Unit – I : Elementary Data Structures

- Data structures
- Asymptotic complexity
- Abstract data type:
  - Array,
  - Stacks,
  - Queues,
  - Linked Lists, and their applications.

Background

- In computer terms, a data structure is a Specific way to store and organize data in a computer's memory so that these data can be used efficiently later. Data may be arranged in many different ways such as the logical or mathematical model for a particular organization of data is termed as a data structure.
- The variety of a particular data model depends on the two factors –
  - Firstly, it must be loaded enough in structure to reflect the actual relationships of the data with the real world object.
  - Secondly, the formation should be simple enough so that anyone can efficiently process the data each time it is necessary.

Categories of Data Structure

- Linear Data Structure
- Non-linear Data Structure

Linear Data Structure

- A data structure is said to be linear if its elements combine to form any specific order. There are basically two techniques of representing such linear structure within memory.
- First way is to provide the linear relationships among all the elements represented by means of linear memory location. These linear structures are termed as arrays.
- The second technique is to provide the linear relationship among all the elements represented by using the concept of pointers or links. These linear structures are termed as linked lists.

Non linear Data Structure

- This structure is mostly used for representing data that contains a hierarchical relationship among various elements.
- Examples of Non Linear Data Structures are listed below:
- Graphs
- family of trees and
- table of contents

Characteristics of a Data Structure

- **Correctness** − Data structure implementation should implement its interface correctly.
- **Time Complexity** − Running time or the execution time of operations of data structure must be as small as possible.
- **Space Complexity** − Memory usage of a data structure operation should be as little as possible.

Need for Data Structure

- As applications are getting complex and data rich, there are three common problems that applications face now-a-days.
- **Data Search** − Consider an inventory of 1 million(10^6) items of a store. If the application is to search an item, it has to search an item in 1 million(10^6) items every time slowing down the search. As data grows, search will become slower.
- **Processor speed** − Processor speed although being very high, falls limited if the data grows to billion records.
- **Multiple requests** − As thousands of users can search data simultaneously on a web server, even the fast server fails while searching the data.

Think

- A Library
- Organized Structure
- Computer thinks only in step by step

Pseudocode

- Pseudocode is an English-like representation of the algorithm logic.
- It consists of an extended version of the basic algorithmic constructs: sequence, selection, and iteration.

Asymptotic complexity

- For example, let us say, we want to store marks of all students in a class, we can use an array to store them. This helps in reducing the use of number of variables as we don't need to create a separate variable for marks of every subject. All marks can be accessed by simply traversing the array.

Asymptotic complexity

- Suppose you have a bicycle, a motorcycle and a helicopter. You spend most time when you ride bicycle, lesser with motorcycle and least with helicopter to cover the same distance. So you say that big-O notation is minimum for helicopter and maximum for bicycle.

  As you increase the distance to be covered, the time difference between bicycle, car and helicopter will increase. The bigger the distance, the more saving with fastest mode of travel.

Asymptotic complexity

Asymptotic complexity

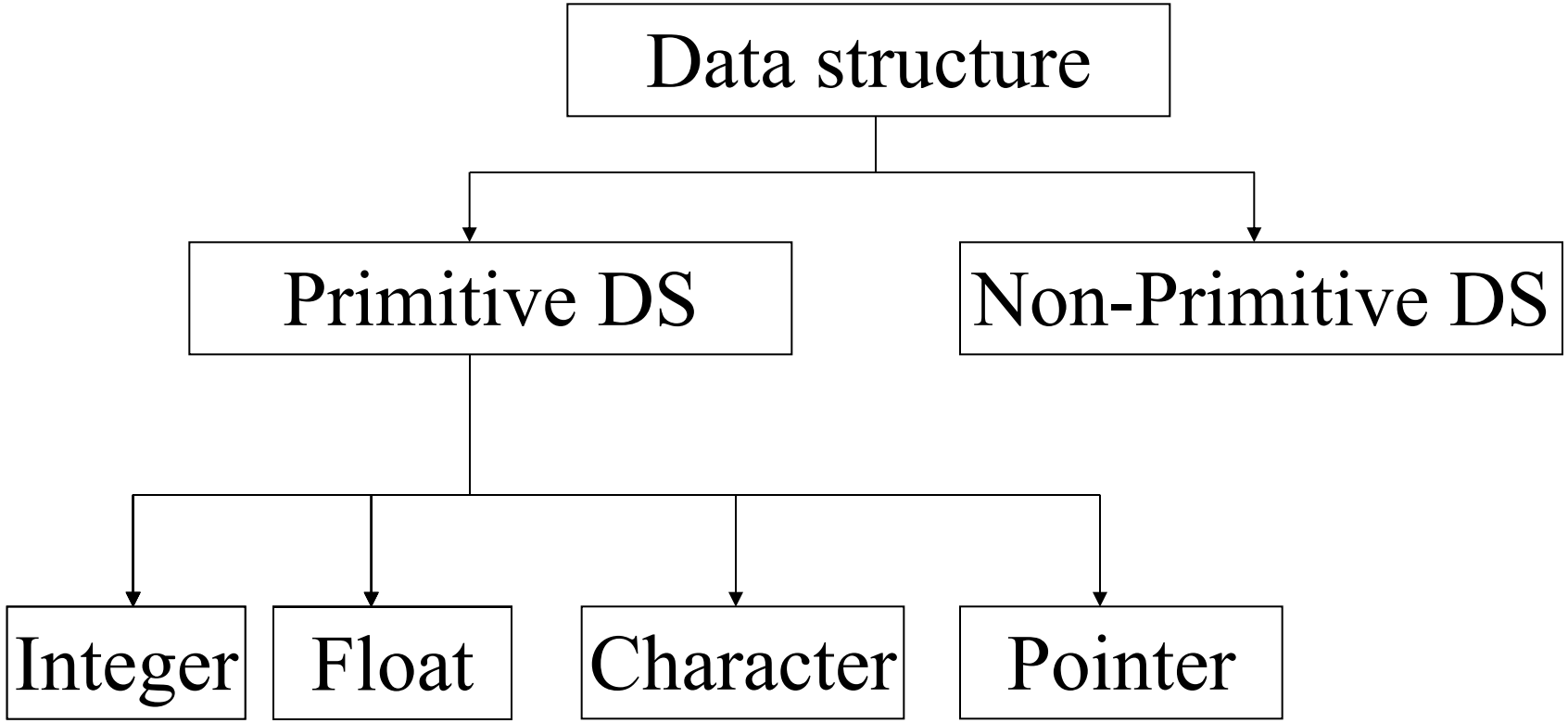
- Big-O notation tell you the **order of time or space** (memory space). It tells you the relation between time/space and your data size. The lesser the order the faster the operation (or lesser the space requirement)

Array

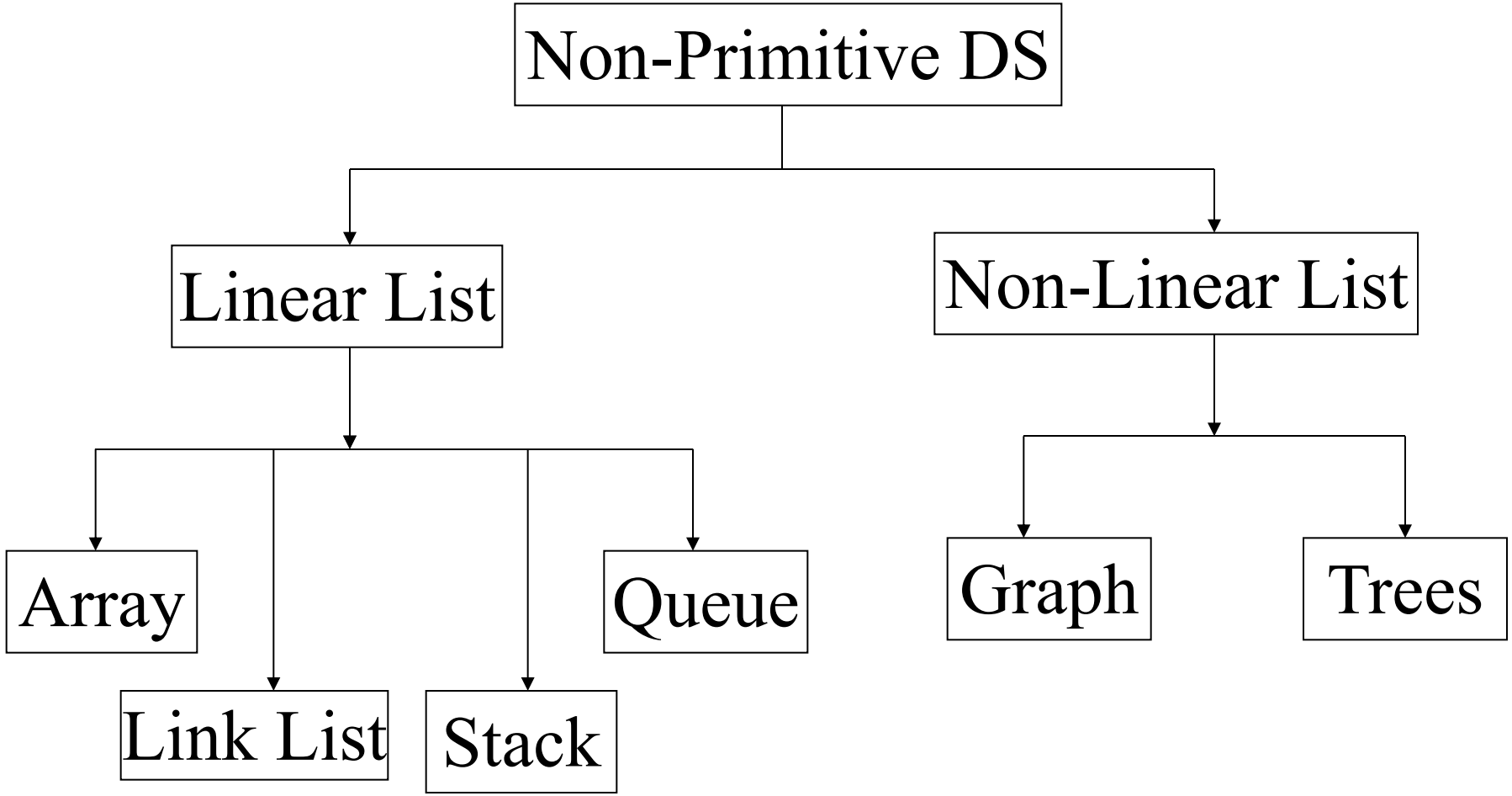
```
Let size of array be n.
Accessing Time: O(1) [This is possible because elements
                      are stored at contiguous locations]
Search Time:    O(n) for Sequential Search:
                O(log n) for Binary Search [If Array is sorted]
Insertion Time: O(n) [The worst case occurs when insertion
                     happens at the Beginning of an array and
                     requires shifting all of the elements]
Deletion Time: O(n) [The worst case occurs when deletion
                     happens at the Beginning of an array and
                     requires shifting all of the elements]
```

