

Data Structure and Algorithm (DSA)

Presented by: Er. Aruna Chhatkuli
Nepal College of Information Technology,
Balkumari, Lalitpur

Introduction to Data Structure and Algorithm:

Review of C and C++

Arrays

- Arrays are most frequently used in programming.
- Mathematical problems like matrix, algebra and etc, can be easily handled by arrays.
- An array is a collection of homogeneous data elements described by a single name.
- Each element of an array is referenced by a subscripted variable or value, called subscript or index enclosed in parenthesis.
- If an element of an array is referenced by single subscript, then the array is known as one dimensional array or linear array and if two subscripts are required to reference an element, the array is known as two dimensional array and so on.
- Analogously the arrays whose elements are referenced by two or more subscripts are called multi dimensional arrays.

Introduction to Data Structure and Algorithm:

One Dimensional Array

➤ One-dimensional array (or linear array) is a set of 'n' finite numbers of homogenous data elements such as:

- I. The elements of the array are referenced respectively by an index set consisting of 'n' consecutive numbers.
- II. The elements of the array are stored respectively in successive memory locations.

'n' number of elements is called the length or size of an array.

The elements of an array 'A' may be denoted in C as A [0], A [1], A [2],A[n-1]. The number 'n' in A [n] is called a subscript or an index and A[n] is called a subscripted variable. If 'n' is 10, then the array elements A [0], A [1].....A[9] are stored in sequential memory locations as follows :

| | | | | | | | | | |
|------|------|------|------|------|------|------|------|------|------|
| A[0] | A[1] | A[2] | A[3] | A[4] | A[5] | A[6] | A[7] | A[8] | A[9] |
|------|------|------|------|------|------|------|------|------|------|

Introduction to Data Structure and Algorithm:

- In C, array can always be read or written through loop.
- To read a one-dimensional array, it requires one loop for reading and writing the array,
- For example:
- For reading an array of 'n' elements
for (i = 0; i < n; i ++)
scanf ("%d", &a[i]);

For writing an array

```
for (i = 0; i < n; i ++)  
printf ("%d", a[i]);
```

Insertion in One-Dimensional Array

Insertion of a new element in an array can be done in two ways:

1. Insertion at the end of an array

- Providing the memory space allocated for the array enough to accommodate the additional element can easily do insertion at the end of an array.

2. Insertion at the required position

- For inserting the element at required position, element must be moved downwards to new locations.
- To accommodate the new element and keep the order of the elements. For inserting an element into a linear array `insert(a, len, pos, num)` where `a` is a linear array, `len` be total numbers of elements with an array, `pos` is the position at which number `num` will be inserted.

