Computer Science &
Information Technology

Programming and Data Structures

Comprehensive Theory

with Solved Examples and Practice Questions

MADE EASY Publications

MADE EASY Publications Pvt. Ltd.

Corporate Office: 44-A/4, Kalu Sarai (Near Hauz Khas Metro Station), New Delhi-110016
E-mail: infomep@madeeasy.in
Contact: 011-45124660, 8860378007
Visit us at: www.madeeasypublications.org

Programming and Data Structures

First Edition: 2015
Second Edition : 2016
Third Edition : 2017
Fourth Edition : 2018
Fifth Edition : 2019
Sixth Edition : 2020
Seventh Edition : 2021
Eighth Edition : 2022

Contents

Programming & Data Structures

■■■■

Programming and Data Structures

Goal of the Subject

Computer Science is not the study of programming. Programming, however, is an important part of what a computer scientist does. Programming is often the way that we create a representation for our solutions. Therefore, this language representation and the process of creating it becomes a fundamental part of the discipline.

A data structure is a specialized format for organizing and storing data. To manage the complexity of problems and the problem-solving process, computer scientists use abstractions to allow them to focus on the "big picture" without getting lost in the details. By creating models of the problem domain, we are able to utilize a better and more efficient problem-solving process. The implementation of an abstract data type, often referred to as a data structure, will require that we provide a physical view of the data using some collection of programming constructs and primitive data types.

General data structure types include the array, the file, the record, the table, the tree, and so on. Any data structure is designed to organize data to suit a specific purpose so that it can be accessed and worked with in appropriate ways. In computer programming, a data structure may be selected or designed to store data for the purpose of working on it with various algorithms.

Introduction

In this book we tried to keep the syllabus of Software Programming and Data structures around the GATE syllabus. Each topic required for GATE is crisply covered with illustrative examples and each chapter is provided with Student Assignment at the end of each chapter so that the students get the thorough revision of the topics that he/she had studied. This subject is carefully divided into seven chapters as described below.

1. **Programming Methodology:** In this chapter we will study about the different segments and their organization, variables and their scope, flow of control in a program, function evaluation types, storage classes, and pointers and finally we discuss the application of pointers.
2. **Arrays:** In this chapter we will study properties and application of arrays, accessing methods for two and three dimensional arrays and finally we discuss the arrays in the form of special matrices.
3. **Stack:** In this chapter we will study the ADT of stack, operations on stack, applications and different types of notations evaluated by stack and finally we discuss the tower of Hanoi (application).
4. **Queue:** In this chapter we will study about the Queue, operations on queue, applications and finally we discuss different types of queues.
5. **Linked Lists:** In this chapter we will study types and applications of linked list, operations on linked list, priority queue and finally we discuss implementation of stack, queue and priority queue using lists.
6. **Trees:** In this chapter we introduce trees, their applications, types of trees (BST, B-tree, and AVL), different types tree traversals and finally we discuss operations on trees.
7. **Hashing Techniques:** In this chapter we introduce the Hash function, collision resolution techniques and comparisons of different collision techniques.

■■■■

CHAPTER 02

Arrays

2.1 Definition of Array

- A collection of items having same data type stored in contiguous memory allocation.
- An array is derived data type in c programming language which can store similar type of data in continuous memory location. Data may be primitive type (int, char, float, double…), address of union, structure, pointer, function or another array.
- Importance: Array implementation is important because:
  (a) Most assembly languages have no concept of arrays.
  (b) From an array, any other data structure we might want, can be built.

2.2 Declaration of Array

There are various array in which we can declare an array.

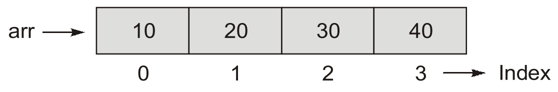
(i) Array declaration by specifying size:

Example: 1. int arr[10];
2. int n = 10;
int arr[n];

(ii) Array declaration by initializing elements:

Example: int arr[ ] = {10, 20, 30, 40}

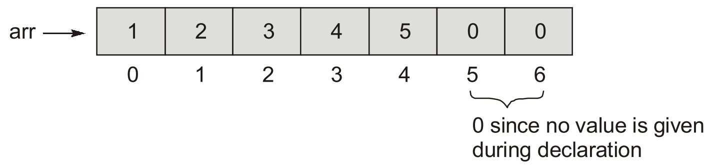
Compiler creates an array of size 4



(iii) Array declaration by specifying size and initializing:

Example: int arr[7] = {1, 2, 3, 4, 5}

Compiler creates an array of size 7



2.3 Properties of Array

- Each element is of the same size (char = 1 byte, integer = 2 byte or 4 word).
- Elements are stored continuously, with the first element stored at the smallest memory address.

