Pre-requisite based Study Material

for

Data and File Structures

(MCA-102)

by

Dr. Sunil Pratap Singh

(Assistant Professor, BVICAM, New Delhi)

April, 2021

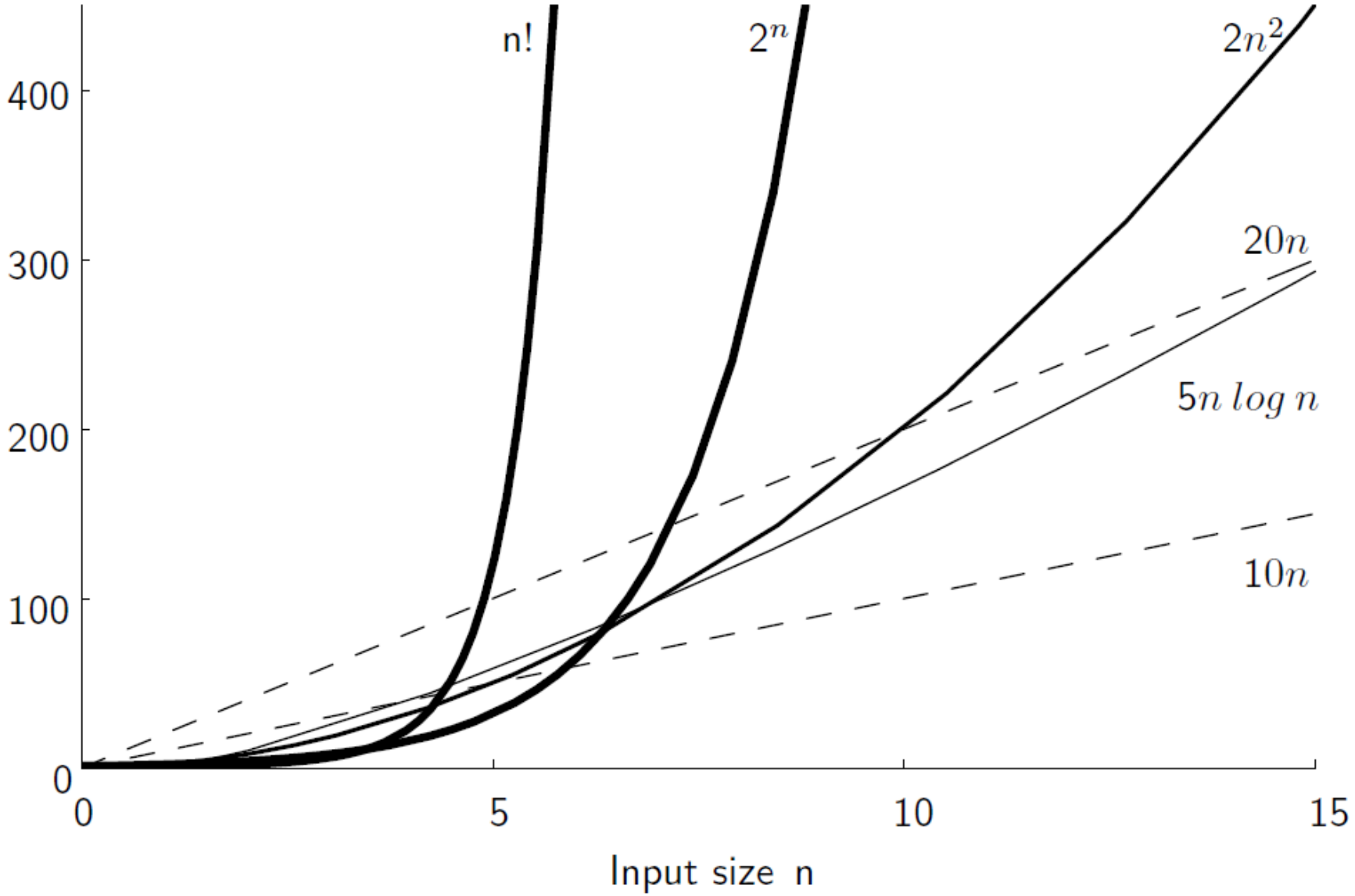
Algorithm

- Algorithm: An algorithm is a finite set of instructions which, if followed, accomplish a particular task. Every algorithm should have following properties:
  - It must be **correct**. In other words, it must compute the desired function, converting each input to the correct output.
  - It is composed of a series of **concrete steps**. Concrete means that the action described by that step is completely understood — and doable — by the person or machine that must perform the algorithm.
  - There can be **no ambiguity** as to which step will be performed next.
  - It must be composed of a **finite** number of steps.
  - It must **terminate**. In other words, it may not go into an infinite loop.