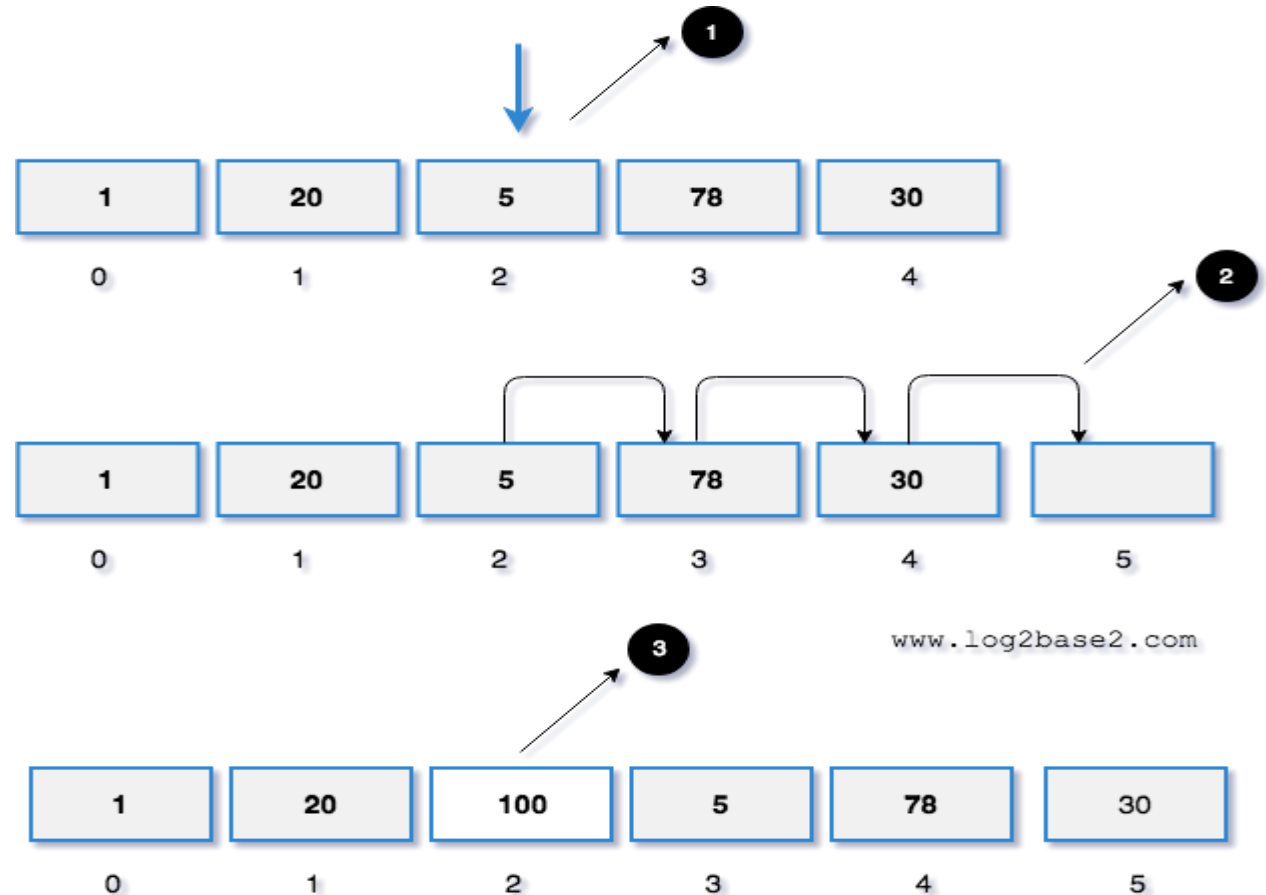
Insertion in One-Dimensional Array

ALGORITHM

- i. Get the **element value** which needs to be inserted.
- ii. Get the **position** value.
- iii. Check whether the position value is valid or not.
- iv. If it is **valid**,
 - a. Shift all the elements from the last index to position index by 1 position to the **right**.
 - b. insert the new element in **arr[position]**
- v. Otherwise,
 - a. Invalid Position

Visual Representation

- Let's take an array of 5 integers.
- 1, 20, 5, 78, 30.
- If we need to insert an element 100 at position 2, the execution will be,



Multi Dimensional Array

- If we are reading or writing two-dimensional array, two loops are required. Similarly the array of 'n' dimensions would require 'n' loops.
- The structure of the two dimensional array is illustrated in the following figure: `int A[3][3];`

| | | |
|---------|---------|---------|
| A[0][0] | A[0][1] | A[0][2] |
| A[1][0] | A[1][1] | A[1][2] |
| A[2][0] | A[2][1] | A[2][2] |

For reading an array of 'r' rows and 'c'

```
for (i = 0; i < r; i ++)  
    for( j = 0; j < c; j++)  
        scanf ("%d", &a[i][j]);
```

For writing an array

```
for (i = 0; i < r; i ++)  
    for( j = 0; j < c; j++)  
        printf ("%d", a[i][j]);
```


Sparse Array

- A sparse array is an array where nearly all of the elements have the same value (usually zero) and this value is a constant.
- One-dimensional sparse array is called sparse vectors and two-dimensional sparse arrays are called sparse matrix. Following is an example of sparse matrix where only 7 elements are nonzero among 35 elements where 28 elements are zeros.

