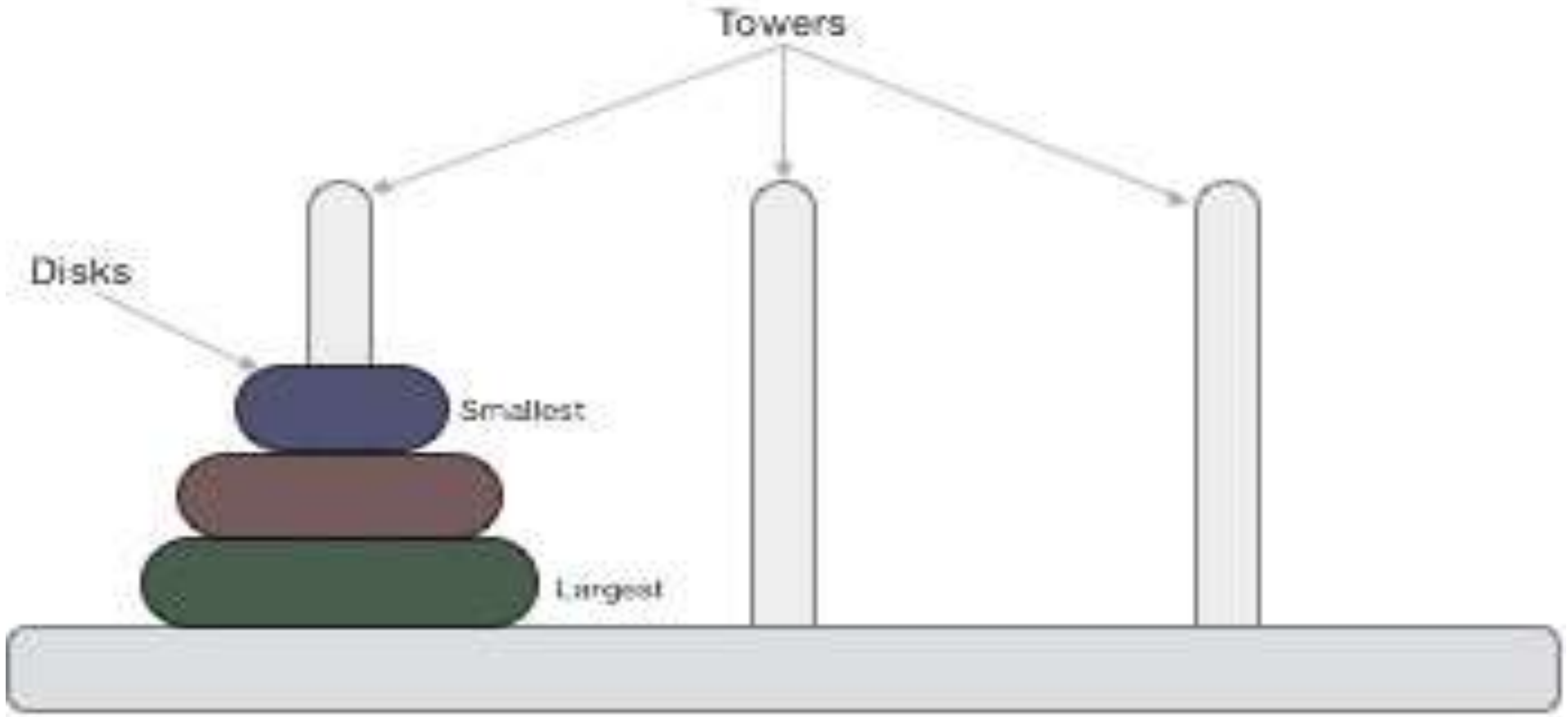
4TH STEP: SIMILARLY, THE FUNCTION FOR VALUE N=2(ITS PREVIOUS FUNCTION) GETS EXECUTED AND ITS ACTIVATION RECORD GETS DELETED. IT COMES OUT FROM THE RECURSIVE FUNCTION TO THE MAIN FUNCTION.



So, in the above example for recursive function fact and value, n=2 recursive stack size excluding the main function is 2. Hence, for value n recursive stack size is n.

TOWER OF HANOI USING RECURSION

Tower of Hanoi, is a mathematical puzzle which consists of three towers (pegs) and more than one rings is as depicted



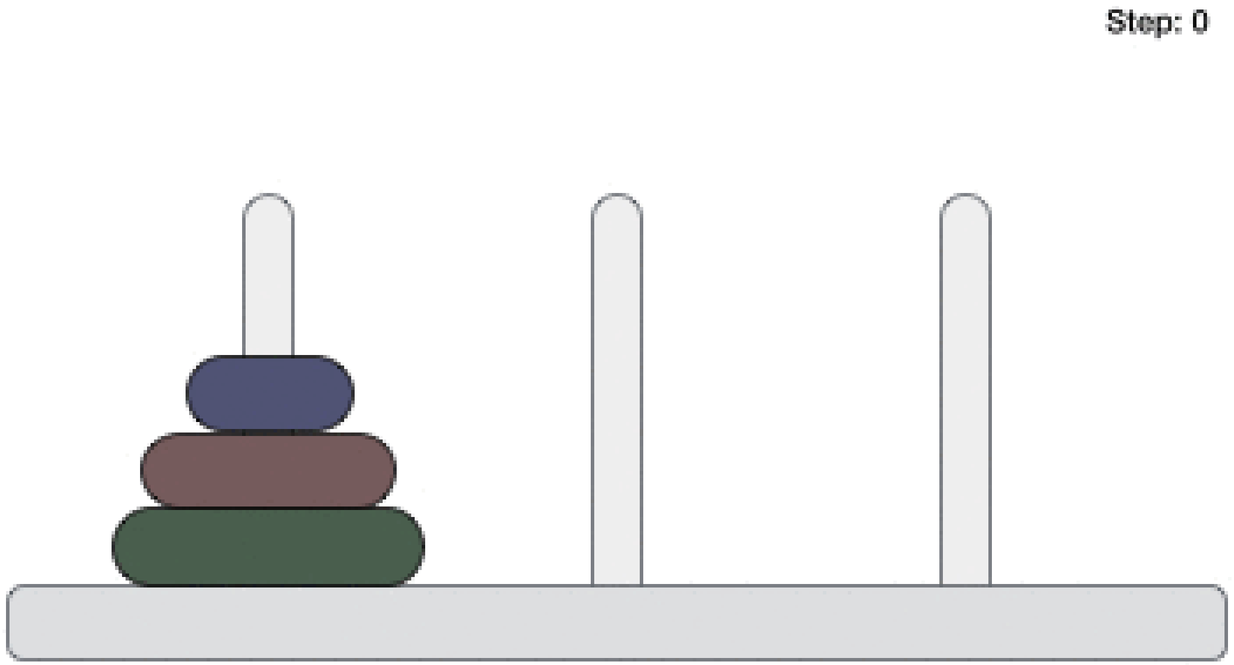
These rings are of different sizes and stacked upon in an ascending order, i.e. the smaller one sits over the larger one. There are other variations of the puzzle where the number of disks increase, but the tower count remains the same.

Rules

The mission is to move all the disks to some another tower without violating the sequence of arrangement. A few rules to be followed for Tower of Hanoi are −

Only one disk can be moved among the towers at any given time.
Only the "top" disk can be removed.
No large disk can sit over a small disk.

TOWER OF HANOI PUZZLE WITH N DISKS CAN BE SOLVED IN MINIMUM $\mathbf{2^N-1}$ STEPS.
THIS PRESENTATION SHOWS THAT A PUZZLE WITH 3 DISKS HAS TAKEN $\mathbf{2^3 - 1 = 7}$ STEPS.



CONVERT INFIX TO POSTFIX NOTATION

An infix and postfix are the expressions. An expression consists of constants, variables, and symbols. Symbols can be operators or parenthesis.

All these components must be arranged according to a set of rules so that all these expressions can be

Examples of expressions are:

5 + 6

A - B

(P * 5)

WHAT IS INFIX NOTATION?

When the operator is written in between the operands, then it is known as **infix notation**. Operand does not have to be always a constant or a variable; it can also be an expression itself.

For example,

(p + q) * (r + s)

In the above expression, both the expressions of the multiplication operator are the operands, i.e., **(p + q)**, and **(r + s)** are the operands.

In the above expression, there are three operators. The operands for the first plus operator are p and q, the operands for the second plus operator are r and s.

While performing the **operations on the expression, we need to follow some set of rules to evaluate the result. In the** above expression, addition operation would be performed on the two expressions, i.e., p+q and r+s, and then the multiplication operation would be performed.

Syntax of infix notation is given below:

<operand> <operator> <operand>

POSTFIX EXPRESSION

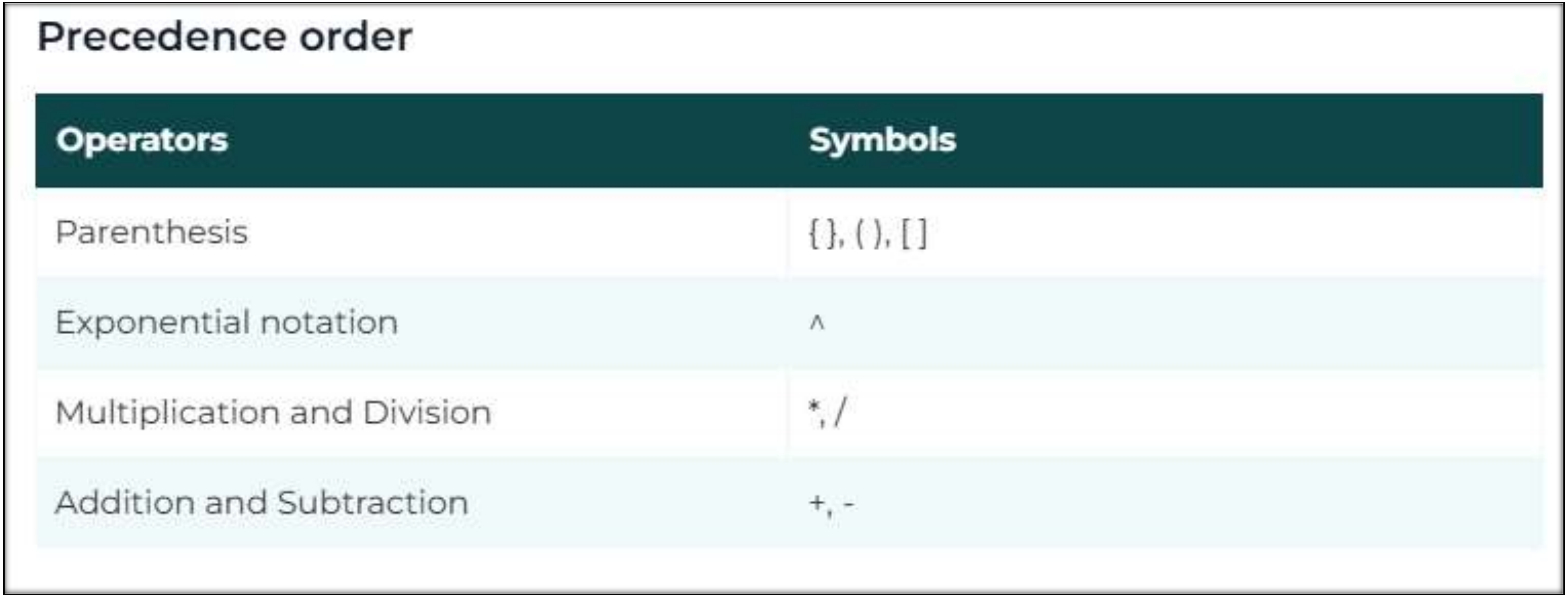
The postfix expression is an expression in which the operator is written after the operands. For example, the postfix expression of infix notation (2+3) can be written as 23+.

Some key points regarding the postfix expression are:

1. In postfix expression, operations are performed in the order in which they have written from left to right.
2. It does not any require any parenthesis.
3. We do not need to apply operator precedence rules and associativity rules.

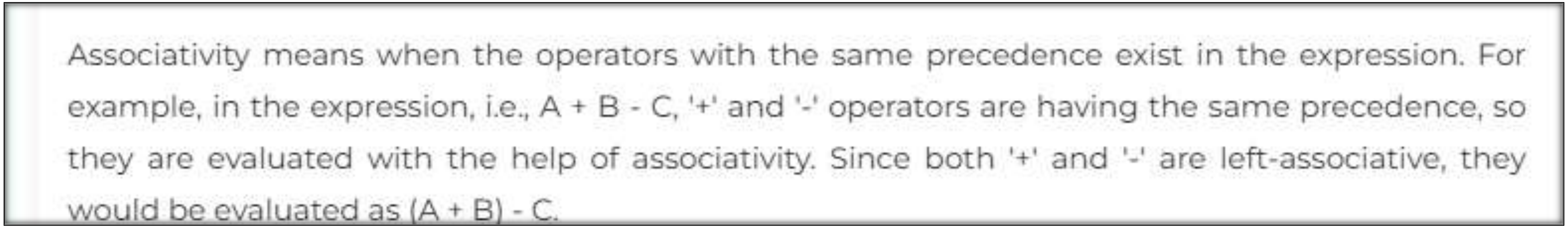
ALGORITHM TO EVALUATE THE POSTFIX EXPRESSION

1. Scan the expression from left to right until we encounter any operator.
2. Perform the operation
3. Replace the expression with its computed value.
4. Repeat the steps from 1 to 3 until no more operators exist.

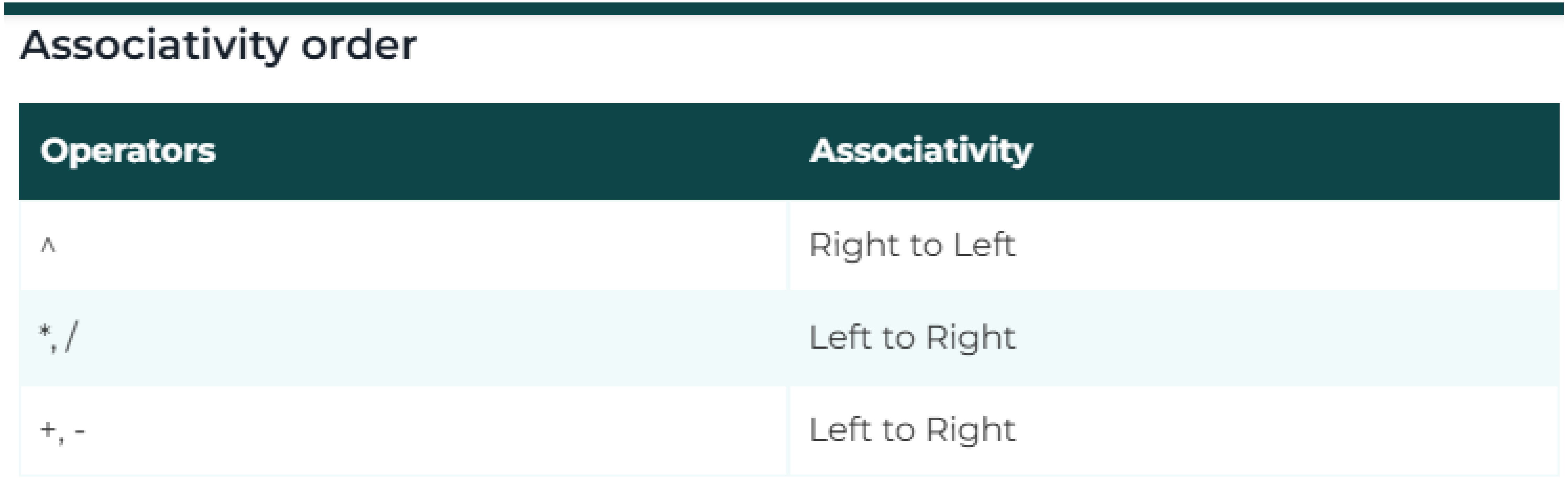


Precedence order

Operators	Symbols
Parenthesis	{ }, ( ), [ ]
Exponential notation	^
Multiplication and Division	*, /
Addition and Subtraction	+, -



Associativity means when the operators with the same precedence exist in the expression. For example, in the expression, i.e., A + B - C, '+' and '-' operators are having the same precedence, so they are evaluated with the help of associativity. Since both '+' and '-' are left-associative, they would be evaluated as (A + B) - C.



Associativity order

Operators	Associativity
^	Right to Left
*, /	Left to Right
+, -	Left to Right

CONVERSION OF INFIX TO PREFIX USING STACK

Priority:	
Highest Priority	: ^
Next	: * / (Same priority)
Next	: + - (same priority)

Rules:

1. No two operator of same priority should be in the stack
2. Highest Priority can’t be placed before the lowest priority
3. Operators within the brackets should be popped out

HOW TO EVALUATE POSTFIX?

- Going from left to right, if you see an operator, apply it to the previous two operands (numbers)
- Example:

$$\underbrace{\underbrace{2\quad \underbrace{\underbrace{10\quad 4\quad *}_{40}\quad 5\quad /}_{8}\quad +}_{10}\quad \underbrace{9\quad 3\quad -}_{6}\quad -}_{4}$$

- Equivalent infix: 2 + 10 * 4 / 5 – (9 – 3)

COMPUTER EVALUATION OF POSTFIX

- Going left to right,
  - If you see a number, put it on the stack
  - If you see an operator, pop two numbers from the stack, do the operation, and put the result back on the stack
    - (The top number on the stack is the operand on the right)

2	10	4	*	5	/	+	9	3	-	-
		4		5				3		
	10	10	40	40	8		9	9	6	
2	2	2	2	2	2	10	10	10	10	4

- The result is the only thing on the stack when done
  - If there's more than one thing, there was an error in the expression

FROM INFIX TO POSTFIX

- Figure out, using the infix rules, which operation to perform next
- Write the new operand or operands in their correct places
- Write the operator at the end
- Postfix does *not* use parentheses, but we'll put them in temporarily to help show the way things are grouped

Step 1: Initially the stack will be empty.

Step 2: Assign an opening and closing paranthesis in the beginning and the end of the expression.

Step 3: Read the character of the expression one by one from left to right.

Step 4: If the character scanned is an operand, print the operand in the result.
Step 5: If the character scanned is an open paranthesis, push it inside the stack.

Step 6: If the character scanned is an operator with higher precedence than the operator in the top of the stack, push the operator into the stack.

Step 7: If the character scanned is a closing paranthesis, pop all the elements one by one from the stack and print it in the result until an opening paranthesis is encountered.

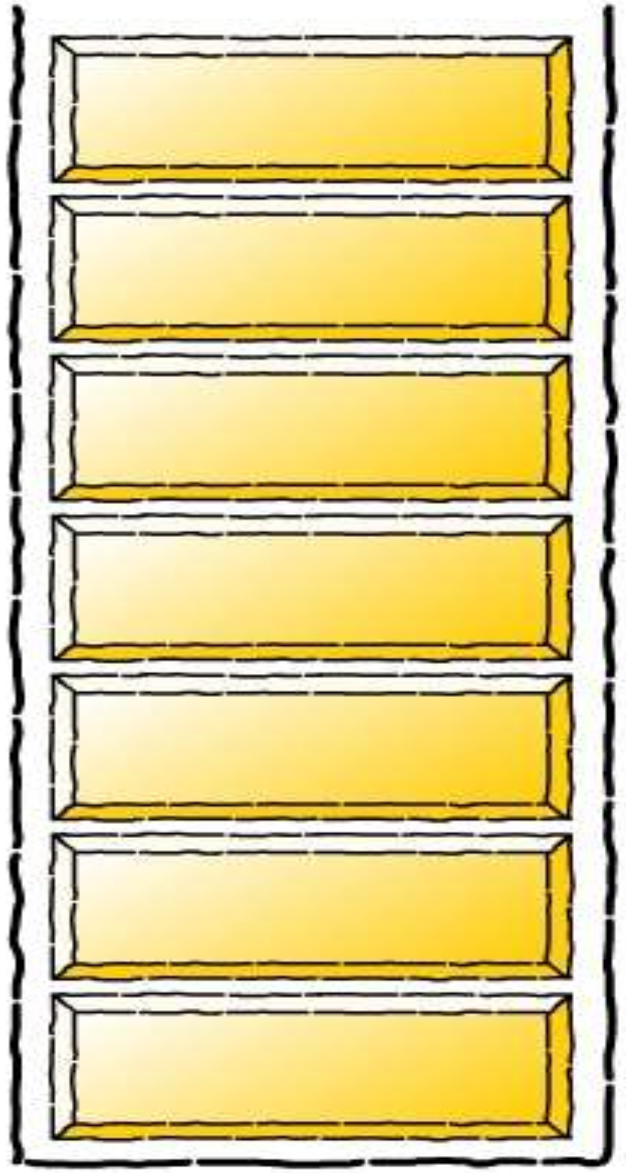
Step8: If the character scanned is an operator with lower precedence or equal precedence to the operator in the top of the stack, pop the operator in the top of the stack and print it in the result and then push the operator scanned into the stack. [Hint: The top most operator in the stack should always have the highest precedence].

Step 9: Repeat the steps 3 to 6 until the stack becomes empty and no more characters to read in the input expression.

Algorithm

- **Step 1** : Scan the Infix Expression from left to right.
- **Step 2** : If the scanned character is an operand, append it with final Infix to Postfix string.
- **Step 3** : Else,
  - **Step 3.1** : If the precedence order of the scanned(incoming) operator is greater than the precedence order of the operator in the stack (or the stack is empty or the stack contains a ‘(‘ or ‘[‘ or ‘{‘), push it on stack.
- **Step 3.2** : Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)
- **Step 4** : If the scanned character is an ‘(‘ or ‘[‘ or ‘{‘, push it to the stack.
- **Step 5** : If the scanned character is an ‘)’or ‘]’ or ‘}’, pop the stack and and output it until a ‘(‘ or ‘[‘ or ‘{‘ respectively is encountered, and discard both the parenthesis.
- **Step 6** : Repeat steps 2-6 until infix expression is scanned.
- **Step 7** : Print the output
- **Step 8** : Pop and output from the stack until it is not empty.

