DATA STRUCTURE AND ALGORITHM

UNIT: ONE

1. DATA TYPES, DATA STRUCTURE AND ABSTRACT DATE TYPE
2. DYNAMIC MEMORY ALLOCATION IN C
3. INTRODUCTION TO ALGORITHMS
4. ASYMPTOTIC NOTATIONS AND COMMON FUNCTIONS

Data Types

- **A data type** is an attribute of data which tells the compiler (or interpreter) how the programmer intends to use the data

or

- **A data-type** defines the type of value an object can have and what operations can be performed on it. A data type should be declared first before being used. Different programming languages support different data-types.
- For example,
  - ✓ C supports char, int, float, long, etc.
  - ✓ Python supports String, List, Tuple, etc.

    Primitive: basic building block (Boolean, integer, float, char etc.)

    Composite: any data type (struct, array, string etc.) composed of primitives or composite types.

    Abstract: data type that is defined by its behaviour (tuple, set, stack, queue, graph etc.)
- If we consider a composite type, such as a 'string', it describes a data structure which contains a sequence of char primitives (characters), and as such is referred to as being a 'composite' type.
- Whereas the underlying implementation of the string composite type is typically implemented using an array data structure

Data Types

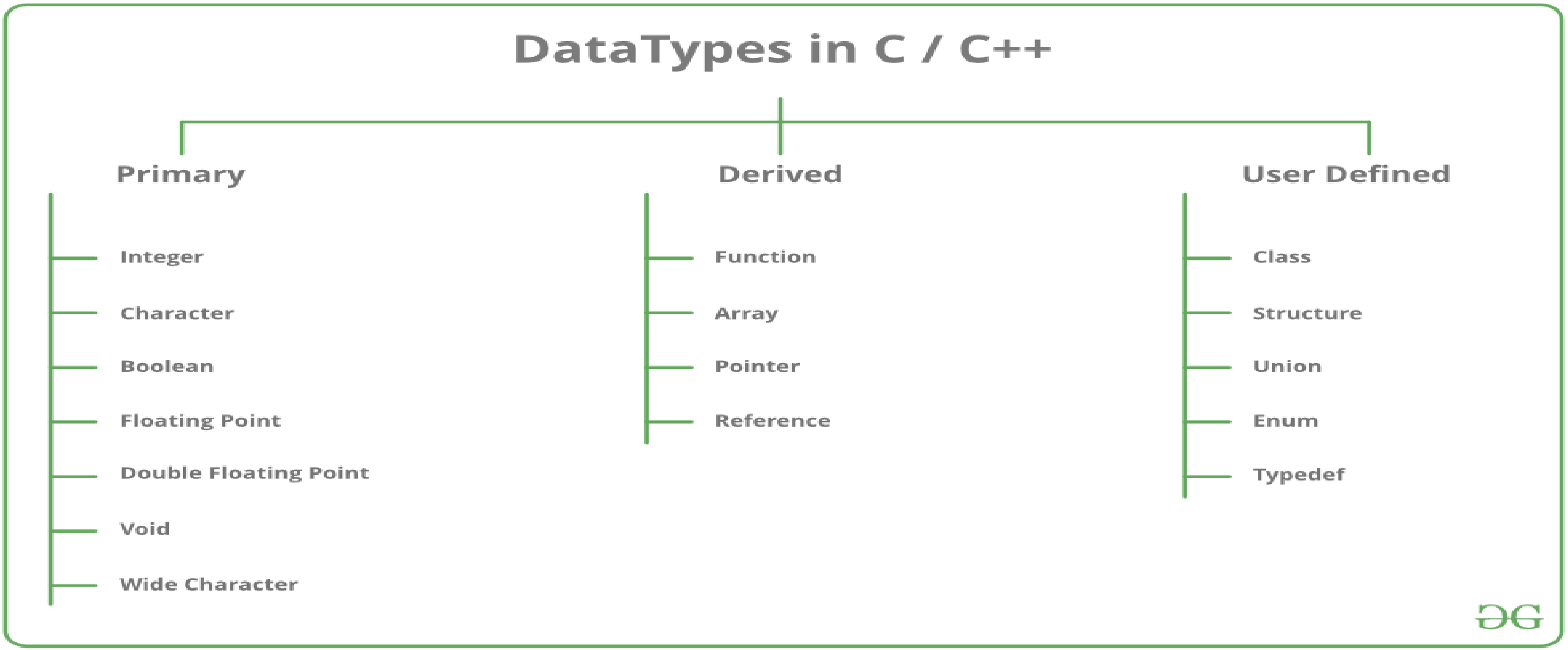
- In a broad sense, there are three types of data types –