Definition 1.2 [Space/Time complexity] The *space complexity* of an algorithm is the amount of memory it needs to run to completion. The *time complexity* of an algorithm is the amount of computer time it needs to run to completion. □

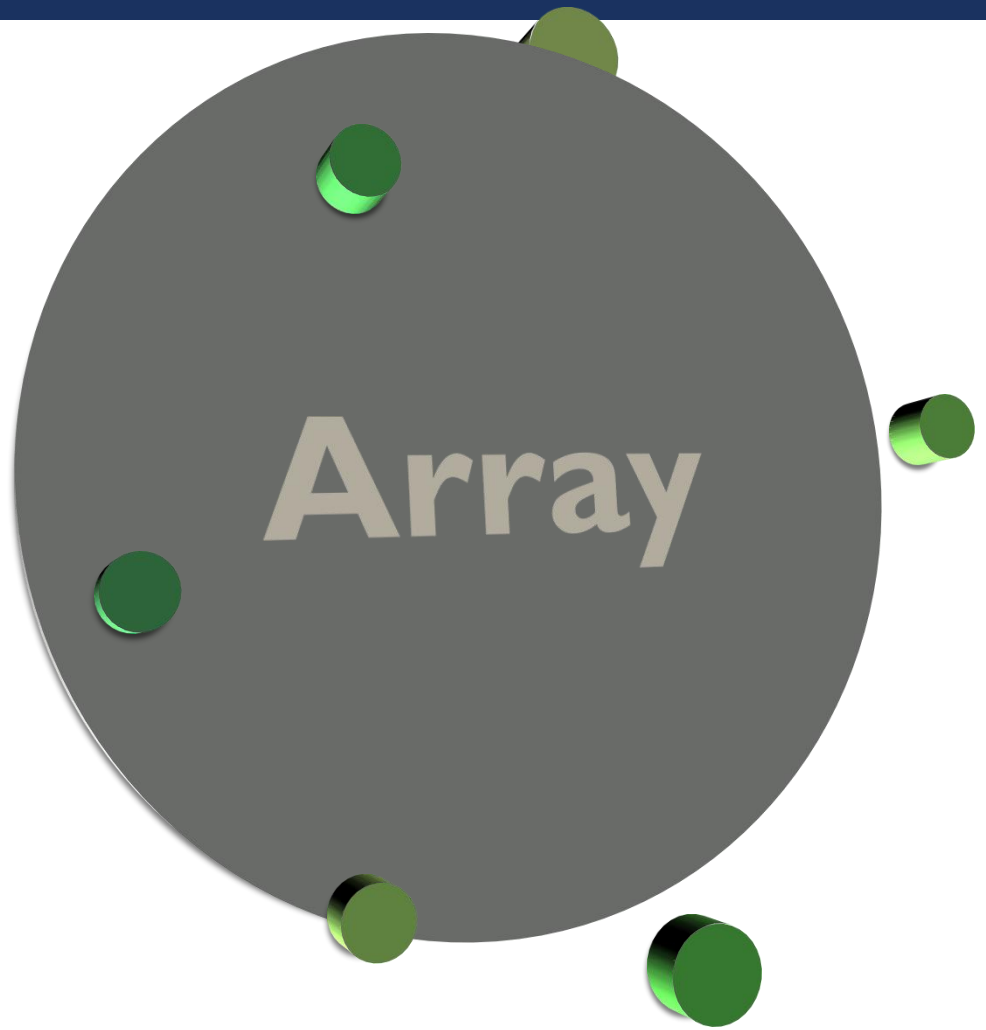
Classification of Data Structure



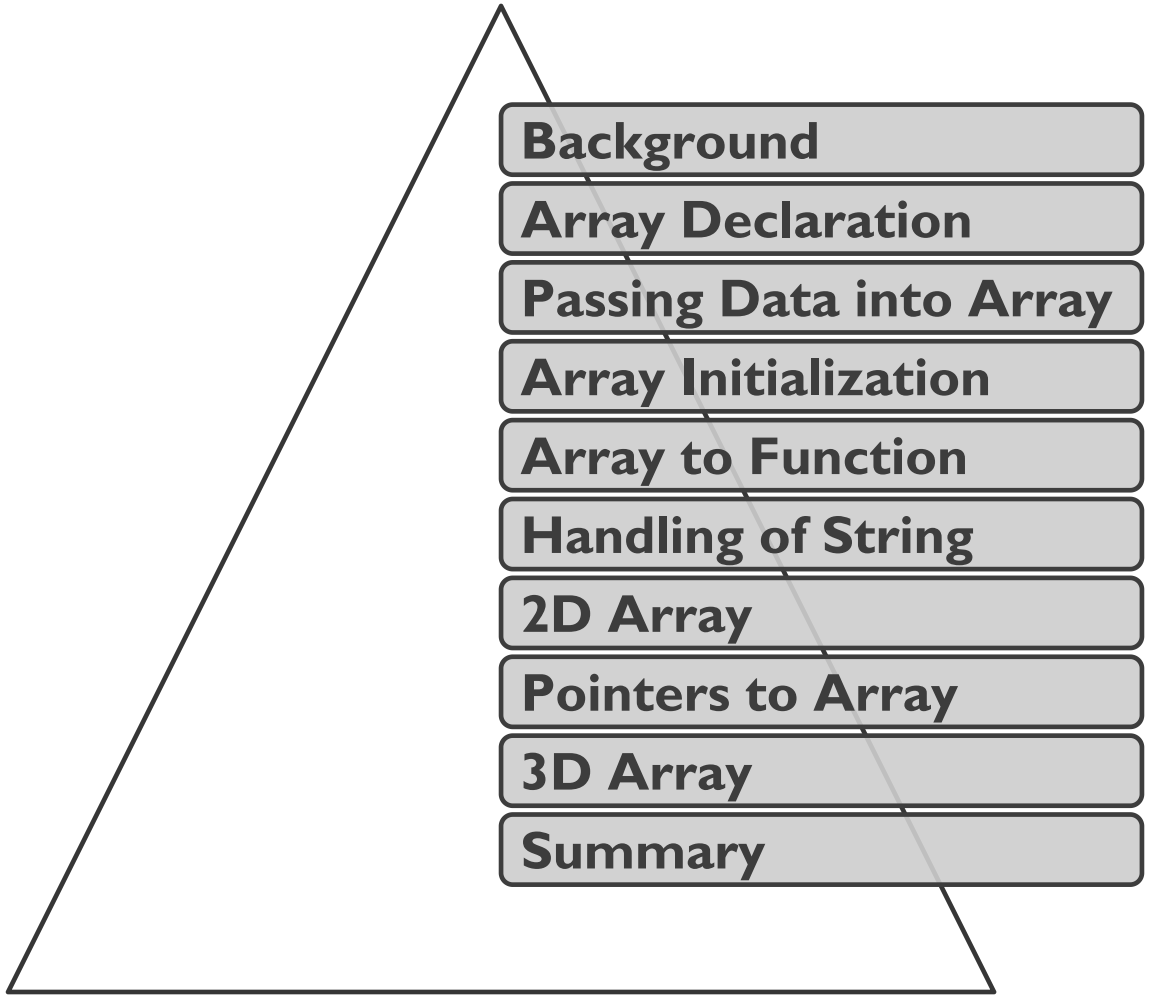
Classification of Data Structure

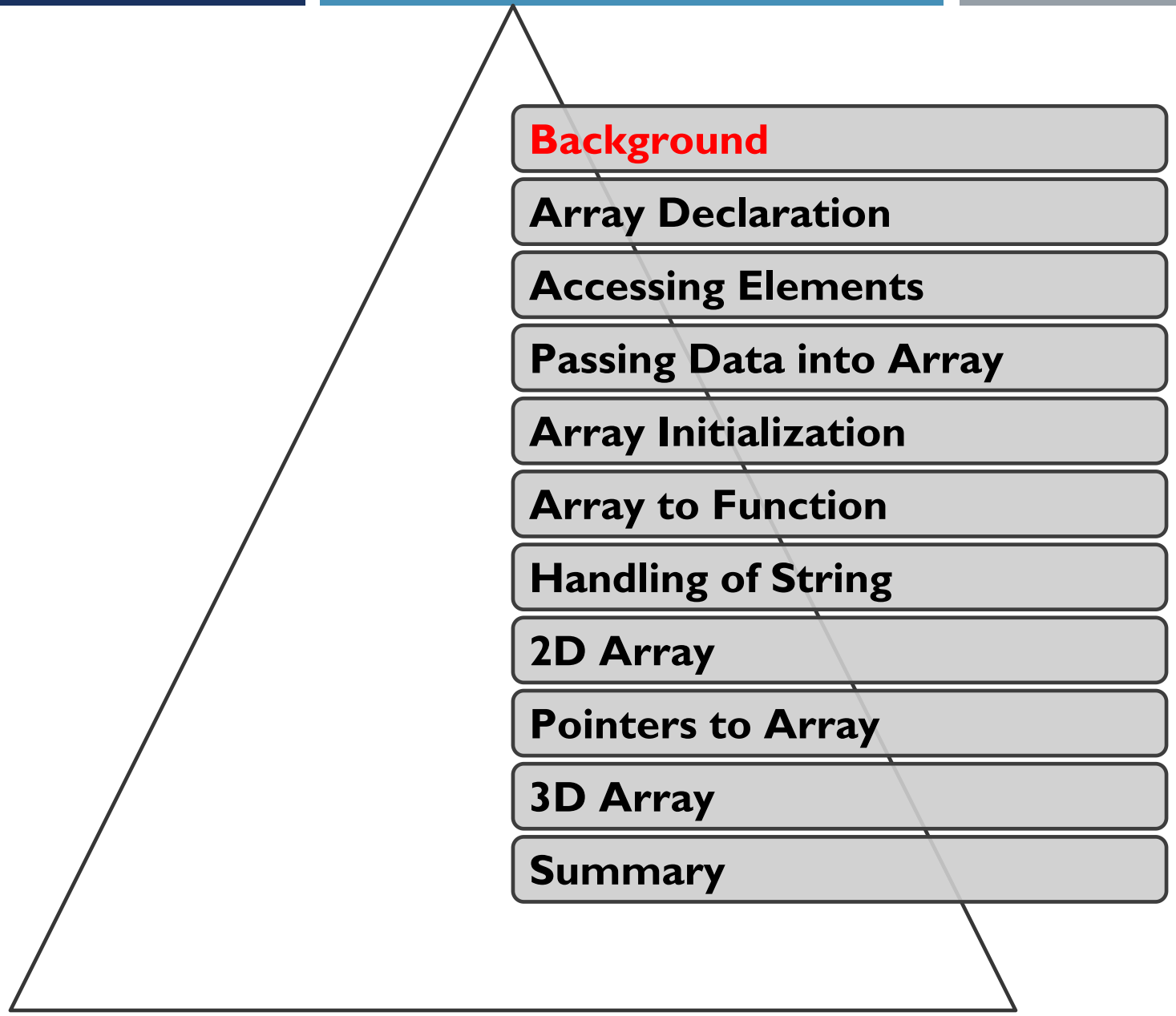


Todays Agenda



Detailed Agenda

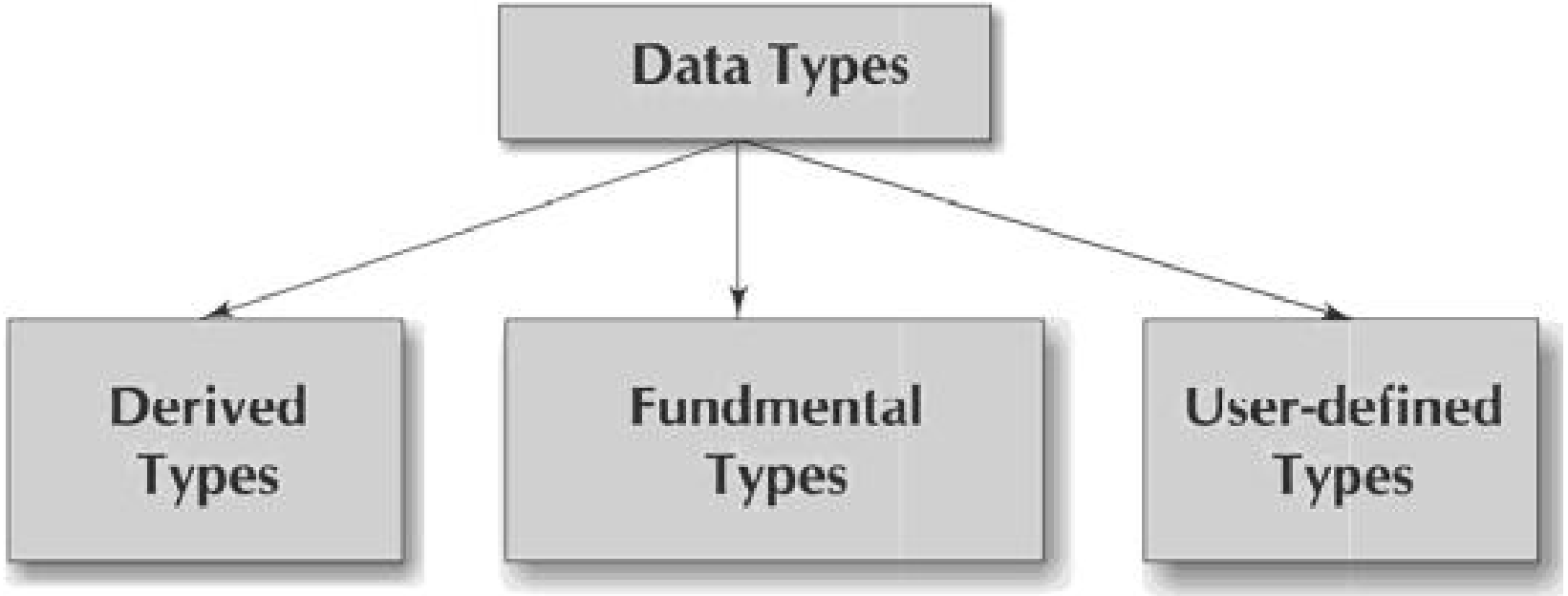



Background
Array Declaration
Accessing Elements
Passing Data into Array
Array Initialization
Array to Function
Handling of String
2D Array
Pointers to Array
3D Array
Summary

Introduction

- Fundamental Data types
  - int, char, float, double
    - Can Store one value at a time
    - Handle limited amounts of data
- To Process large amount of data, we need a powerful data type that would facilitate efficient storing, accessing and manipulation of data items.
- C supports a derived data types known as array

DERIVED DATA TYPES



Basics

- An array is a flexible structure for storing a sequence of values all of the same type.
  - A structure that holds multiple values of the same type.
  - The values stored in an array are called elements. The individual elements are accessed using an integer index [SUB SCRIPT].
  - An integer indicating the position of a value in a data structure
    - Indexed starting with 0

Example

```
main( )
{
        int  x ;
        x = 5 ;
        x = 10 ;
        printf ( "\nx = %d", x ) ;
}
```

- Because when a value 10 is assigned to **x**, the earlier value of **x**, i.e. 5, is lost. Thus, ordinary variables are capable of holding only one value at a time.

- However, there are situations in which we would want to store more than one value at a time in a single variable.

Background

- Suppose we wish to arrange the percentage marks obtained by 100 students in ascending order.
- Two options to store these marks in memory:

1. **Construct 100 variables to store percentage marks obtained by 100 different students, i.e. each variable containing one student's marks.**

2. **Construct one variable (called array or subscripted variable) capable of storing or holding all the hundred values.**

Background