NOTE: Thus the whole trick in assembly language is **(a)** To allocate correct amount of space for an array and **(b)** An address tells the location of an element.

Example: `int arr[5];` , `char arr[5];` , `float arr[5];` , `long double arr[5];` , `char * arr[5];` , `int (arr[])();` and `double ** arr[5];`

2.3.1 Array is Useful When

We have to store large number of data of similar type. If we have large number of similar kind of variable then it is very difficult to remember name of all variables and write the program. Below program without array and its equivalent program with array is shown.

Program without Array	Program with Array

`#include<stdio.h>`
`int main(){`
    `int ax=1, b=2, cg=5, dff=7, am=8, raja=0, rani=11, xxx=5, yyy=90, avg;`
    `avg=(ax+b+cg+dff+am+raja+rani +xxx+yyy)/12;`
    `printf("%d",avg);`
    `return 0;`
`}`
	`#include<stdio.h>`
	`int main(){`
	    `int arr[]={1,2,5,7,8,0,11,5,90};`
	    `int i,avg;`
	    `for(i=0;i<12;i++){`
	        `avg=avg+arr[i];`
	    `}`
	    `printf("%d",avg/12);`
	    `return 0;`
	`}`

2.3.2 Advantage of Using Array

1. An array provides single name. So it easy to remember the name of all element of an array.
2. Array name gives base address of an array. So with the help increment operator we can visit one by one all the element of an array.
3. Array issued to implement many data structures like linked list, stack, tree etc.
4. Array allows random access to elements along with sequential access.

   Example: int arr[ ] = {10, 20, 30}

   printf("%d", arr[1]);

   Output → 20 (print element present at index 1)

2.3.3 Disadvantage of Using Array

1. Array is of fixed size so the number of elements to be stored in array should be known in advance. Once the size is declared, it cannot be modified.
2. Insertion and deletion are quite difficult in array as elements are stored in consecutive memory locations and shifting operation is costly.

2.3.4 Array of Pointers in C

Array whose content is address of another variable is known as array of pointers.

***Example* - 2.1** What will be output of following program?

Code

```
#include<stdio.h>
int main ( )
{
   int arr[ ] = {5, 10,20, 40, 30}
   int i, *arrofpointers[5];
   for (i = 0, i < 5, i++)
   {
      arrofpointer[i] = & arr[i];
   }
   for (i = 0, i < 5; i++)
   {
   printf("%d",* arrofpointers[i]);
   }
   return 0;
}
```

Solution

Output: 5, 10, 20, 40, 80

Here, arr is array of integer, arrofpointers is array of pointers.

First for loop is used to store the address of integers present in arr[ ] into arrofpointers[ ].

Second for loop is used to print the value of integers through arrofpointers[ ] array. To access the value from address, '*' operator is used.

2.3.5 Complex Arrays in C

- Declaration of an array of size five which can store address of such functions whose parameter is void data type and return type is also void data type: `void(arr[5])();`
- Declaration of an array of size five which can store address of such function which has two parameter of int data type and return type is float data type: `float(arr[5])(int,int);`
- Declaration of an array of size two which can store the address of printf or scanf function: `int(arr[2])(const char*,...);`

NOTE: Prototype of printf function is: `int printf(const char*,...);`

2.3.6 Different Type of Array in C

- Array of integer: An array which can hold integer data type is known as array of integer. *Example:* `int arr[ ];`
- Array of character: An array which can hold character data type is known as array of character. *Example:* `char ch[3];`
- Array of union: An array which can hold address of union data type is known as array of union.

Example-2.2 What will be output of following program?

Code

```
#include<stdio.h>
union A{
   char p;
   float *const q;
};
int main(){
   union A arr[10];
   printf("%d",sizeof arr);
   return 0;
}
```

Solution

Output: 80

Union is a special data type that allows to store different data types in same memory location.

Size of union = size of largest member of union.