  - ✓ **Fundamental(basics) data types** – These are the predefined data types which are used by the programmer directly to store only one value as per requirement, i.e., integer type, character type, or floating type. For example – int, char, float, etc.

  - ✓ **Derived data types** – These data types are derived using built-in data type which are designed by the programmer to store multiple values of same type as per their requirement. For example – Array, Pointer, function, list, etc.

  - ✓ **User-defined data types** – These data types are derived using built-in data types which are wrapped into a single a data type to store multiple values of either same type or different type or both as per the requirement. For example – Class, Structure, etc.

Data Types


DataTypes in C / C++
Primary
Integer
Character
Boolean
Floating Point
Double Floating Point
Void
Wide Character
Derived
Function
Array
Pointer
Reference
User Defined
Class
Structure
Union
Enum
Typedef

Data Structure

- ✓ Data structure is a way of organizing all data items and establishing relationship among those data items
- ✓ Data structures are the building blocks of a program.
- ✓ To develop a program of an algorithm we should select an appropriate data structure for that algorithm. Therefore, data structure is represented as:

  Algorithm + Data structure = Program
- ✓ A data structure should be seen as a logical concept that must address two fundamental concerns.
  - **First, how the data will be stored, and**
  - **Second, what operations will be performed on it.**
- ✓ Data structure mainly specifies the following four things:
  - **Organization of data.**
  - **Accessing methods**
  - **Degree of associativity**
  - **Processing alternatives for information**

Data Structure

Data structures are divided into two types:

- ✓ **Primitive Data Structures** are the basic data structures that directly operate upon the machine instructions.
- ✓ They have different representations on different computers.
- ✓ Integers, floating point numbers, character constants, string constants and pointers come under this category.

- ✓ **Non-primitive data structures** are more complicated data structures and are derived from primitive data structures.
- ✓ They emphasize on grouping same or different data items with relationship between each data item.
- ✓ Arrays, lists and files come under this category.

