Lecture 2

Arrays

(Part 1)

Suppose there are two students in a class. You need to store their marks.

```
int mark1;
int mark2;
```

Introduction

- Suppose we have 20 students in a class and we have been asked to write a program that reads and prints the marks of all the 20 students.
- In this program, we will need 20 integer variables with different names, as shown in Figure 1.

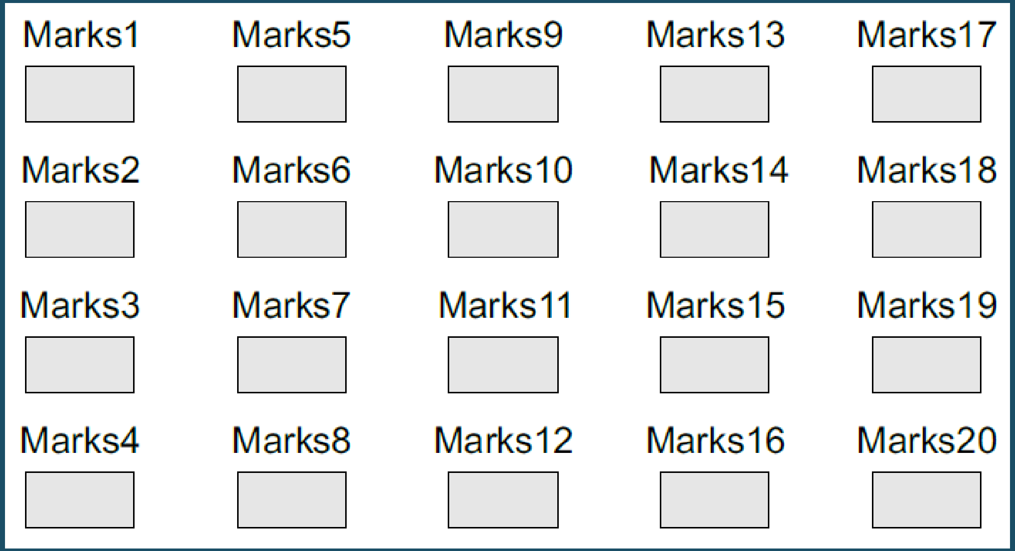


Fig. 1: *Twenty variables for 20 students*

Introduction

- But would it be possible to follow this approach if we have to read and print the marks of students,
  - in the entire course (say 100 students)
  - in the entire college (say 500 students)
  - in the entire university (say 10,000 students)

- The answer is no, definitely not! To process a large amount of data, we need a data structure known as **array**.

Introduction

- An array is a collection of similar data elements.
- These data elements have the same data type.
- The elements of the array are stored in consecutive memory locations and are referenced by an index (also known as the subscript).
- The subscript is an ordinal number which is used to identify an element of the array.