INFIX TO POSTFIX CONVERSION

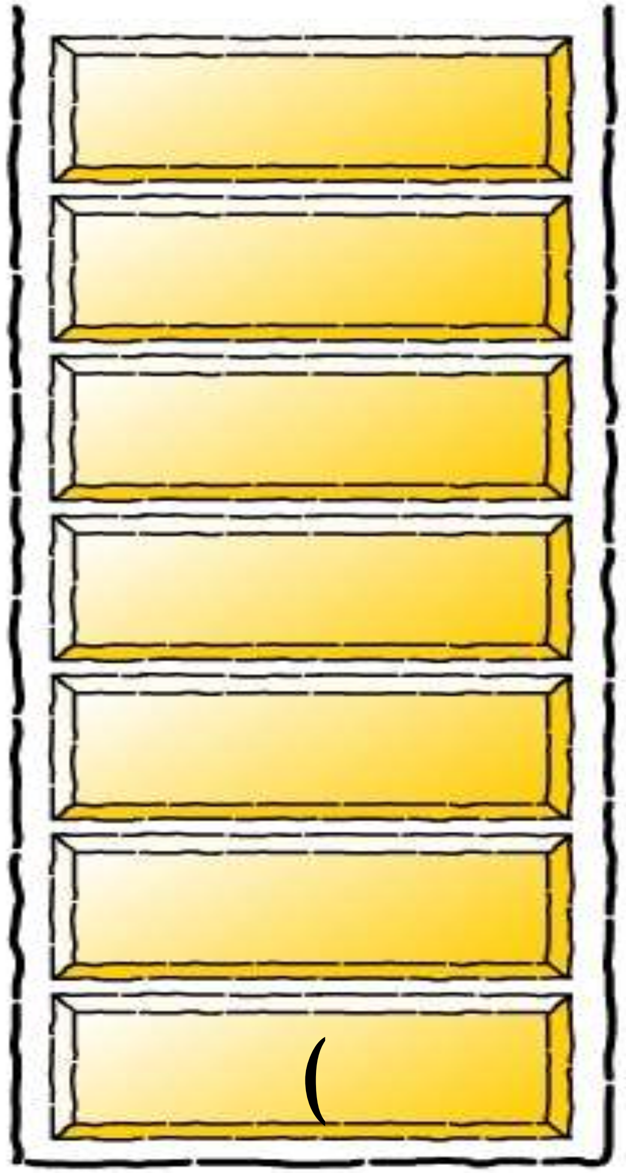


Infix expression

(a + b - c) * d – (e + f)

Postfix expression

INFIX TO POSTFIX CONVERSION



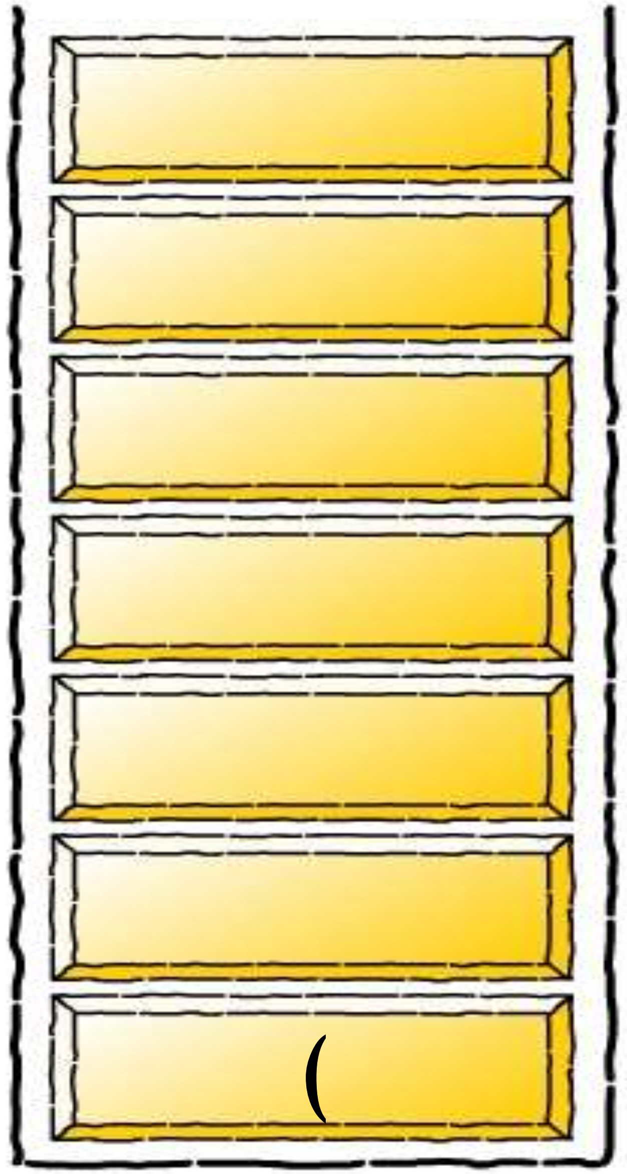
(

Infix expression

a + b - c) * d – (e + f)

Postfix expression

INFIX TO POSTFIX CONVERSION



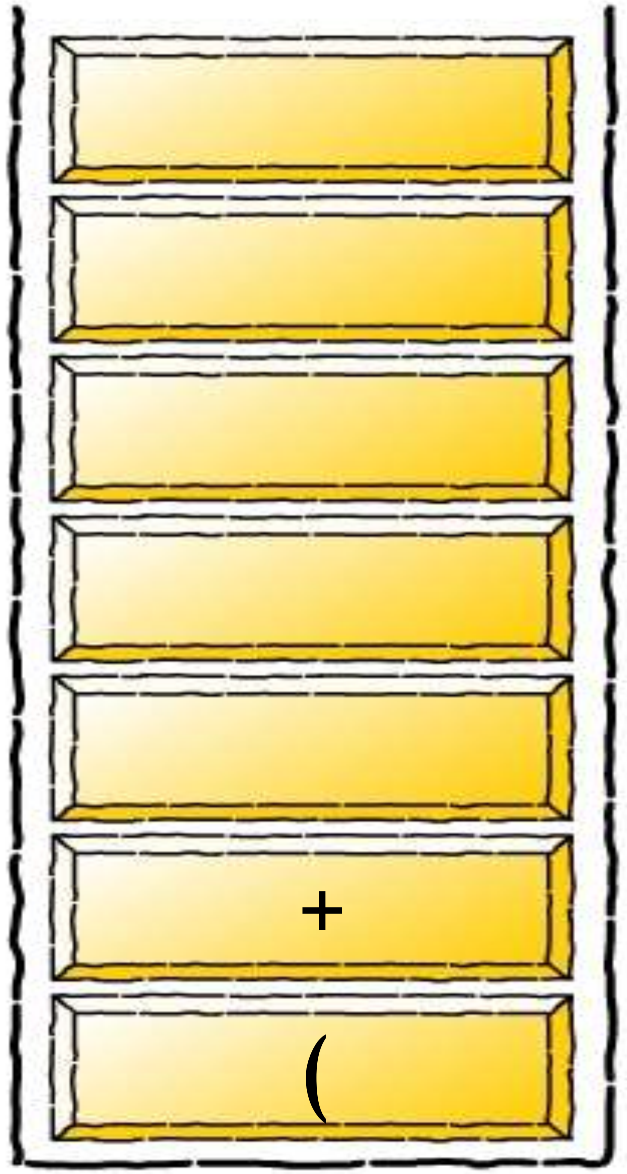
Infix expression

+ b - c) * d – (e + f)

Postfix expression

a

INFIX TO POSTFIX CONVERSION



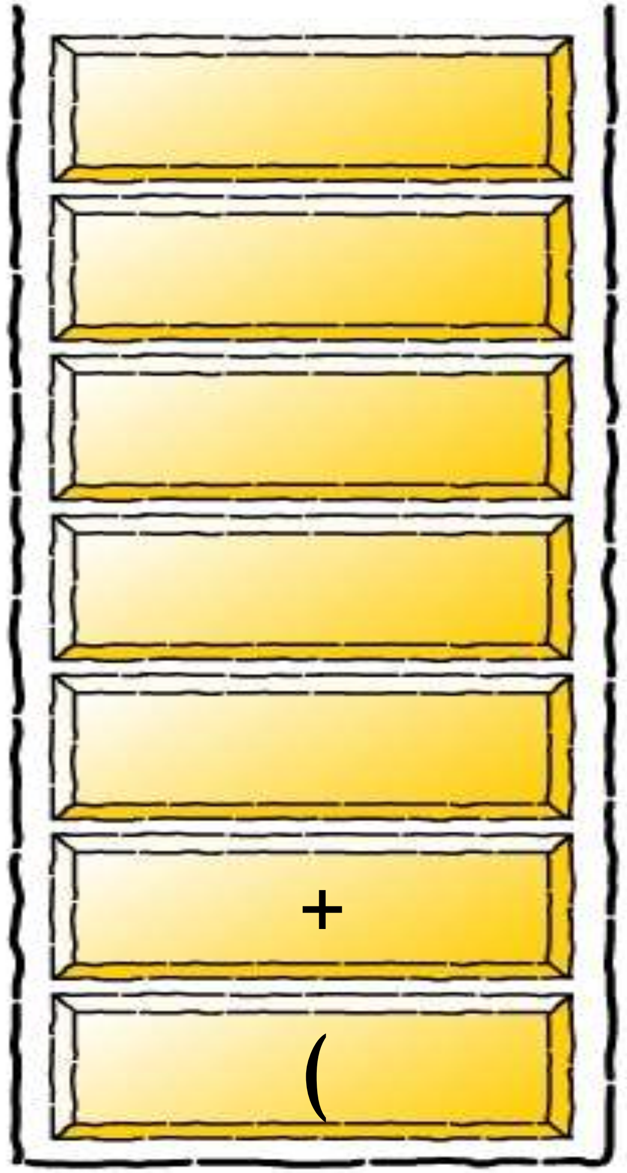
Infix expression

b - c) * d – (e + f)

Postfix expression

a

INFIX TO POSTFIX CONVERSION



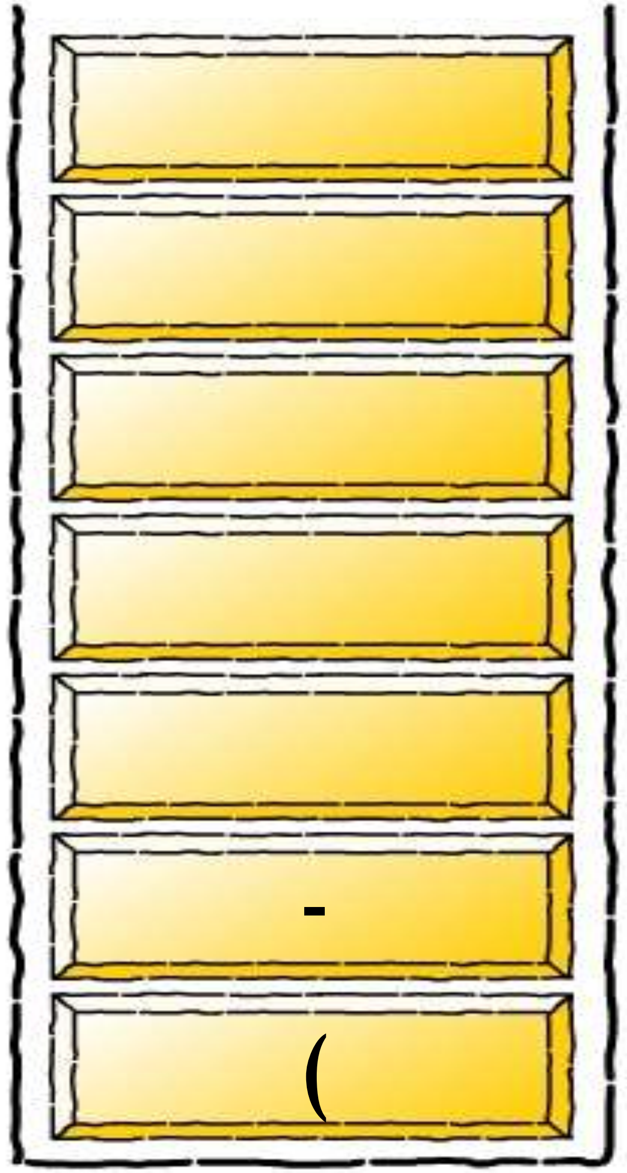
Infix expression

- c) * d – (e + f)

Postfix expression

a b

INFIX TO POSTFIX CONVERSION



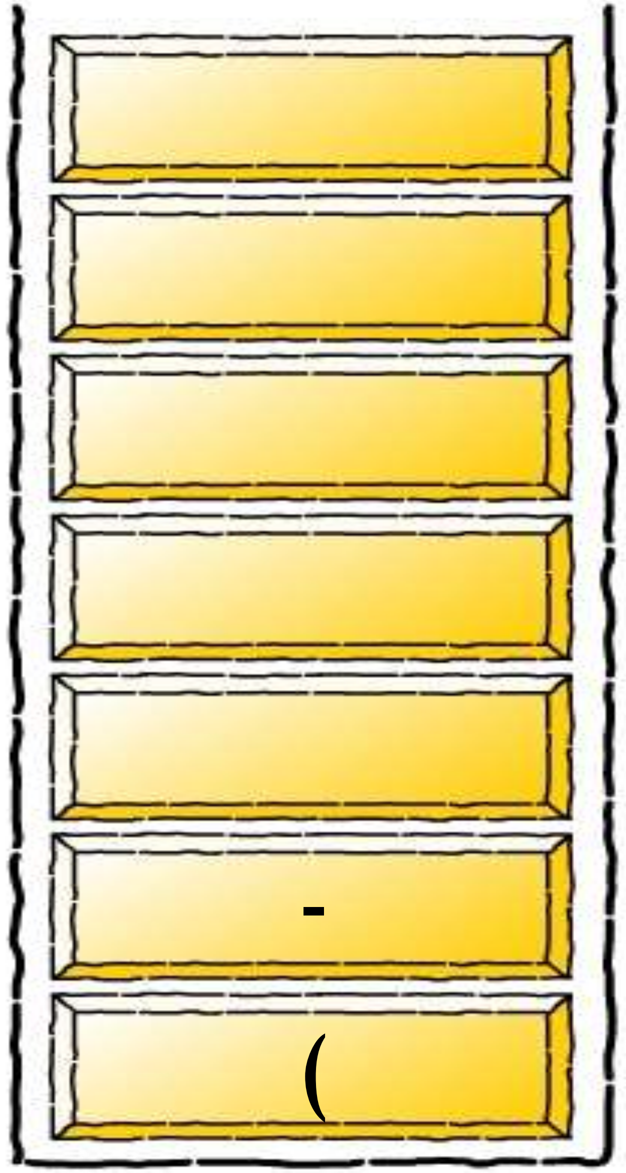
Infix expression

c) * d – (e + f)

Postfix expression

a b +

INFIX TO POSTFIX CONVERSION



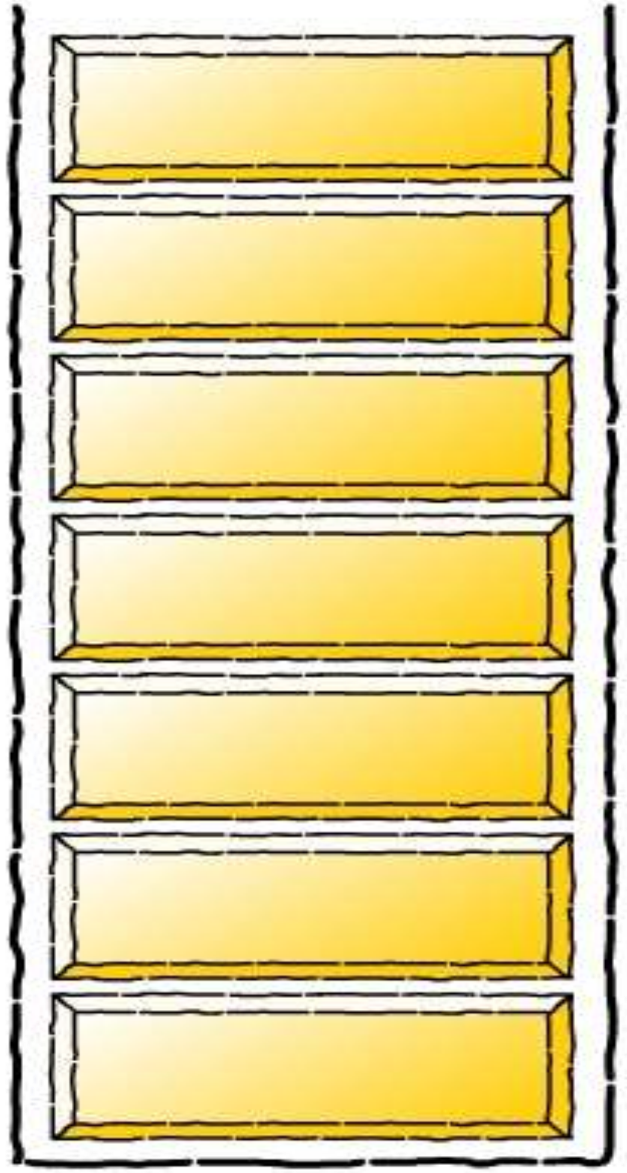
Infix expression

) * d – (e + f)

Postfix expression

a b + c

INFIX TO POSTFIX CONVERSION



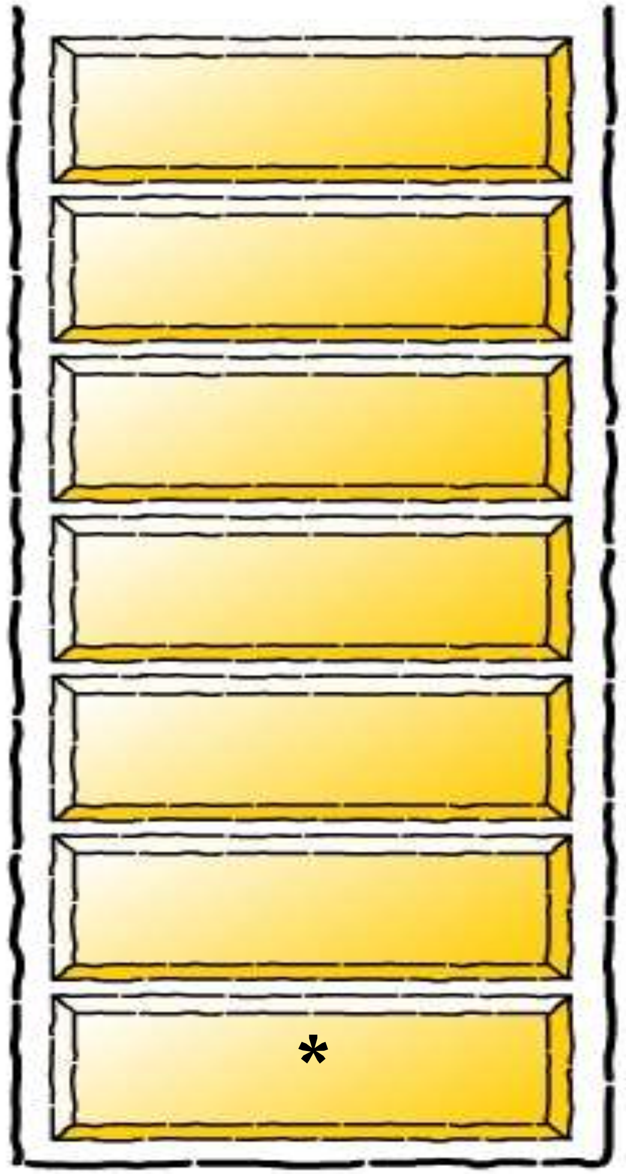
Infix expression

* d – (e + f)

Postfix expression

a b + c -

INFIX TO POSTFIX CONVERSION



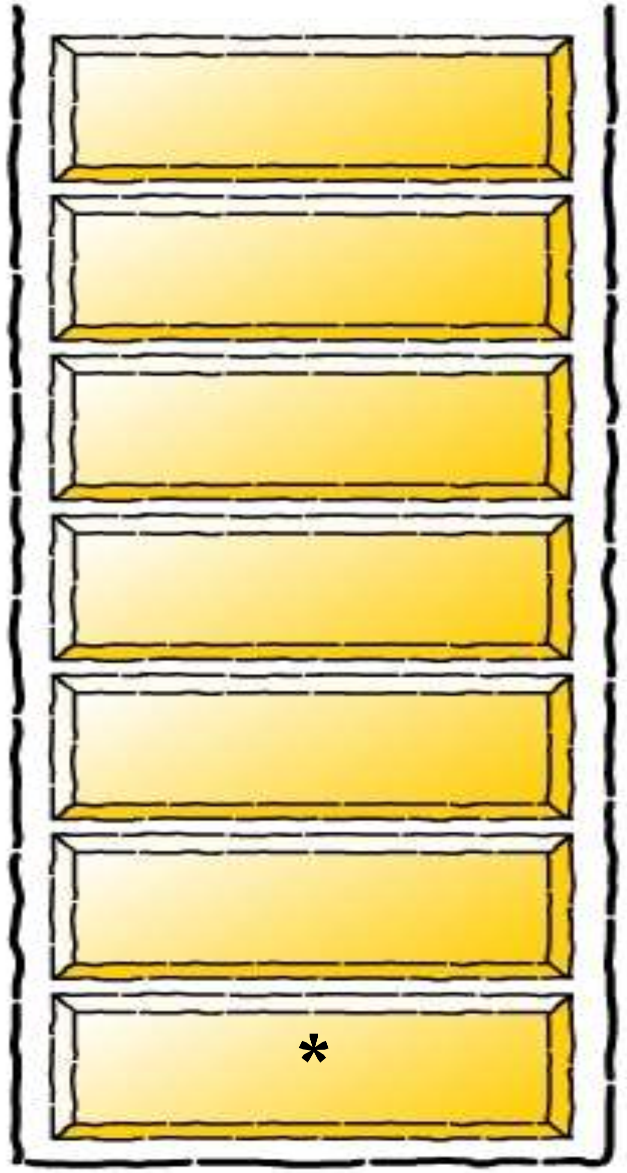
Infix expression

d – (e + f)

Postfix expression

a b + c -

INFIX TO POSTFIX CONVERSION



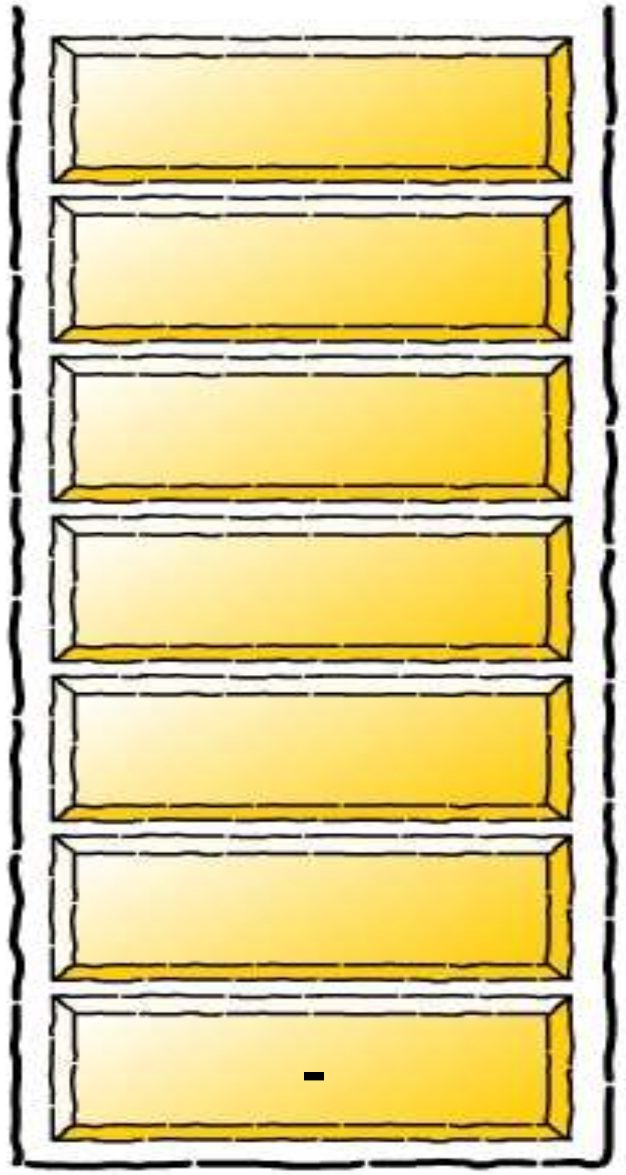
Infix expression

- (e + f)

Postfix expression

a b + c - d

INFIX TO POSTFIX CONVERSION



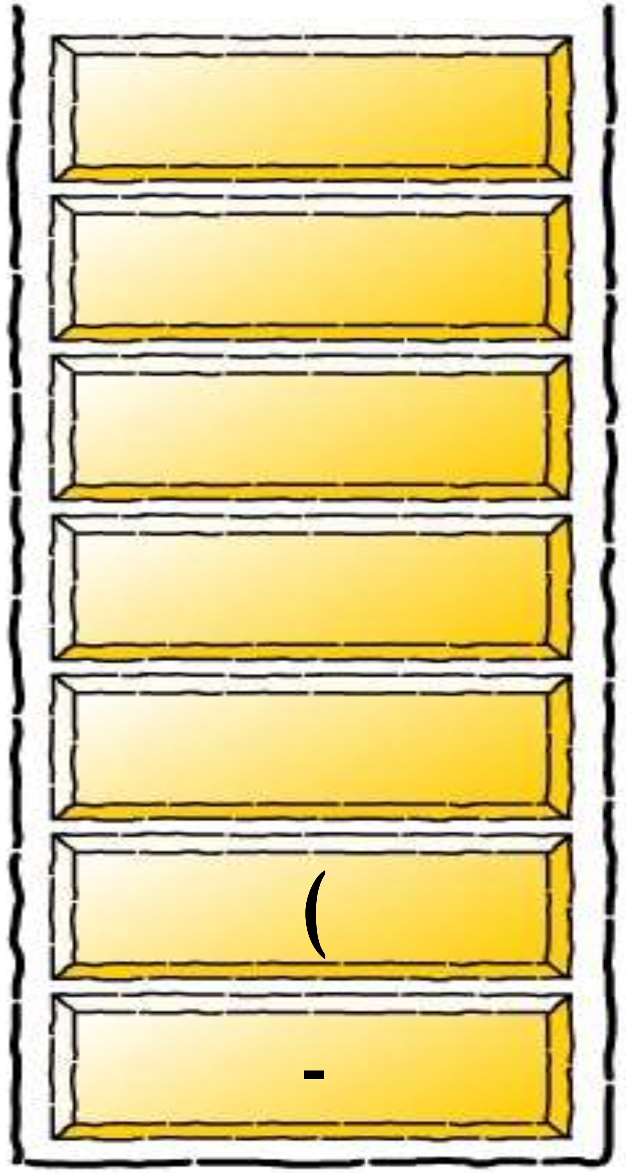
Infix expression

(e + f)

Postfix expression

a b + c - d *

INFIX TO POSTFIX CONVERSION



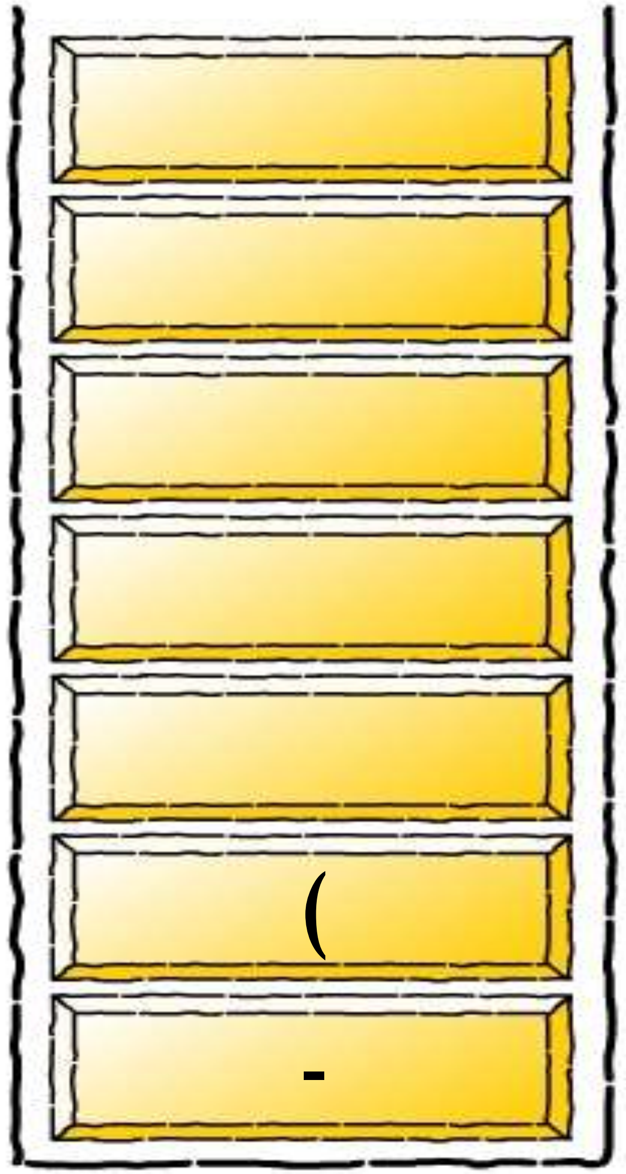
Infix expression

e + f)