- A simple reason for this is, it would be much easier to handle one variable than handling 100 different variables.

- Formal definition of an array—**An array is a collective name given to a group of 'similar quantities'.**
  - Percentage marks of 100 students, or salaries of 300 employees, or ages of 50 employees.

For example

- marks obtained by five students.
- per = { 48, 88, 34, 23, 96 }
  - If we want to refer to the second number of the group, the usual notation used is per2. Similarly, the fourth number of the group is referred as per4.
- However, in C, the fourth number is referred as **per[3].** This is because in C the counting of elements begins with 0 and not with 1. Thus, in this example **per[3]** refers to 23 and **per[4]** refers to 96.
- In general, the notation would be **per[i]**, where, **i** can take a value 0, 1, 2, 3, or 4, depending on the position of the element being referred.
  - Here **per** is the subscripted variable (array), whereas **i** is its subscript.

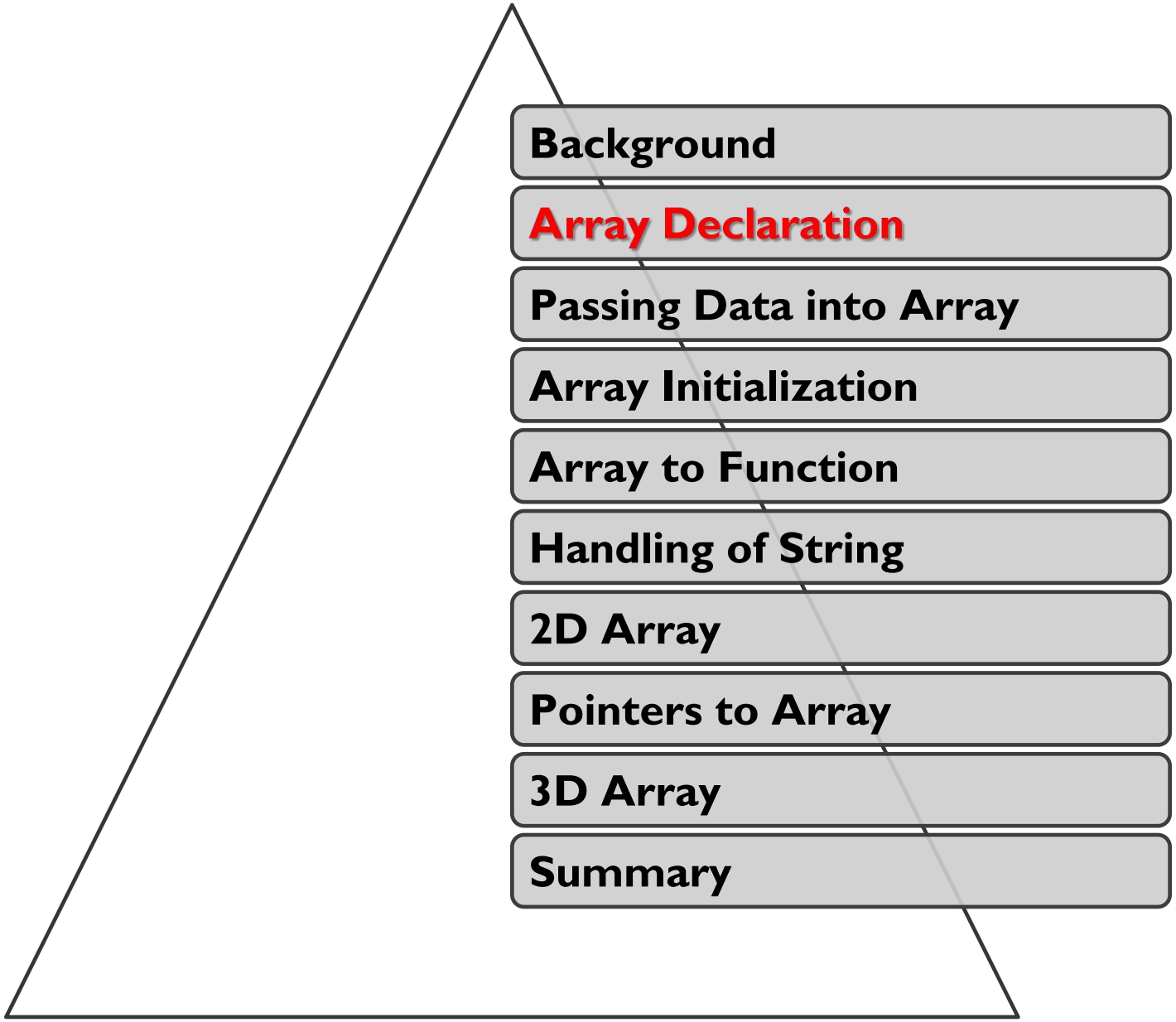
FIND AVERAGE MARKS OBTAINED BY A CLASS OF 30 STUDENTS IN A TEST.

```
main( )
{
      int  avg, sum = 0 ;
      int  i ;
      int  marks[30] ;  /* array declaration */

      for ( i = 0 ; i <= 29 ; i++ )
      {
            printf ( "\nEnter marks " ) ;
            scanf ( "%d", &marks[i] ) ;  /* store data in array */
      }

      for ( i = 0 ; i <= 29 ; i++ )
            sum = sum + marks[i] ;  /* read data from an array*/

      avg = sum / 30 ;
      printf ( "\nAverage marks = %d", avg ) ;
}
```


Background
Array Declaration
Passing Data into Array
Array Initialization
Array to Function
Handling of String
2D Array
Pointers to Array
3D Array
Summary

Array Declaration

- Like other variables an array needs to be declared so that the compiler will know what kind of an array and how large an array we want.
- int marks[30] ;
  - **int** specifies the type of the variable, just as it does with ordinary variables and the word **marks** specifies the name of the variable.
  - The **[30]** however is new. The number 30 tells how many elements of the type **int** will be in our array. This number is often called the 'dimension' of the array.
  - The bracket [ ] tells the compiler that we are dealing with an array.