Here,

Union A {
  char p; 1 byte
  float *const q; 8 byte
}

As there is a array of union, so total size = 10×8 = 80.

- **Array of structure:** An array which can hold address of structure data type is known as array of structure.

Example-2.3 What will be output of following program?

Code

```
#include<stdio.h>
typedef struct madeeasy{
   char *name;
   int roll;
   }s;
int main(){
   s arr[2]={{"made",10},{"easy",11}};
   printf("%s %d",arr[0]);
   return 0;
}
```

Solution

Output: made 10

Structure is a user defined data type in c which allows us to combine data of different types together under a single name. Size of structure = size of member_1 + size of member_2 ..... + size of member *n*.

Here,

struct madeeasy {
  char *name; 8 byte (pointer)
  int roll; 4/12 byte
}s;

As the array is of structure as shown below:

$$\text{arr}\ \underbrace{\boxed{\text{made}}\boxed{10}}_{0}\underbrace{\boxed{\text{easy}}\boxed{11}}_{11}$$

So, output is 'made 10'.

- **Array of string:** An array which can hold string data type is known as array of integer.
  Example: `string name[4];`
- **Array of array:** An array which can hold address of another array is known as array of array.
- **Array of address of integer:** An array which can hold address integer data type is known as array of address of integer.

2.3.7 Pointer to Array

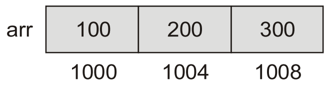
A pointer which holds base address of an array or address of any element of an array is known as pointer to array.

Example-2.4 What will be output of following program?

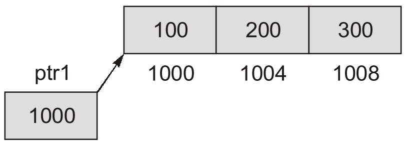
```
#include<stdio.h>
int main(){
   int arr[5]={100,200,300};
   int *ptr1=arr;
   printf("%d",*(ptr1+2),);
return 0;
}
```

Solution:

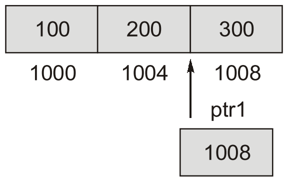
Let array arr be stored from memory location 1000.



ptr1 is a pointer that points to arr, i.e., store address of arr.



(ptr1 + 2) strips 2 elements in the given array.



*(ptr1 + 2) gives the value stored at 1008, i.e., 300

2.4 Accessing Elements of an Array

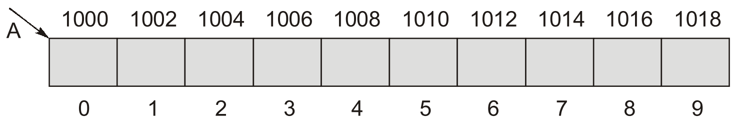
In general we need to know: (a) Where the array starts (called the Base address) (b) Size of an element in bytes (to get a byte address) and (c) What the first element is numbered (first index).

2.4.1 One Dimensional Array

Byte address of element [X] = base address + size (X-first index)

Example-2.5 A[0, ... 9], base address = 1000, size of element = 2 byte. Find location of A[5].

Solution:



$$\text{Location}(A[5]) = \underset{\substack{\text{Base}\\\text{address}}}{1000} + \underbrace{\underset{\substack{\text{Number of element}\\\text{to be shipped}\\\text{to reach index 5}}}{(5-0)} \times \underset{\substack{\text{Size of}\\\text{each element}}}{2}}_{\substack{\text{Total memory to be crossed}\\\text{or skipped to reach given index}}}$$

$$= 1000 + 10 = 1010$$

Example - 2.6 A[–5 ... + 5], Base address (BA) = 999, Size of element = 100 bytes. Find the location of A[–1], A[+5]?

Solution:

$$\begin{aligned} L(A[-1]) &= 999 + [(-1) - (-5)] \times 100 \\ &= 999 + 100 \times 4 = 1399 \\ L(A[5]) &= 999 + [5 - (-5)] \times 100 \\ &= 999 + 10 \times 100 = 1999 \end{aligned}$$

NOTE: Total number of element = Last Index – First Index + 1

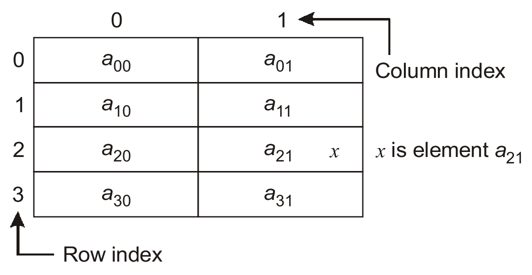
2.4.2 Two-Dimensional Arrays

How to map a 2-D array into a 1-D array memory:

Terminology $r \times c$ array, ***where*** r = rows and c = columns

element $[y, x]$, ***where*** y is row number and x is column number.