Analysis of Algorithms

- "Analysis of Algorithms" is concerned primarily with determining the memory (space) and time requirements (complexity) of an algorithm.
- The most important factor affecting running time is normally size of the input.
- For a given input size n, we often express the time T to run the algorithm as a function of n, written as T(n).
- It is always assumed that T(n) is a non-negative value.
- "Growth Rate" for an algorithm is the rate at which the cost of the algorithm grows as the size of its input grows.
- T(n) describes the growth rate for the running time of the algorithm.

Growth Rate of Algorithms



Costs for Growth Rates

n	$\log\log n$	$\log n$	n	$n \log n$	n^2	n^3	2^n
16	2	4	2^4	$2 \cdot 2^4 = 2^5$	2^8	2^{12}	2^{16}
256	3	8	2^8	$8 \cdot 2^8 = 2^{11}$	2^{16}	2^{24}	2^{256}
1024	≈ 3.3	10	2^{10}	$10 \cdot 2^{10} \approx 2^{13}$	2^{20}	2^{30}	2^{1024}
64K	4	16	2^{16}	$16 \cdot 2^{16} = 2^{20}$	2^{32}	2^{48}	$2^{64\mathrm{K}}$
1M	≈ 4.3	20	2^{20}	$20 \cdot 2^{20} \approx 2^{24}$	2^{40}	2^{60}	$2^{1\mathrm{M}}$
1G	≈ 4.9	30	2^{30}	$30 \cdot 2^{30} \approx 2^{35}$	2^{60}	2^{90}	$2^{1\mathrm{G}}$

Common Growth Rate Functions

- 1 (constant): growth is independent of the problem size n.
- Log n (logarithmic): growth increases slowly compared to the problem size (binary search)
- n (linear): directly proportional to the size of the problem.
- n * log n (n log n): typical of some divide and conquer approaches
- n^2 (quadratic): typical in nested loops
- n^3 (cubic): more nested loops
- 2^n (exponential): growth is extremely rapid

$$1 < \log n < n < n \log n < n^2 < n^3 < 2^n < n!$$

Best, Worst, and Average Cases

- Consider an example of Sequential Search algorithm:
  - Algorithm begins at the first position in the array and looks at each value in turn until K (searched item) is found.
  - Once K is found, the algorithm stops.
  - There is a wide range of possible running times for the sequential search algorithm.
  - The first integer in the array could have value K, and so only one integer is examined. In this case the running time is short. This is the best case for this algorithm.
  - Alternatively, if the last position in the array contains K, then the running time is relatively long, because the algorithm must examine n values. This is the worst case for this algorithm, because sequential search never looks at more than n values.

Best, Worst, and Average Cases

- If we implement sequential search as a program and run it many times on many different arrays of size n, or search for many different values of K within the same array, we expect the algorithm on average to go halfway through the array before finding the value we seek.
- On average, the algorithm examines about $\frac{n}{2}$ values. We call this the average case for this algorithm.

Best, Worst, and Average Cases

- Should we study the best case?
  - Normally we are not interested in the best case, because this might happen only rarely and generally is too optimistic for a fair characterization of the algorithm's running time.
  - In other words, analysis based on the best case is not likely to be representative of the behavior of the algorithm.
  - However, there are rare instances where a best-case analysis is useful — in particular, when the best case has high probability of occurring.

Best, Worst, and Average Cases

- Should we study the worst case?
  - The advantage to analyzing the worst case is that we know for certain that the algorithm must perform at least that well.
  - This is especially important for real-time applications, such as for the computers that monitor an air traffic control system.
  - Here, it would not be acceptable to use an algorithm that can handle n airplanes quickly enough most of the time, but which fails to perform quickly enough when all n airplanes are coming from the same direction.
  - For other applications — particularly when we wish to aggregate the cost of running the program many times on many different inputs — worst-case analysis might not be a representative measure of the algorithm's performance.

Best, Worst, and Average Cases

- Should we study the average case?
  - Often we prefer to know the average-case running time.
  - This means that we would like to know the typical behavior of the algorithm on inputs of size n.
  - Unfortunately, average-case analysis is not always possible.
  - Average-case analysis first requires that we understand how the actual inputs to the program are distributed with respect to the set of all possible inputs to the program.
  - For example, it was stated previously that the sequential search algorithm on average examines half of the array values. This is only true if the element with value K is equally likely to appear in any position in the array. If this assumption is not correct, then the algorithm does not necessarily examine half of the array values in the average case.

Best, Worst, and Average Cases

- Summary
  - For real-time applications we are likely to prefer a worst-case analysis of an algorithm.
  - Otherwise, we often desire an average-case analysis if we know enough about the distribution of our input to compute the average case. If not, then we must resort to worst-case analysis.