Declaration of Arrays

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

- Arrays are declared using the following syntax:

```
type name[size];
```

Declaration of Arrays

- Example: `int marks[10];`

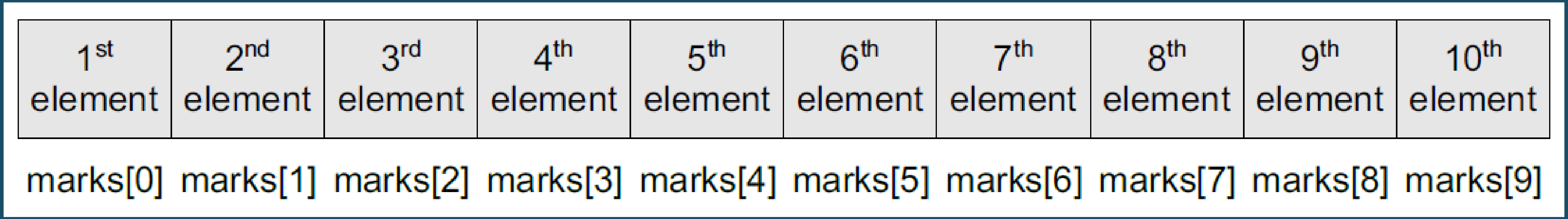


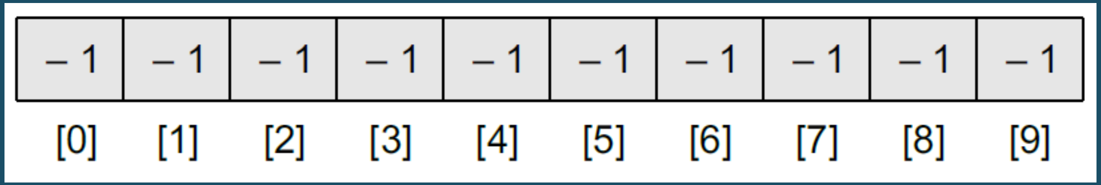
Fig. 2: *Memory representation of an array of 10 elements (in C Language)*

Accessing the Elements of an Array

- To access all the elements, we must use a loop.
- That is, we can access all the elements of an array by varying the value of the subscript into the array.

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

Code to initialize each element of the array to −1



Array **`marks`** *after executing the code given*

Accessing the Elements of an Array

Calculating the Address of Array Elements:

- The array name is a symbolic reference to the address of the first byte of the array.

- When we use the array name, we are actually referring to the first byte of the array.

- The subscript or the index represents the offset from the beginning of the array to the element being referenced.

- That is, with just the array name and the index, C can calculate the address of any element in the array.

Accessing the Elements of an Array

Calculating the Address of Array Elements:

- Since an array stores all its data elements in consecutive memory locations, storing just the base address, that is the address of the first element in the array, is sufficient.
- The address of other data elements can simply be calculated using the base address. The formula to perform this calculation is,

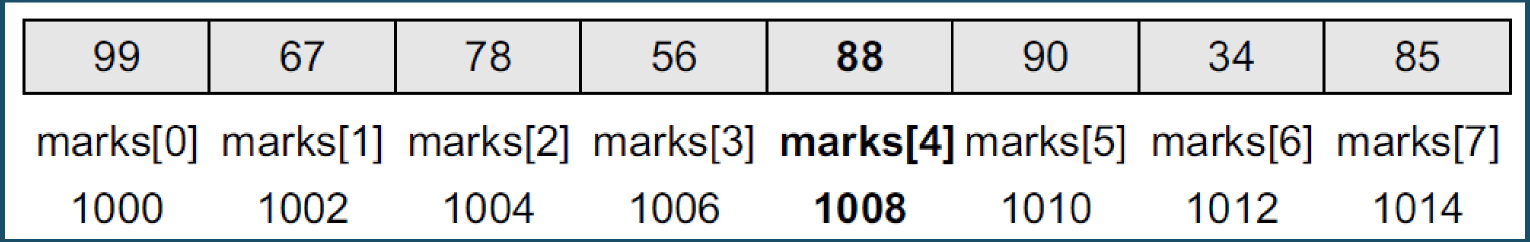
  Address of data element **A[k] = BA(A) + w(k – lower_bound)**
- Here, A is the array, k is the index of the element of which we have to calculate the address, BA is the base address of the array A, and w is the size of one element in memory, for example, size of int is 2 bytes.

Accessing the Elements of an Array

Calculating the Address of Array Elements:

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

Solution:



99	67	78	56	**88**	90	34	85
marks[0]	marks[1]	marks[2]	marks[3]	**marks[4]**	marks[5]	marks[6]	marks[7]
1000	1002	1004	1006	**1008**	1010	1012	1014

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

```
marks[4] = 1000 + 2(4 – 0)
         = 1000 + 2(4)
         = 1008
```

Accessing the Elements of an Array

Calculating the Length of an Array:

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

```
Length = upper_bound – lower_bound + 1
```

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

Accessing the Elements of an Array

Calculating the Length of an Array:

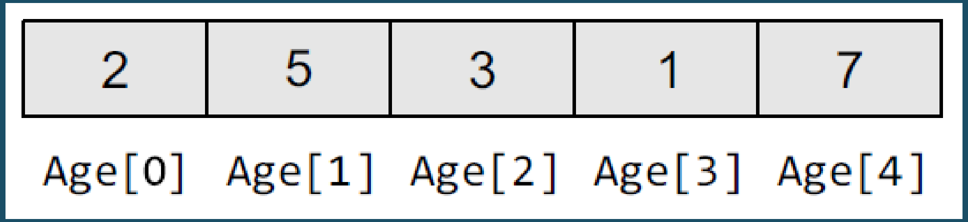
Example: Let Age[5] be an array of integers such that

```
    Age[0] = 2, Age[1] = 5, Age[2] = 3, Age[3] = 1, Age[4] = 7
```

Show the memory representation of the array and calculate its length.