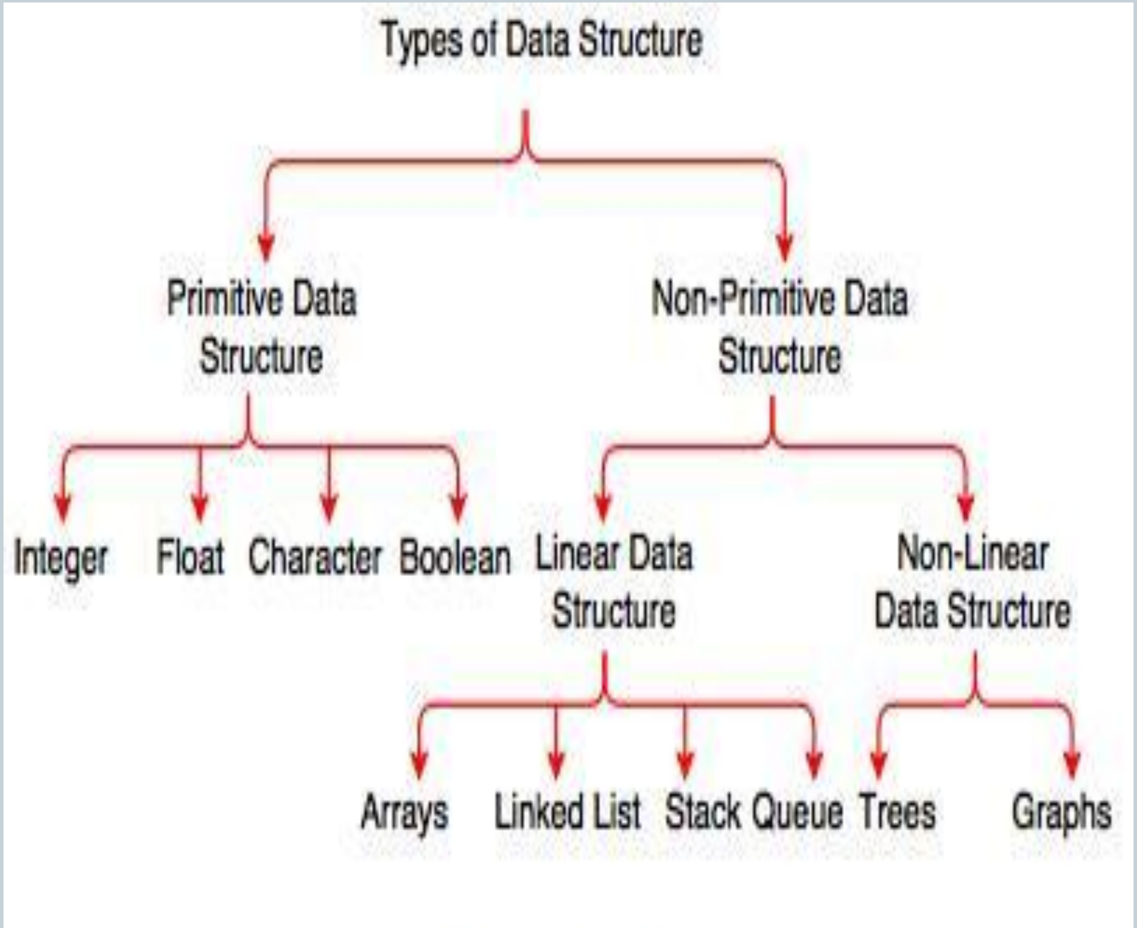


Fig. Types of Data Structure

Data Structure

- ✓ A **Linear data structure** have data elements arranged in sequential manner and each member element is connected to its previous and next element.
- ✓ There are two ways to represent a linear data structure in memory,
  - Static memory allocation
  - Dynamic memory allocation
- ✓ The possible operations on the linear data structure are Traversal, Insertion, Deletion, Searching, Sorting and Merging.
- ✓ Examples of Linear Data Structure are Stack and Queue
  - **Stack** is a data structure in which insertion and deletion operations are performed at one end only.
    - The insertion operation is referred to as **'PUSH'** and deletion operation is referred to as **'POP'** operation.
    - A stack is also called as Last in First out **(LIFO)** data structure.
  - **Queue** is a data structure which permits the insertion at one end and Deletion at another end.
    - The end at which deletion occurs is known as **FRONT** end and another end at which insertion occurs is known as a **REAR** end.
    - A queue is also called a First in First out **(FIFO)** data structure

Data Structure

- ✓ **Nonlinear data structures** are those data structures in which data items are not arranged in a sequence.
- ✓ Examples of Non-linear Data Structure are Tree and Graph.
  - **Tree** can be defined as a finite set of data items (nodes) in which data items are arranged in branches and sub-branches according to requirements.
    - Trees represent the hierarchical relationship between various elements
    - Tree consists of nodes connected by an edge, the node represented by a circle and edge lives connecting to the circle.
  - **Graph** is a collection of nodes (Information) and connecting edges (Logical relation) between nodes.

Abstract Data Types

- ✓ **An abstract data type** is defined by its behaviour (semantics) from the point of view of a user, of the data, specifically in terms of possible values, possible operations on data of this type, and the behaviour of these operations.
- ✓ An abstract data type is a data type whose representation is hidden from, and of no concern to the application code.
- ✓ For example, when writing application code, we don't care how strings are represented: we just declare variables of type String, and manipulate them by using string operations.