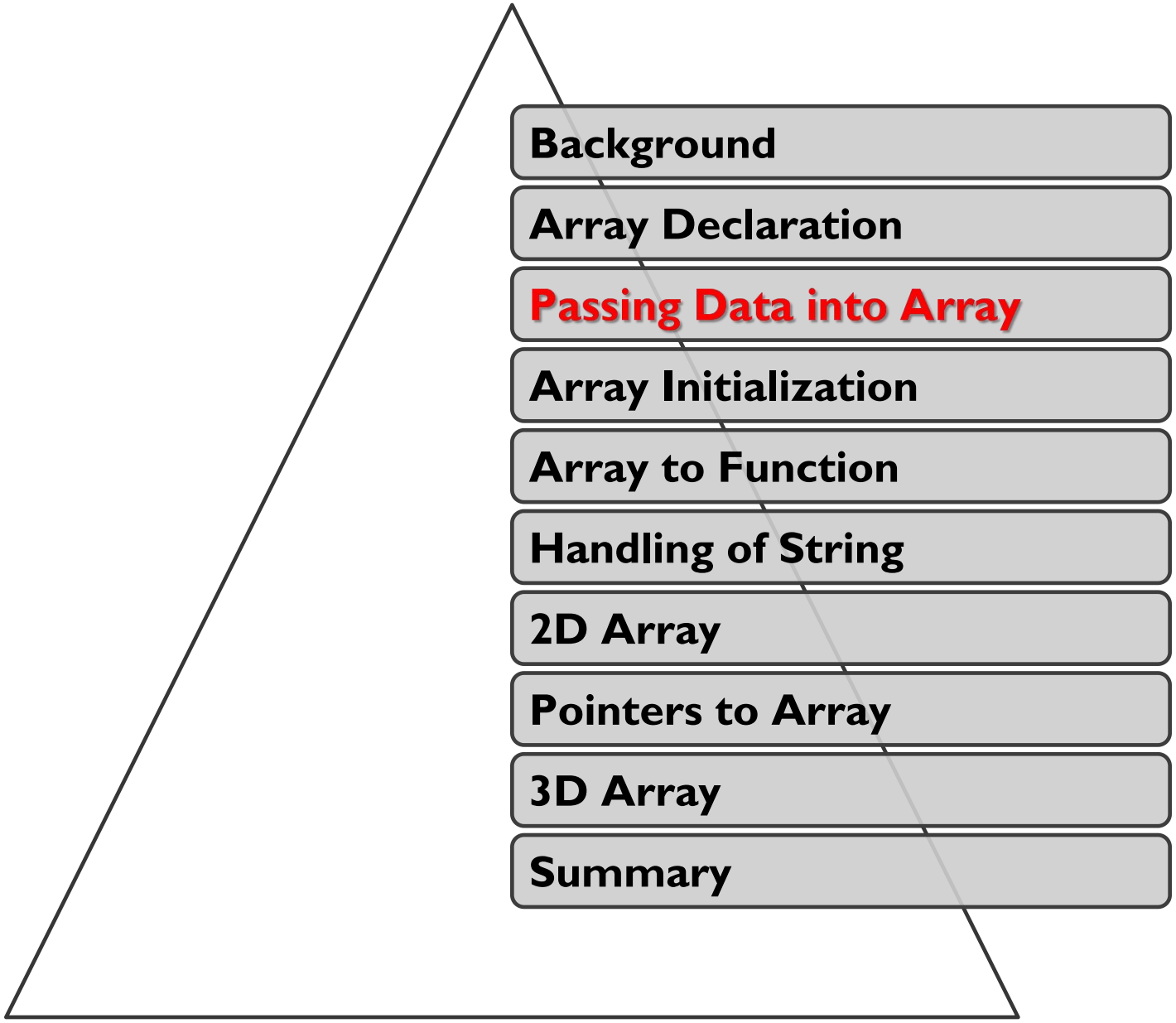
Accessing Elements of an Array

- Once an array is declared, let us see how individual elements in the array can be referred.

- This is done with subscript, the number in the brackets following the array name. This number specifies the element's position in the array.

Accessing Elements of an Array

- All the array elements are numbered, starting with 0. Thus, **marks[2]** is not the second element of the array, but the third.
- In our program we are using the variable **i** as a subscript to refer to various elements of the array.
  - This variable can take different values and hence can refer to the different elements in the array in turn. This ability to use variables as subscripts is what makes arrays so useful.


Background
Array Declaration
Passing Data into Array
Array Initialization
Array to Function
Handling of String
2D Array
Pointers to Array
3D Array
Summary

Entering Data into an Array

```
for ( i = 0 ; i <= 29 ; i++ )
{
    printf ( "\nEnter marks " ) ;
    scanf ( "%d", &marks[i] ) ;
}
```

- The code that places data into an array:
- The **for** loop causes the process of asking for and receiving a student's marks from the user to be repeated 30 times.
- The first time through the loop, **i** has a value 0, so the **scanf( )** function will cause the value typed to be stored in the array element **marks[0],** the first element of the array.

Entering Data into an Array

```
for ( i = 0 ; i <= 29 ; i++ )
{
        printf ( "\nEnter marks " ) ;
        scanf ( "%d", &marks[i] ) ;
}
```

- This process will be repeated until **i** becomes 29.
  - This is last time through the loop, which is a good thing, because there is no array element like **marks[30]**.

Entering Data into an Array

- In **scanf( )** function, we have used the "address of" operator (&) on the element **marks[i]** of the array, just as we have used it earlier on other variables **(&a**, for example).

- Passing the address of this particular array element to the **scanf( )** function, rather than its value; which is what **scanf( )** requires.

READING DATA FROM AN ARRAY

Reading Data from an Array

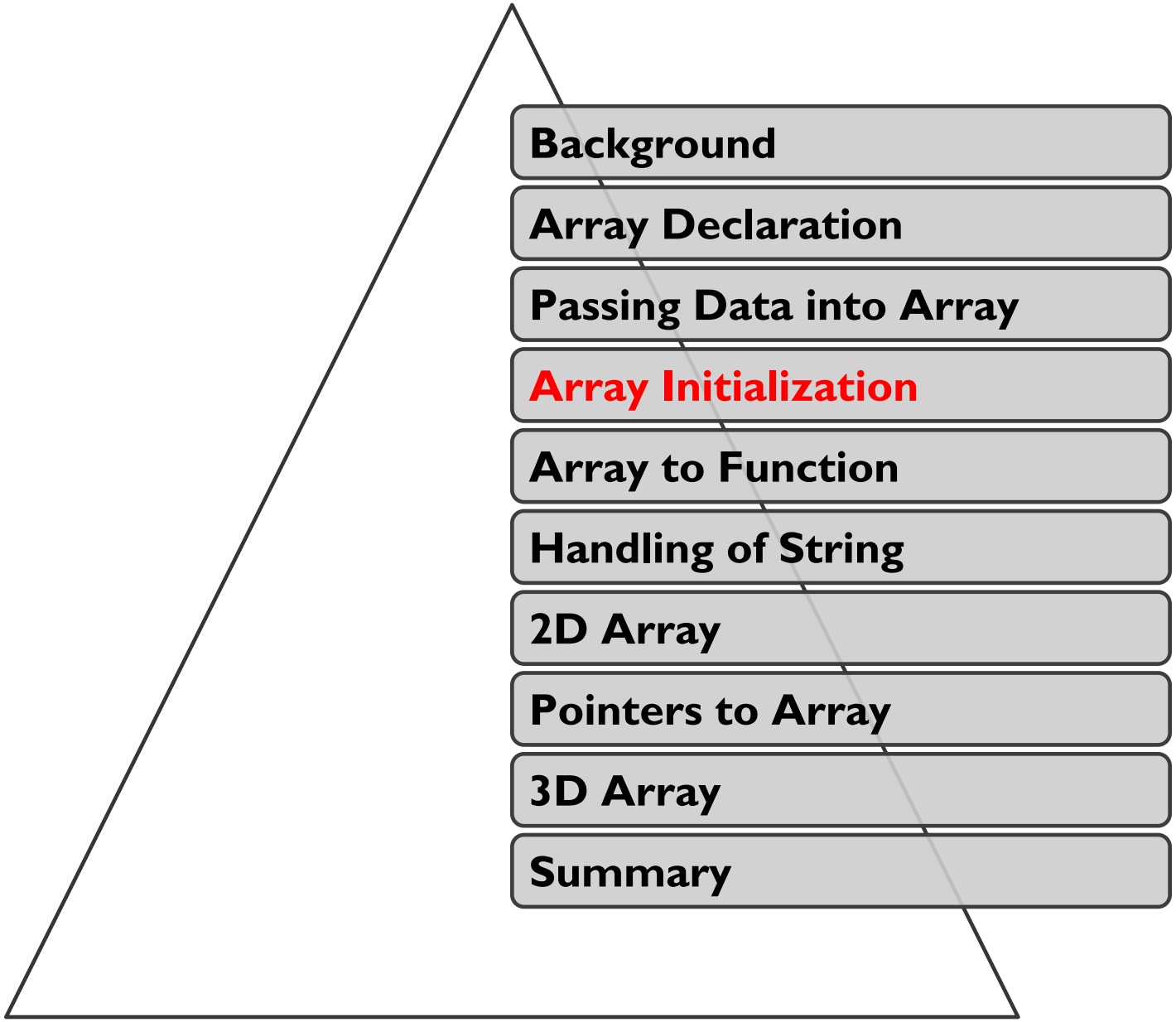
```
for ( i = 0 ; i <= 29 ; i++ )
      sum = sum + marks[i] ;

avg = sum / 30 ;
printf ( "\nAverage marks = %d", avg ) ;
```

- Data back out of the array and uses it to calculate the average.
- The **for** loop is much the same, but now the body of the loop causes each student's marks to be added to a running total stored in a variable called **sum**.
- When all the marks have been added up, the result is divided by 30, the number of students, to get the average.

Sum-up

- An array is a collection of similar elements.
- The first element in the array is numbered 0, so the last element is 1 less than the size of the array.
- An array is also known as a subscripted variable.
- Before using an array its type and dimension must be declared.
- However big an array its elements are always stored in contiguous memory locations. This is a very important point


Background
Array Declaration
Passing Data into Array
Array Initialization
Array to Function
Handling of String
2D Array
Pointers to Array
3D Array
Summary

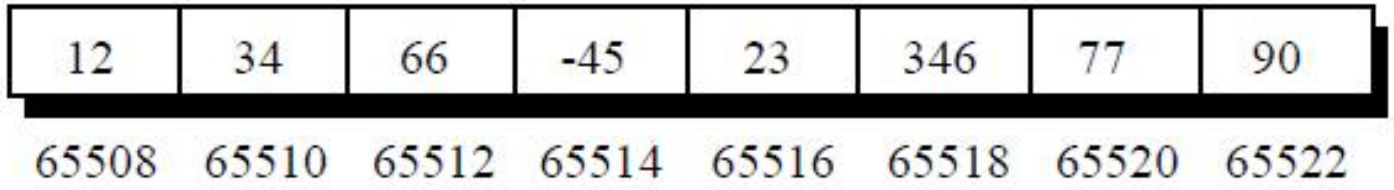
Array Initialization

- How to initialize an array while declaring it.
  - int num[6] = { 2, 4, 12, 5, 45, 5 } ;
  - int n[ ] = { 2, 4, 12, 5, 45, 5 } ;
  - float press[ ] = { 12.3, 34.2 -23.4, -11.3 } ;

- Till the array elements are not given any specific values, they are supposed to contain garbage values.
- If the array is initialized where it is declared, mentioning the dimension of the array is optional.

