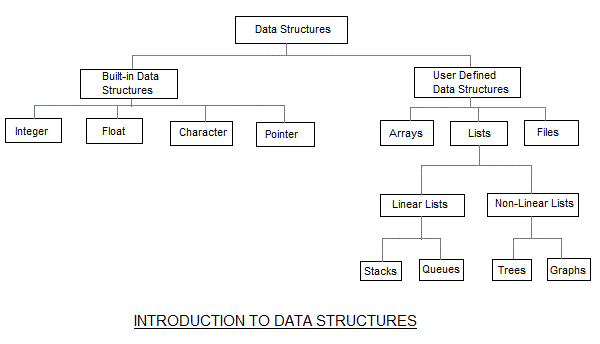
Chapter 1

Data Structure is a way of collecting and organising data in such a way that we can perform operations on these data in an effective way. Data Structures is about rendering data elements in terms of some relationship, for better organization and storage. For example, we have data player's name "Virat" and age 26. Here "Virat" is of **String** data type and 26 is of **integer** data type.

We can organize this data as a record like **Player** record. Now we can collect and store player's records in a file or database as a data structure. For example: "Dhoni" 30, "Gambhir" 31, "Sehwag" 33

In simple language, Data Structures are structures programmed to store ordered data, so that various operations can be performed on it easily.



**Basic types of Data Structures**

As we discussed above, anything that can store data can be called as a data strucure, hence Integer, Float, Boolean, Char etc, all are data structures. They are known as **Primitive Data Structures**.

Then we also have some complex Data Structures, which are used to store large and connected data. Some example of **Abstract Data Structure** are :

* Linked List
* Tree
* Graph
* Stack, Queue etc.

All these data structures allow us to perform different operations on data. We select these data structures based on which type of operation is required

#### Initialization of Pointer variable

**Pointer Initialization** is the process of assigning address of a variable to **pointer** variable. Pointer variable contains address of variable of same data type. In C language **address operator** & is used to determine the address of a variable. The & (immediately preceding a variable name) returns the address of the variable associated with it.

int a = 10 ;

int \*ptr ; *//pointer declaration*

ptr = &a ; *//pointer initialization*

or,

int \*ptr = &a ; *//initialization and declaration together*

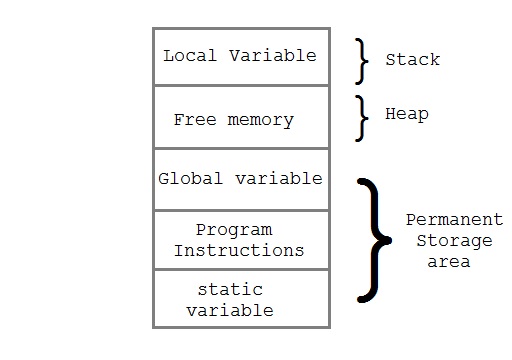
### Dynamic Memory Allocation

The process of allocating memory at runtime is known as **dynamic memory allocation**. Library routines known as "memory management functions" are used for allocating and freeing memory during execution of a program. These functions are defined in **stdlib.h**.

|  |  |
| --- | --- |
| **Function** | **Description** |
| malloc() | allocates requested size of bytes and returns a void pointer pointing to the first byte of the allocated space |
| calloc() | allocates space for an array of elements, initialize them to zero and then return a void pointer to the memory |
| free | releases previously allocated memory |
| realloc | modify the size of previously allocated space |

Memory Allocation Process

**Global** variables, **static** variables and program instructions get their memory in **permanent** storage area whereas **local** variables are stored in area called **Stack**. The memory space between these two region is known as **Heap** area. This region is used for dynamic memory allocation during execution of the program. The size of heap keep changing.



#### Allocating block of Memory

**malloc()** function is used for allocating block of memory at runtime. This function reserves a block of memory of given size and returns a pointer of type void. This means that we can assign it to any type of pointer using typecasting. If it fails to locate enough space it returns a NULL pointer.

**Example using malloc() :**

int \*x;

x = (int\*)malloc(50 \* sizeof(int)); *//memory space allocated to variable x*

free(x); *//releases the memory allocated to variable x*

**calloc()** is another memory allocation function that is used for allocating memory at runtime. **calloc** function is normally used for allocating memory to derived data types such as **arrays** and **structures**. If it fails to locate enough space it returns a NULL pointer.

**Example using calloc() :**

struct employee

{

char \*name;

int salary;

};

typedef struct employee emp;

emp \*e1;

e1 = (emp\*)calloc(30,sizeof(emp));

**realloc()** changes memory size that is already allocated to a variable.

**Example using realloc() :**

int \*x;

x=(int\*)malloc(50 \* sizeof(int));

x=(int\*)realloc(x,100); *//allocated a new memory to variable* ***x***

**What is Algorithm ?**

An algorithm is a finite set of instructions or logic, written in order, to accomplish a certain predefined task. Algorithm is not the complete code or program, it is just the core logic(solution) of a problem, which can be expressed either as an informal high level description as **pseudocode** or using a **flowchart**.