Postfix expression

a b + c - d *

INFIX TO POSTFIX CONVERSION



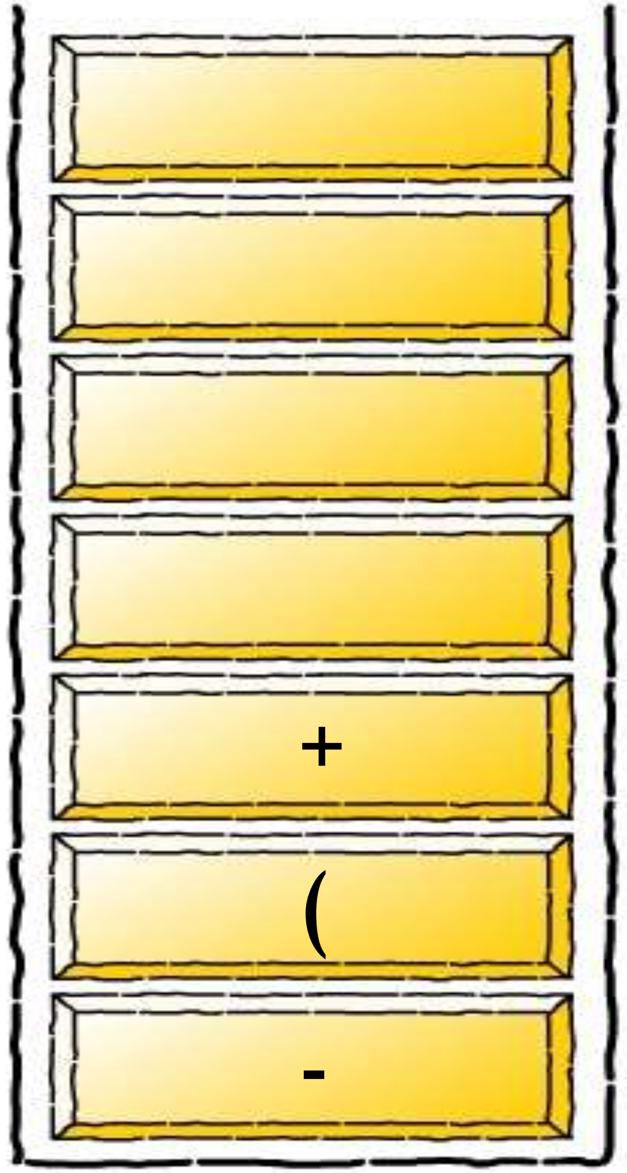
Infix expression

+ f)

Postfix expression

a b + c - d * e

INFIX TO POSTFIX CONVERSION



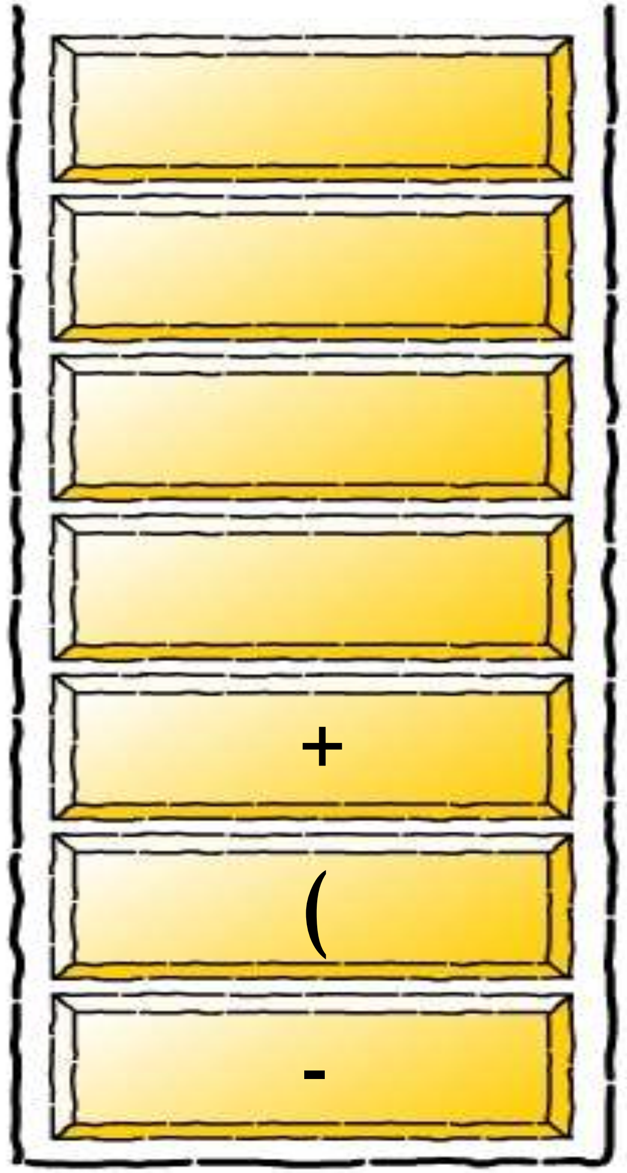
Infix expression

f)

Postfix expression

a b + c - d * e

INFIX TO POSTFIX CONVERSION



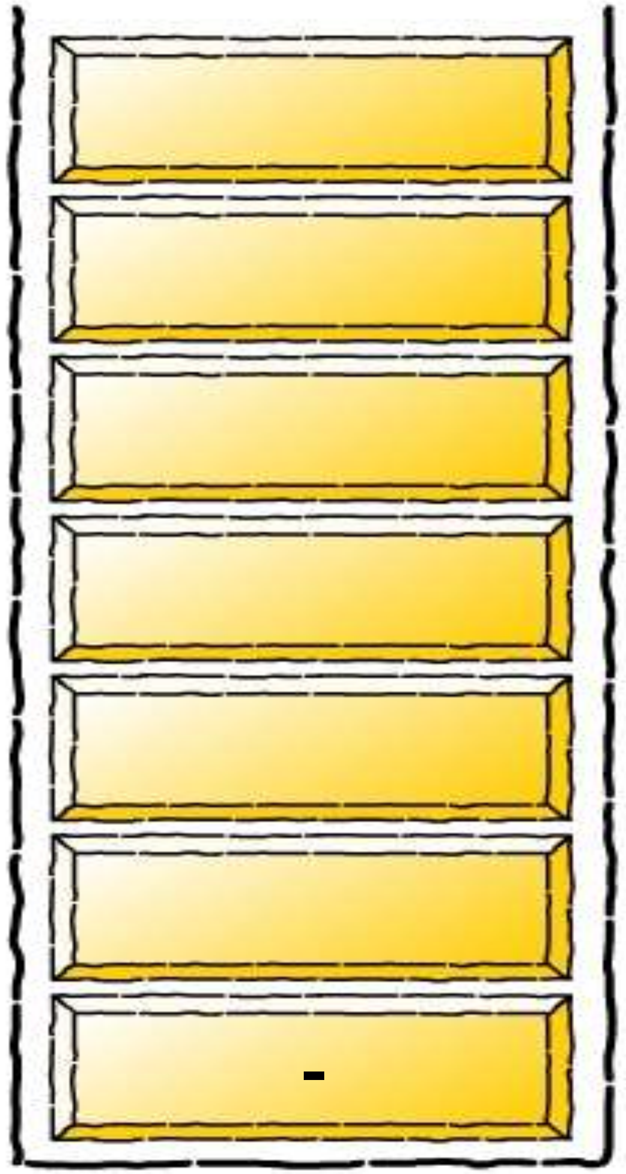
Infix expression

)

Postfix expression

a b + c - d * e f

INFIX TO POSTFIX CONVERSION

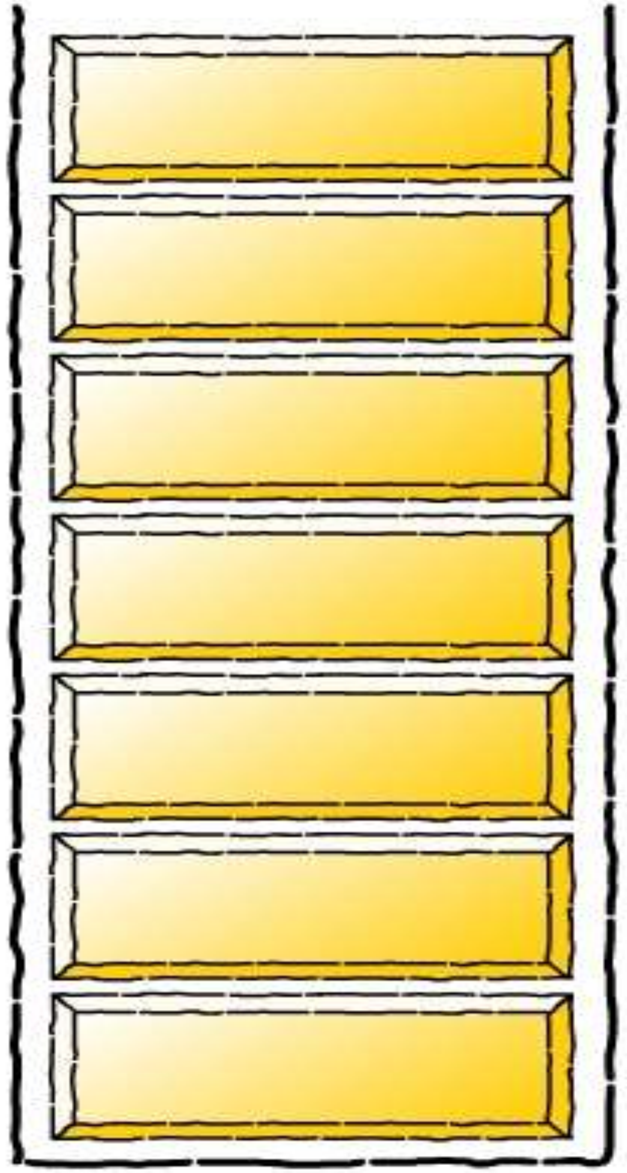


Infix expression

Postfix expression

a b + c - d * e f +

INFIX TO POSTFIX CONVERSION



Infix expression

Postfix expression

a b + c - d * e f + -

INFIX TO POSTFIX-ALGORITHM

```
void infixtopsotfix(char exp[])
{
 char *e, x;
 e = exp;
 while(*e != '\0')
 {
  if(isalnum(*e))
   printf("%c",*e);
  else if(*e == '(')
   push(*e);
  else if(*e == ')')
  {
    while((x = pop()) != '(')
     printf("%c", x);
  }
   else
  {
    while(priority(stack[top]) >=
  priority(*e))
     printf("%c",pop());
    push(*e);
   }
   e++;
  }
  while(top != -1)
  {
    printf("%c",pop());
  }
}
```

INFIX TO POSTFIX-ALGORITHM

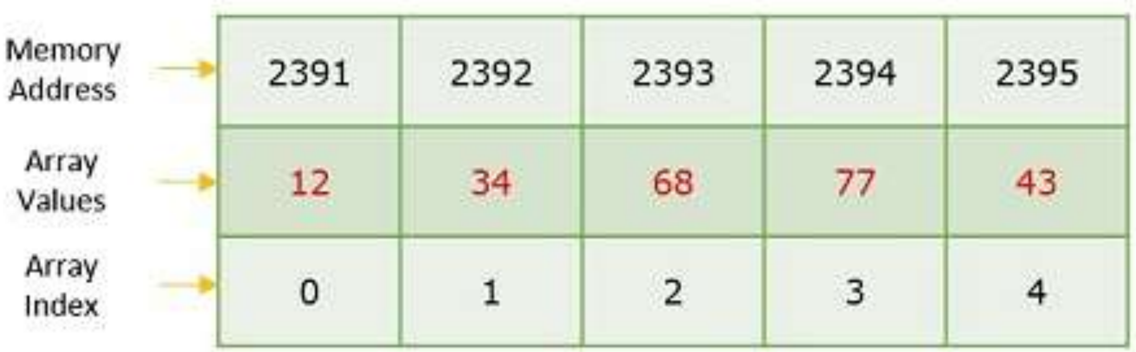
```
int priority(char x)
{
    if(x == '(')
        return 0;
    if(x == '+' || x == '-')
        return 1;
    if(x == '*' || x == '/')
        return 2;
}
```

```
char stack[20];
int top = -1;
void push(char x)
{
    stack[++top] = x;
}

char pop()
{
    if(top == -1)
        return -1;
    else
        return stack[top--];
}
```

ARRAYS

An array is a type of linear data structure that is defined as a collection of elements with same or different data types.

They exist in both single dimension and multiple dimensions. These data structures come into picture when there is a necessity to store multiple elements of similar nature together at one place.



Memory Address →	2391	2392	2393	2394	2395
Array Values →	12	34	68	77	43
Array Index →	0	1	2	3	4

The difference between an array index and a memory address is that the array index acts like a key value to label the elements in the array. However, a memory address is the starting address of free memory available.

Following are the important terms to understand the concept of Array.

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.

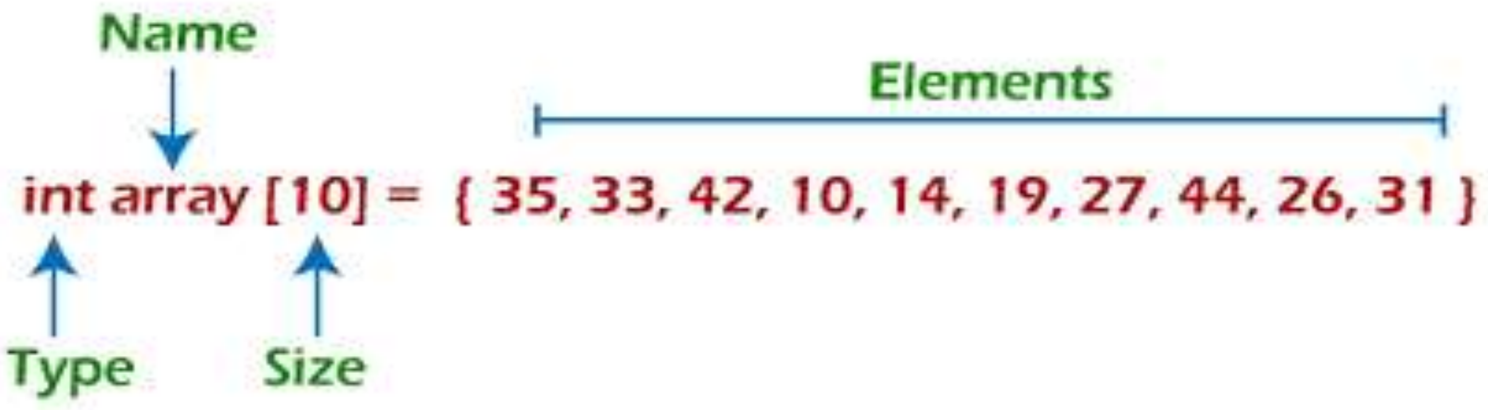
Arrays are useful because -

1. Sorting and searching a value in an array is easier.
2. Arrays are best to process multiple values quickly and easily.

Arrays are good for storing multiple values in a single variable - In computer programming, most cases require storing a large number of data of a similar type. To store such an amount of data, we need to define a large number of variables. It would be very difficult to remember the names of all the variables while writing the programs. Instead of naming all the variables with a different name, it is better to define an array and store all the elements into it.

ARRAY DECLARATION

data_type array_name[array_size]={elements separated by commas} or,
    data_type array_name[array_size];



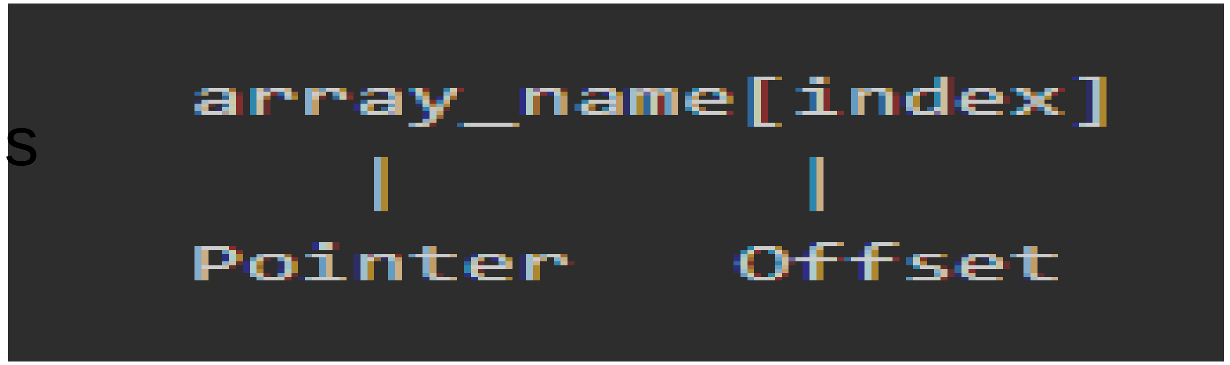
```
int array [10] = { 35, 33, 42, 10, 14, 19, 27, 44, 26, 31 }
```

NEED FOR ARRAYS

Arrays are used as solutions to many problems from the small sorting problems to more complex problems like travelling salesperson problem.

Arrays provide **O(1)** random access lookup time. That means, accessing the 1st index of the array and the 1000th index of the array will both take the same time. This is due to the fact that array comes with a pointer and an offset value. The pointer points to the right location of the memory and the offset value shows how far to look in the said memory.

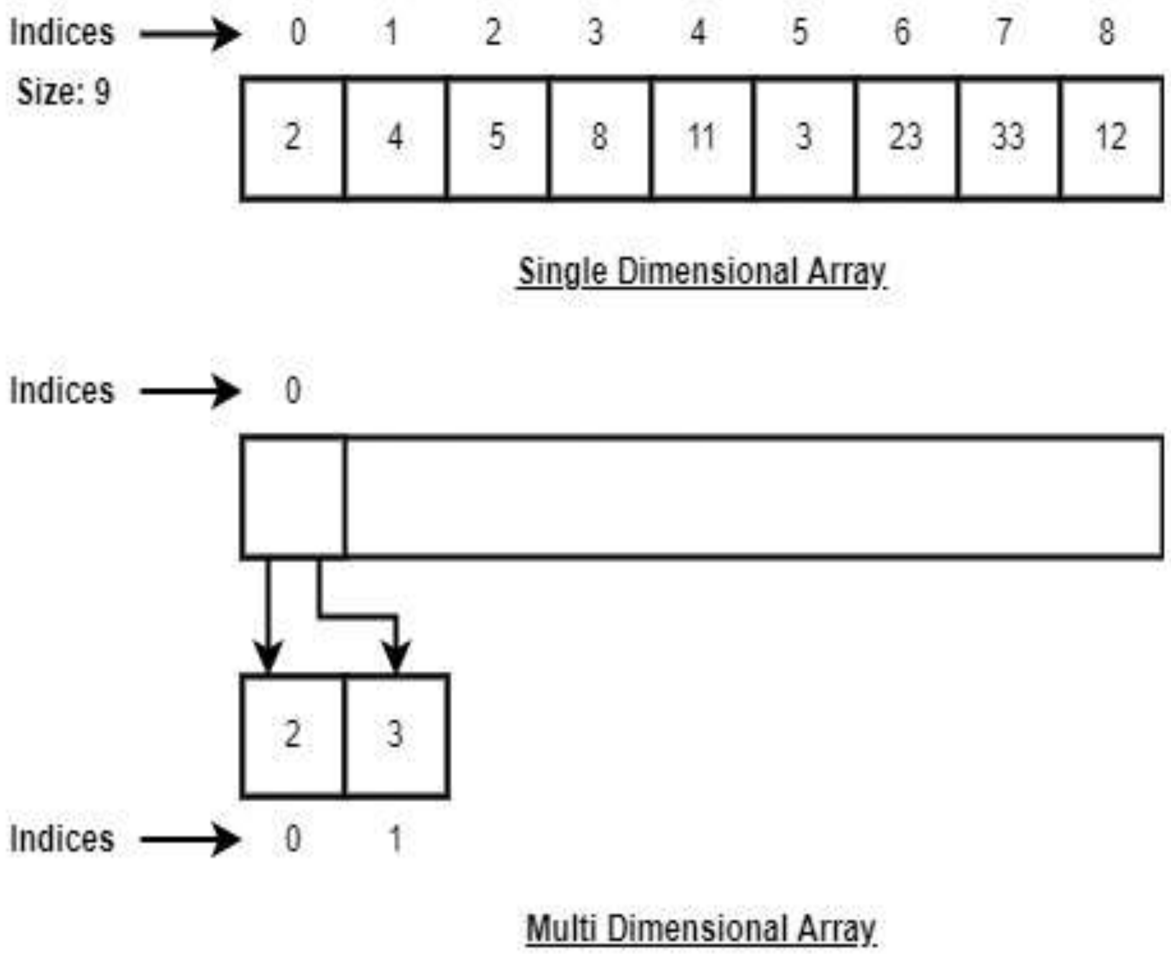
Therefore, in an array with 6 elements, to access the 1st element, array is pointed towards the 0th index. Similarly, to access the 6th element, array is pointed towards the 5th index.



```
array_name[index]
    |         |
Pointer    Offset
```

ARRAY REPRESENTATION

1. Arrays are represented as a collection of buckets where each bucket stores one element. These buckets are indexed from '0' to 'n-1', where n is the size of that particular array. For example, an array with size 10 will have buckets indexed from 0 to 9.
2. This indexing will be similar for the multidimensional arrays as well. If it is a 2-dimensional array, it will have sub-buckets in each bucket. Then it will be indexed as array_name[m][n], where m and n are the sizes of each level in the array.



Index starts with 0.
Array length is 9 which means it can store 9 elements.
Each element can be accessed via its index. For example, we can fetch an element at index 6 as 23.

ACCESSING ARRAY ELEMENTS USING INDEX

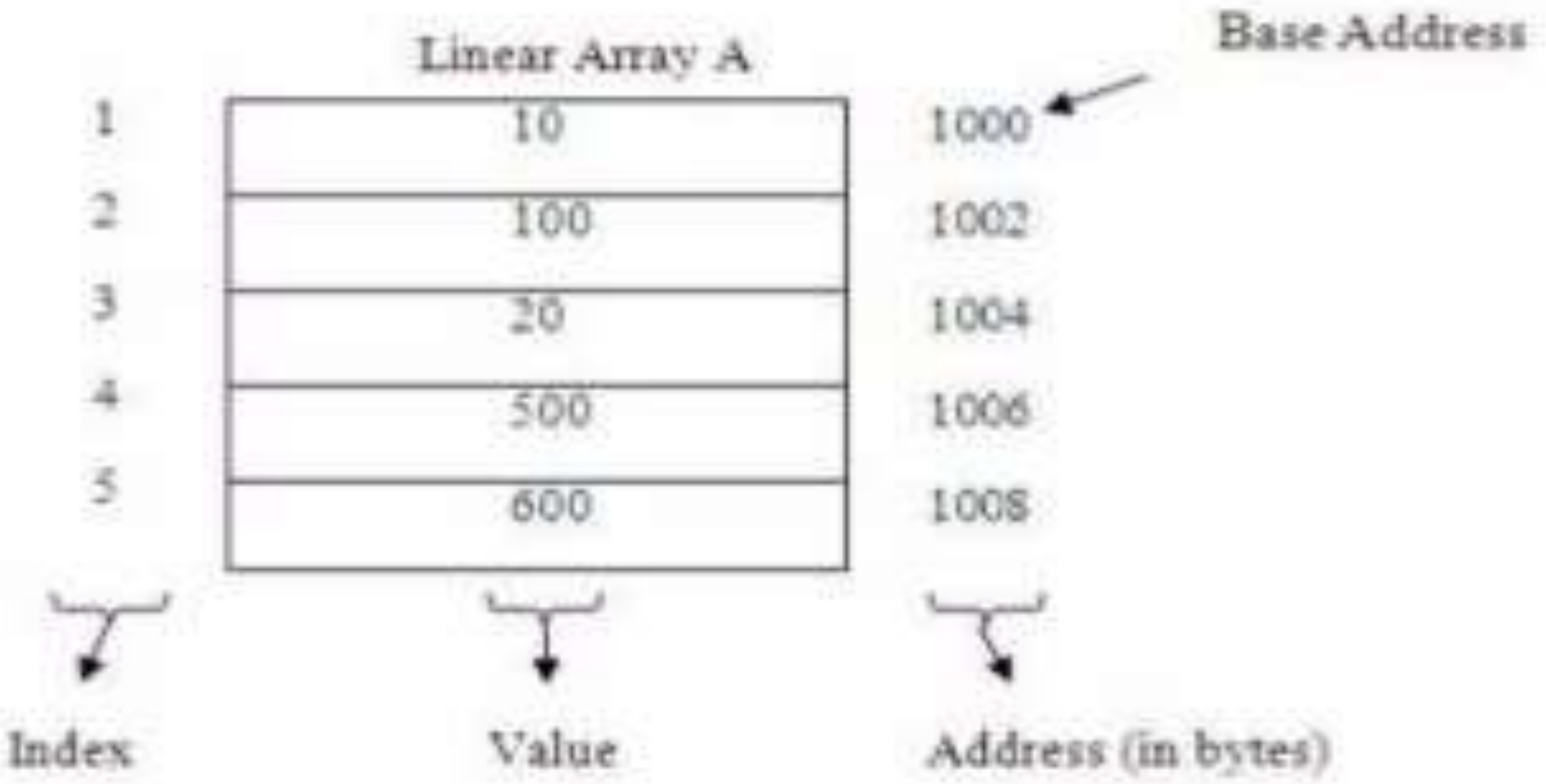
```
int main()
{
int arr[5] = {10, 20, 30, 40, 50};
//printing 3rd element which is index 2
printf("3rd element = %d\n", arr[2]);
 //printing 5th element which is index 4
 printf("5th element = %d\n", arr[4]);
 return 0;
 }
```

TIME COMPLEXITY ANALYSIS

1. Using the index value, we can access the array elements in constant time.
2. So the time complexity is O(1) for accessing an element in the array.

