Data Structures & Algorithms

01: Arrays

(Course Description; Random Initializations)

Dr Ram Prasad Krishnamoorthy

Associate Professor
School of Computing and Data Science

ram.krish@saiuniversity.edu.in

Course Details

Course Details

Outline

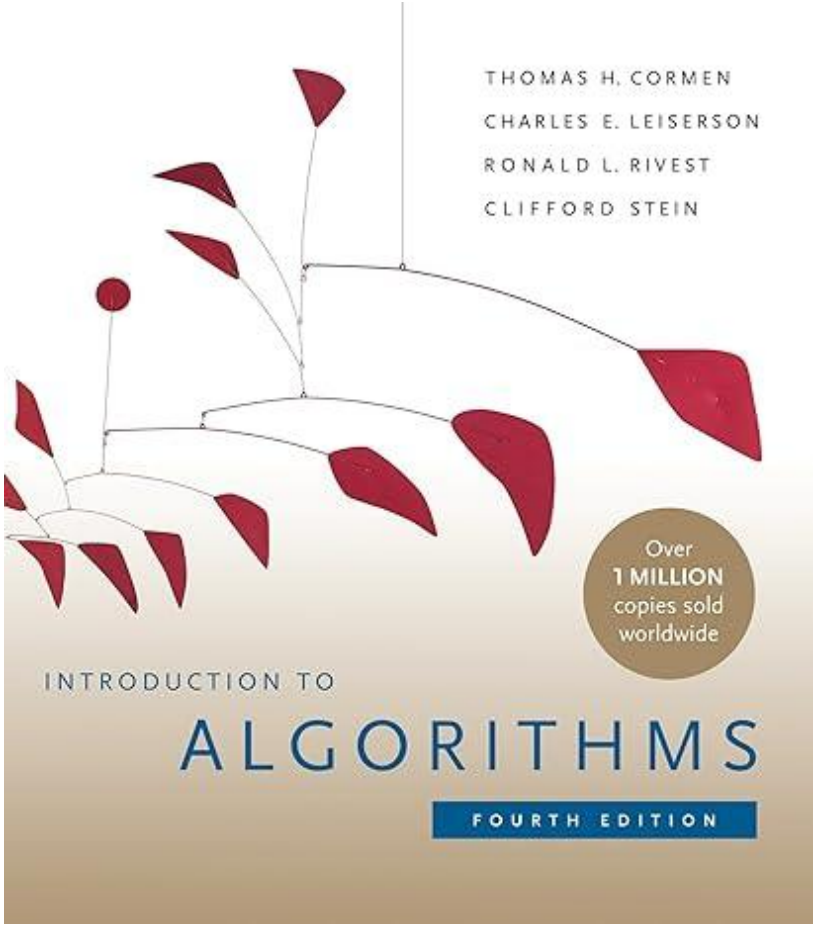
- **Data Structures - I**
  - Stacks, queues, linked lists
- **Data Structures - II**
  - Graphs, trees, binary search trees
  - Binary heaps, hash tables
- **Asymptotic analysis**
  - Time and space complexity
- **Searching algorithms**
  - Linear search, binary search
- **Sorting algorithms**
  - Bubble sort, selection sort
  - Insertion sort, merge sort, quick sort
- **Algorithm design techniques**
  - Greedy Algorithms
  - Divide and conquer
  - Dynamic programming
- **Graph algorithms**
  - Graph traversals
  - Minimum spanning trees
  - Shortest paths

Course Details

Reference Textbook

Introduction to Algorithms
Fourth Edition
Thomas H. Cormen, Charles E. Leiserson,
Ronald L. Rivest and Clifford Stein

Note: You can also consider Third Edition
(4e is currently costly)