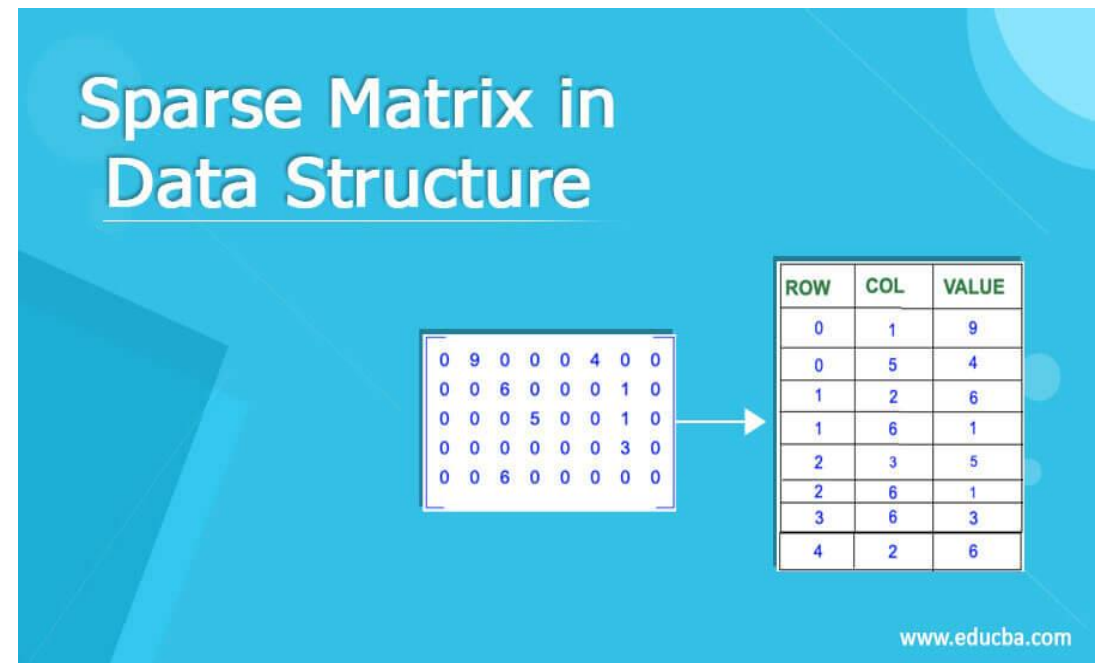
0 0 8 0 0 0 0

0 1 0 0 0 9 0

0 0 0 3 0 0 0

0 3 0 0 0 4 0

0 0 0 0 7 0 0



Structure

- Structures (also called structs) are a way to group several related variables into one place. Each variable in the structure is known as a **member** of the structure.
- Unlike an [array](#), a structure can contain many different data types (int, float, char, etc.).

Create a Structure

```
struct MyStructure {           // Structure declaration
    int myNum;                 // Member (int variable)
    char myLetter;             // Member (char variable)
};                             // End the structure
with a semicolon
```

Structure

To access the structure, you must create a variable of it.

- Use the **struct** keyword inside the **main()** method, followed by the name of the structure and then the name of the structure variable:

Create a struct variable with the name "s1":

```
struct myStructure {  
    int myNum;  
    char myLetter;  
};  
  
int main() {  
    struct myStructure s1;  
    return 0;  
}
```

Structure

Access Structure Members

To access members of a structure, use the dot syntax (.):

```
// Create a structure called myStructure
```

```
struct myStructure {  
    int myNum;  
    char myLetter;  
};
```

```
int main() {
```

```
    // Create a structure variable of myStructure called s1
```

```
    struct myStructure s1;
```

```
    // Assign values to members of s1
```

```
    s1.myNum = 13;
```

```
    s1.myLetter = 'B';
```

```
    // Print values
```

```
    printf("My number: %d\n", s1.myNum);
```

```
    printf("My letter: %c\n", s1.myLetter);
```

```
    return 0;
```

```
}
```

Union

- A **union** is a special data type available in C that allows to store different data types in the same memory location.
- We can define a union with many members, but only one member can contain a value at any given time.
- Unions provide an efficient way of using the same memory location for multiple-purpose.
- Example:

```
union Data {  
    int i;  
    float f;  
    char str[20];  
} data;
```

Union

```
#include <stdio.h>
#include <string.h>
union Data {
    int i;
    float f;
    char str[20]; };
int main( ) {
    union Data data;
    data.i = 10;
    data.f = 220.5;
    strcpy( data.str, "C Programming");
    printf( "data.i : %d\n", data.i);
```

```
printf( "data.f : %f\n", data.f);
printf( "data.str : %s\n", data.str);
return 0;
}
```