Represent a Linear Array in memory

- The elements of linear array are stored in consecutive memory locations. It is shown below:

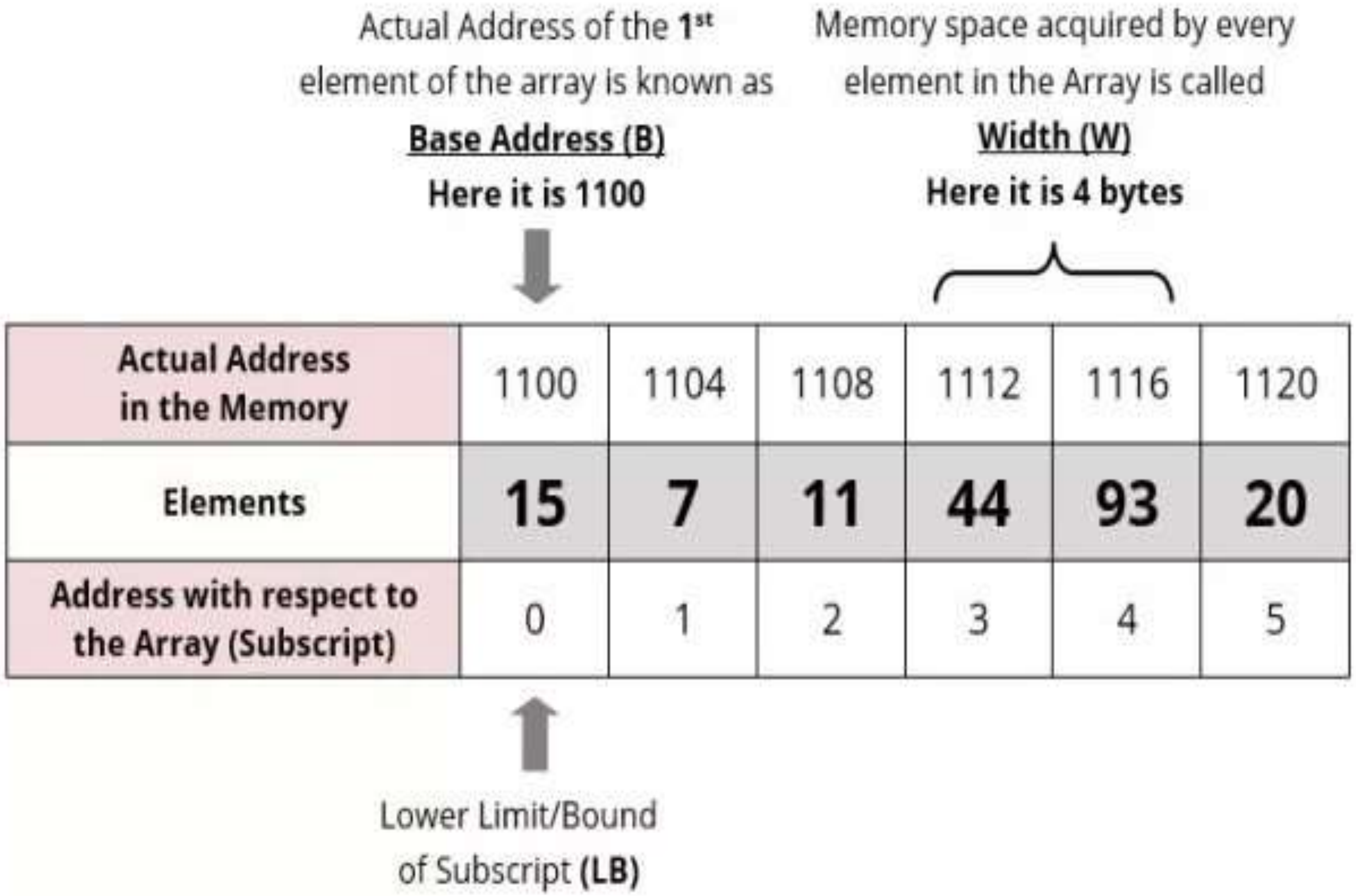


Linear Array A

Base Address

Index	Value	Address (in bytes)
1	10	1000
2	100	1002
3	20	1004
4	500	1006
5	600	1008

ADDRESS CALCULATION IN ARRAY



Actual Address of the 1st element of the array is known as **Base Address (B)**

Here it is 1100

Memory space acquired by every element in the Array is called **Width (W)**

Here it is 4 bytes

Actual Address in the Memory	1100	1104	1108	1112	1116	1120
Elements	**15**	**7**	**11**	**44**	**93**	**20**
Address with respect to the Array (Subscript)	0	1	2	3	4	5

Lower Limit/Bound of Subscript **(LB)**

ARRAY ADDRESS CALCULATION

- Array of an element of an array say “A[I]” is calculated using the following formula:

Address of A [I] = B + W * (I – LB)

Where,

- B = Base address
- W = Storage Size of one element stored in the array (in byte)
- I = Subscript of element whose address is to be found
- LB = Lower limit / Lower Bound of subscript, if not specified assume 0 (zero)

EXAMPLE

Given the base address of an array **B[1300…..1900]** as 1020 and size of each element is 2 bytes in the memory. Find the address of **B[1700]**.

Solution:

The given values are: B = 1020, LB = 1300, W = 2, I =1700

Address of A[I] = B + W * (I – LB)

= 1020 + 2 * (1700 – 1300)

= 1020 + 2 * 400

= 1020 + 800

= 1820 **[Ans]**

BASIC OPERATIONS

1. Traversal - This operation is used to print the elements of the array.
2. Insertion - It is used to add an element at a particular index.
3. Deletion - It is used to delete an element from a particular index.
4. Search - It is used to search an element using the given index or by the value.
5. Update - It updates an element at a particular index.

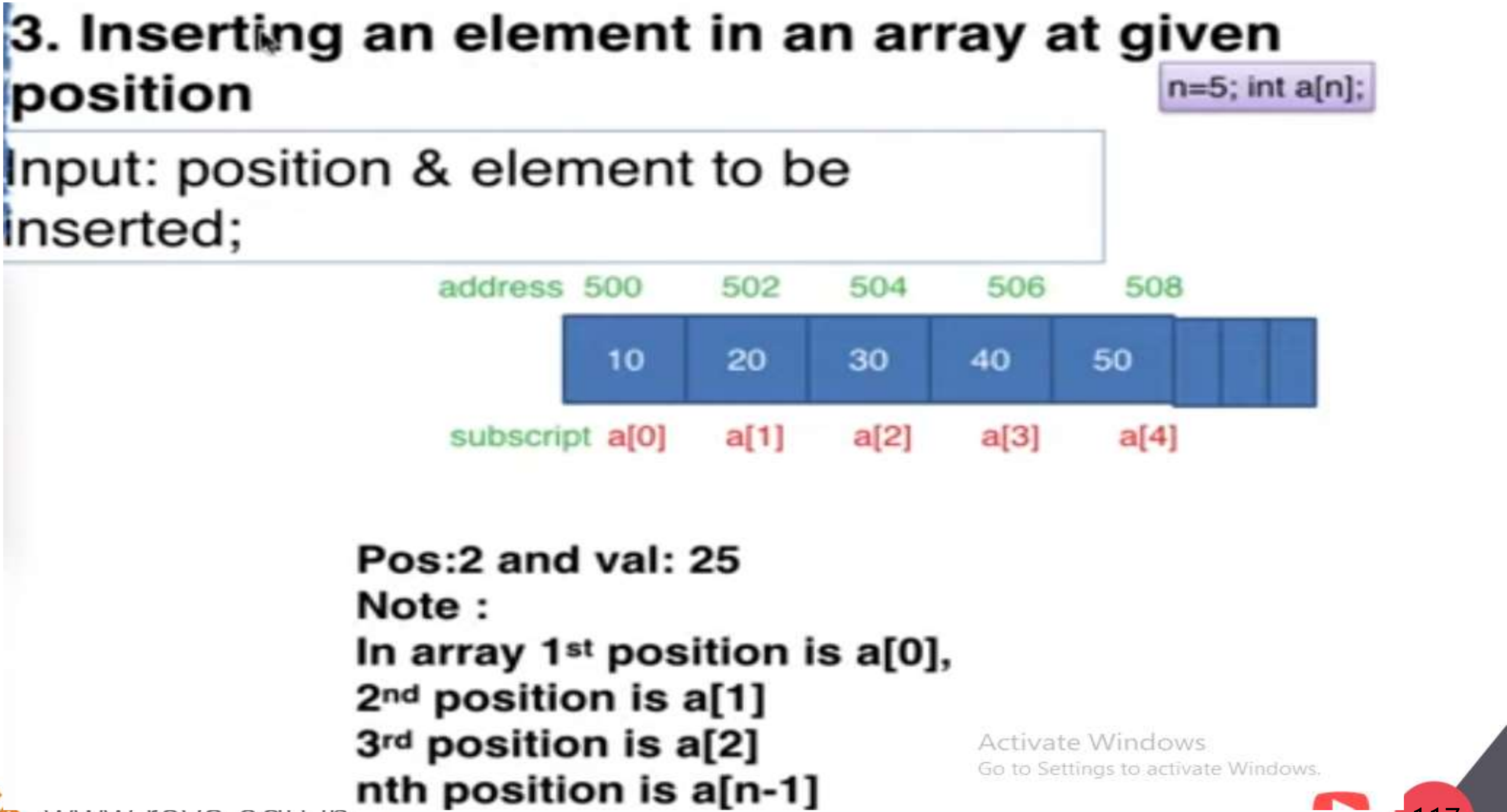
TRAVERSAL OPERATION

This operation is performed to traverse through the array elements. It prints all array elements one after another. We can understand it with the below program -

```
#include <stdio.h>
void main() {
 int Arr[5] = {18, 30, 15, 70, 12};
int i;
printf("Elements of the array are:\n");
   for(i = 0; i<5; i++) {
    printf("Arr[%d] = %d,  ", i, Arr[i]);
   }  }
```

OUTPUT:

```
Elements of the array are:
Arr[0]=18,
Arr[1]=30,Arr[2]=15,Arr[3]=70,Arr[4]=12
```



3. Inserting an element in an array at given position

n=5; int a[n];

Input: position & element to be inserted;

address	500	502	504	506	508			
	10	20	30	40	50			
subscript	a[0]	a[1]	a[2]	a[3]	a[4]			

Pos:2 and val: 25

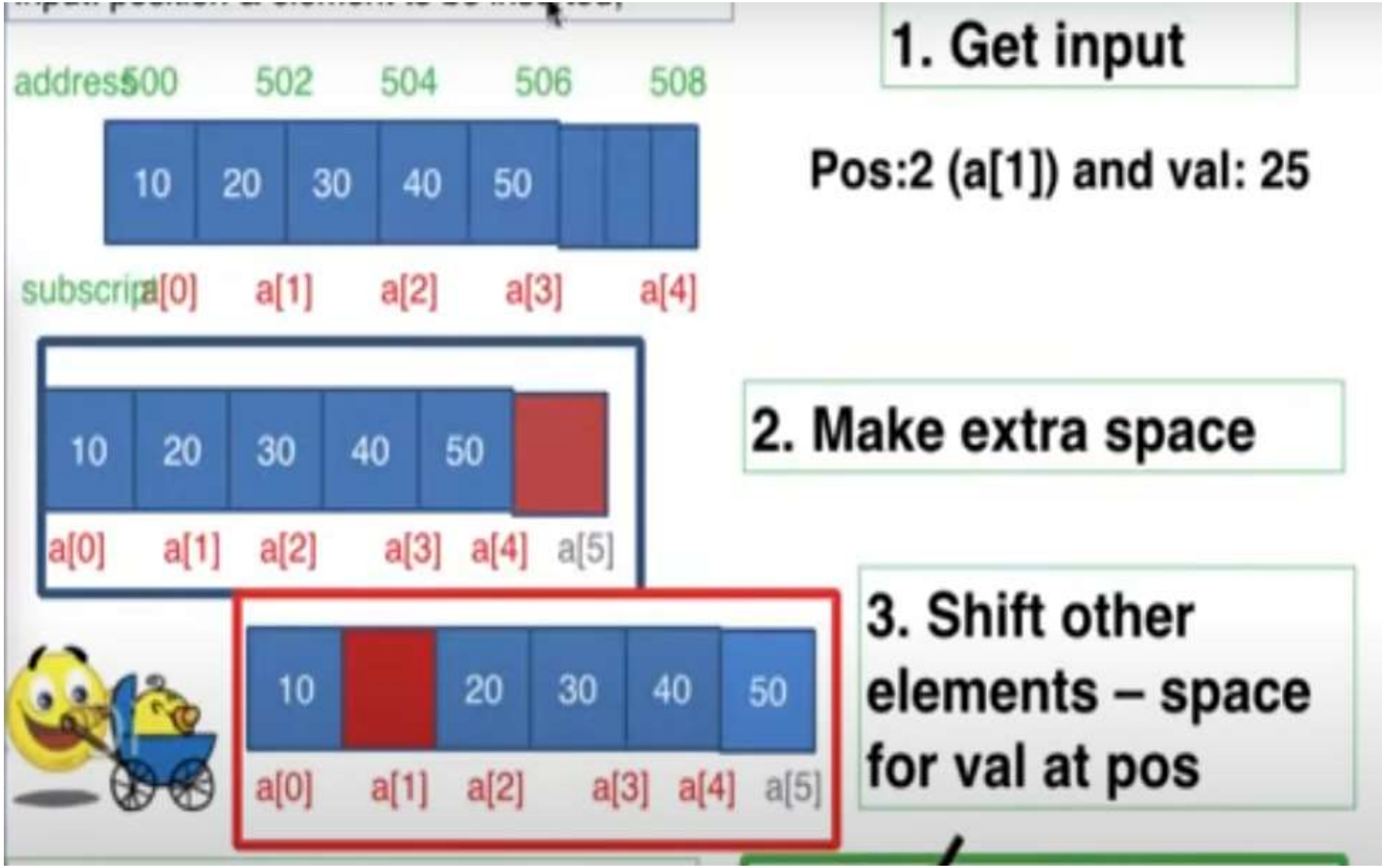
Note :

In array 1st position is a[0],

2nd position is a[1]

3rd position is a[2]

nth position is a[n-1]

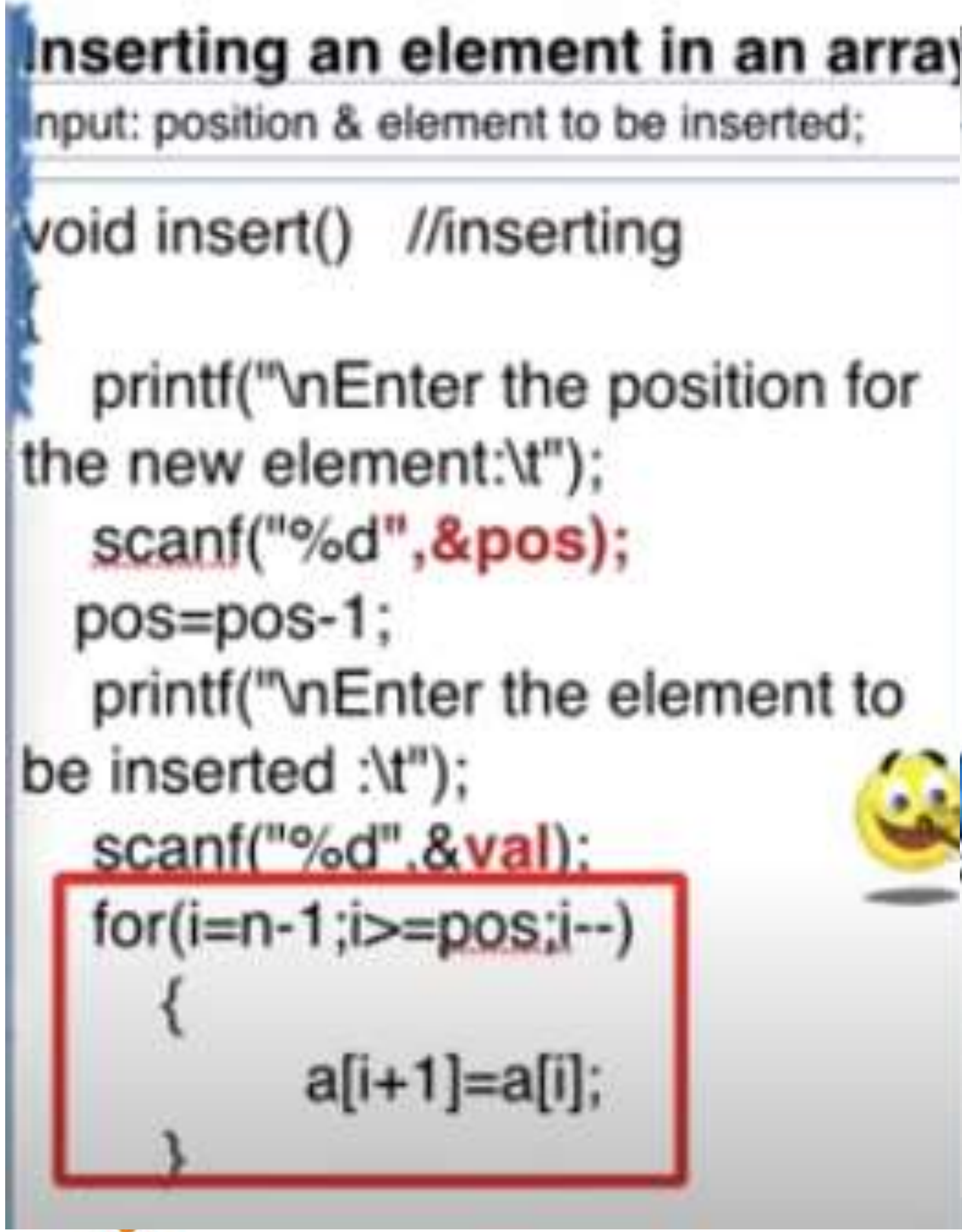


1. Get input

Pos:2 (a[1]) and val: 25

2. Make extra space

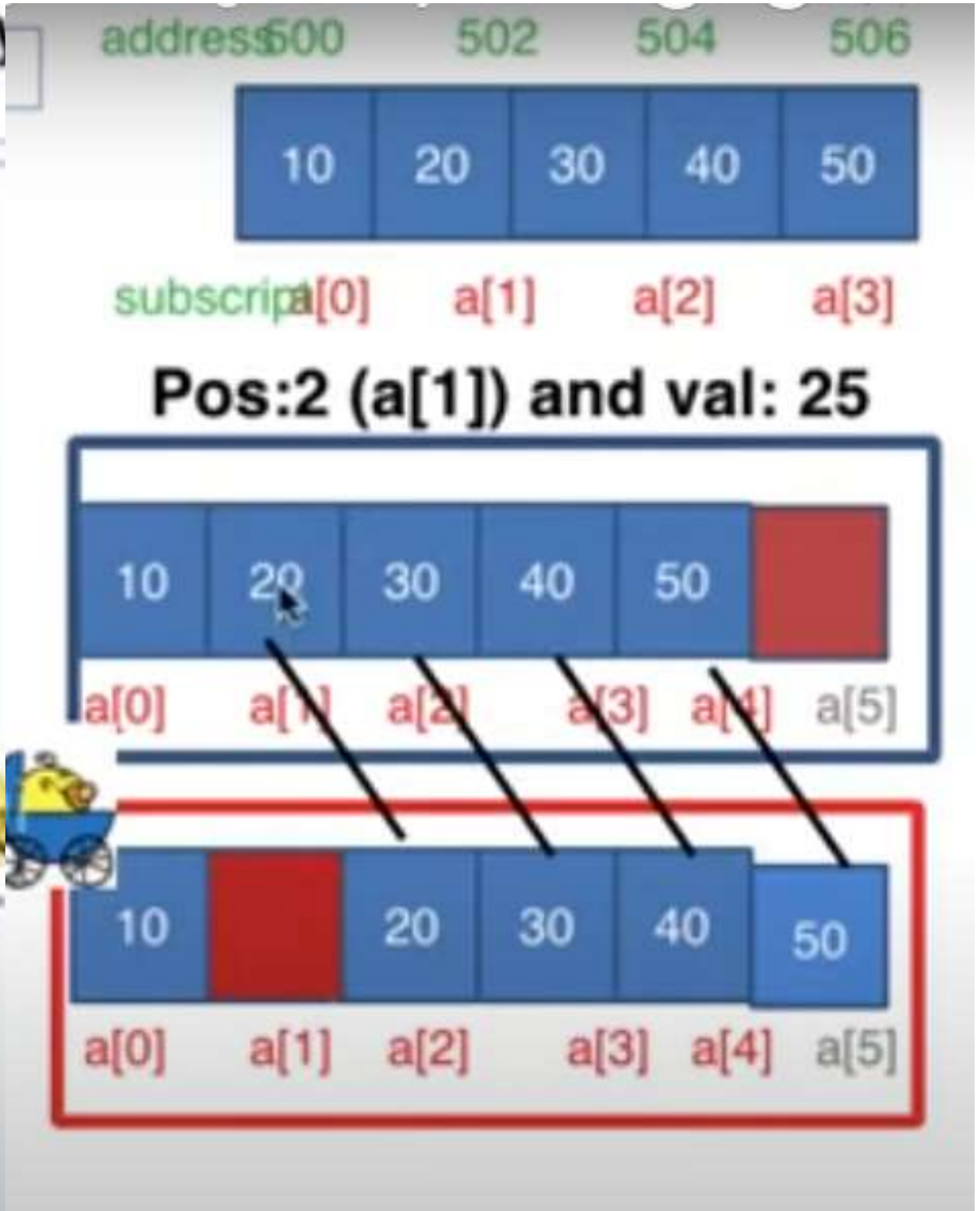
3. Shift other elements – space for val at pos



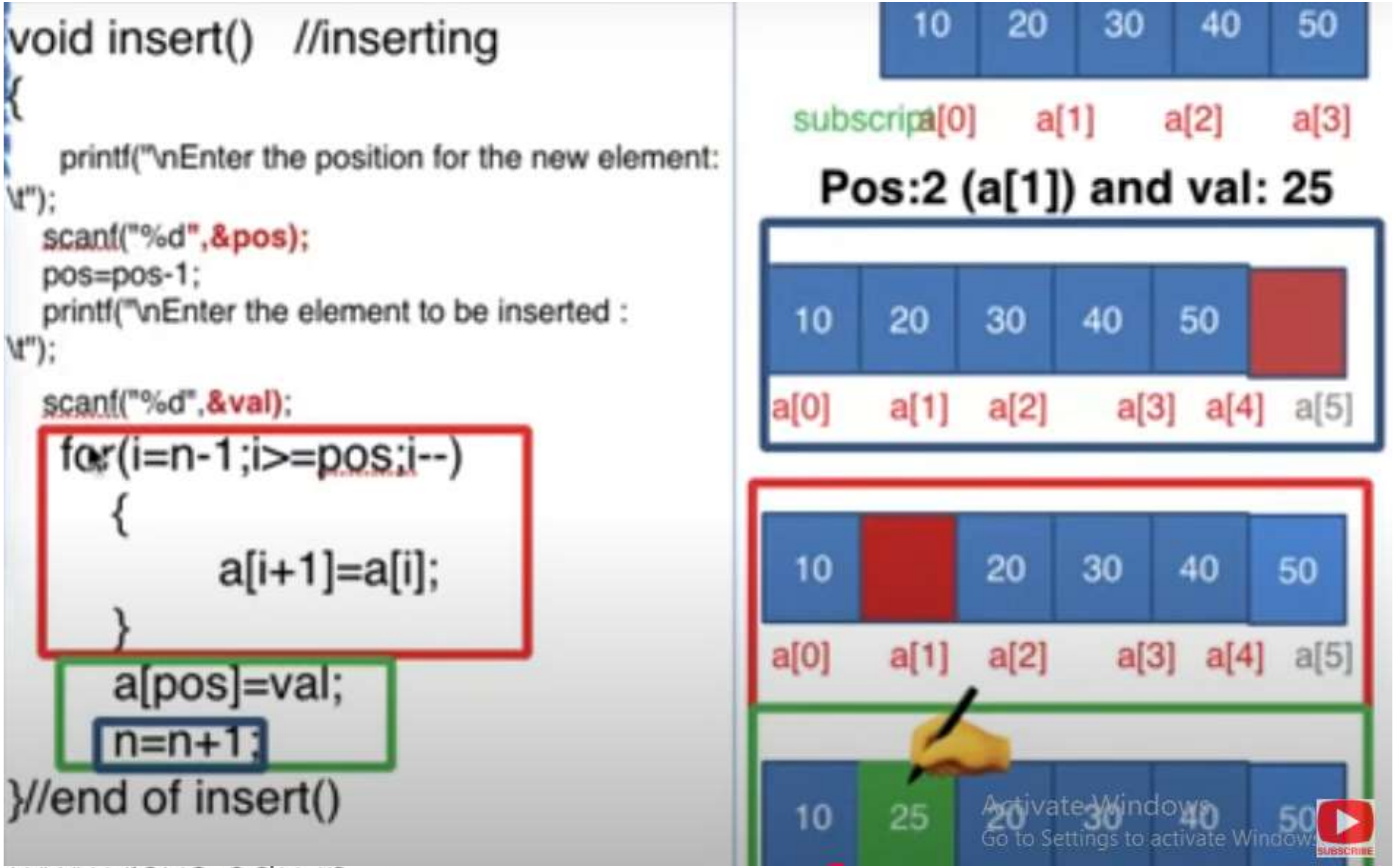
Inserting an element in an array

Input: position & element to be inserted;

```
void insert()   //inserting
{
   printf("\nEnter the position for
the new element:\t");
   scanf("%d",&pos);
  pos=pos-1;
   printf("\nEnter the element to
be inserted :\t");
   scanf("%d",&val);
   for(i=n-1;i>=pos;i--)
     {
          a[i+1]=a[i];
     }
```



Pos:2 (a[1]) and val: 25



```
void insert()   //inserting
{
    printf("\nEnter the position for the new element:
\t");
   scanf("%d",&pos);
   pos=pos-1;
   printf("\nEnter the element to be inserted :
\t");

   scanf("%d",&val);
    for(i=n-1;i>=pos;i--)
      {
              a[i+1]=a[i];
      }
      a[pos]=val;
      n=n+1;
}//end of insert()
```

Pos:2 (a[1]) and val: 25

10 20 30 40 50
subscript a[0] a[1] a[2] a[3]
10 20 30 40 50
a[0] a[1] a[2] a[3] a[4] a[5]
10 20 30 40 50
a[0] a[1] a[2] a[3] a[4] a[5]
10 25 20 30 40 50
Activate Windows
Go to Settings to activate Windows
SUBSCRIBE

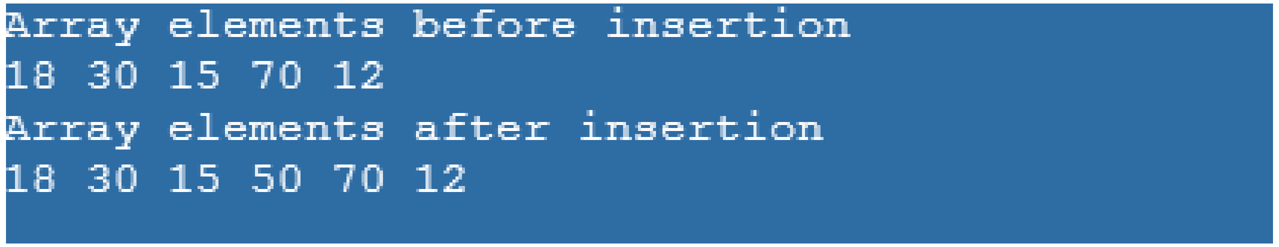
INSERTION OPERATION

This operation is performed to insert one or more elements into the array.