Array Elements in Memory

- int arr[8];



- What happens in memory when we make this declaration?

- 16 bytes get immediately reserved in memory
- Since the array is not being initialized, all eight values present in it would be garbage values. This so happens because the storage class of this array is assumed to be **auto**.
- If the storage class is declared to be **static** then all the array elements would have a default initial value as zero. Whatever be the initial values, all the array elements would always be present in contiguous memory locations.

Bounds Checking

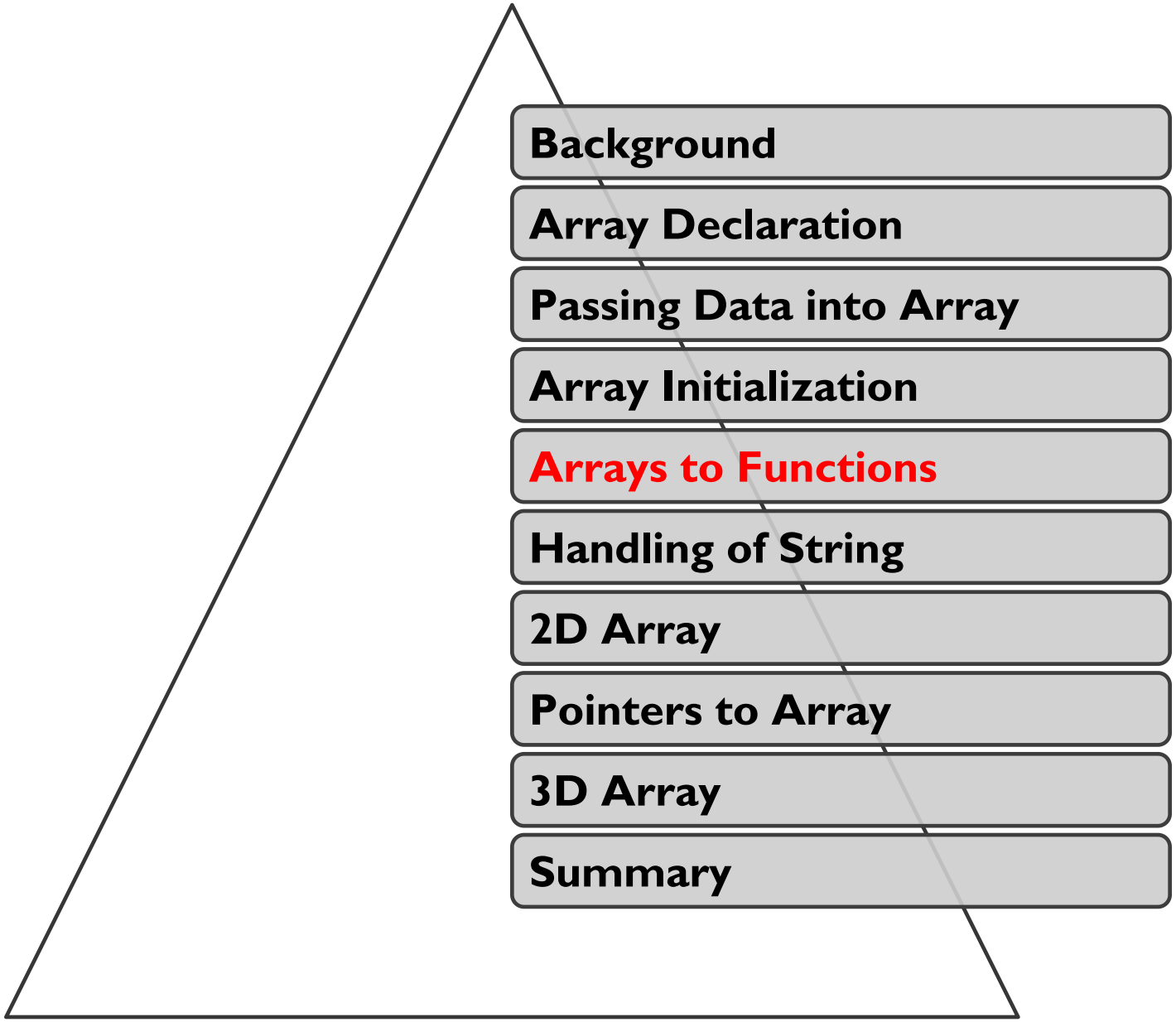
- In C there is no check to see if the subscript used for an array exceeds the size of the array.
- Data entered with a subscript exceeding the array size will simply be placed in memory outside the array; probably on top of other data, or on the program itself.
  - This will lead to unpredictable results, to say the least, and there will be no error message to warn you that you are going beyond the array size.
  - In some cases the computer may just hang.

Bounds Checking

```
main( )
{
        int  num[40], i ;

        for ( i = 0 ; i <= 100 ; i++ )
                num[i] = i ;
}
```

we do not reach beyond the array size is entirely the programmer's botheration and not the compiler's.


Background
Array Declaration
Passing Data into Array
Array Initialization
Arrays to Functions
Handling of String
2D Array
Pointers to Array
3D Array
Summary

Passing Array Elements to a Function

- Array elements can be passed to a function by calling the function by value, or by reference.

- In the call by value we pass values of array elements to the function, whereas in the call by reference we pass addresses of array elements to the function.

Call by value

```
main( )
{
      int  i ;
      int  marks[ ] = { 55, 65, 75, 56, 78, 78, 90 } ;

      for ( i = 0 ; i <= 6 ; i++ )
            display ( marks[i] ) ;
}

display ( int  m )
{
      printf ( "%d ", m ) ;
}
```

55 65 75 56 78 78 90

we are passing an individual array element at a time to the function **display( )** and getting it printed in the function **display( ).**

Note that since at a time only one element is being passed, this element is collected in an ordinary integer variable **m**, in the function **display( )**.

CALL BY REFERENCE

```
main( )
{
    int i ;
    int marks[ ] = { 55, 65, 75, 56, 78, 78, 90 } ;

    for ( i = 0 ; i <= 6 ; i++ )
        disp ( &marks[i] ) ;
}

disp ( int *n )
{
    printf ( "%d ", *n ) ;
}
```

55 65 75 56 78 78 90

we are passing addresses of individual array elements to the function **display( )**. Hence, the variable in which this address is collected (**n**) is declared as a pointer variable.

And since **n** contains the address of array element, to print out the array element we are using the 'value at address' operator (*).

PASSING AN ENTIRE ARRAY TO A FUNCTION

Background

- We saw two programs—
  - one in which we passed individual elements of an array to a function, and
  - another in which we passed addresses of individual elements to a function.

- Now see how to pass an entire array to a function rather than its individual elements.

/* Demonstration of passing an entire array to a function */

```
main( )
{
int num[ ] = { 24, 34, 12, 44, 56, 17 } ;
dislpay ( &num[0], 6 ) ;
}
display ( int *j, int n )
{
int i ;
for ( i = 0 ; i <= n - 1 ; i++ )
{
printf ( "\n element = %d", *j ) ;
j++ ;  /* increment pointer to point to next element */
} }
```

the **display( )** function is used to print out the array elements. Note that the address of the zeroth element is being passed to the **display( )** function.

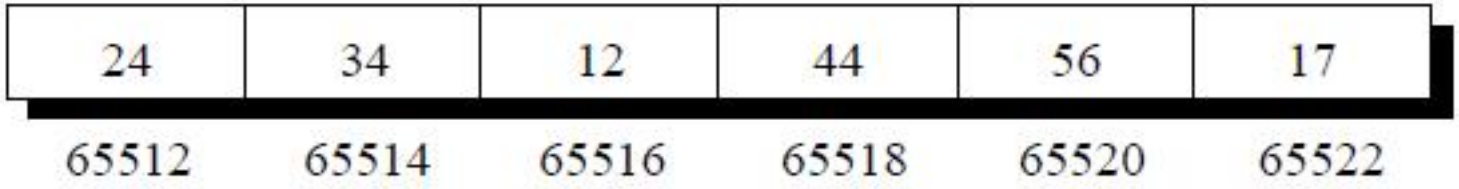
The **for** loop is same as the one used in the earlier program to access the array elements using pointers. Thus, just passing the address of the zeroth element of the array to a function is as good as passing the entire array to the function.

It is also necessary to pass the total number of elements in the array, otherwise the **display( )** function would not know when to terminate the **for** loop.

Note that the address of the zeroth element (many a times called the base address) can also be passed by just passing the name of the array.

Explanation

```
main( )
{
int num[ ] = { 24, 34, 12, 44, 56, 17 } ;
dislpay ( &num[0], 6 ) ;
}
display ( int *j, int n )
{
int i ;
for ( i = 0 ; i <= n - 1 ; i++ )
{
printf ( "\n element = %d", *j ) ;
j++ ;   /* increment pointer to point to next element */
} }
```



int num[ ] = { 24, 34, 12, 44, 56, 17 } ;

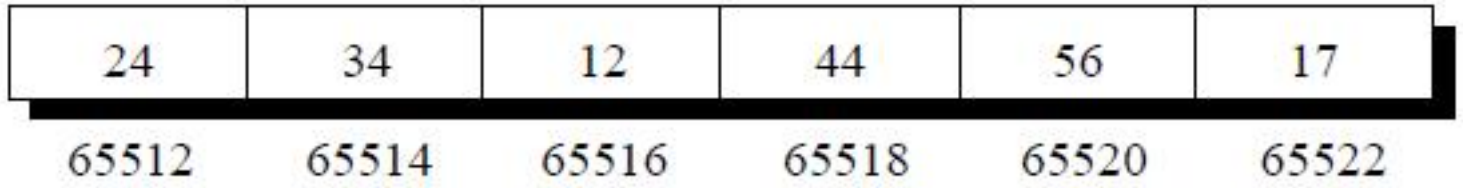
- Mentioning the name of the array we get its base address. Thus, by saying ***num** we would be able to refer to the zeroth element of the array, that is, 24.
  - ***num** and ***(num + 0)** both refer to 24.