Solution:

The memory representation of the array Age[5] is given as below.



`Length = upper_bound – lower_bound + 1`

Here, `lower_bound = 0, upper_bound = 4`

Therefore, `length = 4 – 0 + 1 = 5`

Storing Values in Arrays

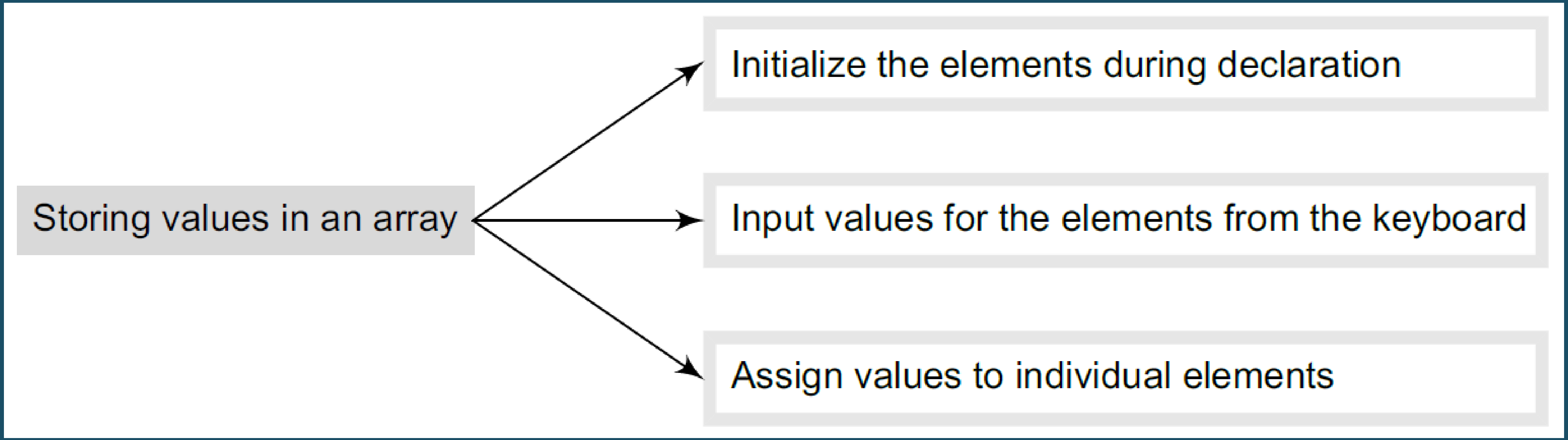

Storing values in an array
Initialize the elements during declaration
Input values for the elements from the keyboard
Assign values to individual elements

***Fig. 3:** three ways to store values in an array*

Storing Values in Arrays

Initializing Arrays during Declaration:

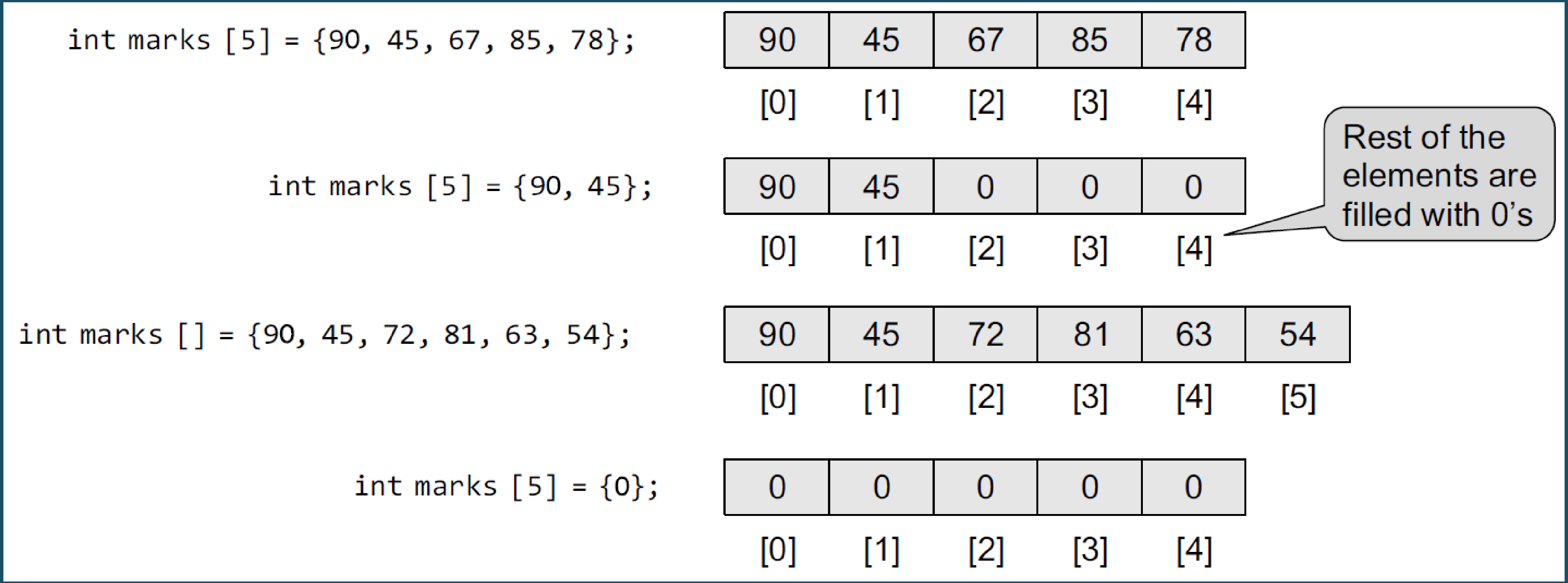


Fig. 4: *Initialization of array elements*

Storing Values in Arrays

Inputting Values from the Keyboard:

```
int i, marks[10];
for(i=0; i<10; i++)
    scanf("%d", &marks[i]);
```

Storing Values in Arrays

Note that we cannot assign one array to another array, even if the two arrays have the same type and size. To copy an array, you must copy the value of every element of the first array into the elements of the second array.

```
int i, arr1[10], arr2[10];
arr1[10] = {0,1,2,3,4,5,6,7,8,9};
for(i=0;i<10;i++)
     arr2[i] = arr1[i];
```

Example: *Code to copy an array at the individual element level*

Operations On Arrays

- ✓ Traversing an array
- ✓ Inserting an element in an array
- ✓ Searching an element in an array
- ✓ Deleting an element from an array
- ✓ Merging two arrays
- ✓ Sorting an array in ascending or descending order

Traversing an Array

- Traversing an array means accessing each and every element of the array for a specific purpose.
- Traversing the data elements of an array A can include
  - ✓ printing every element,
  - ✓ counting the total number of elements,
  - ✓ or performing any process on these elements.

```
Step 1: [INITIALIZATION] SET I = lower_bound
Step 2: Repeat Steps 3 to 4 while I <= upper_bound
Step 3: Apply Process to A[I]
Step 4: SET I = I + 1
[END OF LOOP]
Step 5: EXIT
```

Algorithm for array traversal

Inserting an Element in an Array

- If an element has to be inserted at the end of an existing array, then the task of insertion is quite simple.
- We just have to add 1 to the upper_bound and assign the value. (Here, we assume that the memory space allocated for the array is still available.)
- For example, if an array is declared to contain 10 elements, but currently it has only 8 elements, then obviously there is space to accommodate two more elements. But if it already has 10 elements, then we will not be able to add another element to it.

```
Step 1: Set upper_bound = upper_bound + 1
Step 2: Set A[upper_bound] = NEW_VAL
Step 3: EXIT
```

Algorithm to append a new element to an existing array

Inserting an Element in an Array

Example: Data[] is an array that is declared as int Data[20]; and contains the following values:

Data[] = {12, 23, 34, 45, 56, 67, 78, 89, 90, 100};

(a) Calculate the length of the array.