Result will be:

data.i : 1917853763

data.f:4122360580327794860452759994
368.000000

data.str : C Programming

Union

```
#include <stdio.h>
#include <string.h>
union Data {
    int i;
    float f;
    char str[20]; };
int main( ) {
    union Data data;
    data.i = 10;
    printf( "data.i : %d\n", data.i);
    data.f = 220.5;
    printf( "data.f : %f\n", data.f);
```

```
strcpy( data.str, "C Programming");
printf( "data.str : %s\n", data.str);
return 0; }
```

Result will be:

data.i : 10

data.f : 220.500000

data.str : C Programming

Pointers

- A **pointer** is a variable that **stores** the **memory address** of another variable as its value.
- A **pointer variable points** to a **data type** (like `int`) of the same type, and is created with the `*` operator.

Example

```
• int myAge = 43; // An int variable  
int* ptr = &myAge; // A pointer variable, with the name  
ptr, that stores the address of myAge
```

```
// Output the value of myAge (43)  
printf("%d\n", myAge);
```

```
// Output the memory address of myAge (0x7ffe5367e044)  
printf("%p\n", &myAge);
```

```
// Output the memory address of myAge with the pointer  
(0x7ffe5367e044)  
printf("%p\n", ptr);
```


Memory Allocation in C

There are two types of memory allocations in C.

1. Static memory allocations or Compile time
 2. Dynamic memory allocations or Run time
- In *Static Memory Allocation*, memory is allocated at compile time, that can't be modified while executing program and is generally used in array.
 - In *Dynamic Memory Allocation*, memory is allocated at run time, that can be modified while executing program and is generally used in linked list.

Memory Allocation in C

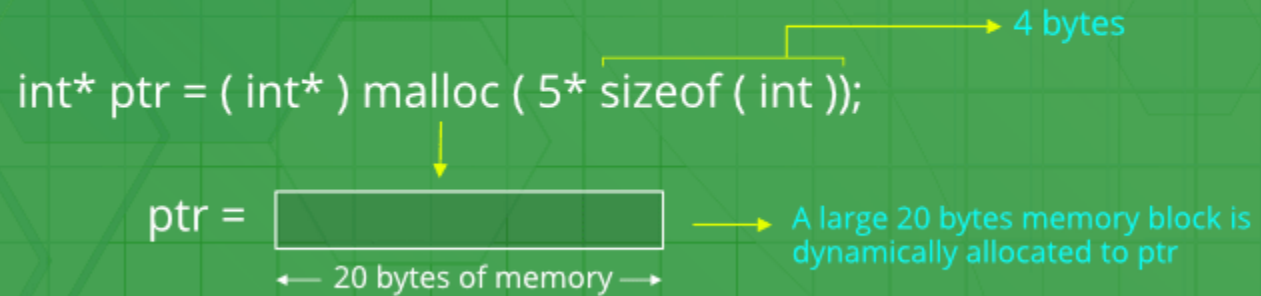
- **Methods used for Dynamic memory allocation:**

| Method | Syntax | Uses |
|-----------|--|---|
| malloc() | $P = (\text{cast_type}^*)$ malloc(byte_size) | To allocate a single block of requested memory. |
| calloc() | $P = (\text{cast_type}^*)$ calloc(number, byte_size) | To allocate multiple block of requested memory. |
| realloc() | $P = \text{realloc}(P, \text{new_size})$ | To allocate the memory occupied by malloc() or calloc() function. |
| free() | free(P) | To free the dynamically allocated memory. |

Memory Allocation in C

Malloc()

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



ptr =



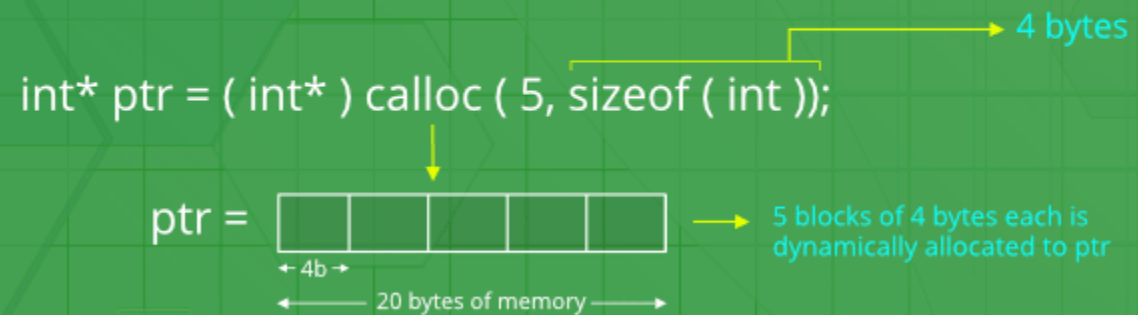
← 20 bytes of memory →

A large 20 bytes memory block is dynamically allocated to ptr



Calloc()