As per the requirements, an element can be added at the beginning, end, or at any index of the array.

Now, let's see the implementation of inserting an element into the array.

```
int main()
{    int arr[20] = { 18, 30, 15, 70, 12 };     int i, x, pos, n = 5;
    printf("Array elements before insertion\n");
    for (i = 0; i < n; i++)
      printf("%d ", arr[i]);
   printf("\n");
   x = 50; // element to be inserted
    pos = 4;
    n++;
      for (i = n-1; i >= pos; i--)
        arr[i] = arr[i - 1];
    arr[pos - 1] = x;
    printf("Array elements after insertion\n");
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
    return 0;
}
```



```
Array elements before insertion
18 30 15 70 12
Array elements after insertion
18 30 15 50 70 12
```

Insertion in an array refers to the process of adding a new element to a specific position within the array while shifting the existing elements to make room for the new element.

Here's a simple algorithm in simple language to perform an insertion in an array:

1. Define the array and the element to be inserted.
2. Determine the position where the element should be inserted.
3. Shift the elements after the insertion position to the right by one index to make room for the new element.
4. Insert the new element at the desired position.
5. Update the length of the array.

ALGORITHM TO INSERT AN ELEMENT IN AN ARRAY

Here size is the array size. Position (pos) is location where element to be inserted and Item is new value in the array

```
Step 1: Start
Step 2: [Initialize variable ] Set i = size - 1
Step 3: Repeat Step 4 and 5  While i >= pos-1
Step 4: [Move ith element forward. ] set arr[i+1] = arr[i]
Step 5: [Decrease counter. ] Set i = i - 1
Step 6: [End of step 3 loop. ]
Step 7: [Insert element. ] Set arr[pos-1] = item
Step 8: Stop
```

DELETION OPERATION

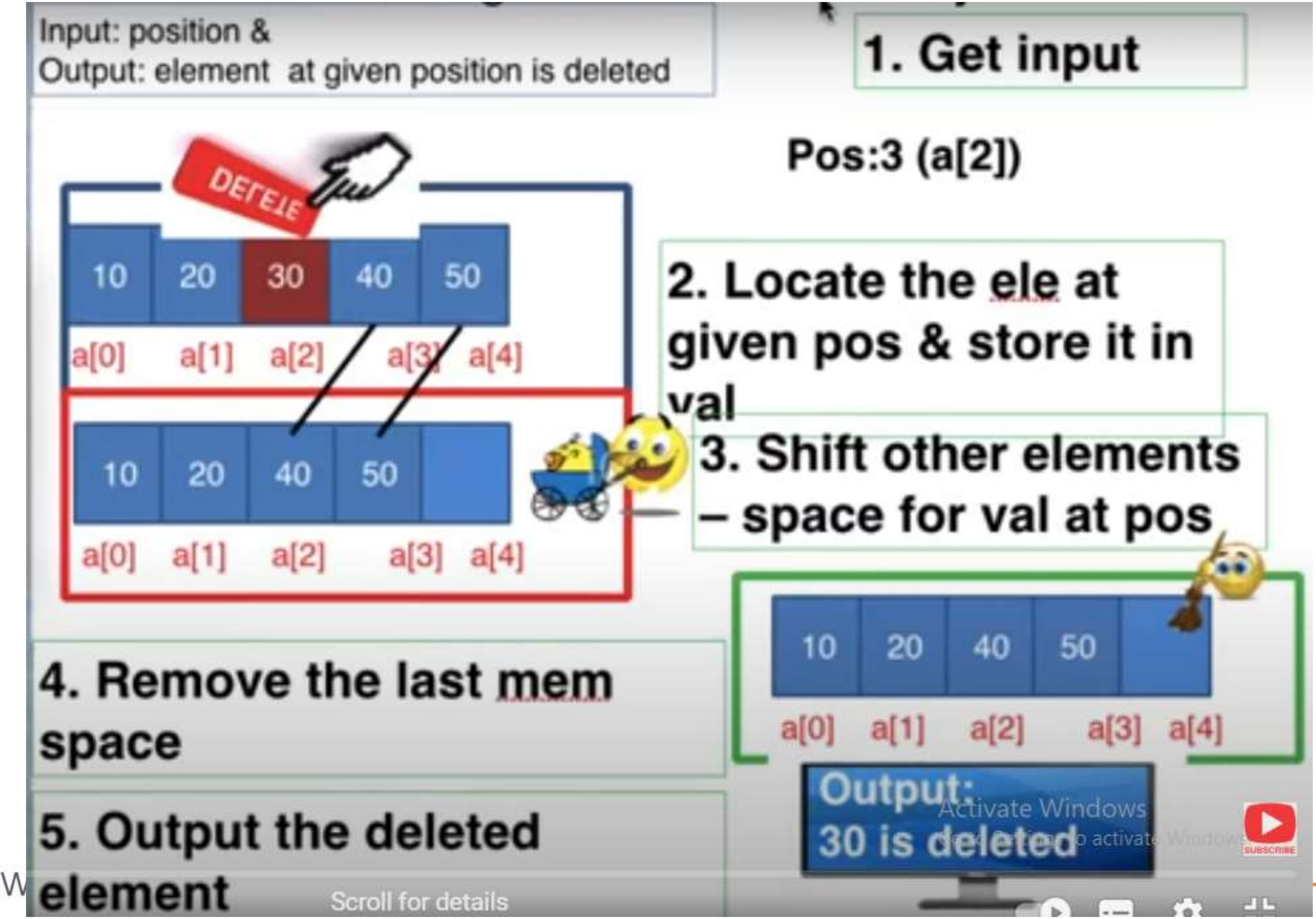
Deletion in an array means removing an element from an array and shifting the remaining elements to fill the empty space.

Here is the algorithm to delete an element from an array in C:

1. Initialize the array and the index of the element to be deleted.
2. Traverse the array from the index of the element to be deleted until the end of the array.
3. Shift each element one position to the left.
4. Decrement the size of the array by 1.

Algorithm to Delete an element from an Array:

```
Step 1: Start
Step 2: [Initialize  variable] Set i = pos - 1
Step 3: Repeat Step 4 and 5 for i = pos - 1  to i < size
Step 4: [Move ith element left side ] set a[i] = a[i+1]
Step 5: [Incerement ] Set i = i + 1
Step 6: [End of step 03 loop. ]
Step 7: [Update Array Size] set size = size - 1
Step 8: Stop
```



Input: position &
Output: element at given position is deleted

1. Get input

Pos:3 (a[2])

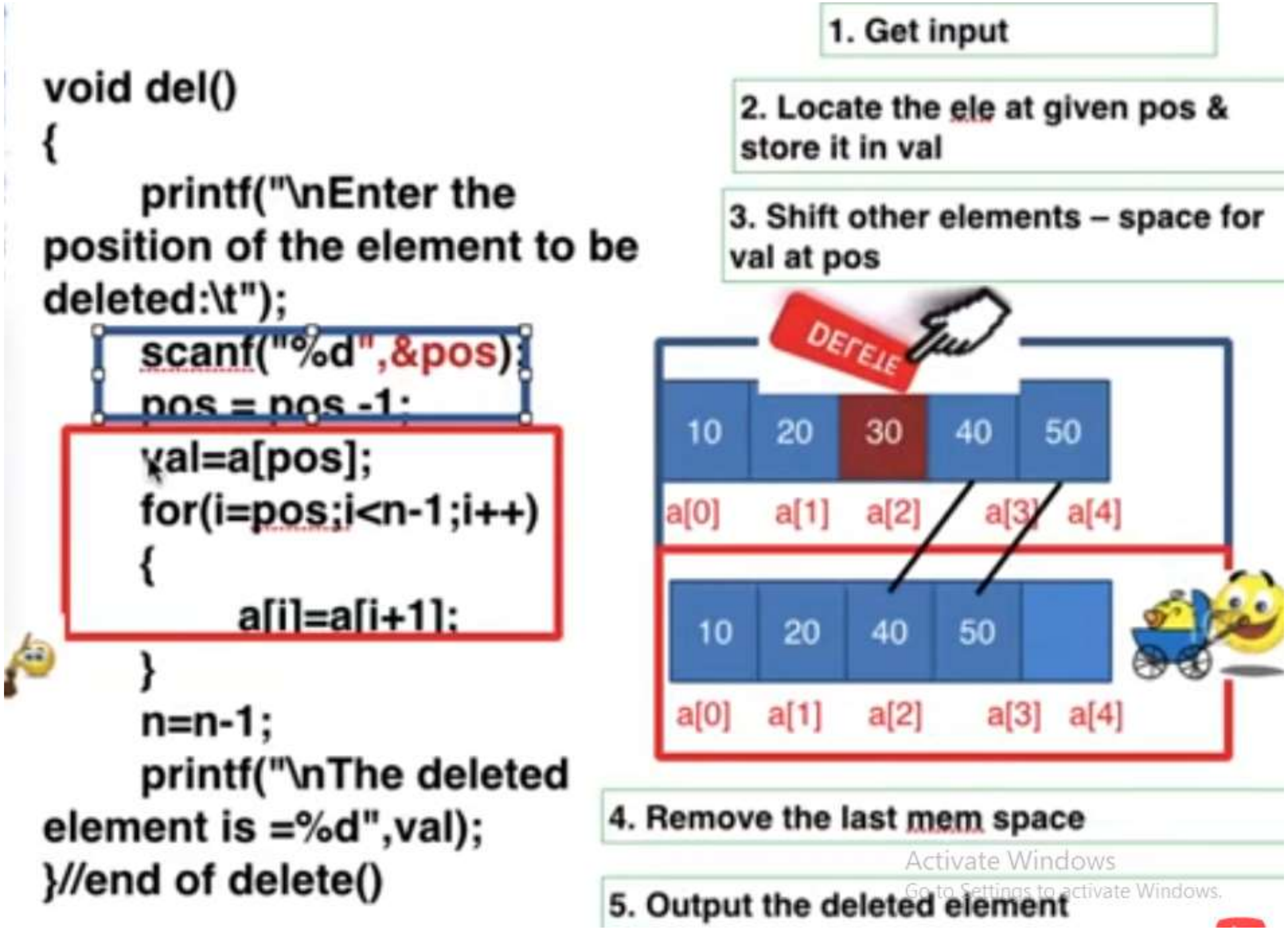
2. Locate the ele at given pos & store it in val

3. Shift other elements – space for val at pos

4. Remove the last mem space

5. Output the deleted element

Output:
30 is deleted



```
void del()
{
        printf("\nEnter the
position of the element to be
deleted:\t");
        scanf("%d",&pos);
        pos = pos -1;
        val=a[pos];
        for(i=pos;i<n-1;i++)
        {
                a[i]=a[i+1];
        }
        n=n-1;
        printf("\nThe deleted
element is =%d",val);
}//end of delete()
```

1. Get input

2. Locate the ele at given pos & store it in val

3. Shift other elements – space for val at pos

4. Remove the last mem space

5. Output the deleted element

Advantages of Array

1. Array provides the single name for the group of variables of the same type. Therefore, it is easy to remember the name of all the elements of an array.
2. Traversing an array is a very simple process; we just need to increment the base address of the array in order to visit each element one by one.
3. Any element in the array can be directly accessed by using the index.

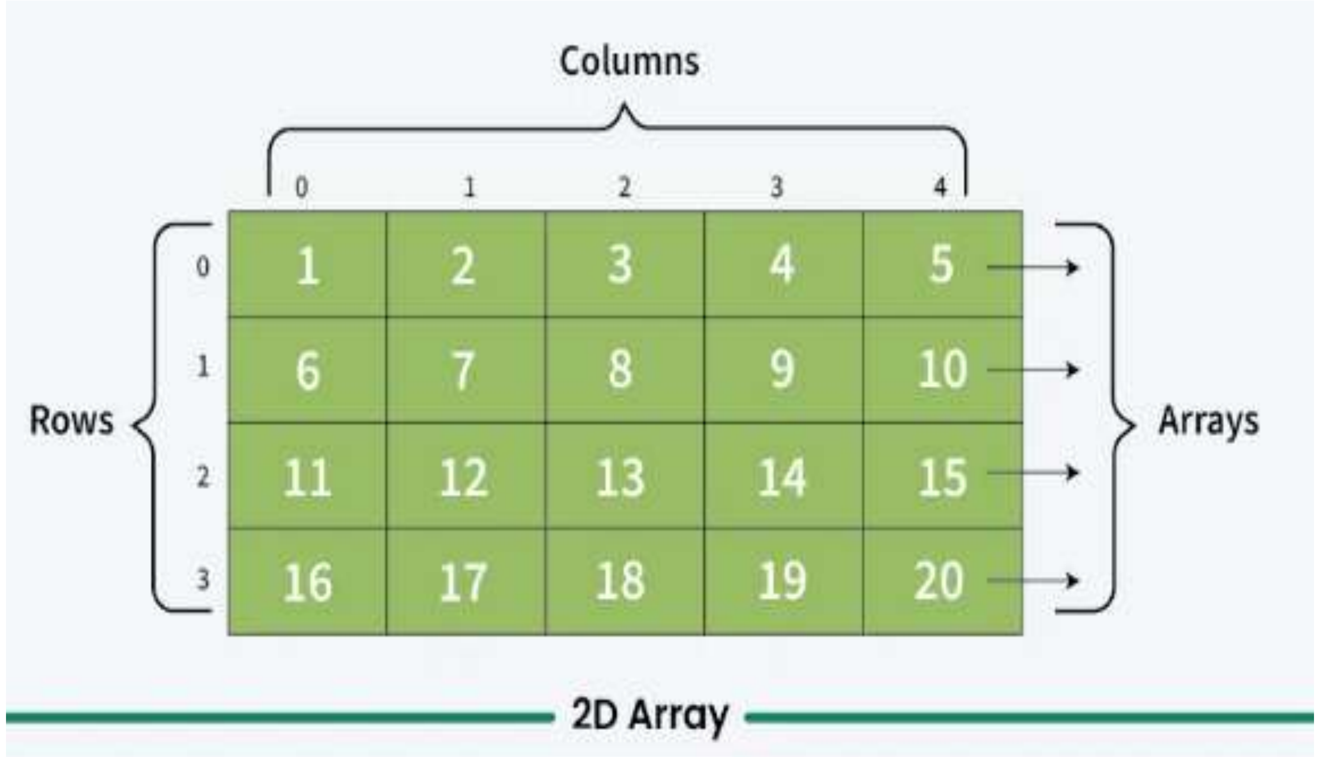
Disadvantages of Array

Array is homogenous. It means that the elements with similar data type can be stored in it.
In array, there is static memory allocation that is size of an array cannot be altered.
There will be wastage of memory if we store less number of elements than the declared size.

2D ARRAY

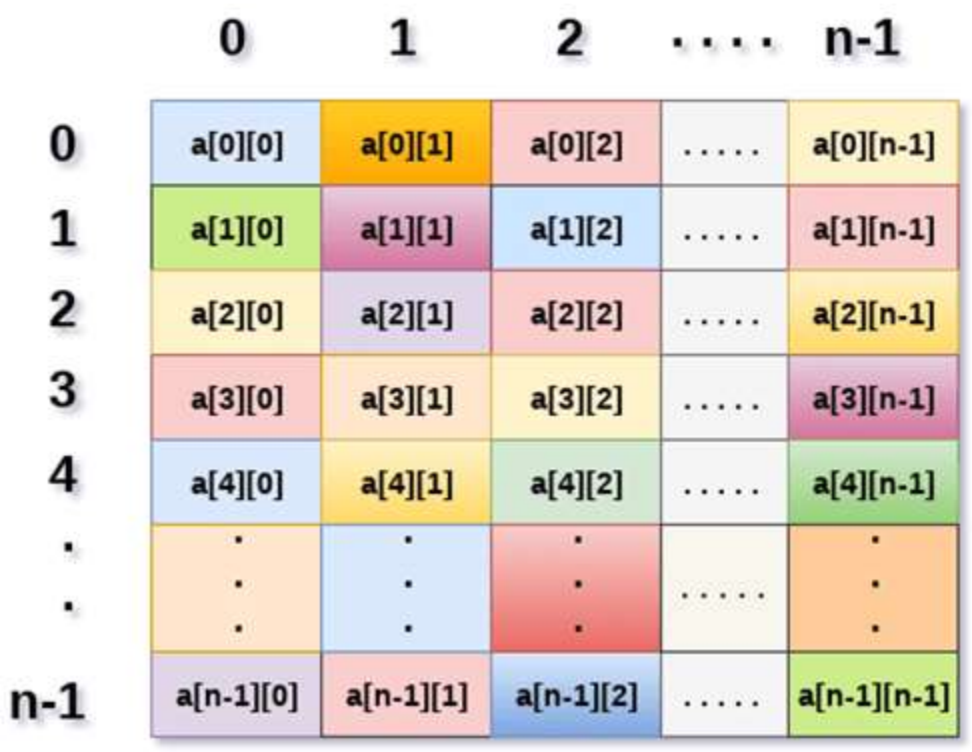
2D array can be defined as an array of arrays. The 2D array is organized as matrices which can be represented as the collection of rows and columns.

```
int arr[max_rows][max_columns];
```



	0	1	2	3	4
0	1	2	3	4	5
1	6	7	8	9	10
2	11	12	13	14	15
3	16	17	18	19	20

2D Array

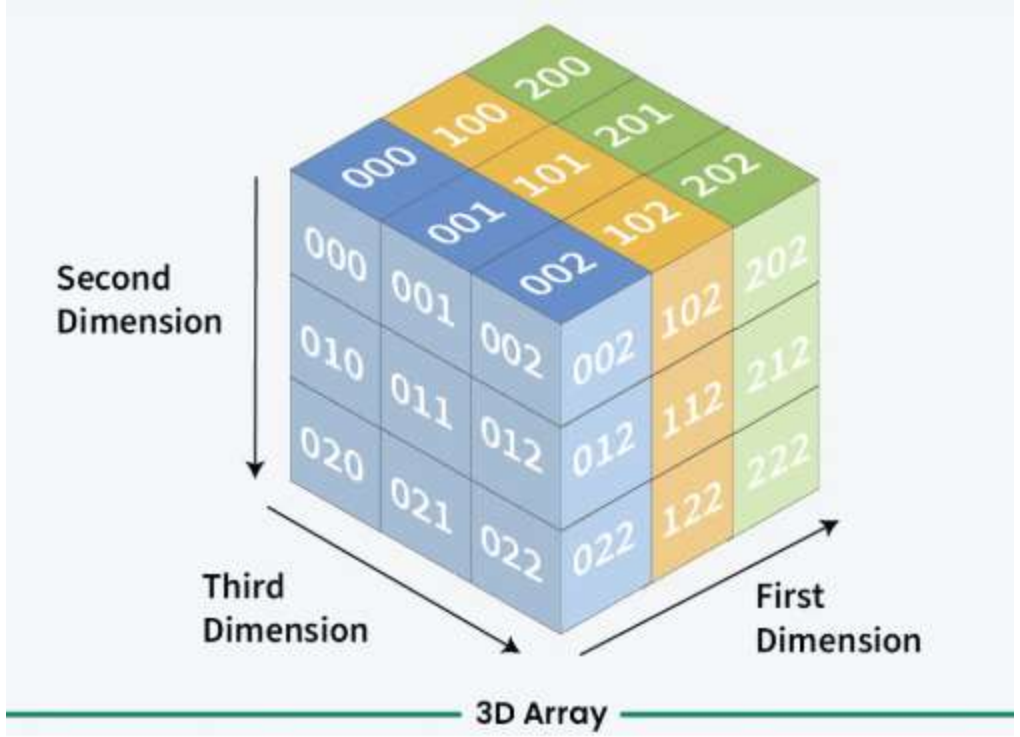


	0	1	2	....	n-1
0	a[0][0]	a[0][1]	a[0][2]	.....	a[0][n-1]
1	a[1][0]	a[1][1]	a[1][2]	.....	a[1][n-1]
2	a[2][0]	a[2][1]	a[2][2]	.....	a[2][n-1]
3	a[3][0]	a[3][1]	a[3][2]	.....	a[3][n-1]
4	a[4][0]	a[4][1]	a[4][2]	.....	a[4][n-1]
. .	. . .	. . .	. . .	.....	. . .
n-1	a[n-1][0]	a[n-1][1]	a[n-1][2]	.....	a[n-1][n-1]

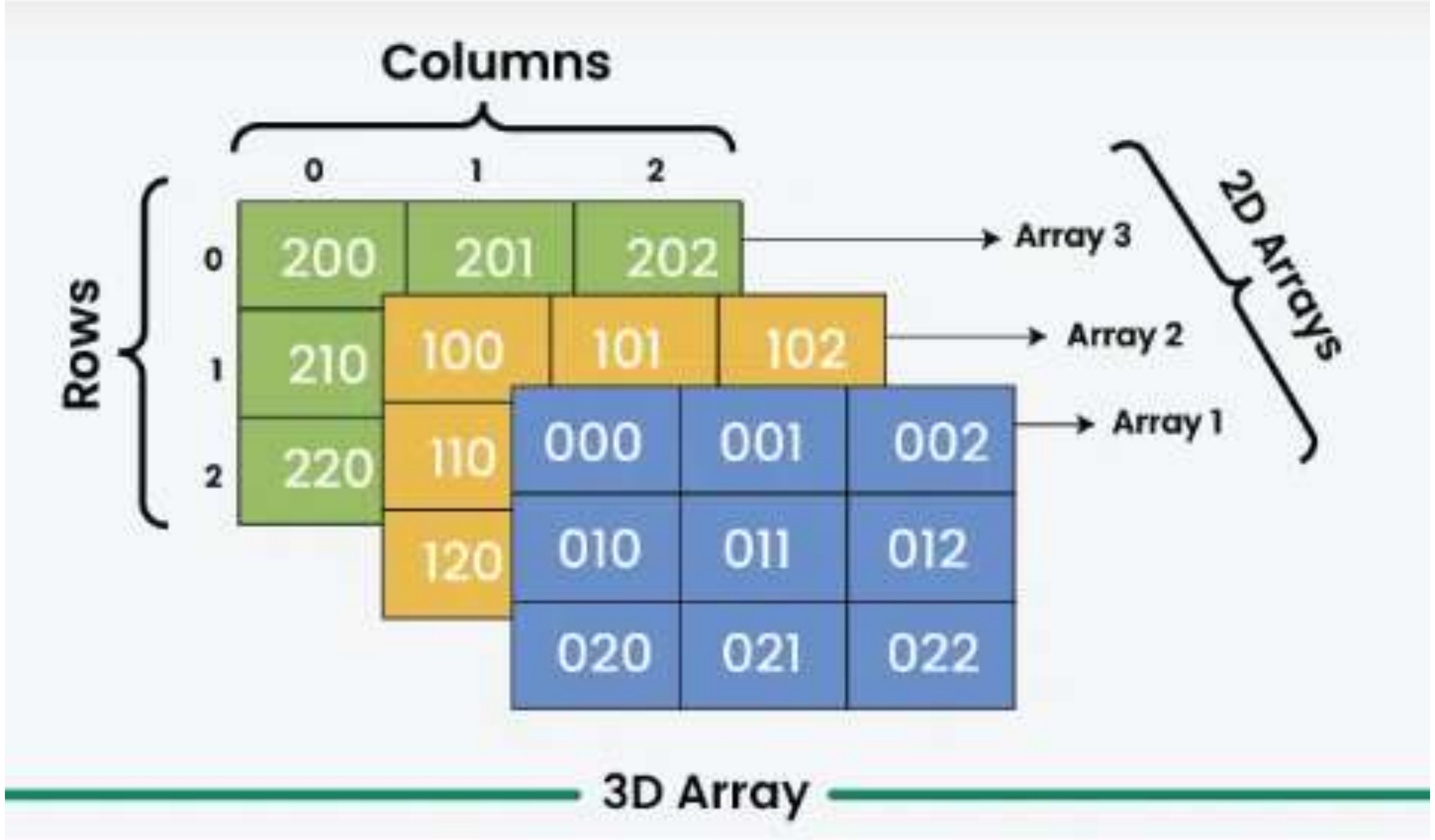
a[n][n]

THREE-DIMENSIONAL (3D) ARRAY

A **Three-Dimensional Array** or **3D** array in C is a collection of two-dimensional arrays. It can be visualized as multiple 2D arrays stacked on top of each other.