- Similarly, by saying ***(num + 1)** refers the first element of the array, that is, 34. In fact, this is what the C compiler does internally. When we say, **num[i]**, the C compiler internally converts it to ***(num + i)**.
  - following notations are same:
    - num[i], *(num + i), *(i + num), i[num]

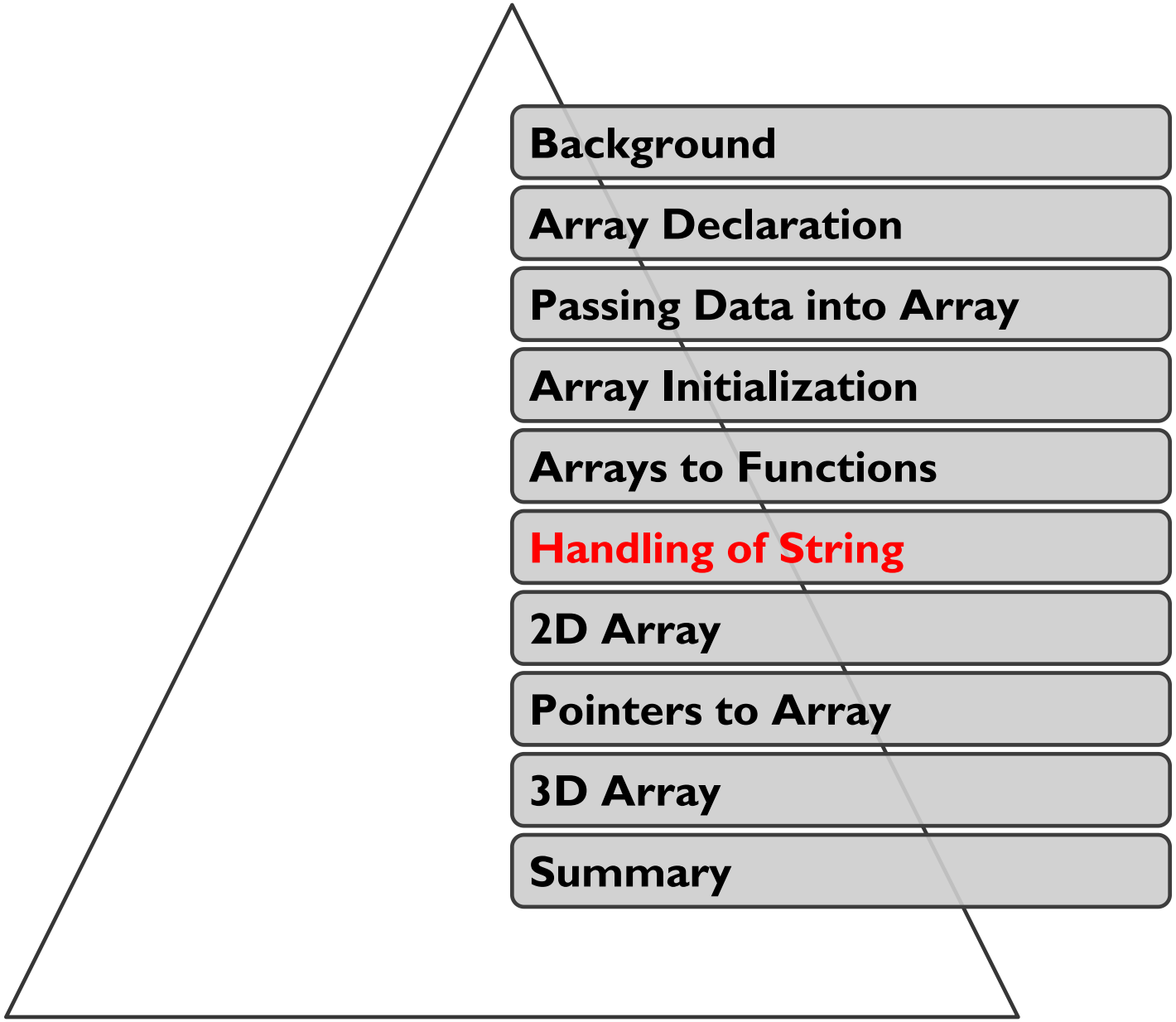
Accessing array elements in different ways

```
main( )
{
    int  num[ ] = { 24, 34, 12, 44, 56, 17 } ;
    int  i ;

    for ( i = 0 ; i <= 5 ; i++ )
    {
        printf ( "\naddress = %u ", &num[i] ) ;
        printf ( "element = %d %d ", num[i], *( num + i ) ) ;
        printf ( "%d %d", *( i + num ), i[num] ) ;
    }
}
```




```
address = 65512 element = 24 24 24 24
address = 65514 element = 34 34 34 34
address = 65516 element = 12 12 12 12
address = 65518 element = 44 44 44 44
address = 65520 element = 56 56 56 56
address = 65522 element = 17 17 17 17
```

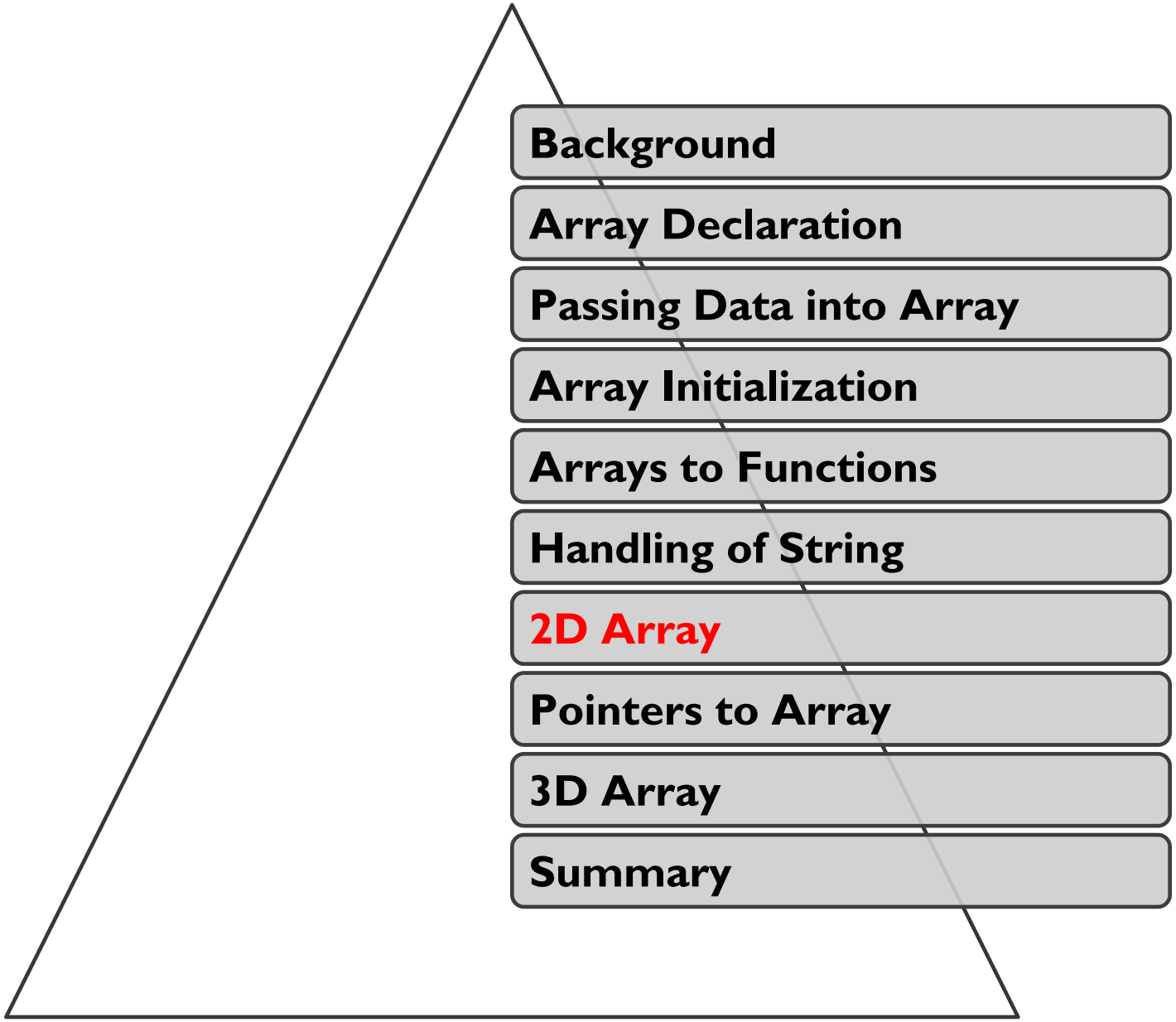

Background
Array Declaration
Passing Data into Array
Array Initialization
Arrays to Functions
Handling of String
2D Array
Pointers to Array
3D Array
Summary

Handling of Strings

- When a string constant is assigned to an external or a static character array as a part of array definition
  - The array size specification is usually omitted
  - The proper size will be assigned automatically
  - The null character \0 is automatically added at the end of every string
- Example
  - char color[] = “RED”;
  - char color[4] = “RED”;

Handling of Strings

- Example
  - char color[3] = “RED”;
- The above assignment is incorrect
  - color[0] = ‘R’
  - color[1] = ‘E’
  - color[2]= ‘D’
  - No assignment of null character \0


Background
Array Declaration
Passing Data into Array
Array Initialization
Arrays to Functions
Handling of String
2D Array
Pointers to Array
3D Array
Summary

Two Dimensional Arrays

- It is also possible for arrays to have two or more dimensions.
  - The two dimensional array is also called a matrix.

```
main( )
{
    int  stud[4][2] ;
    int  i, j ;

    for ( i = 0 ; i <= 3 ; i++ )
    {
        printf ( "\n Enter roll no. and marks" ) ;
        scanf ( "%d %d", &stud[i][0], &stud[i][1] ) ;
    }

    for ( i = 0 ; i <= 3 ; i++ )
        printf ( "\n%d %d", stud[i][0], stud[i][1] ) ;
}
```

- the first part through a **for** loop we read in the values of roll no. and marks,
- whereas, in second part through another **for** loop we print out these values.

scanf ("%d %d", &stud[i][0], &stud[i][1]) ;

- In **stud[i][0]** and **stud[i][1]** the first subscript of the variable **stud**, is row number which changes for every student.
  - The second subscript tells which of the two columns are we talking about—
  - The zeroth column which contains the roll no. or the first column which contains the marks.
- Remember the counting of rows and columns begin with zero.

Complete array arrangement

	col. no. 0	col. no. 1
row no. 0	1234	56
row no. 1	1212	33
row no. 2	1434	80
row no. 3	1312	78

Thus, 1234 is stored in **stud[0][0]**, 56 is stored in **stud[0][1]** and so on. The above arrangement highlights the fact that a two-dimensional array is nothing but a collection of a number of one-dimensional arrays placed one below the other.

Row Rule

- The array elements have been stored row wise and accessed row wise.
- However, you can access the array elements column wise as well.
- Traditionally, the array elements are being stored and accessed row wise; therefore we would also stick to the same strategy.

Initializing a 2-Dimensional Array

```
int  stud[4][2] = {
                    { 1234, 56 },
                    { 1212, 33 },
                    { 1434, 80 },
                    { 1312, 78 }
                  } ;
```

or even this would work...

```
int stud[4][2] = { 1234, 56, 1212, 33, 1434, 80, 1312, 78 } ;
```

of course with a corresponding loss in readability.