Solution: Length of the array = number of elements

Therefore, length of the array = 10

Inserting an Element in an Array

Example: Data[] is an array that is declared as int Data[20]; and contains the following values:

Data[] = {12, 23, 34, 45, 56, 67, 78, 89, 90, 100};

(b) Find the upper_bound and lower_bound.

Solution: By default, lower_bound = 0 and upper_bound = 9

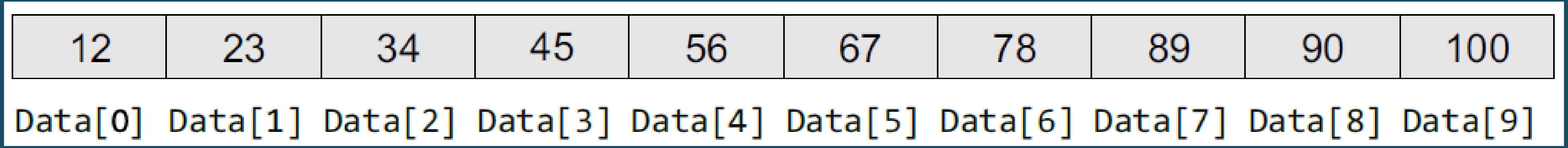
Inserting an Element in an Array

Example: Data[] is an array that is declared as int Data[20]; and contains the following values:

Data[] = {12, 23, 34, 45, 56, 67, 78, 89, 90, 100};

(c) Show the memory representation of the array.

Solution:



12	23	34	45	56	67	78	89	90	100
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]	Data[7]	Data[8]	Data[9]

Inserting an Element in an Array

Example: Data[] is an array that is declared as int Data[20]; and contains the following values:

Data[] = {12, 23, 34, 45, 56, 67, 78, 89, 90, 100};

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

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

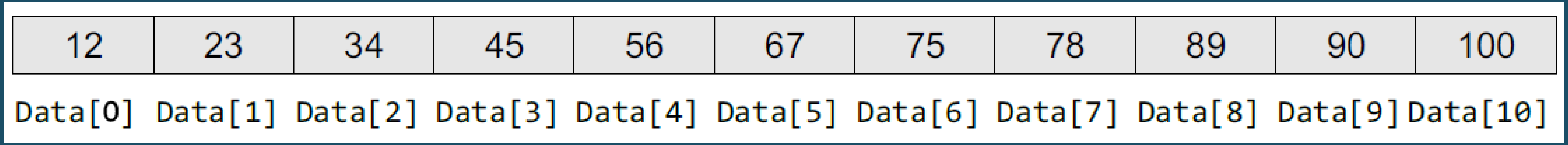
Inserting an Element in an Array

Example: Data[] is an array that is declared as int Data[20]; and contains the following values:

Data[] = {12, 23, 34, 45, 56, 67, 78, 89, 90, 100};

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

Solution:



12	23	34	45	56	67	75	78	89	90	100
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]	Data[7]	Data[8]	Data[9]	Data[10]

Inserting an Element in an Array

Algorithm to Insert an Element in the Middle of an Array

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

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

b) N, the number of elements in the array

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

d) VAL, the value that has to be inserted

Inserting an Element in an Array

Algorithm to Insert an Element in the Middle of an Array

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

Algorithm to insert an element in the middle of an array.

Inserting an Element in an Array

Algorithm to Insert an Element in the Middle of an Array

Initial Data[] is given as below.

45	23	34	12	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]

Calling INSERT (Data, 6, 3, 100) will lead to the following processing in the array:

45	23	34	12	56	20	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]

45	23	34	12	56	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]

45	23	34	12	12	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]

45	23	34	100	12	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]