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



ptr =



← 4b →

← 20 bytes of memory →

5 blocks of 4 bytes each is dynamically allocated to ptr



Memory Allocation in C

Free()

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

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



Memory Allocation in C

Realloc()

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

4 bytes

ptr =



← 20 bytes of memory →

A large 20 bytes memory block is dynamically allocated to ptr

```
ptr = realloc ( ptr, 10* sizeof( int ));
```

ptr =



← 40 bytes of memory →

The size of ptr is changed from 20 bytes to 40 bytes dynamically



Memory Allocation in C

Static memory allocation has following drawbacks.

- If you try to read 15 elements of an array whose size is declared as 10, then first 10 values and other five consecutive unknown random memory values will be read.
- Again if you try to assign values to 15 elements of an array whose size is declared as 10, then first 10 elements can be assigned and the other 5 elements cannot be assigned/accessed.
- The second problem with static memory allocation is that if you store less number of elements than the number of elements for which you have declared memory, and then the rest of the memory will be wasted.

Memory Allocation in C

Note:

The memory allocated using malloc() function contains garbage values, the memory allocated by calloc() function contains the value zero.

Class:

- C++ is an object-oriented programming language.
- In object-oriented programming, a class is a template definition of the methods and variables in a particular kind of object.
- Thus, an object is a specific instance of a class; it contains real values instead of variables. The class is one of the defining ideas of object-oriented programming.
- Everything in C++ is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an **object**. The car has **attributes**, such as weight and color, and **methods**, such as drive and brake.
- Attributes and methods are basically **variables** and **functions** that belongs to the class. These are often referred to as "class members".
- A class is a user-defined data type that we can use in our program, and it works as an object constructor, or a "blueprint" for creating objects.

Class:

Example:

Create a class called "MyClass":

```
class MyClass {           // The class
    public:                // Access specifier
    int myNum;             // Attribute (int variable)
    string myString;       // Attribute (string variable)
};
```

Example explained:

- The **class** keyword is used to create a class called **MyClass**.
- The **public** keyword is an **access specifier**, which specifies that members (attributes and methods) of the class are accessible from outside the class. You will learn more about [access specifiers](#) later.
- Inside the class, there is an integer variable **myNum** and a string variable **myString**. When variables are declared within a class, they are called **attributes**.
- At last, end the class definition with a semicolon **;**.

Class:

Create an Object:

- In C++, an object is created from a class. We have already created the class named `MyClass`, so now we can use this to create objects.
- To create an object of `MyClass`, specify the class name, followed by the object name.
- To access the class attributes (`myNum` and `myString`), use the dot syntax (`.`) on the object:

Class:

Example:

Create an object called "myObj" and access the attributes:

```
class MyClass {           // The class
    public:                // Access specifier
        int myNum;         // Attribute (int variable)
        string myString;   // Attribute (string variable)
};

int main() {
    MyClass myObj;        // Create an object of MyClass

    // Access attributes and set values
    myObj.myNum = 15;
    myObj.myString = "Some text";

    // Print attribute values
    cout << myObj.myNum << "\n";
    cout << myObj.myString;
    return 0;
}
```

Class:

- **Class member functions:** A member function of a class is a function that has its definition or its prototype within the class definition like any other variable.
- **Class access modifiers:** A class member can be defined as public, private or protected. By default members would be assumed as private.
- **Constructor & destructor:** A class constructor is a special function in a class that is called when a new object of the class is created. A destructor is also a special function which is called when created object is deleted.
- **Pointer to C++ classes:** A pointer to a class is done exactly the same way a pointer to a structure is. In fact a class is really just a structure with functions in it.

Thank you!!!!