Abstract Data Types

- An **Abstract Data Type (ADT)** is an abstract concept defined by axioms that represent some data and operations on that data.
  - Common **ADT** Examples
    - List
    - Stack
    - Queue
- For example, integers are an ADT, defined as the values ..., −2, −1, 0, 1, 2, ..., and by the operations of addition, subtraction, multiplication, and division, together with greater than, less than, etc.,

Dynamic Memory Allocation in C

- ✓ An array is a collection of a fixed number of values. Once the size of an array is declared, you cannot change it.
- ✓ Sometimes the size of the array you declared may be insufficient.
- ✓ To solve this issue, you can allocate memory manually during run-time. This is known as dynamic memory allocation in C programming.
- ✓ To allocate memory dynamically, library functions are:
  - ✓ **malloc()**
  - ✓ **calloc()**
  - ✓ **realloc()**
  - ✓ **free()**
- ✓ These functions are defined in the **<stdlib.h>** header file

Dynamic Memory Allocation in C

C malloc()

- ✓ The name "malloc" stands for memory allocation.
- ✓ The malloc() function reserves a block of memory of the specified number of bytes. And, it returns a pointer of void which can be casted into pointers of any form.
- ✓ Syntax of malloc()

  ptr = (castType*) malloc(size);
- ✓ Example

  ptr = (float*) malloc(100 * sizeof(float));
- ✓ The above statement allocates 400 bytes of memory. It's because the size of float is 4 bytes. And, the pointer ptr holds the address of the first byte in the allocated memory
- ✓ The expression results in a NULL pointer if the memory cannot be allocated.

Dynamic Memory Allocation in C

C calloc()

- ✓ The name "calloc" stands for contiguous allocation.
- ✓ The malloc() function allocates memory and leaves the memory uninitialized, whereas the calloc() function allocates memory and initializes all bits to zero.
- ✓ Syntax of calloc()

```
ptr = (castType*)calloc(n, size);
```

- ✓ Example:

```
ptr = (float*) calloc(25, sizeof(float));
```

- ✓ The above statement allocates contiguous space in memory for 25 elements of type float

Dynamic Memory Allocation in C

C free()

- ✓ Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on their own. You must explicitly use free() to release the space.
- ✓ Syntax of free()

  free(ptr);
- ✓ This statement frees the space allocated in the memory pointed by ptr.

Dynamic Memory Allocation in C

C realloc()

- ✓ If the dynamically allocated memory is insufficient or more than required, you can change the size of previously allocated memory using the realloc() function.
- ✓ Syntax of realloc()

```
ptr = realloc(ptr, x);
```

Here, ptr is reallocated with a new size x.

Example 1: malloc() and free()

// Program to calculate the sum of n numbers entered by the user

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
   int n, i, *ptr, sum = 0;

   printf("Enter number of elements: ");
   scanf("%d", &n);

   ptr = (int*) malloc(n * sizeof(int));

   // if memory cannot be allocated
   if(ptr == NULL)
   {
      printf("Error! memory not allocated.");
      exit(0);
   }
   printf("Enter elements: ");
   for(i = 0; i < n; ++i)
   {
      scanf("%d", ptr + i);
      sum += *(ptr + i);
   }

   printf("Sum = %d", sum);

   // deallocating the memory
   free(ptr);

   return 0;
}
```

Here, we have dynamically allocated the memory for n number of int

Example 2: calloc() and free()

// Program to calculate the sum of n numbers entered by the user

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int n, i, *ptr, sum = 0;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    ptr = (int*) calloc(n, sizeof(int));
    if(ptr == NULL)
    {
        printf("Error! memory not allocated.");
        exit(0);
    }

    printf("Enter elements: ");
    for(i = 0; i < n; ++i)
    {
        scanf("%d", ptr + i);
        sum += *(ptr + i);
    }

    printf("Sum = %d", sum);
    free(ptr);
    return 0;
}
```