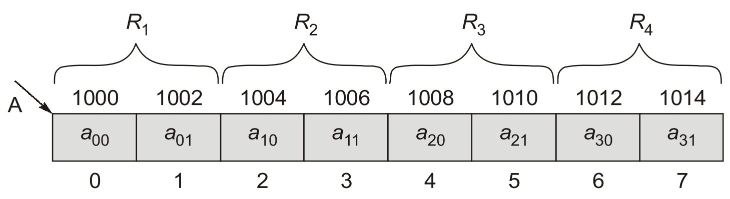
Example: A[0 ... 3, 0 ... 1] i.e. A[4] [2]



Mapping this 4 × 2 array into memory.

There are two possibilities:

- **Row Major Ordering:** Rows are all together.



Suppose array A[lb_1 ..... ub_1] [lg_2 ... ub_2] has base address 'BA', 'S' is size of each element array is stored in row major order, then location of some element A[i][j] is:

location (A[i][j]) = BA + [(i – lb_1) × number of columns + (j – lb_2)]) * S

2.4.6 Strictly Lower Triangular Matrix

A lower triangular matrix having as along with the diagonal as well as the upper portion i.e., a matrix A = [$a_{i,j}$] such that $a_{i,j} = 0$ for $i \geq j$.

$$\text{Written explicitly } L = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ a_{21} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{bmatrix}$$

2.4.7 Strictly Upper Triangular Matrix

A upper triangular matrix as along the diagonal as well as the upper portion i.e., a matrix $A = [a_{i,j}]$ such that $a_{i,j} = 0$ for $i \leq j$

$$\text{Written explicit } U = \begin{bmatrix} 0 & a_{12} & \cdots & a_n \\ 0 & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}$$

Summary

- Array is a collection of homogeneous elements stored in contiguous memory location.
- **Properties:** Static in nature, Compile time early binding and user friendly.
- **1-D Array:** Let '*A*' be an array of *n* elements. Address of an element *A*[*i*]:
  Base(A) + (*i* – start index) × size of element.
- **2-D Array:** Let *A*[*m*][*n*] be a 2-D array with *m* rows and *n* columns.
  Address of an element *A*[*i*][*j*] in Row Major order:
  Base (*A*) + (*j* – start index) × size of element + (*i* – start index) × size of element × *n*
  Address of an element A[*i*][*j*] in Column Major order:
  Base (*A*) + (*i* – start index) × size of element + (*j* – start index) × *m* × size of element.
- **Insertion:**
  [best case] At end takes constant time i.e. $\Omega(1)$
  [worst case] At beginning takes $O(n)$ time
  [Average case] In middle takes $\theta(n)$ time
- **Deletion:**
  [best case] At end takes constant time i.e. $\Omega(1)$
  [worst case] At beginning takes $O(n)$ time
  [Average case] In middle takes $\theta(n)$ time
- **Lower Triangular** [◺]$_{n\times n}$:

  (*i*) Size $= n + \dfrac{n^2 - n}{2} = \dfrac{n(n+1)}{2}$

  (*ii*) Row Major Order (RMO): $(j-1) + \dfrac{i(i-1)}{2}$

  (*iii*) Column Major Order (CMO): $a[i][j] = (i-j) + \left[(j-1)n - \dfrac{(j-1)(j-2)}{2}\right]$

- Upper Triangular [◹]$_{n\times n}$:
  - (*i*) Size = $\dfrac{n(n+1)}{2}$
  - (*ii*) Row Major Order (RMO): $a[i][j] = (j-i) + \left[(i-1)n - \dfrac{(i-1)(i-2)}{2}\right]$
  - (*iii*) Column Major Order (CMO): $a[i][j] = \left[(i-1) + \dfrac{j(j-1)}{2}\right]$
- Strictly lower triangular [◺]$_{n\times n}$
  - (*i*) Size = $\dfrac{n^2-n}{2}$
  - (*ii*) Row Major Order (RMO): $a[i][j] = (j-1) + \dfrac{(i-1)(i-2)}{2}$
  - (*iii*) Column Major Order (CMO): $a[i][j] = (i-1) + \dfrac{(j-1)(j-2)}{2}$