An algorithm is said to be efficient and fast, if it takes less time to execute and consumes less memory space. The performance of an algorithm is measured on the basis of following properties :

1. Time Complexity
2. Space Complexity

**Space Complexity**

Its the amount of memory space required by the algorithm, during the course of its execution. Space complexity must be taken seriously for multi-user systems and in situations where limited memory is available.

An algorithm generally requires space for following components :

* **Instruction Space :** Its the space required to store the executable version of the program. This space is fixed, but varies depending upon the number of lines of code in the program.
* **Data Space :** Its the space required to store all the constants and variables value.
* **Environment Space :** Its the space required to store the environment information needed to resume the suspended function.

## Time Complexity of Algorithms

Time complexity of an algorithm signifies the total time required by the program to run to completion. The time complexity of algorithms is most commonly expressed using the **big O notation**.

Time Complexity is most commonly estimated by counting the number of elementary functions performed by the algorithm. And since the algorithm's performance may vary with different types of input data, hence for an algorithm we usually use the **worst-case Time complexity** of an algorithm because that is the maximum time taken for any input size.

#### Calculating Time Complexity

Now lets tap onto the next big topic related to Time complexity, which is How to Calculate Time Complexity. It becomes very confusing some times, but we will try to explain it in the simplest way.

Now the most common metric for calculating time complexity is Big O notation. This removes all constant factors so that the running time can be estimated in relation to **N**, as N approaches infinity. In general you can think of it like this :

statement;

Above we have a single statement. Its Time Complexity will be **Constant**. The running time of the statement will not change in relation to N.

for(i=0; i < N; i++)

{

statement;

}

The time complexity for the above algorithm will be **Linear**. The running time of the loop is directly proportional to N. When N doubles, so does the running time.

for(i=0; i < N; i++)

{

for(j=0; j < N;j++)

{

statement;

}

}

This time, the time complexity for the above code will be **Quadratic**. The running time of the two loops is proportional to the square of N. When N doubles, the running time increases by N \* N.

while(low <= high)

{

mid = (low + high) / 2;

if (target < list[mid])

high = mid - 1;

else if (target > list[mid])

low = mid + 1;

else break;

}

This is an algorithm to break a set of numbers into halves, to search a particular field(we will study this in detail later). Now, this algorithm will have a **Logarithmic** Time Complexity. The running time of the algorithm is proportional to the number of times N can be divided by 2(N is high-low here). This is because the algorithm divides the working area in half with each iteration.

void quicksort(int list[], int left, int right)

{

int pivot = partition(list, left, right);

quicksort(list, left, pivot - 1);

quicksort(list, pivot + 1, right);

}

Taking the previous algorithm forward, above we have a small logic of Quick Sort(we will study this in detail later). Now in Quick Sort, we divide the list into halves every time, but we repeat the iteration N times(where N is the size of list). Hence time complexity will be **N\*log( N )**. The running time consists of N loops (iterative or recursive) that are logarithmic, thus the algorithm is a combination of linear and logarithmic.

**NOTE :** In general, doing something with every item in one dimension is linear, doing something with every item in two dimensions is quadratic, and dividing the working area in half is logarithmic.

#### Types of Notations for Time Complexity

Now we will discuss and understand the various notations used for Time Complexity.

1. **Big Oh** denotes "*fewer than or the same as*" <expression> iterations.
2. **Big Omega** denotes "*more than or the same as*" <expression> iterations.
3. **Big Theta** denotes "*the same as*" <expression> iterations.
4. **Little Oh** denotes "*fewer than*" <expression> iterations.
5. **Little Omega** denotes "*more than*" <expression> iterations.

#### Understanding Notations of Time Complexity with Example

**O(expression)** is the set of functions that grow slower than or at the same rate as expression.

**Omega(expression)** is the set of functions that grow faster than or at the same rate as expression.

**Theta(expression)** consist of all the functions that lie in both O(expression) and Omega(expression).

Suppose you've calculated that an algorithm takes f(n) operations, where,

f(n) = 3\*n^2 + 2\*n + 4. // n^2 means square of n

Since this polynomial grows at the same rate as **n^2**, then you could say that the function **f** lies in the set **Theta(n^2)**. (It also lies in the sets **O(n^2)** and **Omega(n^2)** for the same reason.)

The simplest explanation is, because **Theta** denotes *the same as* the expression. Hence, as **f(n)** grows by a factor of **n^2**, the time complexity can be best represented as **Theta(n^2)**.