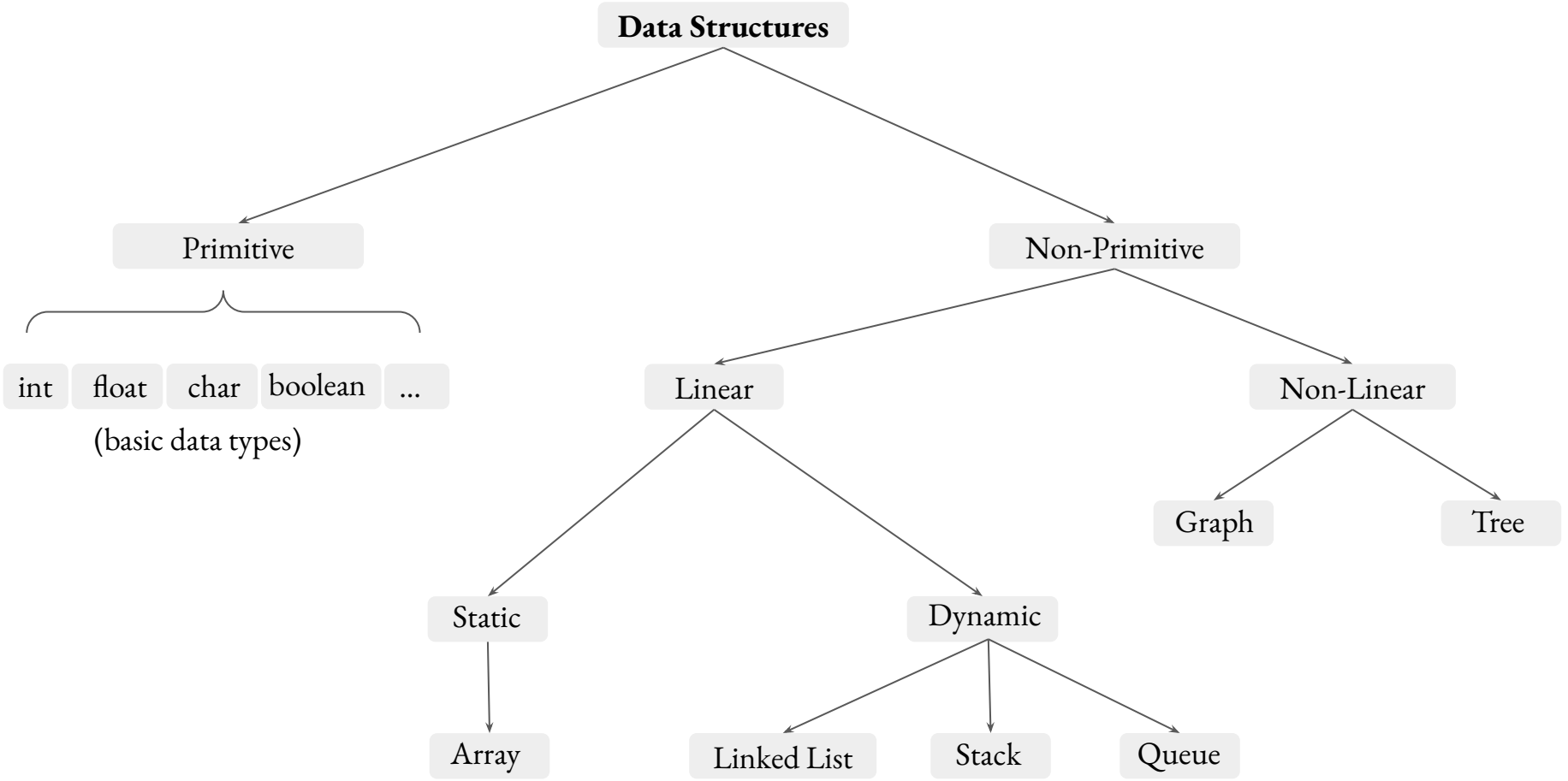
Course Details

Evaluation

1. Class Tests - *45% (3 best out of 4 tests; 15% each)*
2. Assignments - *25%*
3. End-sem evaluation - *30%*

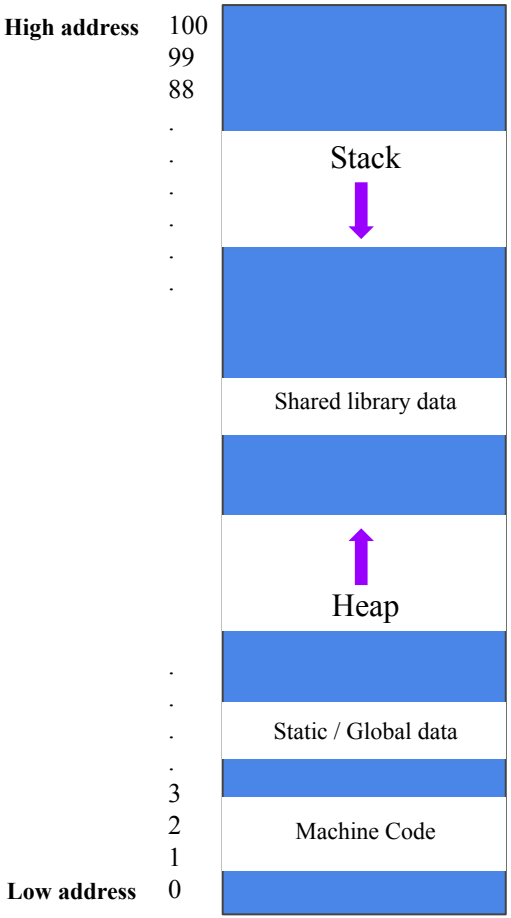
Data Structures

Deals with efficient storage and retrieval of data
in the computer's main memory.


Data Structures
Primitive
Non-Primitive
int
float
char
boolean
...
(basic data types)
Linear