Student's Assignment

Q.1 Advantage of array is
(a) Linear access (b) Random access
(c) Sequential access (d) All the above

Q.2 Consider the following single dimensional array int *a*[5] = {10, 20, 30, 40, 50}. Array is stored starting from location 1000 (size of int = 2 byte)? What is location is *a*[4]?
(a) 1000 (b) 1008
(c) 1005 (d) 1002

Q.3 Consider the following single dimensional array declaration

A[1... 1000]

Base address 1000 and size of element 4. Find the location of A[50]
(a) 1186 (b) 1126
(c) 1096 (d) 1196

Q.4 Consider the 2 dimensional array. A [– 8...12, –4...16]. Calculate the address of A[1, 3]. Assume that it is stored in a row major order. Each element occupies 4-byte and starting address of array 2000.
(a) 2780 (b) 2784
(c) 2776 (d) 2782

Q.5 Main ( )
{

int *a*[3] [4] = $\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$

printf("\ *n*% *u*% *u*% *u*", *a*[0] + 1, * (*a*[0] + 1), *(*(*a* + 0) + 1)));

}

What is output of the above program? Assume array begin at address 10.
(a) 12, 2, 2 (b) 10, 2, 2
(c) 12, 2, 4 (d) 12, 4, 4

Q.6 Professor Pradhumn decides to make quick sort stable by changing each key *A*[*i*] in array *A*[1: *n*] to (*n**A[*i*]) + *i* – 1, so that all the new keys are distinct (call the modified array *A'*[1 : *n*] and then sorting *A'* (Assume *A* is subset of integers). Then which of the following is true?
(a) *A'* contains distinct elements and the original keys can be restored by computing $\left\lceil \dfrac{A'[i]}{n} \right\rceil$ for each *i*.
(b) *A'* contains distinct elements and the original keys can be restored by computing $\left\lfloor \dfrac{A'[i]}{n} \right\rfloor$ for each *i*.

(c) A′ does not always contain distinct keys.
(d) A′ contains distinct elements but it is not possible to restore original keys.

Q.7 In a compact single dimensional array representation for lower triangular matrices (i.e. all the elements above the diagonal are zero of size $n \times n$, non-zero elements (i.e., elements of the lower triangle) of each row are stored one after another, starting from the first row, the index of the $(i, j)^{th}$ element of the lower triangular matrix in the new representation is

(a) $i + j$
(b) $i + 1 - 1$
(c) $(j-1)+\frac{i(i-1)}{2}$
(d) $i+\frac{j(j-1)}{2}$

Q.8 Main( )
{

$$\text{int } a[3][4] = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

printf("\ n% u% u", a +1, & a + 1);
}

What is the output of the above program? Assume array begin at address 54572.

(a) 54572, 54580
(b) 54580, 54596
(c) 54572, 54596
(d) 54572, 65490

Q.9 Suppose we want to arrange the n numbers stored in an array such that all negative value occur before all positive ones, minimum number of exchanges required in the worst case is

(a) $n - 1$
(b) n
(c) $n + 1$
(d) $\frac{n}{2}$

Q.10 Let A[1 : n] be an array such that A[i] = i. An algorithm randomly permutes the elements of A, call the resulting array A′. Let X denote the number of locations such that A′[i] = i. What is expectation of X?

(a) n^2
(b) $\frac{n}{2}$
(c) n
(d) 1

Q.11 Consider the function given below, which should return the index of first zero in input array of length 'n' if present else return –1.

```
int index of zero (int[ ] array, int n) {
for (int i = 0; [P] ; i++);
    if (i = = n)
        return –1;
    return i;
}
```

Which of the should be place in code at $\boxed{\text{P}}$, so that code will work fine?

(a) array[i]! = 0 && $i \le n$
(b) array[i]! = 0 && $i < n$
(c) ! array[i] = 0 && $i < n$
(d) ! array[i] = = 0 || n

Q.12 Consider the following C code snippet:

```
main ( )
{
    int S[6] = {128, 256, 512, 1024, 2048, 4096};
    int *x = (int *) (& S + 1);
    printf("%d", x);
}
```

Let the size of int is 4 bytes; the array starts from 2000 onwards. Then the o/p generated by the above code is ________.