Example 3: realloc()

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
   int *ptr, i , n1, n2;
   printf("Enter size: ");
   scanf("%d", &n1);
   ptr = (int*) malloc(n1 * sizeof(int));
   printf("Addresses of previously allocated
memory: ");
   for(i = 0; i < n1; ++i)
       printf("%u\n",ptr + i);
   printf("\nEnter the new size: ");
   scanf("%d", &n2);
   // rellocating the memory
   ptr = realloc(ptr, n2 * sizeof(int));
   printf("Addresses of newly allocated
memory: ");
   for(i = 0; i < n2; ++i)
       printf("%u\n", ptr + i);
   free(ptr);
   return 0;
  }
```

Algorithm

- ✓ An **algorithm** is a precise specification of a sequence of instructions to be carried out in order to solve a given problem. Each instruction tells what task is to be done.
- ✓ There should be a finite number of instructions in an algorithm and each instruction should be executed in a finite amount of time.
- ✓ Properties of Algorithms:
  - **Input:** A number of quantities are provided to an algorithm initially before the algorithm begins. These quantities are inputs which are processed by the algorithm.
  - **Definiteness**: Each step must be clear and unambiguous.
  - **Effectiveness**: Each step must be carried out in finite time.
  - **Finiteness**: Algorithms must terminate after finite time or step
  - **Output**: An algorithm must have output
  - **Correctness:** Correct set of output values must be produced from the each set of inputs.

Algorithm

✓ Design an algorithm to add two numbers and display the result.

```
1. START
2. declare three integers a, b & c
3. define values of a & b
4. add values of a & b
5. store output of step 4 to c
6. print c
7. STOP
```

Algorithm Complexity

- ✓ Suppose X is treated as an algorithm and N is treated as the size of input data, the time and space implemented by the Algorithm X are the two main factors which determine the efficiency of X.

  - **Time Factor –** The time is calculated or measured by counting the number of key operations such as comparisons in sorting algorithm.
  - **Space Factor –** The space is calculated or measured by counting the maximum memory space required by the algorithm.

- ✓ The complexity of an algorithm f(N) provides the running time and / or storage space needed by the algorithm with respect of N as the size of input data

Algorithm Complexity

- ✓ **Space complexity** of an algorithm represents the amount of memory space needed the algorithm in its life cycle.
- ✓ Space needed by an algorithm is equal to the sum of the following two components
  - A fixed part that is a space required to store certain data and variables (i.e. simple variables and constants, program size etc.), that are not dependent of the size of the problem.
  - A variable part is a space required by variables, whose size is totally dependent on the size of the problem. For example, recursion stack space, dynamic memory allocation etc.
- ✓ Space complexity **S(p)** of any algorithm p is **S(p) = A + Sp(I)** Where **A** is treated as the fixed part and **S(I)** is treated as the variable part of the algorithm which depends on instance characteristic **I**. Following is a simple example that tries to explain the concept
- ✓ Algorithm