Initializing a 2-Dimensional Array

It is important to remember that while initializing a 2-D array it is necessary to mention the second (column) dimension, whereas the first dimension (row) is optional.

Thus the declarations,

```
int  arr[2][3] = { 12, 34, 23, 45, 56, 45 } ;
int  arr[ ][3] = { 12, 34, 23, 45, 56, 45 } ;
```

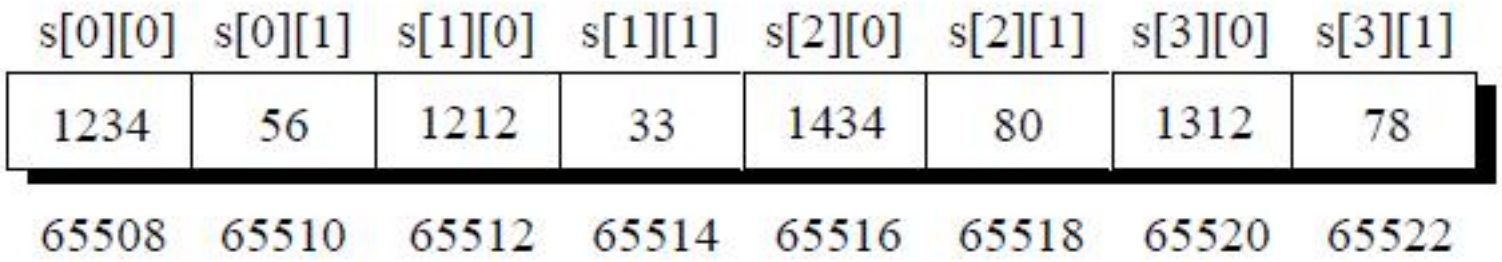
are perfectly acceptable,

whereas,

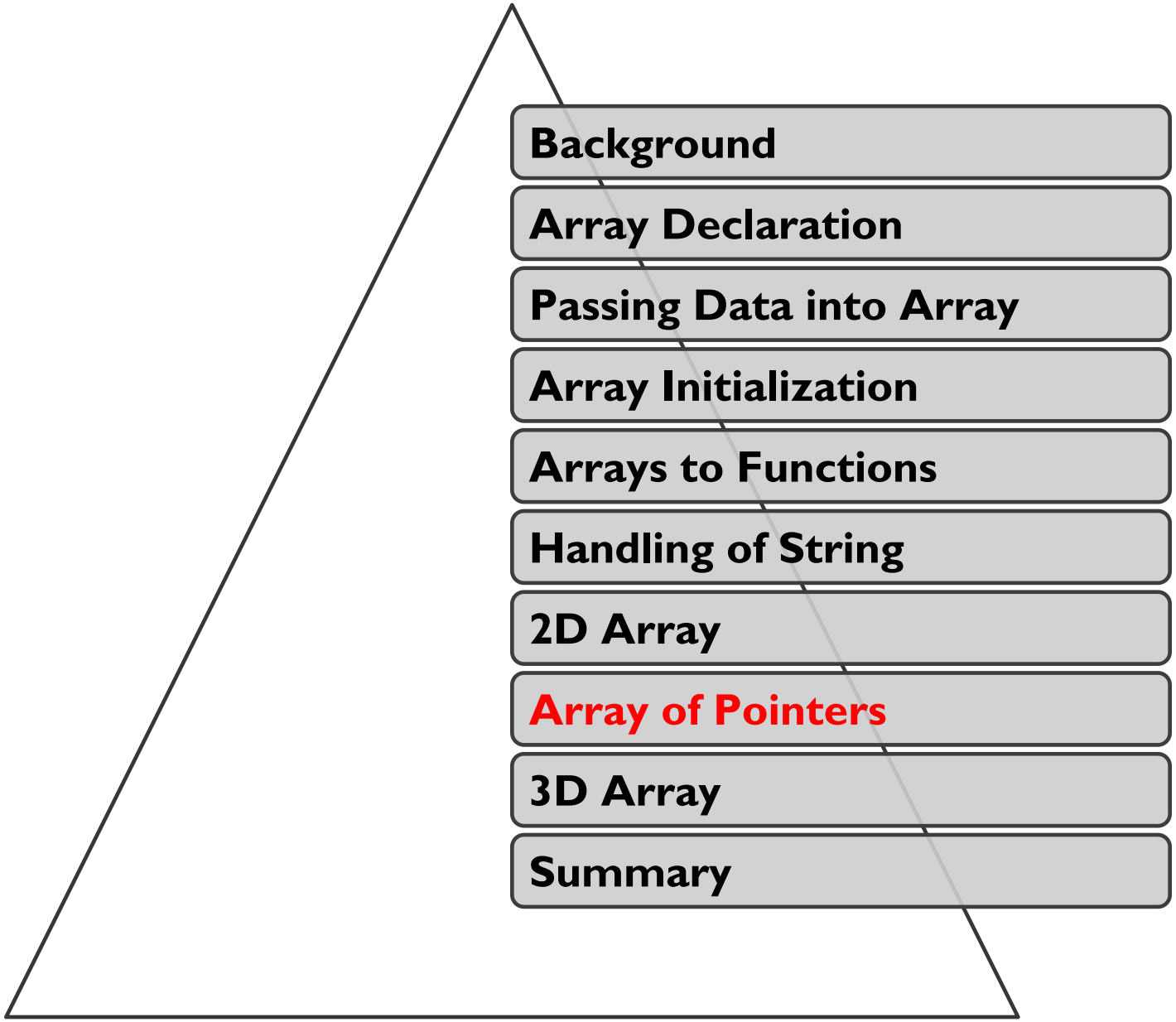
```
int  arr[2][ ] = { 12, 34, 23, 45, 56, 45 } ;
int  arr[ ][ ] = { 12, 34, 23, 45, 56, 45 } ;
```

would never work.

Memory Map of a 2-D Array



- The array arrangement is conceptually true.
  - This is because memory doesn't contain rows and columns.
- In memory whether it is a one-dimensional or a two-dimensional array the array elements are stored in one continuous chain.

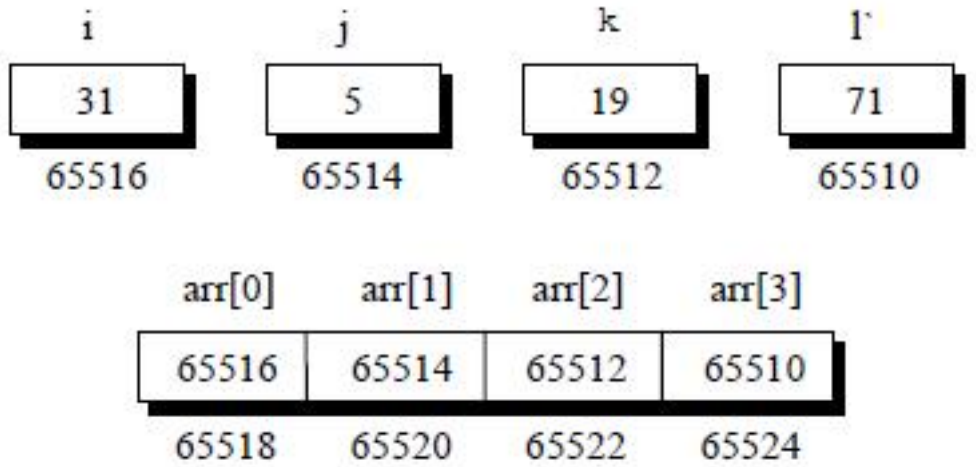

Background
Array Declaration
Passing Data into Array
Array Initialization
Arrays to Functions
Handling of String
2D Array
Array of Pointers
3D Array
Summary

Array of Pointers

- The way there can be an array of **int** s or an array of **floats**, similarly there can be an array of pointers.
  - Since a pointer variable always contains an address, an array of pointers would be nothing but a collection of addresses.
- The addresses present in the array of pointers can be addresses of isolated variables or addresses of array elements or any other addresses.
- All rules that apply to an ordinary array apply to the array of pointers as well.

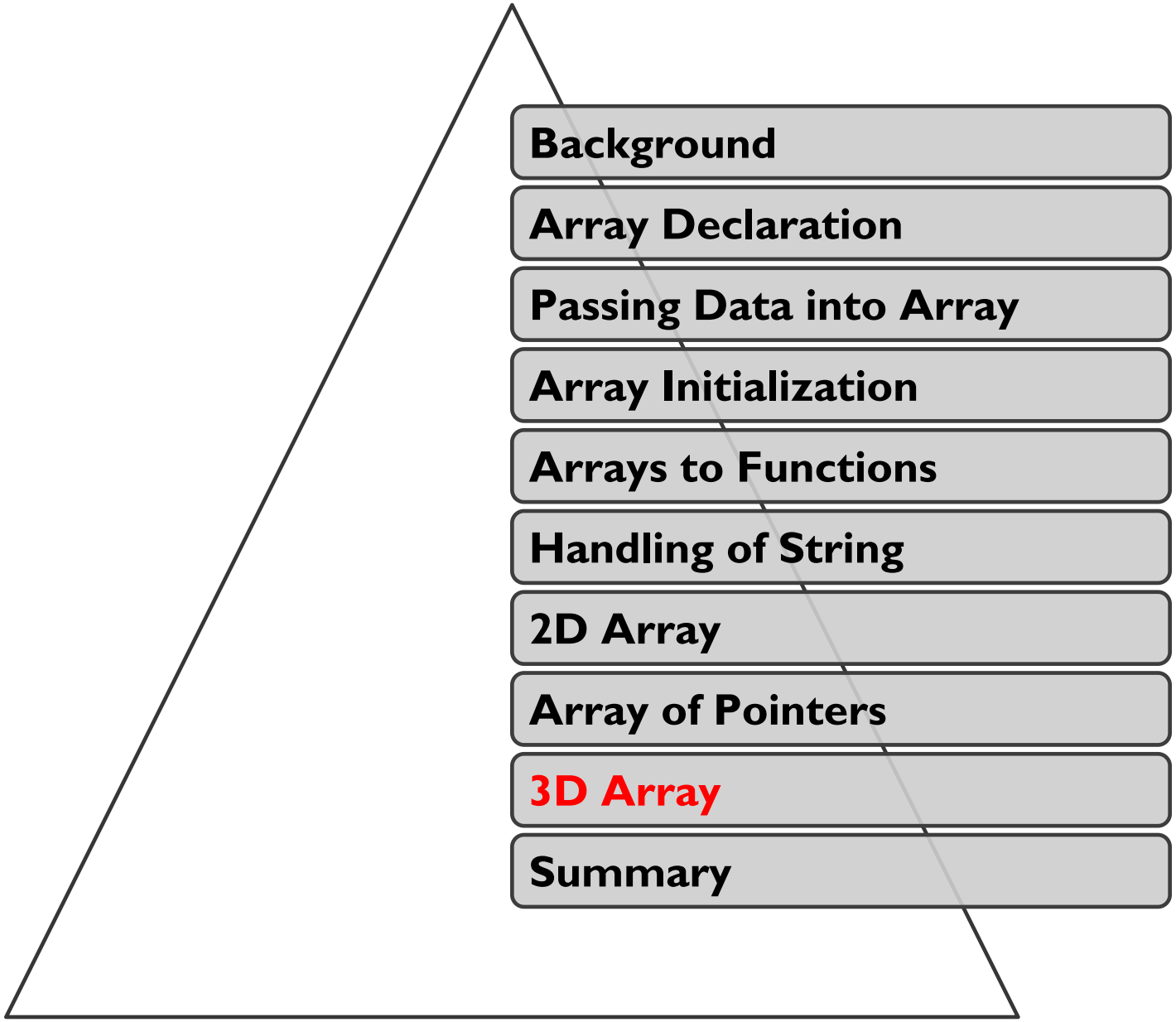
Array of Pointers

```
main( )
{
int *arr[4] ; /* array of integer pointers */
int i = 31, j = 5, k = 19, l = 71, m ;
arr[0] = &i ;
arr[1] = &j ;
arr[2] = &k ;
arr[3] = &l ;
for ( m = 0 ; m <= 3 ; m++ )
printf ( "%d ", * ( arr[m] ) ) ;
}
```



arr contains addresses of isolated **int** variables **i**, **j**, **k** and **l**.