Q.13 Consider the integer arrayA[1 ..... 100, 1 ..... 100] in which the elements are stored in Z representation. An example of a 5 × 5 array in Z representation is shown below:

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ 2 & & & & a_{24} & \\ 3 & & & a_{33} & & \\ 4 & & a_{42} & & & \\ 5 & a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{array}$$

If the base address of A is starting from 1000 onwards, size of each element is 1 bytes and A is stored in Row Major Order, then the address corresponding to A[100] [55] is ________.

Q.14 In a lower triangular matrices (size 15 × 15) representation of compact single dimensional array, non-zero elements (i.e. elements of the lower triangle) of each row are stored one after another, starting from the first row. Assume each integer

take 1B. The array stored in row major order and first element of array is stored at location 1000, then the address of element $a[10]$ $[6]$ is __________ B.
[Note: Only lower triangular elements of the matrix are stored in contiguous array]

Q.15 Consider the following C program:

```
#include <stdio.h>
#include <conio.h>
int main ( )
{
    int arr [2][3][2] = {{{1, 2}, {3, 4}, {5, 6}}, {{7, 8},
    {9, 10}, {11, 12}}}
    printf("%d%d", a[1] – a[0], a[1][0] – a[0][0]);
    return 0;
}
```

The output produced by above C programm is __________.

Q.16 Consider 3 dimensional Array A[90] [30] [40] stored in linear array in column major order. If the base address starts at 10, The location of A [20] [20] [30] is __________. (Assume the first element is stored at A[1][1][1] and each element take 1 memory location)

Q.17 Consider the following C-fragment where size of int is 1 B (Assume starting address of array is 1000)

```
int main ( )
{
    int S[6] = {10, 20, 30, 40, 50, 60};
    int * Str = (int *) (&S + 1);
    printf("% d", Str);
}
```

The output generated by above code is __________.

Q.18 Consider a 2 dimensional array A[40 ..... 95, 40 ..... 95] in lower triangular matrix representation. The size of each element in the array is 1 byte. If the array is implemented in the memory in the form of row major order and base address of the array is 1000, the address of A[66] [50] will be ________.

Q.19 The minimum size that an array may require to store a binary tree with 'n' nodes is _______.

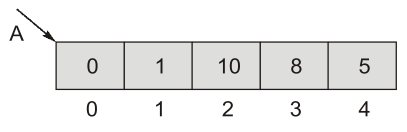
(a) $2^{\lceil \log_2(n+1) \rceil} - 1$
(b) $2^n - 1$
(c) $2^n - n + 1$
(d) $n + 1$

Answer Key:

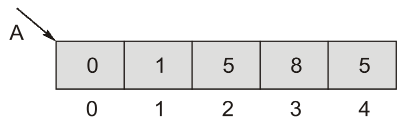
1. (b)	**2.** (b)	**3.** (d)	**4.** (b)	**5.** (a)
6. (b)	**7.** (c)	**8.** (b)	**9.** (d)	**10.** (d)
11. (b)	**12.** (2024)	**13.** (1252)	**14.** (1061)	**15.** (36)
16. (23699)	**17.** (1006)	**18.** (1361)	**19.** (a)	

Student's Assignments Explanations

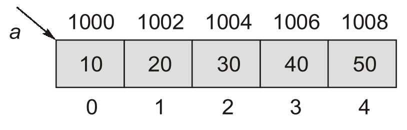
1. (b)
An array can be accessed in random way also.



Either we can access the array sequentially using a loop or can directly access a element using its index.
Example: A[2] = 5
Now array A looks like



2. (b)
Array is stored from location 1000 and size of each element is 2 byte.



$a[4] = 50$ is stored at location 1008.