Non-Linear
Graph
Tree
Static
Dynamic
Array
Linked List
Stack
Queue

C Memory Architecture

C Memory Architecture



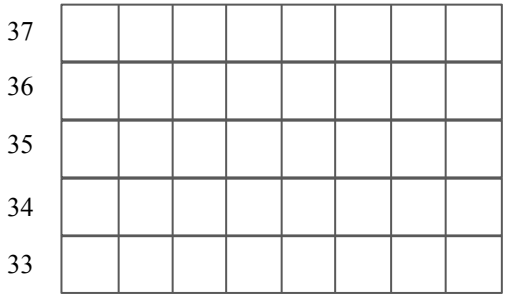
- Stack grows **downwards** when new function is called.
- Shrinks **upwards** when function finishes.

Note:

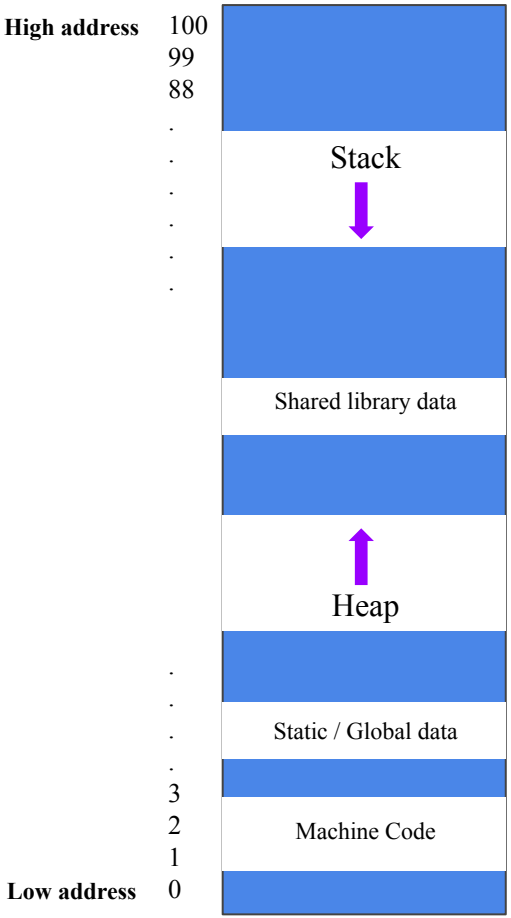
- Each memory address/location stores 8 bit of data.

```
char alpha = 'a'; // ASCII is 97

int  num1= 12;
```