HOW 3D ARRAYS ARE STORED IN THE MEMORY?



ARRAY OF STRINGS

A string is a 1-D array of characters, so an array of strings is a 2-D array of characters. Just like we can create a 2-D array of int, float etc; we can also create a 2-D array of character or array of strings. Here is how we can declare a 2-D array of characters.

```
char ch_arr[3][10] = {      {'s', 'p', 'i', 'k', 'e', '\0'},

                            {'t', 'o', 'm','\0'},
                            {'j', 'e', 'r', 'r', 'y','\0'}
                              };

char ch_arr[3][10] = {          "spike",

                            "tom",
                            "jerry"
              };
```

The first subscript of the array i.e 3 denotes the number of strings in the array and the second subscript denotes the maximum length of the string. Recall the that in C, each character occupies 1 byte of data, so when the compiler sees the above statement it allocates 30 bytes (3*10) of memory

POINTERS

Pointers are the variables that are used to store the location of value present in the memory.

A pointer to a location stores its memory address.

The process of obtaining the value stored at a location being referenced by a pointer is known as dereferencing.

It is the same as the index for a textbook where each page is referred by its page number present in the index.

One can easily find the page using the location referred to there. Such pointers usage helps in the dynamic implementation of various data structures such as stack or list.

WHY DO WE NEED POINTERS IN DATA STRUCTURE?

Optimization of our code and improving the time complexity of one algorithm. Using pointers helps reduce the time needed by an algorithm to copy data from one place to another. Since it used the memory locations directly, any change made to the value will be reflected at all the locations.

Example:

1. Call_by_value needs the value of arguments to be copied every time any operation needs to be performed.
2. Call_by_reference makes this task easier using its memory location to update the value at memory locations.

Control Program Flow: Another use of pointers is to control the program flow. This is implemented by control tables that use these pointers.

These pointers are stored in a table to point to each subroutine’s entry point to be executed one after the other.

These pointers reference the addresses of the various procedures. This helps while working with a recursive procedure or traversal of algorithms where there is a need to store the calling step’s location.

Dynamic Memory Allocation: Many programming languages use dynamic memory allocations to allocate the memory for run-time variables.

For such type of memory, allocations heap is used rather than the stack, which uses pointers.

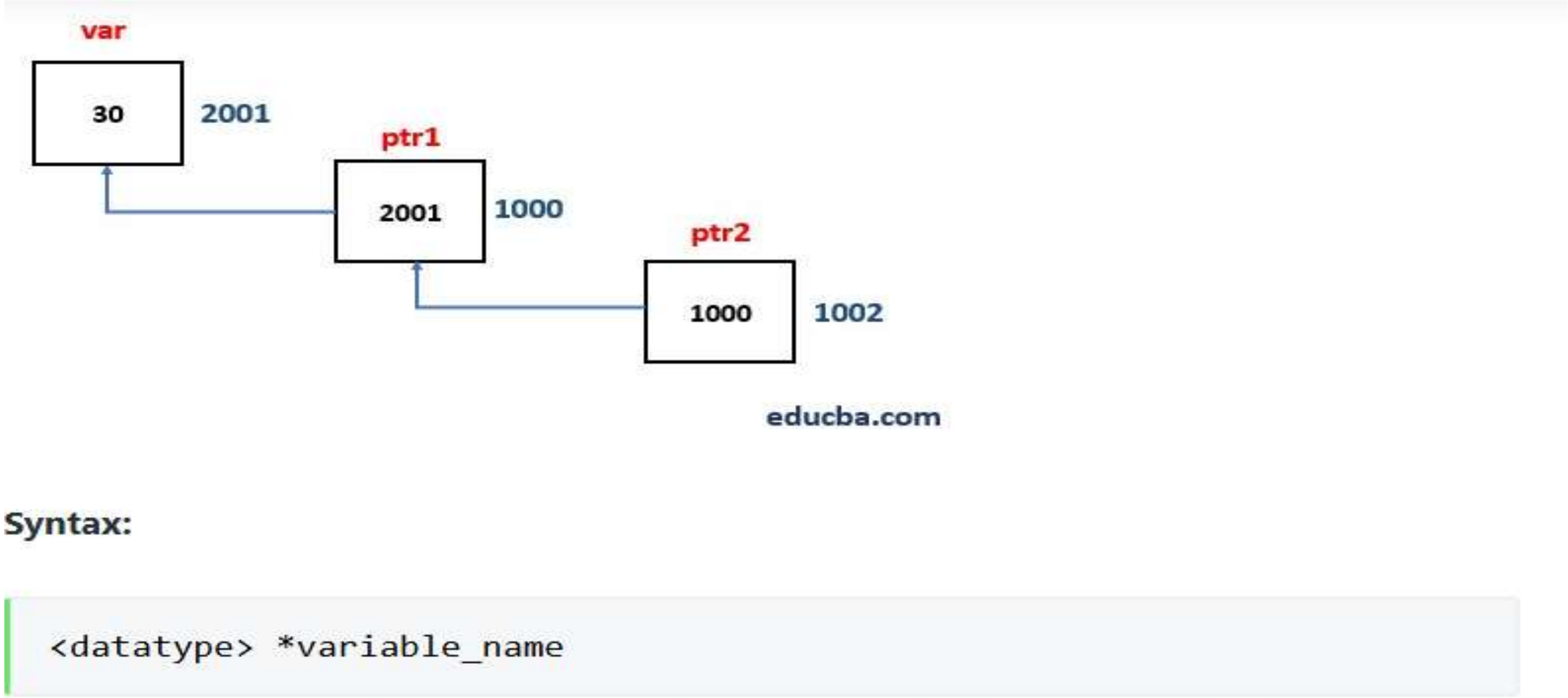
Here pointers hold the address of these dynamically generated data blocks or array of objects.

Many structured or OOPs languages use a heap or free store to provide them with storage locations. The last Node in the linked list is denoted using a NULL pointer that indicates there is no element further in the list.

HOW DO POINTERS WORK IN DATA STRUCTURE?

Pointers are kind of variables that store the address of a variable.

Defining a Pointer



Syntax:

```
<datatype> *variable_name
```

EXAMPLE:

int *ptr1 – ptr1 references to a memory location that holds data of int datatype.
int var = 30;
int *ptr1 = &var; // pointer to var
int **ptr2 = & ptr1; // pointer to pointer variable ptr1

In the above example, ‘&’ is used to denote the unary operator, which returns a variable’s address.
And ‘*’ is a unary operator that returns the value stored at an address specified by the pointer variable,

print(“%d”, *ptr1) // prints 30

DECLARATION OF POINTER VARIABLE

The general syntax of pointer declaration is,

```
datatype *pointer_name;
```

The data type of the pointer and the variable to which the pointer variable is pointing must be the same.

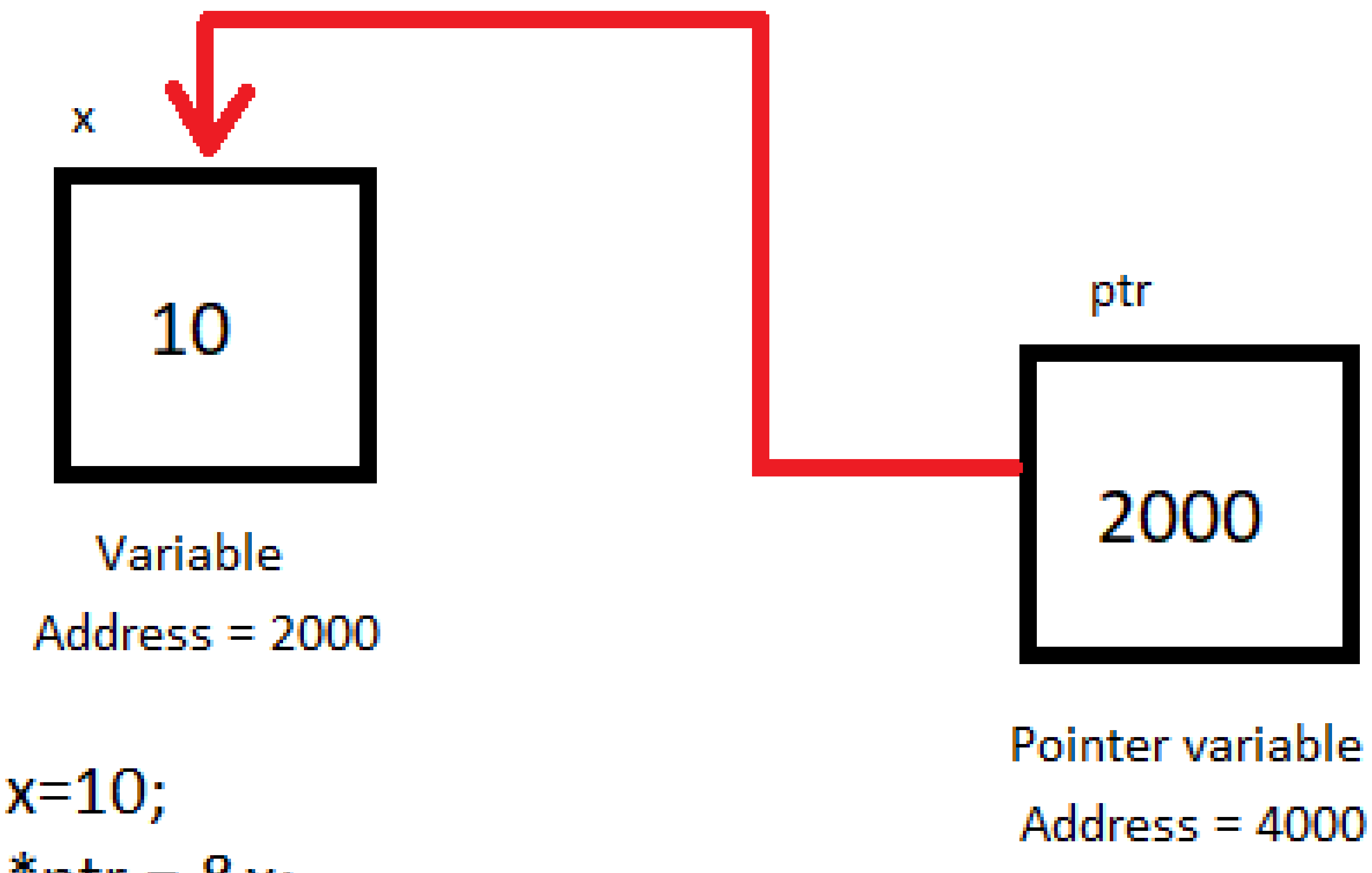
INITIALIZATION OF POINTER VARIABLE

Pointer Initialization is the process of assigning address of a variable to a **pointer** variable.

It contains the address of a variable of the same data type.

address operator & is used to determine the address of a variable. The & (immediately preceding a variable name) returns the address of the variable associated with it.

```
int a = 10;
int *ptr;      //pointer declaration
ptr = &a;      //pointer initialization
```



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

Address of Operator

Pointer variable always points to variables of the same datatype. For example:

```
float a;
int *ptr = &a;       // ERROR, type mismatch
```

While declaring a pointer variable, if it is not assigned to anything then it contains garbage value. Therefore, it is recommended to assign a NULL value to it

A pointer that is assigned a NULL value is called a **NULL pointer**

int *ptr = NULL;

USING THE POINTER OR DEREFERENCING OF POINTER

Once a pointer has been assigned the address of a variable, to access the value of the variable, the pointer is **dereferenced**, using the **indirection operator** or **dereferencing operator** *

```
int main()
{
    int a;
    a = 10;
    int *p = &a;     // declaring and initializing the
pointer

    //prints the value of 'a'
    printf("%d\n", *p);
    printf("%d\n", *&a);
    //prints the address of 'a'
    printf("%u\n", &a);
    printf("%u\n", p);

    printf("%u\n", &p);    //prints address of 'p'

    return 0;
}
```

POINTER ARITHMETIC

We can perform arithmetic operations on the pointers like addition, subtraction, etc.

However, as we know that pointer contains the address, the result of an arithmetic operation performed on the pointer will also be a pointer if the other operand is of type integer.

In pointer-from-pointer subtraction, the result will be an integer value.

- Increment
- Decrement
- Addition
- Subtraction
- Comparison

INCREMENTING POINTER

1. If we increment a pointer by 1, the pointer will start pointing to the immediate next location.
2. This is somewhat different from the general arithmetic since the value of the pointer will get increased by the size of the data type to which the pointer is pointing.
3. We can traverse an array by using the increment operation on a pointer which will keep pointing to every element of the array, perform some operation on that, and update itself in a loop.

The Rule to increment the pointer is given below:

new_address= current_address + i * size_of(data type)

Where i is the number by which the pointer get increased.

32-bit

For 32-bit int variable, it will be incremented by 2 bytes.

64-bit

For 64-bit int variable, it will be incremented by 4 bytes.

EXAMPLE OF INCREMENTING POINTER VARIABLE ON 64-BIT ARCHITECTURE.

```
#include<stdio.h>

int main(){

int number=50;

int *p;//pointer to int

p=&number;//stores the address of number variable

printf("Address of p variable is %u \n",p);

p=p+1;

printf("After increment: Address of p variable is %u \n",p); // in our case, p will get incremented
        by 4 bytes.

return 0;

}
```

OUTPUT

```
Address of p variable is 3214864300
After increment: Address of p variable is 3214864304
```

TRAVERSING AN ARRAY BY USING POINTER

```
#include<stdio.h>

void main ()

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

    int *p = arr;

    int i;

    printf("printing array elements...\n");

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

    {

        printf("%d  ",*(p+i));

   }

}
```

OUTPUT

```
printing array elements...
1 2 3 4 5
```

DECREMENTING POINTER

Like increment, we can decrement a pointer variable. If we decrement a pointer, it will start pointing to the previous location. The formula of decrementing the pointer is given below:

new_address= current_address - i * size_of(data type)

32-bit

For 32-bit int variable, it will be decremented by 2 bytes.

64-bit

For 64-bit int variable, it will be decremented by 4 bytes.

EXAMPLE OF DECREMENTING POINTER VARIABLE ON 64-BIT OS.

```
#include <stdio.h>
void main(){
int number=50;
int *p;//pointer to int
p=&number;//stores the address of number variable
printf("Address of p variable is %u \n",p);
p=p-1;
printf("After decrement: Address of p variable is %u \n",p); // P will now point
                                             to the immediate previous location.
}
```

Output

```
Address of p variable is 3214864300
 After decrement: Address of p variable is 3214864296
```

POINTER ADDITION

We can add a value to the pointer variable. The formula of adding value to pointer is given below:

new_address= current_address + (number * size_of(data type))

32-bit
For 32-bit int variable, it will add 2 * number.

64-bit
For 64-bit int variable, it will add 4 * number.

EXAMPLE OF ADDING VALUE TO POINTER VARIABLE ON 64-BIT ARCHITECTURE.

```
#include<stdio.h>
int main(){
int number=50;
int *p;//pointer to int
p=&number;//stores the address of number variable
printf("Address of p variable is %u \n",p);
p=p+3;   //adding 3 to pointer variable
printf("After adding 3: Address of p variable is %u \n",p);
return 0;
}
```

Output:

```
Address of p variable is 3214864300
 After adding 3: Address of p variable is 3214864312
```

As you can see, the address of p is 3214864300. But after adding 3 with p variable, it is 3214864312, i.e., 4*3=12 increment.

Since we are using 64-bit architecture, it increments 12. But if we were using 32-bit architecture, it was incrementing to 6 only, i.e., 2*3=6. As integer value occupies 2-byte memory in 32-bit OS.

SUBTRACTION OF TWO POINTERS

When you subtract two pointers that point to the same data type, the result is calculated by finding the difference between their addresses and then dividing by the size of the data type to determine how many elements (not bytes) separate the two pointers.

1. You have two integer pointers, ptr1 with an address of 1000 and ptr2 with an address of 1004.
2. When you subtract these two pointers (ptr2 - ptr1), you get a result of 4 bytes, which is the difference in addresses.
3. Since the size of an integer is 4 bytes, you divide the address difference by the size of the data type: (4 / 4), which equals 1. This means there is an increment of 1 integer-sized element between ptr1 and ptr2.

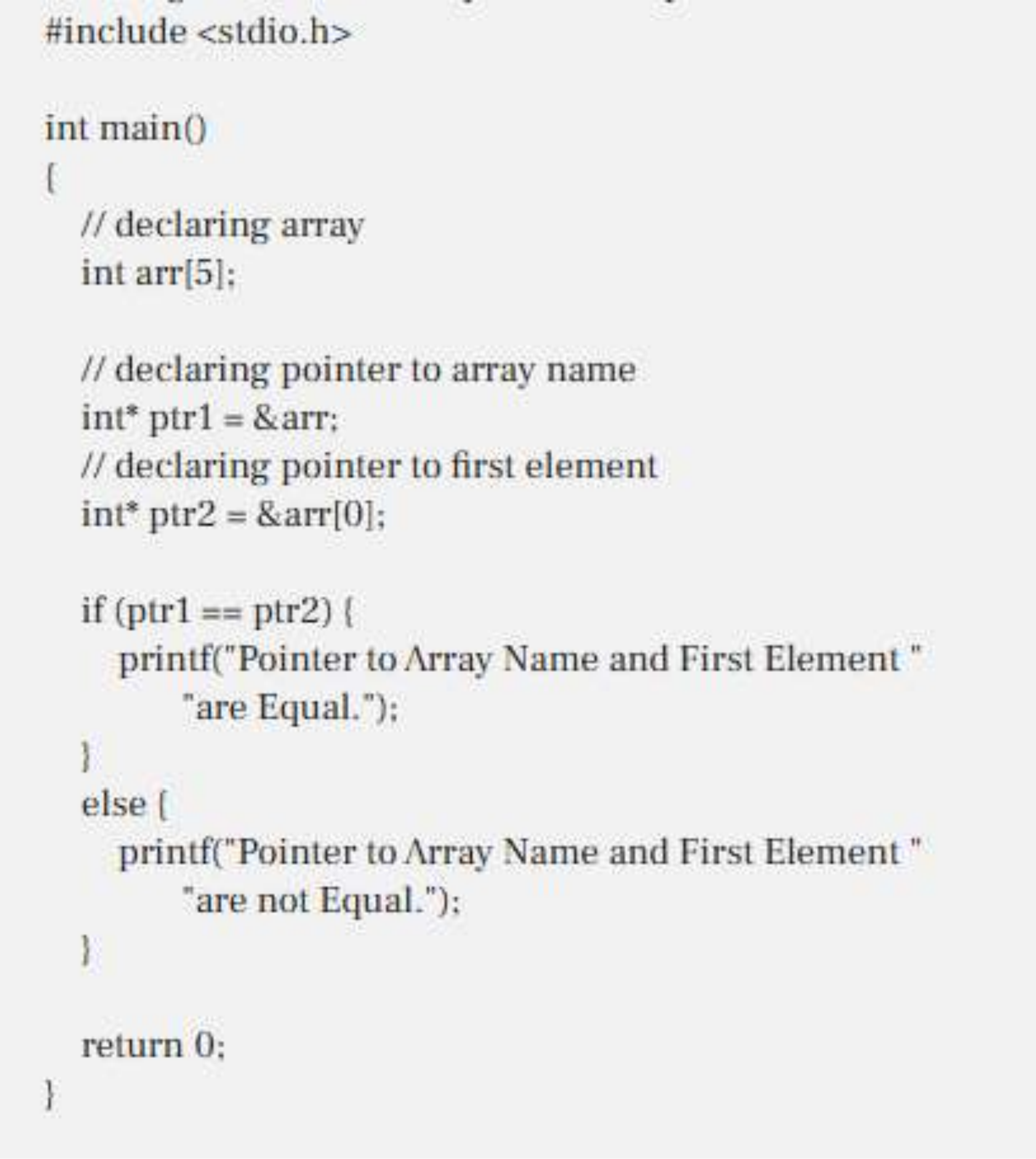
COMPARISON OF POINTERS

You can compare pointers using operators like >, >=, <, <=, ==, and !=. These operators return true for valid conditions and false for unsatisfied conditions.

Step 1: Initialize integer values and point these integer values to the pointers.

Step 2: Use comparison or relational operators to check conditions on the pointer variables.

Step 3: Display the output, which will indicate whether the specified conditions between the pointers are met.



```
#include <stdio.h>

int main()
{
    // declaring array
    int arr[5];

    // declaring pointer to array name
    int* ptr1 = &arr;
    // declaring pointer to first element
    int* ptr2 = &arr[0];

    if (ptr1 == ptr2) {
        printf("Pointer to Array Name and First Element "
               "are Equal.");
    }
    else {
        printf("Pointer to Array Name and First Element "
               "are not Equal.");
    }

    return 0;
}
```

FUNCTION POINTERS

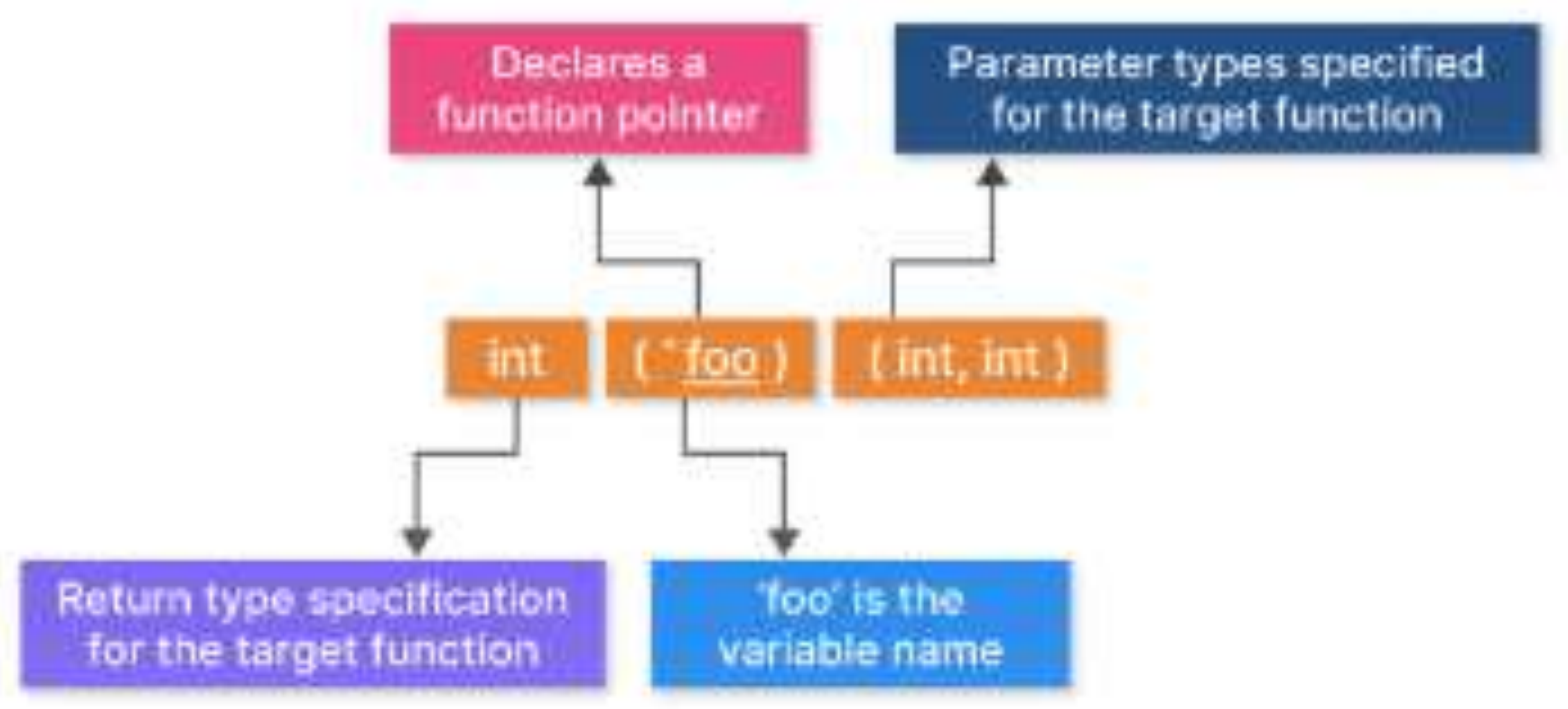
Pointer variables that point to the address of a function are called function pointers .

We can use them to call functions indirectly, pass functions as arguments, and more. They enhance flexibility and modularity in code while removing redundancies.

Function pointers are a powerful feature that allows us to store the address of a function in a pointer variable.

Function pointers are essential in various programming scenarios, such as implementing callback functions, designing state machines, and optimizing performance-critical applications.

DECLARATION OF FUNCTION POINTERS


Declares a function pointer
Parameter types specified for the target function
int
(*foo)
(int, int)
Return type specification for the target function
'foo' is the variable name

(type) (*pointer_name)(parameter);

In the above syntax, the **type** is the variable type which is returned by the function, ***pointer_name** is the function pointer, and the **parameter** is the list of the argument passed to the function.

float (*add)(); // this is a legal declaration for the function pointer
float *add(); // this is an illegal declaration for the function pointer

A function pointer can also point to another function, or we can say that it holds the address of another function.