The **for** loop in the program picks up the addresses present in **arr** and prints the values present at these addresses.


Background
Array Declaration
Passing Data into Array
Array Initialization
Arrays to Functions
Handling of String
2D Array
Array of Pointers
3D Array
Summary

Three-Dimensional Array

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

Three-Dimensional Array

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

- A three-dimensional array can be thought of as an array of arrays of arrays.
- The outer array has three elements, each of which is a two-dimensional array of four one-dimensional arrays, each of which contains two integers. In other words, a one-dimensional array of two elements is constructed first.
- Then four such one dimensional arrays are placed one below the other to give a two dimensional array containing four rows.
- Then, three such two dimensional arrays are placed one behind the other to yield a three dimensional array containing three 2-dimensional arrays.

Three-Dimensional Array

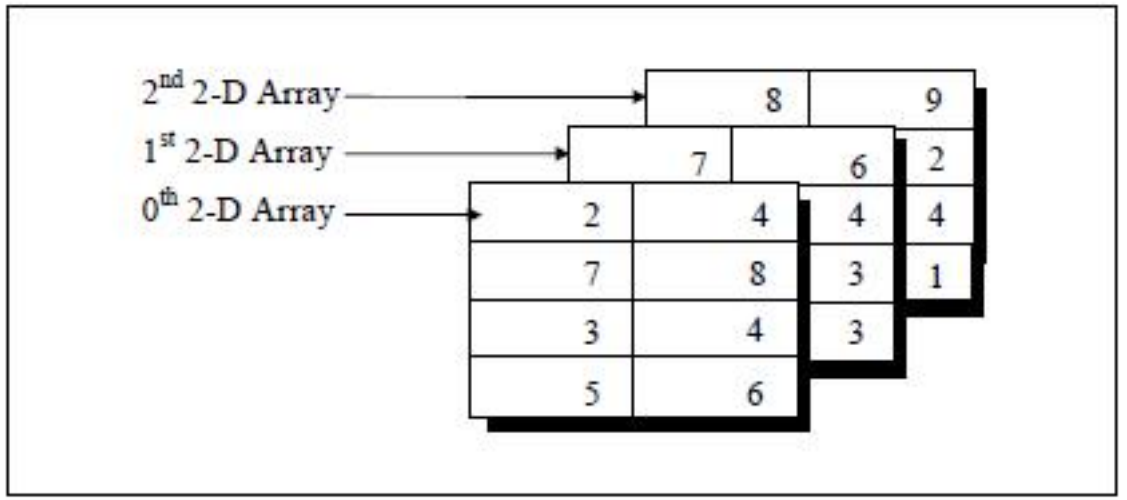
```
int arr[3][4][2] = {
                    {
                        {2, 4},
                        {7, 8},
                        {3, 4},
                        {5, 6}
                    },
                    {
                        {7, 6},
                        {3, 4},
                 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

- The first subscript should be [2], since the element is in third two dimensional array; the second subscript should be [3] since the element is in fourth row of the two-dimensional array; and the third subscript should be [1] since the element is in second position in the one-dimensional array.

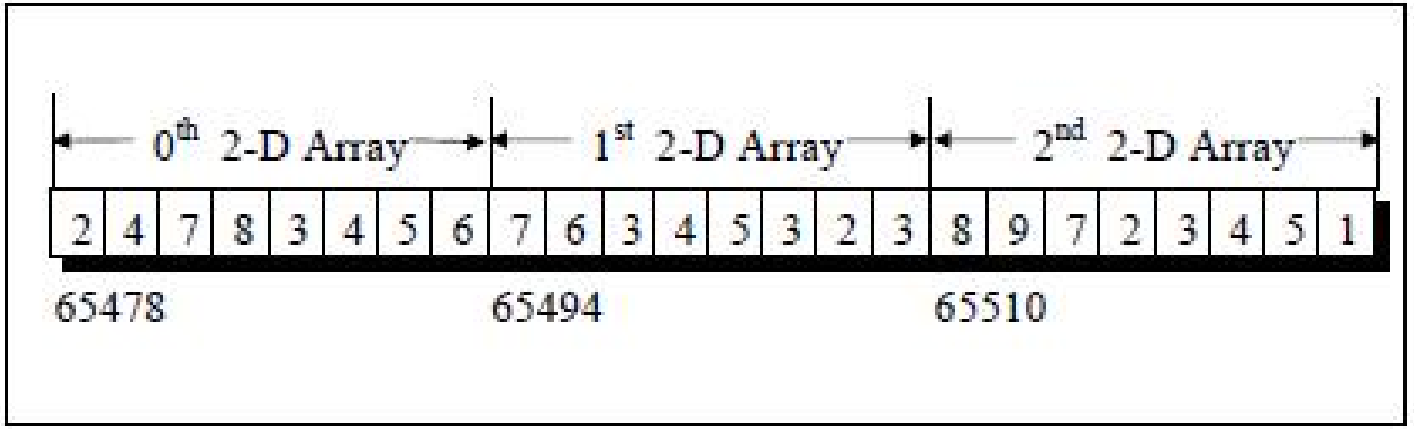
- We can therefore say that the element 1 can be referred as **arr[2][3][1]**. It may be noted here that the counting of array elements even for a 3-D array begins with zero. Can we not refer to this element using pointer notation?

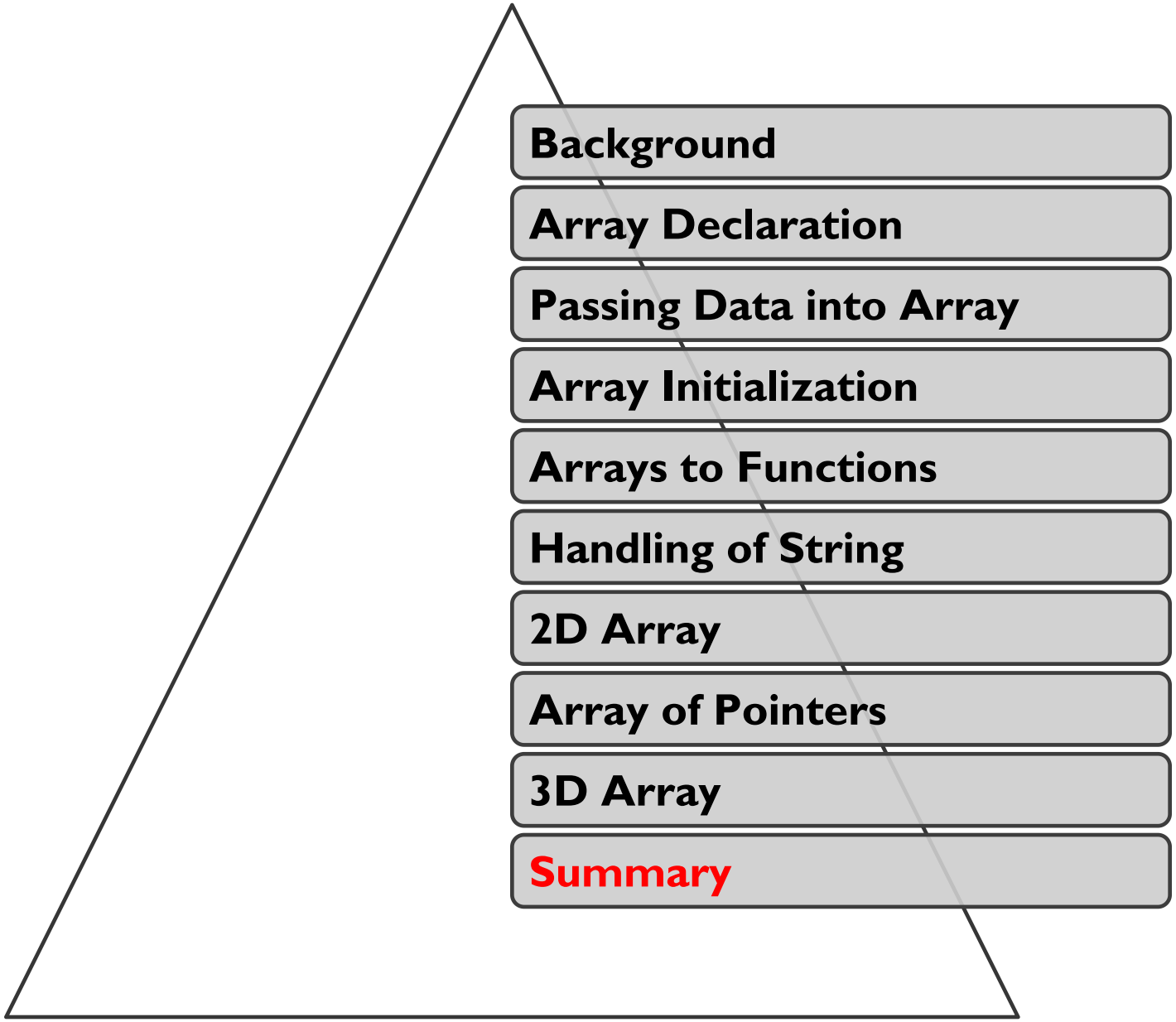
Three-Dimensional Array

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



Three-Dimensional Array




Background
Array Declaration
Passing Data into Array
Array Initialization
Arrays to Functions
Handling of String
2D Array
Array of Pointers
3D Array
Summary

Summary

- An array is similar to an ordinary variable except that it can store multiple elements of similar type.
- Compiler doesn't perform bounds checking on an array.
- The array variable acts as a pointer to the zeroth element of the array. In a 1-D array, zeroth element is a single value, whereas, in a 2-D array this element is a 1-D array.
- On incrementing a pointer it points to the next location of its type.
- Array elements are stored in contiguous memory locations and so they can be accessed using pointers.