C Memory Architecture



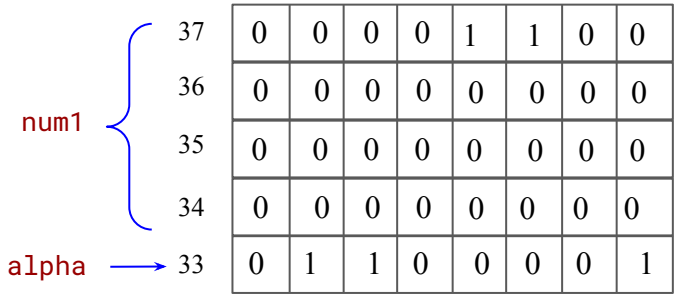
- Stack grows **downwards** when new function is called.
- Shrinks **upwards** when function finishes.

Note:

- Each memory address/location stores 8 bit of data.

```
char alpha = 'a'; // ASCII is 97

int  num1= 12;
```



Array Storage

Arrays

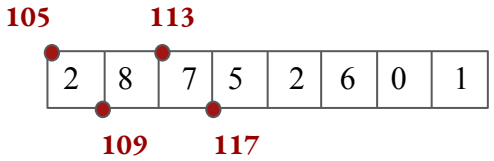
Arrays store elements contiguously in memory.

Assume the following:

$s \Rightarrow$ starting index

$a \Rightarrow$ starting address

$b \Rightarrow$ bytes occupied by each element



$s \Rightarrow 0$

$a \Rightarrow 105$

$b \Rightarrow 4$

Find out the bytes occupied by 3rd index.

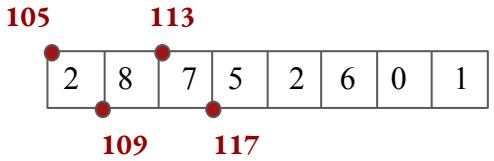
A general equation is also required.

Arrays

Arrays store elements contiguously in memory.

Assume the following:

$s \Rightarrow$ starting index
$a \Rightarrow$ starting address
$b \Rightarrow$ bytes occupied by each element



$s \Rightarrow 0$
$a \Rightarrow 105$
$b \Rightarrow 4$

Find out the bytes occupied by 3rd index.

A general equation is also required.

$105 + 4 \times (3 - 0) = 105 + 12 = 117$ (from)

$105 + 4 \times (3 + 1 - 0) - 1 = 105 + 16 - 1 = 120$ (to)

Arrays

Arrays store elements contiguously in memory.

Assume the following:

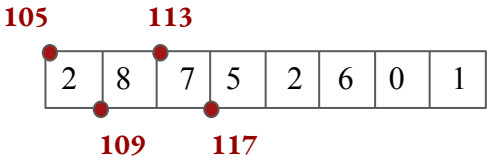
$s \Rightarrow$ starting index
$a \Rightarrow$ starting address
$b \Rightarrow$ bytes occupied by each element

i^{th} index occupies bytes:

$a + b(i - s)$ through $a + b(i + 1 - s) - 1$

The most common values of s are 0 and 1.

if s = 0 $\Rightarrow a + bi$ through $a + b(i + 1) - 1$
if s = 1 $\Rightarrow a + b(i - 1)$ through $a + bi - 1$



$s \Rightarrow 0$
$a \Rightarrow 105$
$b \Rightarrow 4$

Find out the bytes occupied by 3rd index.

A general equation is also required.

105 + 4 x (3 - 0) = 105 + 12 = 117 (from)

105 + 4 x (3 + 1 - 0) - 1 = 105 + 16 - 1 = 120 (to)

Arrays

Matrix

An *m* x *n* matrix has *m* rows and *n* columns.

Two common ways to store a matrix:

- **Row-major**: stored row by row
- **Column major**: stored column by column

Arrays

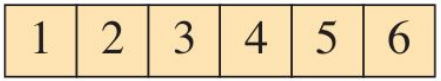
Matrix

An m x n matrix has m rows and n columns.