```
SUM(P, Q)
Step 1 - START
Step 2 - R ← P + Q + 10
Step 3 - Stop
```

  - Here we have three variables P, Q and R and one constant. Hence **S(p)** = 1+3. Now space is dependent on data types of given constant types and variables and it will be multiplied accordingly.

Algorithm Complexity

- **Time Complexity** of an algorithm is the representation of the amount of time required by the algorithm to execute to completion. Time requirements can be denoted or defined as a numerical function **t(N)**, where **t(N)** can be measured as the number of steps, provided each step takes constant time.
- Algorithm
    - SUM(P, Q)
    - Step 1 - START
    - Step 2 - R $\leftarrow$ P + Q + 10
    - Step 3 - Stop
- For example, in case of addition of two n-bit integers, **N** steps are taken. Consequently, the total computational time is **t(N) = c*n**, where **c** is the time consumed for addition of two bits. Here, we observe that **t(N)** grows linearly as input size increases.

Asymptotic Notation

- ✓ **Asymptotic analysis** of an algorithm refers to defining the mathematical boundation/framing of its run-time performance.
- ✓ Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm.
- ✓ Asymptotic analysis is input bound i.e., if there's no input to the algorithm, it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.
- ✓ Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation.
- ✓ Primarily used to describe running time of algorithm,
- ✓ Usually, the time required by an algorithm falls under three types –
  - **Best Case** – Minimum time required for program execution.
  - **Average Case** – Average time required for program execution.
  - **Worst Case** – Maximum time required for program execution

Asymptotic Notation

- Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.
  - O Notation
  - Ω Notation
  - θ Notation

Asymptotic Notation

Big Oh (O) notation:

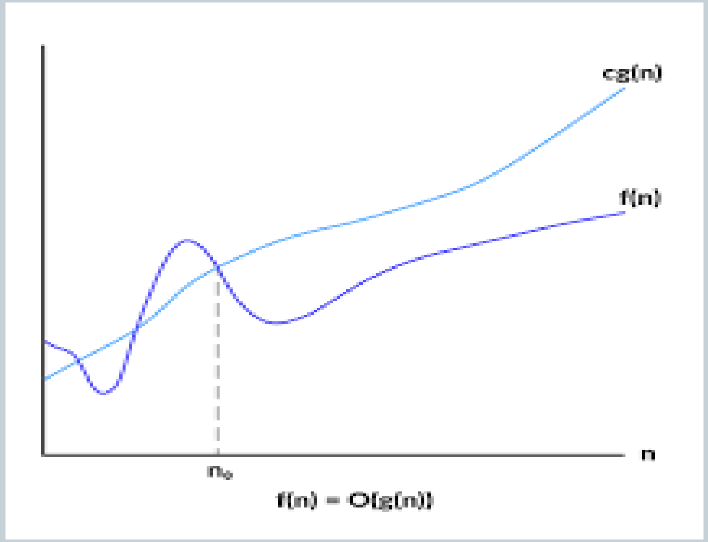