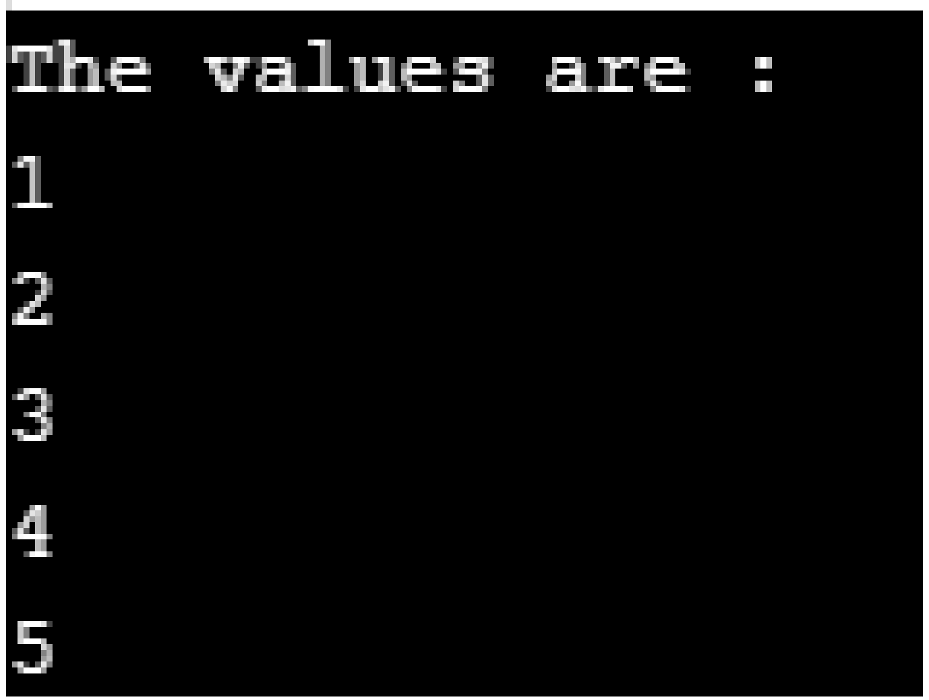
```
float add (int a, int b);  // function declaration
float (*a)(int, int);  // declaration of a pointer to a function
a=add; // assigning address of add() to 'a' pointer
```

Now, 'a' is a pointer pointing to the add() function. We can call the add() function by using the pointer, i.e., 'a'. Let's see how we can do that:

```
a(2, 3);
```

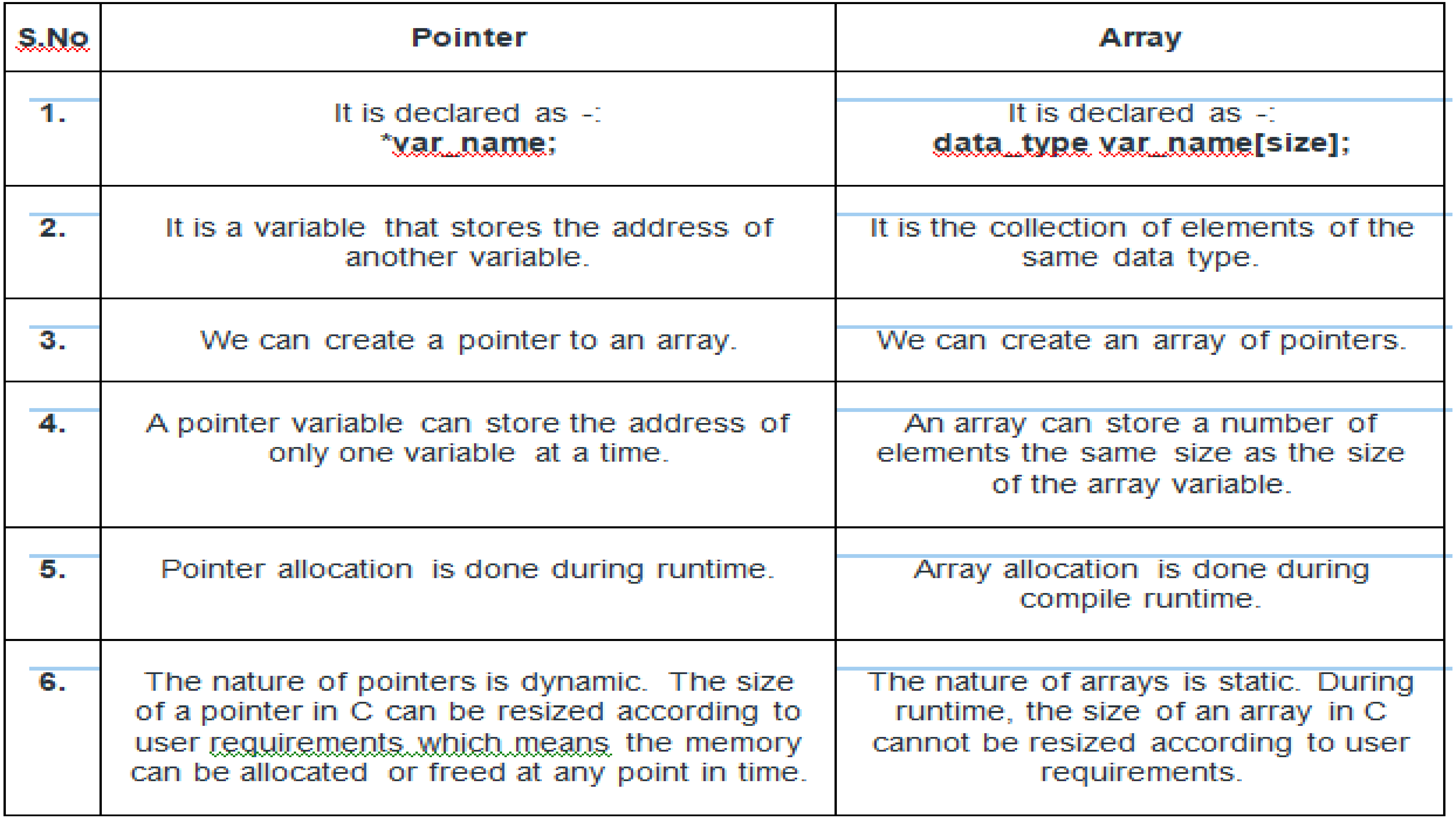
The above statement calls the add() function by using pointer 'a', and two parameters are passed in 'a', i.e., 2 and 3.

```
void display(void (*p)())
{
    for(int i=1;i<=5;i++)
    {
        p(i);
    }
}
void print_numbers(int num)
{
    cout<<num;
}
int main()
{
    void (*p)(int);     // void function pointer declaration
    printf("The values are :");
  display(print_numbers);
    return 0;
}
```



```
The values are :
1
2
3
4
5
```

1. We have defined two functions named 'display()' and print_numbers().
2. Inside the main() method, we have declared a function pointer named as (*p), and we call the display() function in which we pass the print_numbers() function.
3. When the control goes to the display() function, then pointer *p contains the address of print_numbers() function. It means that we can call the print_numbers() function using function pointer *p.
4. In the definition of display() function, we have defined a 'for' loop, and inside the for loop, we call the print_numbers() function using statement p(i). Here, p(i) means that print_numbers() function will be called on each iteration of i, and the value of 'i' gets printed.



S.No	Pointer	Array
1.	It is declared as -: ***var_name;**	It is declared as -: **data_type var_name[size];**
2.	It is a variable that stores the address of another variable.	It is the collection of elements of the same data type.
3.	We can create a pointer to an array.	We can create an array of pointers.
4.	A pointer variable can store the address of only one variable at a time.	An array can store a number of elements the same size as the size of the array variable.
5.	Pointer allocation is done during runtime.	Array allocation is done during compile runtime.
6.	The nature of pointers is dynamic. The size of a pointer in C can be resized according to user requirements which means the memory can be allocated or freed at any point in time.	The nature of arrays is static. During runtime, the size of an array in C cannot be resized according to user requirements.

POINTER TO AN ARRAY

A pointer to an array is a pointer that points to the whole array instead of the first element of the array. It considers the whole array as a single unit instead of it being a collection of given elements.

```
int main()
 {
  int arr[5] = { 1, 2, 3, 4, 5 };
  int *ptr = arr;
  printf("%p\n", ptr);
  return 0;
}
```

Output: 0x7ffde273ac20

SYNTAX OF POINTER TO ARRAY

type**(****ptr*****)**[size];

where,

type: Type of data that the array holds.

ptr: Name of the pointer variable.

size: Size of the array to which the pointer will point.

For example,
int (*ptr)[10];

```c
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
```

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
```

Note that the last number of each of the elements' memory address is different, with an addition of 4.
It is because the size of an `int` type is typically 4 bytes

```
// Create an int variable
int myInt;

// Get the memory size of an int
printf("%lu", sizeof(myInt));
```

Result:

```
4
```

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

// Get the size of the myNumbers array
printf("%lu", sizeof(myNumbers));
```

Result:

```
16
```

you can see that the compiler reserves 4 bytes of memory for each array element, which means that the entire array takes up 16 bytes (4 * 4) of memory storage

DYNAMIC MEMORY ALLOCATION

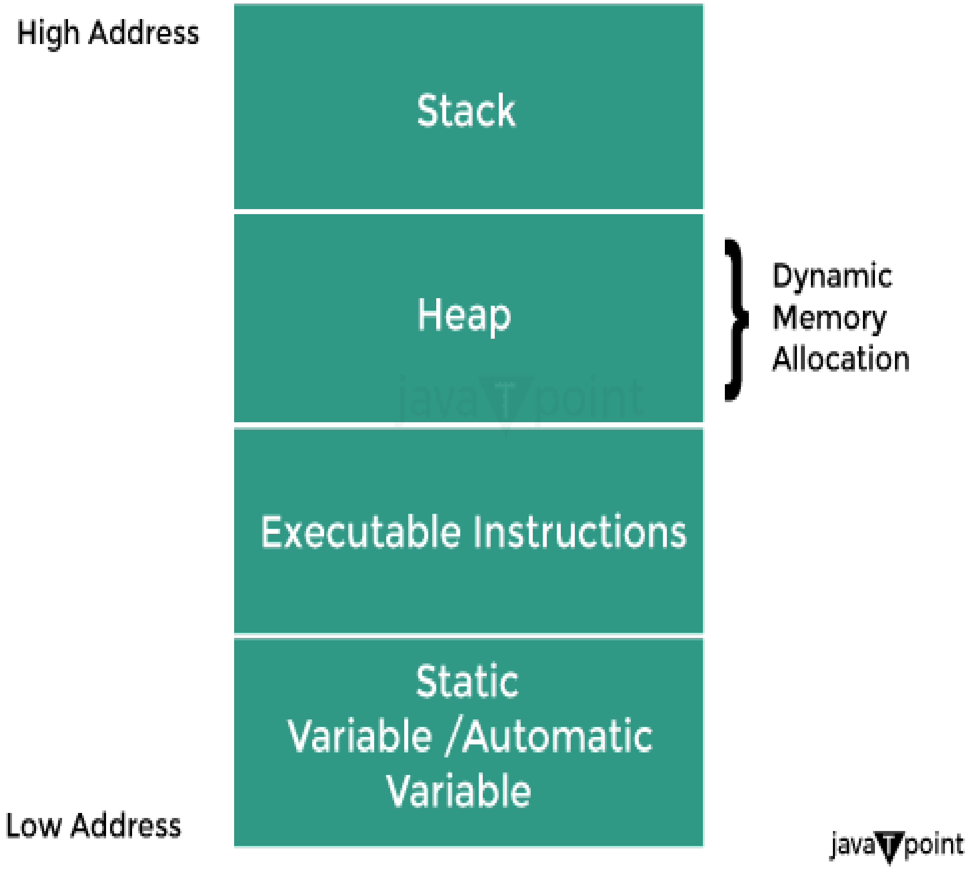
Memory allocation is the process of reserving virtual or physical computer space for a specific purpose (e.g., for computer programs and services to run). Memory allocation is part of the management of computer memory resources, known as memory management.

Dynamic memory allocation is a fundamental concept in data structures and programming.

It allows programs to allocate memory at runtime, providing flexibility and efficiency when working with data structures of varying sizes.

In most programming languages, including C++, memory can be classified into two categories: stack memory and heap memory. Local variables and function calls are stored in the stack memory, whereas the more adaptable heap memory can be allocated and released at runtime.

The process of allocating and releasing memory from the heap is known as dynamic memory allocation. It allows the programmer to manage memory explicitly, providing the ability to create data structures of varying sizes and adjust memory requirements dynamically.



DIFFERENCES

static memory allocation	dynamic memory allocation
memory is allocated at compile time.	memory is allocated at run time.
memory can't be increased while executing program.	memory can be increased while executing program.
used in array.	used in linked list.

METHODS USED FOR DYNAMIC MEMORY ALLOCATION.

malloc()	allocates single block of requested memory.
calloc()	allocates multiple block of requested memory.
realloc()	reallocates the memory occupied by malloc() or calloc() functions.
free()	frees the dynamically allocated memory.

MALLOC() FUNCTION

The malloc() function allocates single block of requested memory.

It doesn't initialize memory at execution time, so it has garbage value initially.

It returns NULL if memory is not sufficient.

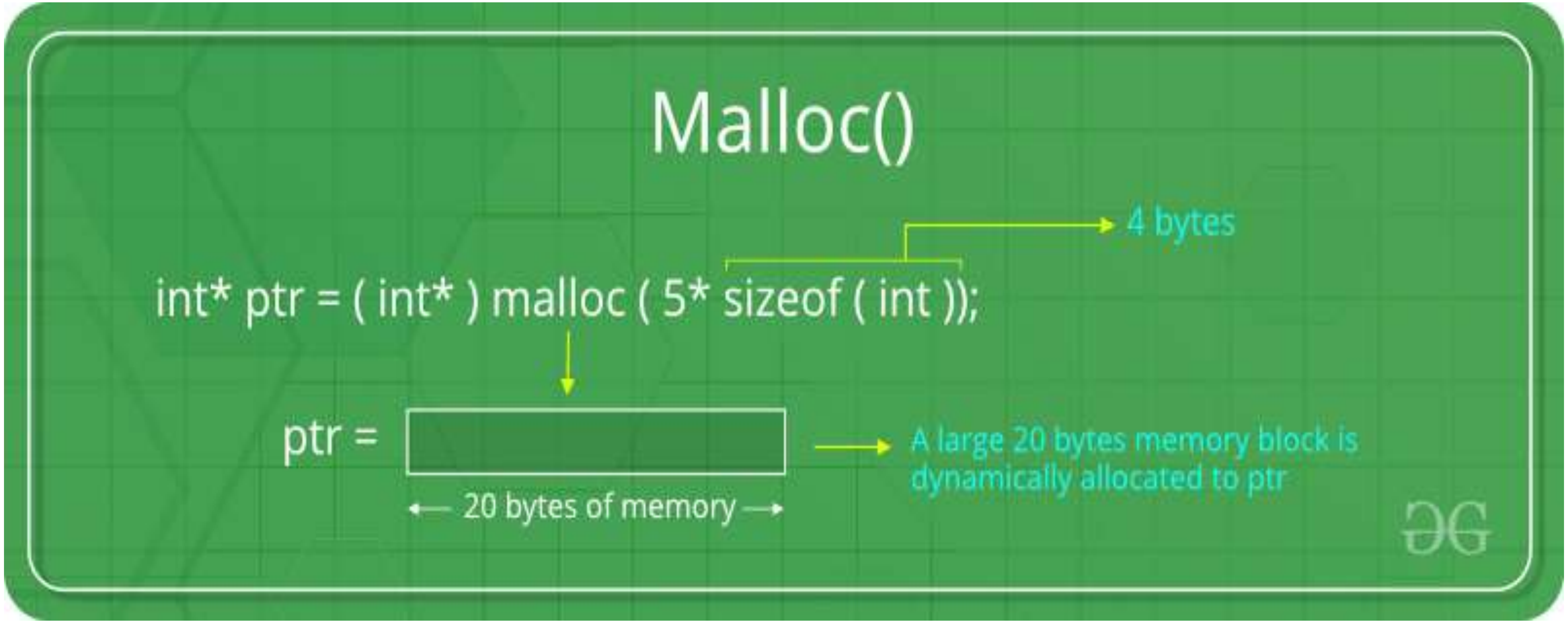
The syntax of malloc() function is given below:

ptr=(cast-type*)malloc(byte-size)

Example:

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

> *Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory. And, the pointer ptr holds the address of the first byte in the allocated memory.*



Malloc()

```
int* ptr = ( int* ) malloc ( 5* sizeof ( int ));
```

A large 20 bytes memory block is dynamically allocated to ptr

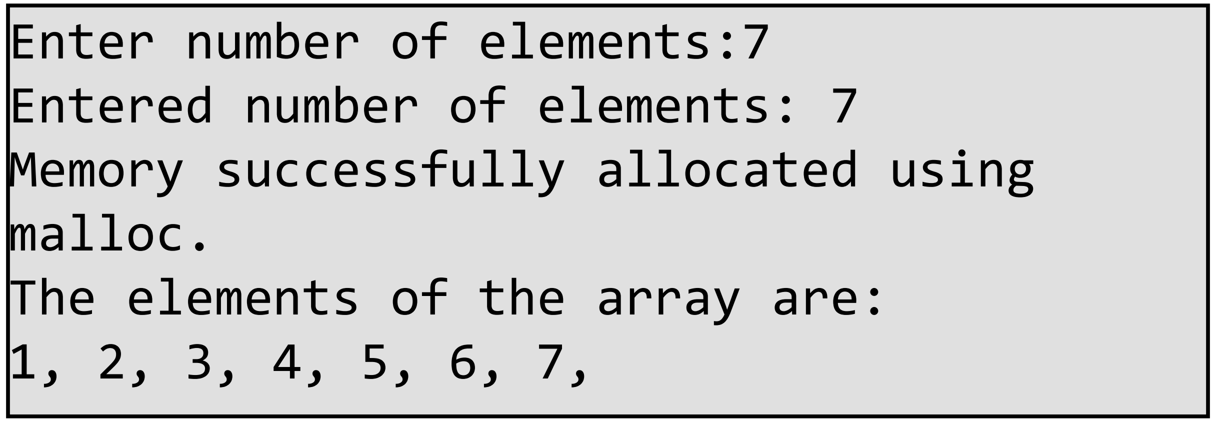
If space is insufficient, allocation fails and returns a NULL pointer.

```
int main(){
  int n,i,*ptr,sum=0;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    ptr=(int*)malloc(n*sizeof(int));  //memory allocated using malloc
    if(ptr==NULL)
    {
        printf("Sorry! unable to allocate memory");
        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;
}
```

```
Enter elements of array: 3
Enter elements of array:
10 10 10
Sum=30
```

```
int main()
{

    // This pointer will hold the
    // base address of the block created
    int* ptr;
    int n, i;

    // Get the number of elements for
the array
    printf("Enter number of elements:");
    scanf("%d",&n);
    printf("Entered number of elements:
%d\n", n);

    // Dynamically allocate memory
using malloc()
    ptr = (int*)malloc(n * sizeof(int));

    // Check if the memory has been
successfully
    // allocated by malloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {
```

```
// Memory has been successfully allocated
        printf("Memory successfully allocated using malloc.\n");

        // Get the elements of the array
        for (i = 0; i < n; ++i) {
            ptr[i] = i + 1;
        }

        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }
    }

    return 0;
}
```



```
Enter number of elements:7
Entered number of elements: 7
Memory successfully allocated using
malloc.
The elements of the array are:
1, 2, 3, 4, 5, 6, 7,
```

CALLOC() FUNCTION

The calloc() function allocates multiple block of requested memory.

It initially initialize all bytes to zero.
It returns NULL if memory is not sufficient.

The syntax of calloc() function is given below:

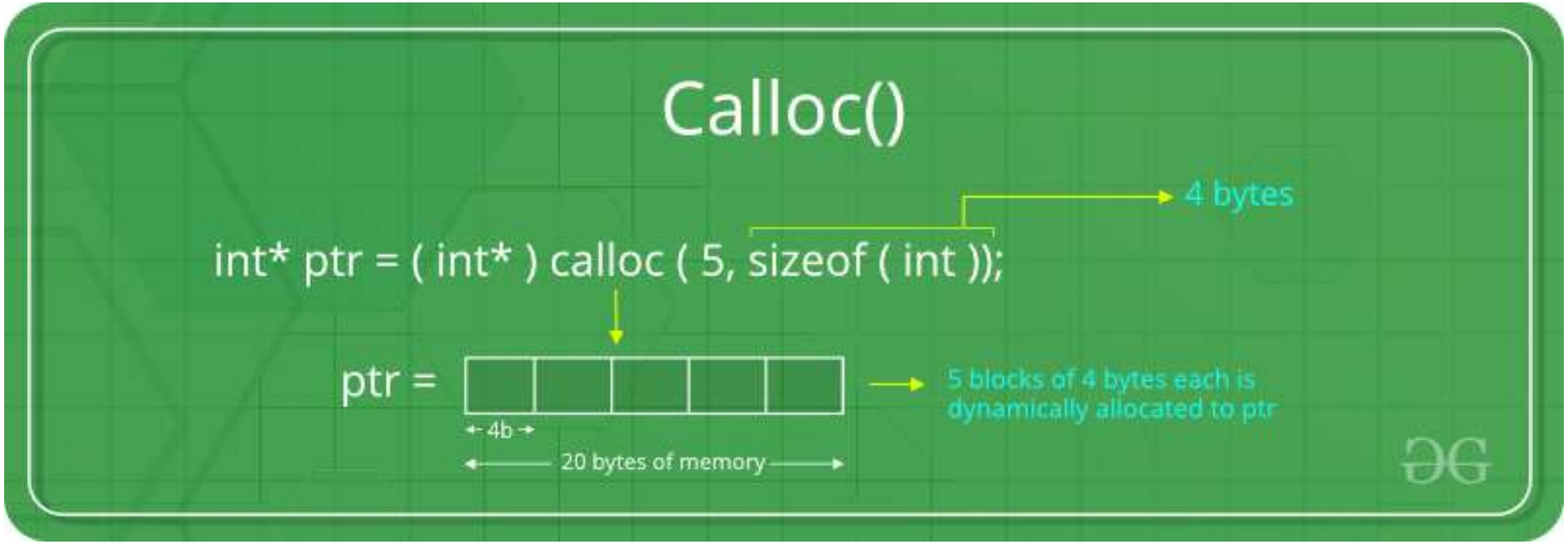
ptr = (cast-type*)calloc(n, element-size);

here, n is the no. of elements and element-size is the size of each element.

Ex:

ptr = (float*) calloc(25, sizeof(float));

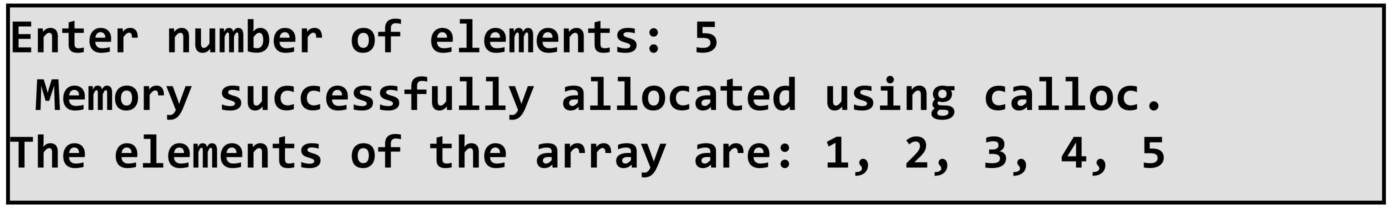
This statement allocates contiguous space in memory for 25 elements each with the size of the float.



```
int* ptr = ( int* ) calloc ( 5, sizeof ( int ));
```

If space is insufficient, allocation fails and returns a NULL pointer.

```
int main()
{

    // This pointer will hold the
    // base address of the block created
    int* ptr;
    int n, i;

    // Get the number of elements for the array
    n = 5;
    printf("Enter number of elements: %d\n", n);

    // Dynamically allocate memory using calloc()
    ptr = (int*)calloc(n, sizeof(int));

    // Check if the memory has been successfully
    // allocated by calloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {
 // Memory has been successfully allocated
        printf("Memory successfully allocated using calloc.\n");

        // Get the elements of the array
        for (i = 0; i < n; ++i) {
            ptr[i] = i + 1;
        }

        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }
    }

    return 0;
}
```



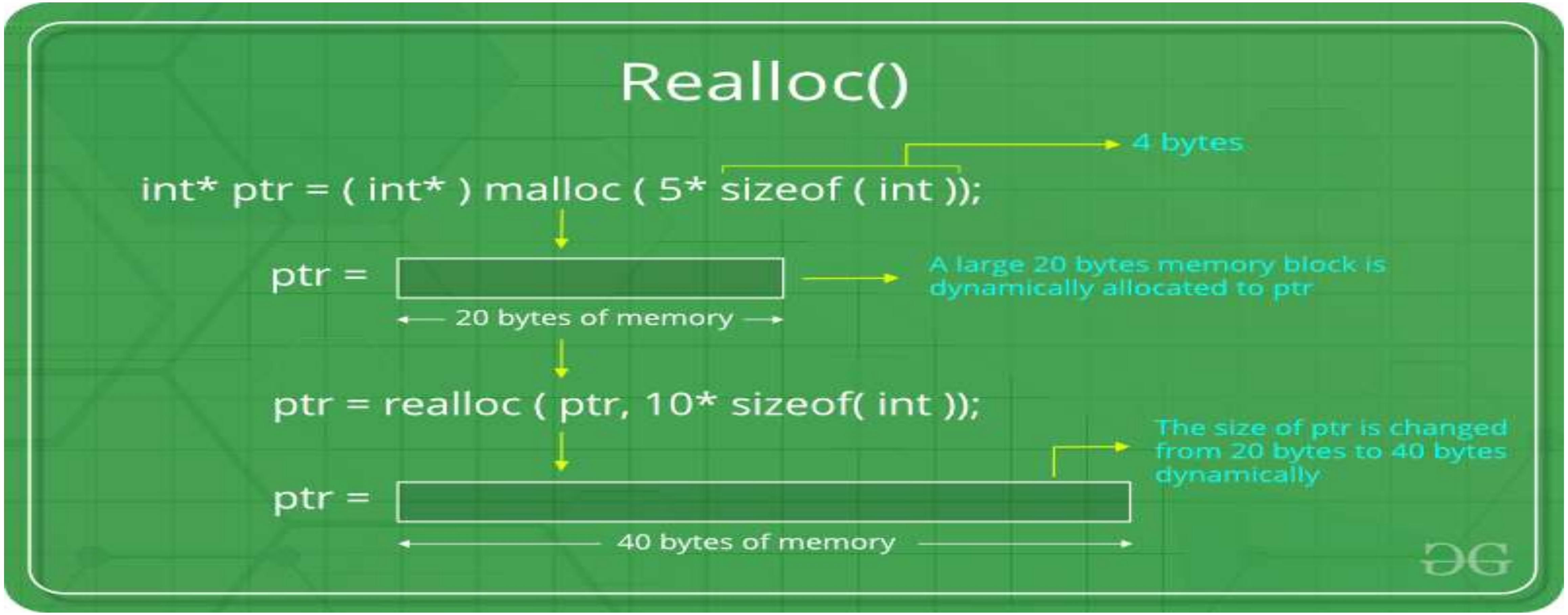
```
Enter number of elements: 5
 Memory successfully allocated using calloc.
The elements of the array are: 1, 2, 3, 4, 5
```

REALLOC() FUNCTION

If memory is not sufficient for malloc() or calloc(), you can reallocate the memory by realloc() function. In short, it changes the memory size.