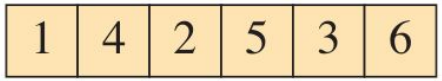
Two common ways to store a matrix:

- **Row-major**: stored row by row
- **Column major**: stored column by column

$$M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$



Row-major



Column-major

Arrays

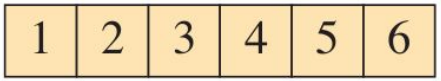
Matrix

An m x n matrix has m rows and n columns.

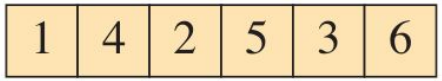
Two common ways to store a matrix:

- **Row-major**: stored row by row
- **Column major**: stored column by column

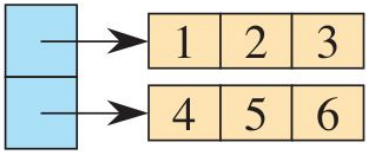
$$M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$



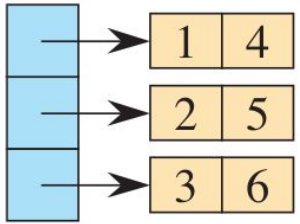
Row-major



Column-major



Row-major



Column-major

Pointer based

Pointer based implementation of Arrays

Random number Generation

`stdlib.h`

Arrays

```c
/* Random number basic code */

printf("%d\n", rand());
printf("%d\n", rand());
printf("RAND_MAX: %d\n", RAND_MAX);
// ( 2 ^ 32 / 2 ) - 1
```

```
1804289383
846930886
RAND_MAX: 2147483647
```

Arrays

```c
/* Random number basic code */

printf("%d\n", rand());
printf("%d\n", rand());
printf("RAND_MAX: %d\n", RAND_MAX);
// ( 2 ^ 32 / 2 ) - 1
```

```c
int rand_bound()
{
   /* Random number between a lower/upper bound */
   int l_bound = 10;
   int u_bound = 100;

   int width = (u_bound - l_bound) + 1;

   int val = 0;

   val = rand() % width + l_bound;

   return val;
}
```

```
1804289383
846930886
RAND_MAX: 2147483647
```

Arrays

```c
/* Random number basic code */

printf("%d\n", rand());
printf("%d\n", rand());
printf("RAND_MAX: %d\n", RAND_MAX);
// ( 2 ^ 32 / 2 ) - 1

printf("%d\n", rand_bound());
printf("%d\n", rand_bound());
```

```c
int rand_bound()
{
   /* Random number between a lower/upper bound */
   int l_bound = 10;
   int u_bound = 100;

   int width = (u_bound - l_bound) + 1;

   int val = 0;

   val = rand() % width + l_bound;

   return val;
}
```

```
1804289383
846930886
RAND_MAX: 2147483647
47
92
```

Arrays

```c
/* Random number basic code */

printf("%d\n", rand());
printf("%d\n", rand());
printf("RAND_MAX: %d\n", RAND_MAX);
// ( 2 ^ 32 / 2 ) - 1

printf("%d\n", rand_bound());
printf("%d\n", rand_bound());

/* After seeding */
printf("After seeding\n");
srand(time(0)); // vs srand(42);

printf("%d\n", rand());
printf("%d\n", rand());
```

```c
int rand_bound()
{
   /* Random number between a lower/upper bound */
   int l_bound = 10;
   int u_bound = 100;

   int width = (u_bound - l_bound) + 1;

   int val = 0;

   val = rand() % width + l_bound;

   return val;
}
```

```
1804289383                  After seeding
846930886                   82612881
RAND_MAX: 2147483647        162735805
47
92
```

Arrays

```c
/* Random number basic code */

printf("%d\n", rand());
printf("%d\n", rand());
printf("RAND_MAX: %d\n", RAND_MAX);
// ( 2 ^ 32 / 2 ) - 1

printf("%d\n", rand_bound());
printf("%d\n", rand_bound());

/* After seeding */
printf("After seeding\n");
srand(time(0)); // vs srand(42);

printf("%d\n", rand());
printf("%d\n", rand());

printf("%d\n", rand_bound());
printf("%d\n", rand_bound());
```

```c
int rand_bound()
{
   /* Random number between a lower/upper bound */
   int l_bound = 10;
   int u_bound = 100;

   int width = (u_bound - l_bound) + 1;

   int val = 0;

   val = rand() % width + l_bound;

   return val;
}
```

```
1804289383                  After seeding
846930886                   82612881
RAND_MAX: 2147483647        162735805
47                          62
92                          97
```