- When we have only asymptotic upper bound then we use **O** notation. A function **f(x)=O(g(x))** (read as **f(x)** is big oh of **g(x)**) iff there exists two positive constants c and x_o such that for all **$x >= x_o$, $f(x) <= c*g(x)$**
- The above relation says that **g(x)** is an upper bound of **f(x)**
- **O(1)** is used to denote constants.
- Example: $f(x)=5x^3+3x^2+4$ Find big **oh(O)** of **f(x)**
  - **solution:$f(x)= 5x^3+3x^2+4<= 5x^3+3x^3+4x^3$ if $x>0$**

    $<=12x^3$

    $=>f(x)<=c.g(x)$

    where c=12 and g(x)=x3

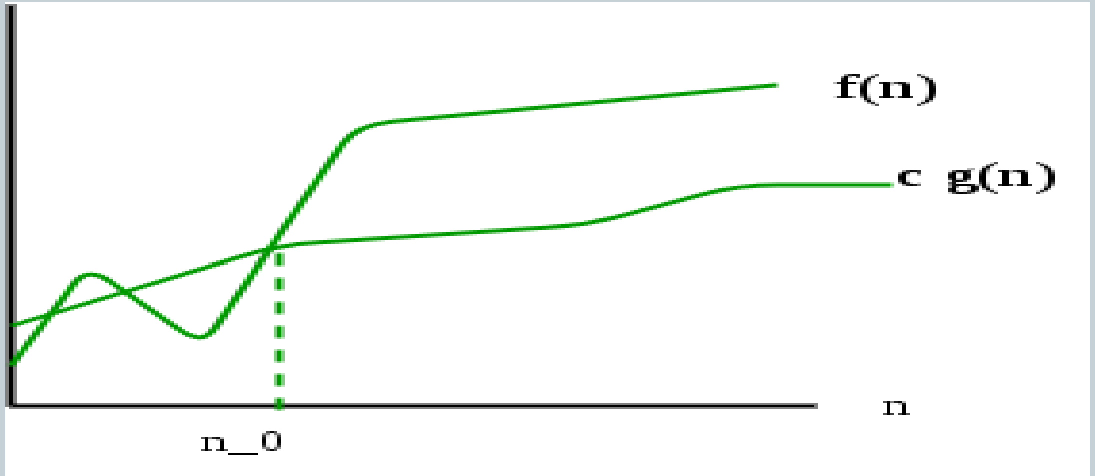
    Thus by definition of big oh O(f(x))=O(x3)



Asymptotic Notation

Big Omega (Ω) notation:

- ✓ Big omega notation gives asymptotic lower bound. A function **f(x) =Ω (g(x))** (read as **g(x)** is big omega of **g(x)**) iff there exists two positive constants c and $\mathbf{x_o}$ such that for all $\mathbf{x >= x_o}$, $\mathbf{0 <= c*g(x) <= f(x)}$.
- ✓ The above relation says that **g(x)** is a lower bound of **f(x)**.



f(n) = Omega(g(n))

Asymptotic Notation

Big Theta (Θ) notation:

- ✓ When we need asymptotically tight bound then we use notation. A function **f(x) = (g(x))** (read as **f(x)** is big theta of **g(x)**) iff there exists three positive constants **c_1, c_2** and **x_0** such that for all **$x >= x_0$, c_1*g(x) <= f(x) <= c_2*g(x)**
- ✓ The above relation says that **f(x)** is order of **g(x)**
- ✓ Example: **f(n)** = **$3n^2 + 4n + 7$, g(n) = n^2**, then prove that **f(n) = (g(n))**.
  - ❑ Proof: let us choose **c_1, c_2** and **n_0** values as **14, 1** and **1** respectively then we can have,

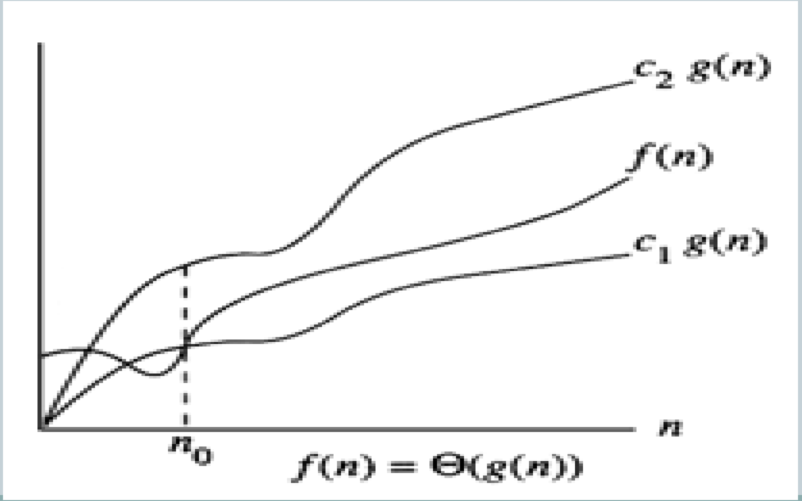
f(n) <= c_1*g(n), n>=n_0 as $3n^2 + 4n + 7 <= 14$*n^2 , and

f(n) >= c_2*g(n), n>=n_0 as 3n2 + 4n + 7 >= 1*n^2

for all n >= 1(in both cases).

So c_2*g(n) <= f(n) <= c_1*g(n) is trivial.

Hence f(n) = Q (g(n)).



Common Asymptotic Notations

✓ Following is a list of some common asymptotic notations

Name	Asymptotic Notations
Constant	$O(1)$
Logarithimic	$O(\log n)$
Linear	$O(n)$
n log n	$O(n \log n)$
Quadratic	$O(n^2)$
Cubic	$O(n^3)$
Polynomial	$n^{O(1)}$
Expotential	$2^{O(n)}$