3. (b)
Base address (BA) = 1000

$$\begin{aligned} \text{Size(s)} &= 4 \text{ byte} \\ \text{lb} &= 1, \text{ ub} = 1000 \\ \text{Location (A[50])} &= \text{BA} + (50 - \text{lb}) \times \text{S} \\ &= 1000 + (50 - 1) \times 4 \\ &= 1000 + 49 \times 4 = 1196 \end{aligned}$$

4. (b)

Base address = 2000

Size of each element(s) = 4 byte

$$lb_1 = -8,\ lb_2 = -4$$
$$lb_1 = 12,\ lb_2 = 16$$

Number of rows (nR) = $ub_1 - lb_1 + 1$
$$= 12 - (-8) + 1 = 21$$

Number of columns (nC) = $ub_2 - lb_2 + 1$
$$= (16) - (-4) + 1 = 21$$

Location (A[1][3]) = BA + [$(i - lb_1)$* nC + $(j - lb_2)$] * S

= 2000 + [(1 – (–8))* 21 + (3 – (–4))]* 4

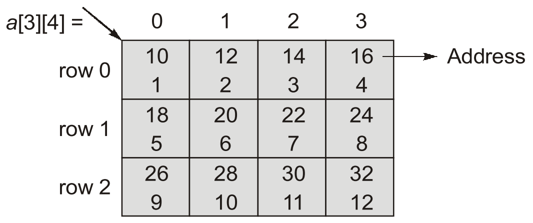
= 2000 + [9 *21 + 7] * 4

= 2784

5. (a)

$$\text{int a[3][4]} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

% *u* - it is used to print the address.



a[3][4] =

	0	1	2	3
row 0	10 1	12 2	14 3	16 4
row 1	18 5	20 6	22 7	24 8
row 2	26 9	28 10	30 11	32 12

(i) *a*[0] +1 ⇒ *a*[0] can be written as *(*a* + 0)

So *a*[0] + 1 = $\underbrace{*(a + 0)}_{\text{Select 0th row}} + \underbrace{1}_{\text{Skip 1 element in that row}}$

= Address of 0th row + 1 × Size of element

= 10 + 2 = 12

(ii) *(*a*[0] +1) ⇒ $\underbrace{*(\underbrace{*(a + 0)}_{\text{Select 0th row}} + \underbrace{1}_{\text{Ship 1 element in that row}})}_{\text{Select the element}}$

Firstly, 0th row selected. In that row, 1 element is stripped (i.e., 1) and second element is selected.

Output → 2

(iii) *(*(*a* + 0) +1) ⇒ Also outputs 2 (same as 2nd).

6. (b)

Let us solve this question with help of an example:

Let array *A* be

4	2	1	4
1	2	3	4

Now calculating array *A*′ using (*n**A[*i*]) + *i* – 1)

$$A'[1] = (4 \times 4 + 1 - 1) = 16$$
$$A'[2] = (4 \times 2 + 2 - 1) = 9$$
$$A'[3] = (4 \times 1 + 3 - 1) = 6$$
$$A'[4] = (4 \times 4 + 4 - 1) = 19$$

A′ contains distinct elements because the values in *A*′ depends on index *i*.

Restoring original value:

$$A[1] = \frac{A'[1]}{n} = \frac{16}{4} = 4$$

$$A[2] = \frac{A'[2]}{n} = \frac{9}{4} = 2.25$$

To get original value, we have to floor of above value.

$$\lfloor 2.25 \rfloor = 2$$

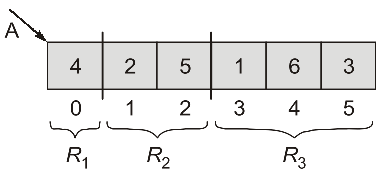
$$A[3] = \left\lfloor \frac{A'[3]}{n} \right\rfloor = \left\lfloor \frac{6}{4} \right\rfloor = \lfloor 1.5 \rfloor = 1$$

$$A[4] = \left\lfloor \frac{A'[4]}{n} \right\rfloor = \left\lfloor \frac{19}{4} \right\rfloor = \lfloor 4.75 \rfloor = 4$$

7. (c)

Let, $n = 3$

$$\begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 1 & 4 & 0 & 0 \\ 2 & 2 & 5 & 0 \\ 3 & 1 & 6 & 3 \end{array}$$



Location (A[*i*][*j*]) = BA + [$\underbrace{(i - 1)\text{ Natural number sum} + (j - 1)}_{\text{Number of elements to be crossed to reach } j\text{th column}}$] × S

$$= 0 + \left[\frac{(i-1)(i-1+1)}{2} + (j-1)\right] \times 1$$

$$= \frac{i(i-1)}{2} + (j-1)$$