Syntax

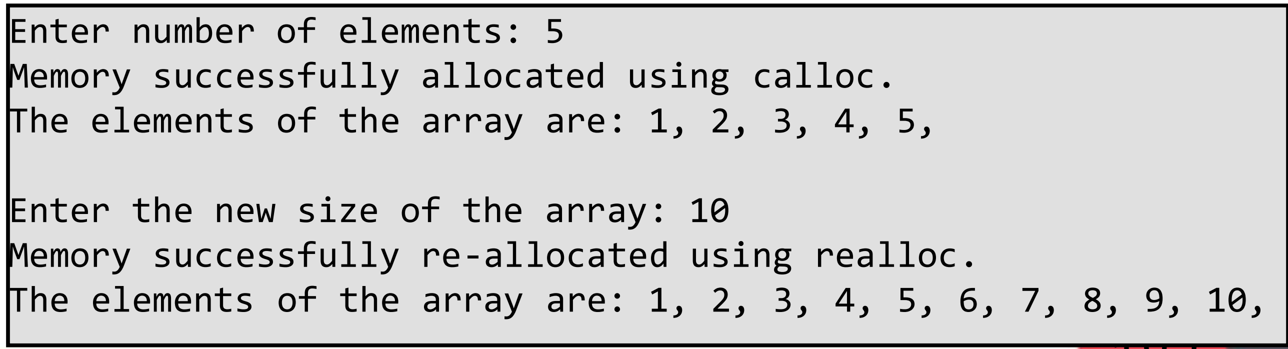
ptr = realloc(ptr, newSize);
where ptr is reallocated with new size 'newSize'.


Realloc()
4 bytes
int* ptr = (int*) malloc (5* sizeof (int));
ptr =
A large 20 bytes memory block is dynamically allocated to ptr
20 bytes of memory
ptr = realloc (ptr, 10* sizeof(int));
The size of ptr is changed from 20 bytes to 40 bytes dynamically
ptr =
40 bytes of memory

If space is insufficient, allocation fails and returns a NULL pointer.

```
int main()
{
    // This pointer will hold the
    // base address of the block created
    int* ptr;
    int n, i;

    // Get the number of elements for the array
    n = 5;
    printf("Enter number of elements: %d\n", n);

    // Dynamically allocate memory using calloc()
    ptr = (int*)calloc(n, sizeof(int));

    // Check if the memory has been successfully
    // allocated by malloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {
        // Memory has been successfully allocated
        printf("Memory successfully allocated using calloc.\n");

        // Get the elements of the array
        for (i = 0; i < n; ++i) {
            ptr[i] = i + 1;
        }
        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }


    return 0;
}
```

```
 // Get the new size for the array
        n = 10;
        printf("\n\nEnter the new size of the array: %d\n", n);

        // Dynamically re-allocate memory using realloc()
        ptr = (int*)realloc(ptr, n * sizeof(int));

          if (ptr == NULL) {
          printf("Reallocation Failed\n");
          exit(0);
        }

        // Memory has been successfully allocated
        printf("Memory successfully re-allocated using realloc.\n");

        // Get the new elements of the array
        for (i = 5; i < n; ++i) {
            ptr[i] = i + 1;
        }

        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }

        free(ptr);
    }
```



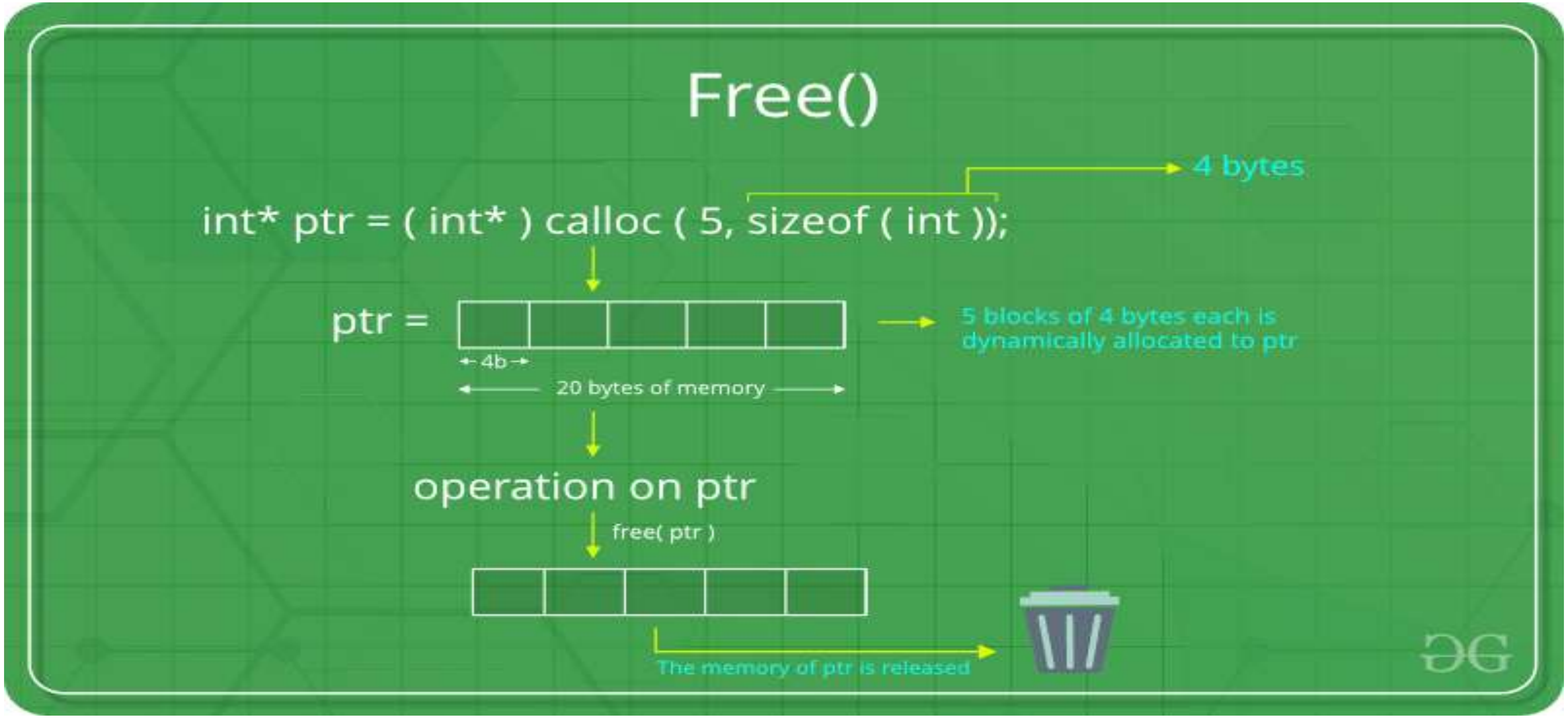
```
Enter number of elements: 5
Memory successfully allocated using calloc.
The elements of the array are: 1, 2, 3, 4, 5,

Enter the new size of the array: 10
Memory successfully re-allocated using realloc.
The elements of the array are: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
```

FREE() FUNCTION

The memory occupied by malloc() or calloc() functions must be released by calling free() function. Otherwise, it will consume memory until program exit.

Syntax

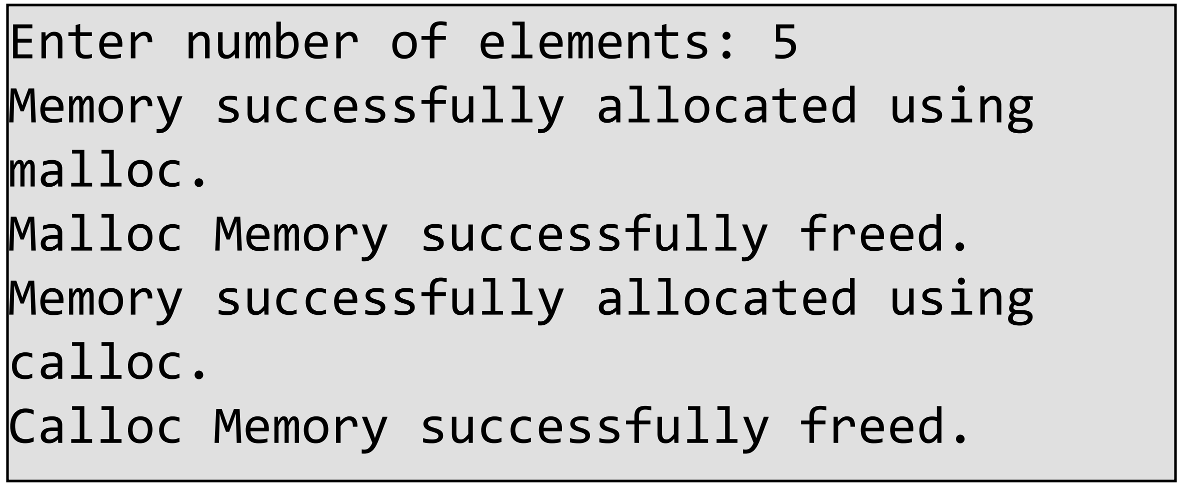
```
free(ptr);
```



```
int* ptr = ( int* ) calloc ( 5, sizeof ( int ));
```

Free()
4 bytes
ptr =
4b
20 bytes of memory
5 blocks of 4 bytes each is dynamically allocated to ptr
operation on ptr
free(ptr)
The memory of ptr is released

```
int main()
{

    // This pointer will hold the
    // base address of the block created
    int *ptr, *ptr1;
    int n, i;

    // Get the number of elements for the array
    n = 5;
    printf("Enter number of elements: %d\n", n);

    // Dynamically allocate memory using malloc()
    ptr = (int*)malloc(n * sizeof(int));

    // Dynamically allocate memory using calloc()
    ptr1 = (int*)calloc(n, sizeof(int));

    // Check if the memory has been successfully
    // allocated by malloc or not
    if (ptr == NULL || ptr1 == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {
// Memory has been successfully allocated
        printf("Memory successfully allocated using malloc.\n");

        // Free the memory
        free(ptr);
        printf("Malloc Memory successfully freed.\n");

        // Memory has been successfully allocated
        printf("\nMemory successfully allocated using calloc.\n");

        // Free the memory
        free(ptr1);
        printf("Calloc Memory successfully freed.\n");
    }

    return 0;
}
```



```
Enter number of elements: 5
Memory successfully allocated using
malloc.
Malloc Memory successfully freed.
Memory successfully allocated using
calloc.
Calloc Memory successfully freed.
```

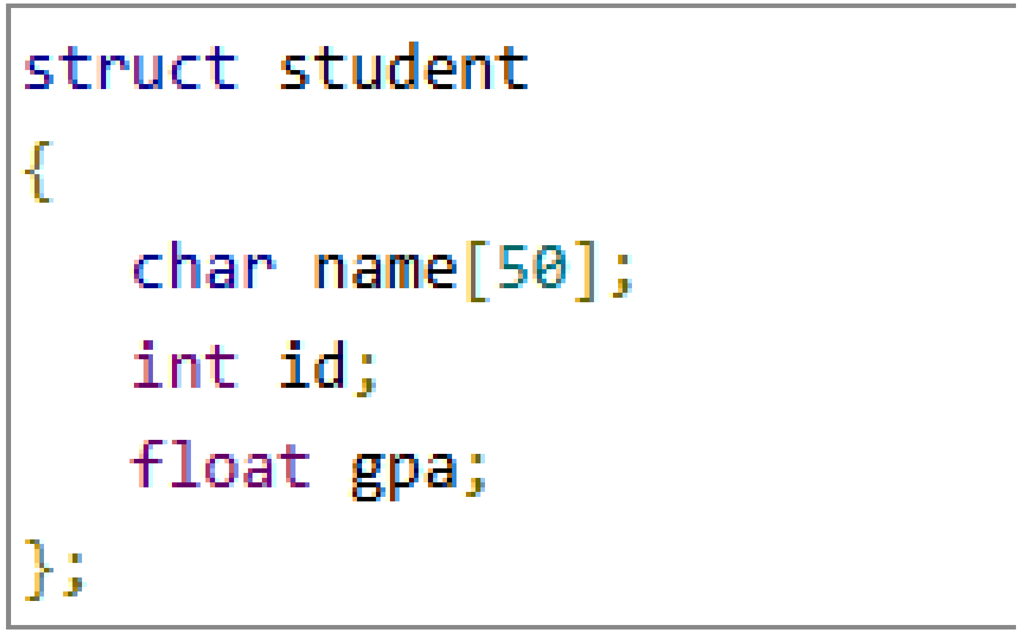
STRUCTURES

Arrays allow you to define the type of variables that can hold many data items of the same type. Similarily In C, a structure is a user-defined data type that groups together different types of data (such as integers, characters, and strings) under a single name.

Syntax for defining a structure

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

For example, you can define a structure called "student" that contains information about a student, such as their name, ID number, and grade point average:



```
struct student
{
   char name[50];
   int id;
   float gpa;
};
```

STRUCTURE DEFINITION

To use structure in our program, we have to define its instance. We can do that by creating variables of the structure type. We can define structure variables using two methods:

1. Structure Variable Declaration with Structure Template

```
 struct structure_name {
data_type member_name1;
data_type member_name1;
....
....
}variable1, varaible2, ...;
```

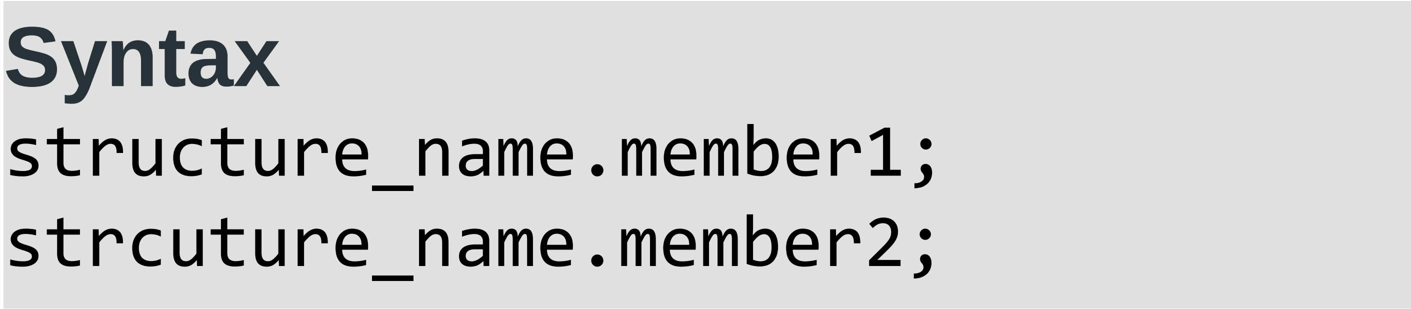
2. Structure Variable Declaration after Structure Template

// structure declared beforehand

struct *structure_name* ***variable1, variable2***, ...;

ACCESS STRUCTURE MEMBERS

We can access structure members by using the **(.) dot operator.**



Syntax

```
structure_name.member1;
strcuture_name.member2;
```

INITIALIZE STRUCTURE MEMBERS

Structure members **cannot be** initialized with the declaration. For example, the following C program fails in the compilation.

```
struct Point
{
int x = 0; // COMPILER ERROR: cannot initialize members here
int y = 0; // COMPILER ERROR: cannot initialize members here
};
```

The reason for the above error is simple. When a datatype is declared, no memory is allocated for it. Memory is allocated only when variables are created.

DEFAULT INITIALIZATION

By default, structure members are not automatically initialized to 0 or NULL. Uninitialized structure members will contain garbage values. However, when a structure variable is declared with an initializer, all members not explicitly initialized are zero-initialized.

```
struct Point
{
int x;
int y;
};

struct Point p = {0}; // Both x and y are initialized to 0
```

We can initialize structure members in 3 ways which are as follows:

1. Using Assignment Operator.
2. Using Initializer List.
3. Using Designated Initializer List.

1. Initialization using Assignment Operator

```
struct structure_name str;
str.member1 = value1;
str.member2 = value2;
str.member3 = value3;
.
.
.
```

2. Initialization using Initializer List

```
struct structure_name str = { value1, value2, value3 };
```

In this type of initialization, the values are assigned in sequential order as they are declared in the structure template.

3. Initialization using Designated Initializer List

Designated Initialization allows structure members to be initialized in any order

```
struct structure_name str = { .member1 = value1,
.member2 = value2, .member3 = value3 };
```

```
// Create a structure
struct myStructure {
  int myNum;
  char myLetter;
  char myString[30];
};

int main() {
  // Create a structure variable and assign values to it
  struct myStructure s1 = {13, 'B', "Some text"};

  // Print values
  printf("%d %c %s", s1.myNum, s1.myLetter, s1.myString);

  return 0;
```

You can also assign one structure to another.

In the following example, the values of s1 are copied to s2:

```
struct myStructure s1 = {13, 'B', "Some text"};
struct myStructure s2;

s2 = s1;
```

```
struct myStructure {
  int myNum;
  char myLetter;
  char myString[30];
};

int main() {
  // Create a structure variable and assign values to it
  struct myStructure s1 = {13, 'B', "Some text"};

  // Modify values
  s1.myNum = 30;
  s1.myLetter = 'C';
  strcpy(s1.myString, "Something else");

  // Print values
  printf("%d %c %s", s1.myNum, s1.myLetter,
s1.myString);

  return 0;
}
```

If you want to change/modify a value, you can use the dot syntax (`.`).
And to modify a string value,
the `strcpy()` function is useful again:

PASS A STRUCTURE AS AN ARGUMENT TO THE FUNCTIONS

When passing structures to or from functions in C, it is important to keep in mind that the entire structure will be copied. This can be expensive in terms of both time and memory, especially for large structures. The passing of structure to the function can be done in two ways:

1. By passing all the elements to the function individually.
2. By passing the entire structure to the function.

EXAMPLE 1: USING CALL BY VALUE METHOD

```c
#include <stdio.h>

 struct car {
   char name[30];
   int price;
};

 void print_car_info(struct car c)
{
    printf("Name : %s", c.name);
    printf("\nPrice : %d\n", c.price);
}

 int main()
{
    struct car c = { "Tata", 1021 };
    print_car_info(c);
    return 0;
}
```

```
Name  : Tata
Price : 1021
```

Time Complexity: O(1)
Auxiliary Space: O(1)

EXAMPLE 2: USING CALL BY REFERENCE METHOD

```
#include <stdio.h>

  struct student {

    char name[50];
    int roll;
    float marks;
};

  void display(struct student* student_obj)
{
    printf("Name: %s\n", student_obj->name);
    printf("Roll: %d\n", student_obj->roll);
    printf("Marks: %f\n", student_obj->marks);
}
int main()
{  struct student st1 = { "Aman", 19, 8.5 };

     display(&st1);        return 0;
}
```

```
Name: Aman
Roll: 19
Marks: 8.500000
```

Time Complexity: O(1)
Auxiliary Space: O(1)

RETURN A STRUCTURE FROM FUNCTIONS

We can return a structure from a function using the **return** Keyword. To return a structure from a function the return type should be a structure only.

```c
struct student {
    char name[20];
    int age;
    float marks;
};

// function to return a structure
struct student get_student_data()
{
    struct student s;


    printf("Enter name: ");
    scanf("%s", s.name);
    printf("Enter age: ");
    scanf("%d", &s.age);
    printf("Enter marks: ");
    scanf("%f", &s.marks);


    return s;
}
```

```c
int main()
{
    // structure variable s1 which has been assigned the
    // returned value of get_student_data
    struct student s1 = get_student_data();
    // displaying the information
    printf("Name: %s\n", s1.name);
    printf("Age: %d\n", s1.age);
    printf("Marks: %.1f\n", s1.marks);


    return 0;
}
```

STRUCTURE AND POINTERS

A structure pointer is a pointer variable that stores the address of a structure. It allows the programmer to manipulate the structure and its members directly by referencing their memory location rather than passing the structure itself.

```
#include <stdio.h>
struct A {
    int var;
};
int main() {
    struct A a = {30};
        // Creating a pointer to the structure
    struct A *ptr;
    // Assigning the address of person1 to the pointer
    ptr = &a;
    // Accessing structure members using the pointer
    printf("%d", ptr->var);

    return 0;
}
```

Explanation: In this example, ptr is a pointer to the structure A. It stores the address of the structure a, and the structure’s member var is accessed using the pointer with the -> operator. This allows efficient access to the structure’s members without directly using the structure variable.

SYNTAX OF STRUCTURE POINTER

The syntax of structure pointer is similar to any other pointer to variable:

struct struct_name ****ptr_name;***

Here, **struct_name** is the name of the structure, and **ptr_name** is the name of the pointer variable.

ACCESSING MEMBER USING STRUCTURE POINTERS

There are two ways to access the members of the structure with the help of a structure pointer:

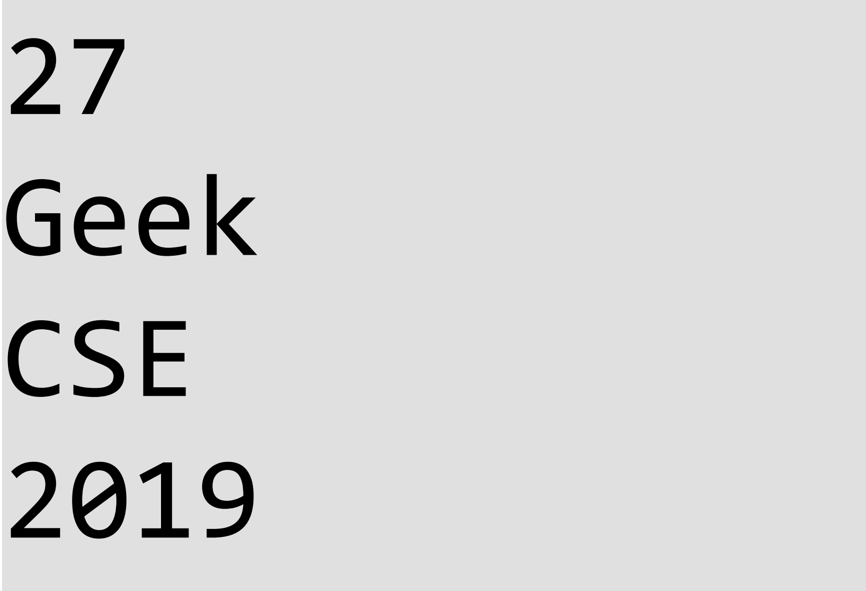
1. Differencing and Using (.) Dot Operator.
2. Using (->) Arrow operator.

Differencing and Using (.) Dot Operator

First method is to first dereference the structure pointer to get to the structure and then use the dot operator to access the member.

BELOW IS THE PROGRAM TO ACCESS THE STRUCTURE MEMBERS USING THE STRUCTURE POINTER WITH THE **HELP OF THE DOT OPERATOR.**

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

struct Student {
    int roll_no;
    char name[30];
    char branch[40];
    int batch;
};

int main() {

    struct Student s1 = {27, "Geek", "CSE", 2019};

      // Pointer to s1
    struct Student* ptr = &s1;

 // Accessing using dot operator
    printf("%d\n", (*ptr).roll_no);
    printf("%s\n", (*ptr).name);
    printf("%s\n", (*ptr).branch);
    printf("%d", (*ptr).batch);

    return 0;
}
```



```
27
Geek
CSE
2019
```

USING (->) ARROW OPERATOR

C language provides an array operator (->) that can be used to directly access the structure member without using two separate operators.

```
#include <stdio.h>
#include <string.h>
struct Student {
    int roll_no;
    char name[30];
    char branch[40];
    int batch;
};
int main() {

    struct Student s1 = {27, "Geek", "CSE", 2019};

      // Pointer to s1
    struct Student* ptr = &s1;

      // Accessing using dot operator
    printf("%d\n", ptr->roll_no);
    printf("%s\n", ptr->name);
    printf("%s\n", ptr->branch);
    printf("%d", ptr->batch);

    return 0;
}
```

```
27
Geek
 CSE
 2019
```

THANK YOU