Arrays

```c
/* Random number basic code */

printf("%d\n", rand());
printf("%d\n", rand());
printf("RAND_MAX: %d\n", RAND_MAX);
// ( 2 ^ 32 / 2 ) - 1

printf("%d\n", rand_bound());
printf("%d\n", rand_bound());

/* After seeding */
printf("After seeding\n");
srand(time(0)); // vs srand(42);

printf("%d\n", rand());
printf("%d\n", rand());

printf("%d\n", rand_bound());
printf("%d\n", rand_bound());
```

```c
int rand_bound()
{
   /* Random number between a lower/upper bound */
   int l_bound = 10;
   int u_bound = 100;

   int width = (u_bound - l_bound) + 1;

   int val = 0;

   val = rand() % width + l_bound;

   return val;
}
```

```
1804289383
846930886
RAND_MAX: 2147483647
47
92
```

```
After seeding
82612881
162735805
62
97
```

Dynamic 1D Array

Arrays

```c
/* Dynamic 1D array to store 12 numbers */
int *ptr_1D = NULL;

ptr_1D = (int *) malloc(12 * sizeof(int));

for (int i = 0; i < 12; i++)
{
    ptr_1D[i] = rand_bound();
}

printf("ptr_1D: ");
for (int i = 0; i < 12; i++)
{
    printf("%d ", ptr_1D[i]);
}
printf("\n");
```

```
ptr_1D: 88 84 47 92 11 76 48 26 11 69 12 78
```

Note: Free the memory allocated for `ptr_1D`

Dynamic 2D Array

Arrays

```
/* Dynamic 2D array; 3 x 4; random elements */

int **ptr_2D = NULL;

ptr_2D = (int **) malloc(3 * sizeof(int *));

for (int i = 0; i < 3; i++)
{
    ptr_2D[i] = (int *) malloc(4 * sizeof(int));
}

for (int r = 0; r < 3; r++)
    for (int c = 0; c < 4; c++)
    {
        ptr_2D[r][c] = rand_bound();
    }
```

Arrays

```c
/* Dynamic 2D array; 3 x 4; random elements */

int **ptr_2D = NULL;

ptr_2D = (int **) malloc(3 * sizeof(int *));

for (int i = 0; i < 3; i++)
{
    ptr_2D[i] = (int *) malloc(4 * sizeof(int));
}

for (int r = 0; r < 3; r++)
    for (int c = 0; c < 4; c++)
    {
        ptr_2D[r][c] = rand_bound();
    }
```

```c
printf("ptr_2D:\n");
for (int r = 0; r < 3; r++)
{
    for (int c = 0; c < 4; c++)
    {
        printf("%3d ", ptr_2D[r][c]);
    }
    printf("\n");
}
```

```
ptr_2D:
 23  14 100  24
 97  46  33  47
 69  31  78  26
```

Arrays

```c
/* Dynamic 2D array; 3 x 4; random elements */

int **ptr_2D = NULL;

ptr_2D = (int **) malloc(3 * sizeof(int *));

for (int i = 0; i < 3; i++)
{
    ptr_2D[i] = (int *) malloc(4 * sizeof(int));
}

for (int r = 0; r < 3; r++)
    for (int c = 0; c < 4; c++)
    {
        ptr_2D[r][c] = rand_bound();
    }
```

```c
printf("ptr_2D:\n");
for (int r = 0; r < 3; r++)
{
    for (int c = 0; c < 4; c++)
    {
        printf("%3d ", ptr_2D[r][c]);
    }
    printf("\n");
}

/* Free the memory */
free(ptr_1D);

for (int i = 0; i < 3; i++)
    free(ptr_2D[i]);
free(ptr_2D);
```

```
ptr_2D:
 23  14 100  24
 97  46  33  47
 69  31  78  26
```