Methods of Estimating/Representing Time Complexity

- Operation Counts
- Step Counts
- Counting Cache Misses
- Asymptotic Notations
- Recurrence Equations
- … … …

Step Counts

- To determine the step count of an algorithm,
  - determine the number of steps per execution (s/e) of each statement
  - determine total number of times (i.e., frequency) each statement is executed
  - combine s/e and frequency to give the total contribution of each statement to the total step count
  - add the contributions of all statements to obtain the step count for the entire algorithm

Step Counts: Example

Algorithm Sum (a,n)	s/e	Frequency	Total Steps
{	0	0	0
`sum = 0;`	1	1	1
`for(i=1; i<=n; i++)`	1	n+1	n+1
`sum = sum + a[i];`	1	n	n
`return sum;`	1	1	1
}	0	0	0
		Total	2n+3

Step Counts: Example

Algorithm MatrixAdd (a,b,c,m,n)	s/e	Frequency	Total Steps
{			
`for(i=1; i<=m; i++)`			
{			
`for(j=1; i<=n; j++)`			
{			
`c[i][j]=a[i][j]+b[i][j];`			
}			
}			
}			
		Total	

2mn+2m+1

Step Counts: Example

```
algorithm Fibonacci(n)                          Step Count
{
  if n <= 1 then                                  ---- 1
    output 'n'
  else
    f2 = 0;                                       ---- 1
    f1 = 1;                                       ---- 1

    for i = 2 to n do                             ---- n
    {
      f = f1 + f2;                                ---- n - 1
      f2 = f1;                                    ---- n - 1
      f1 = f;                                     ---- n - 1
    }
    output 'f'                                    ---- 1
}                                                  --------
                                  Total no of steps=  4n + 1
```

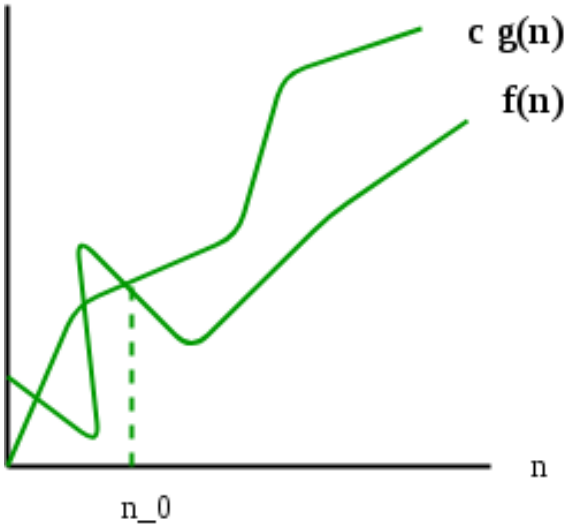
Note that if n ≤ 1 then the step count is just 2 and 4n+1 otherwise.

Asymptotic Analysis/Notations

- Asymptotic notations describes the behavior of the time or space complexity for larger instance characteristics.
- "Upper Bound" indicates the upper or highest growth rate that the algorithm can have.
- Big-oh Notation (O)
  - Big-Oh notation describes an upper bound.
  - Big-Oh notation states a claim about the greatest amount of some resource (usually time) that is required by an algorithm for some class of inputs of size n.

Big-oh (O) Notation

- The notation O(n) is the formal way to express the upper bound of an algorithm's running time.
- It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.
- $f(n) = O(g(n))$ iff there exist positive constants c and n_0 such that: $f(n) \leq c * g(n)$ for all $n, n \geq n_0$

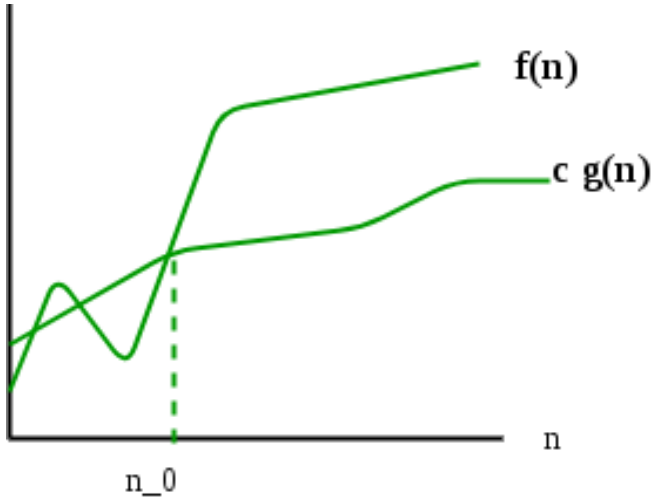


Big-oh Notation (O): Examples

- Examples:
  - $3n + 3 = O(n)$
  - $100n + 6 = O(n)$
  - $1000n^2 + 100n - 6 = O(n^2)$
    - $1000n^2 + 100n - 6 \leq 1001n^2$ for all $n \geq 100$
    - $c = 1001$ and $g(n) = n^2$
  - $6 * 2^n + n^2 = O(2^n)$
    - $6 * 2^n + n^2 \leq 7 * 2^n$ for all $n \geq 4$
    - $c = 7$ and $g(n) = 2^n$
  - $3n + 2 \neq O(1)$
  - $10n^2 + 4n + 2 \neq O(n)$

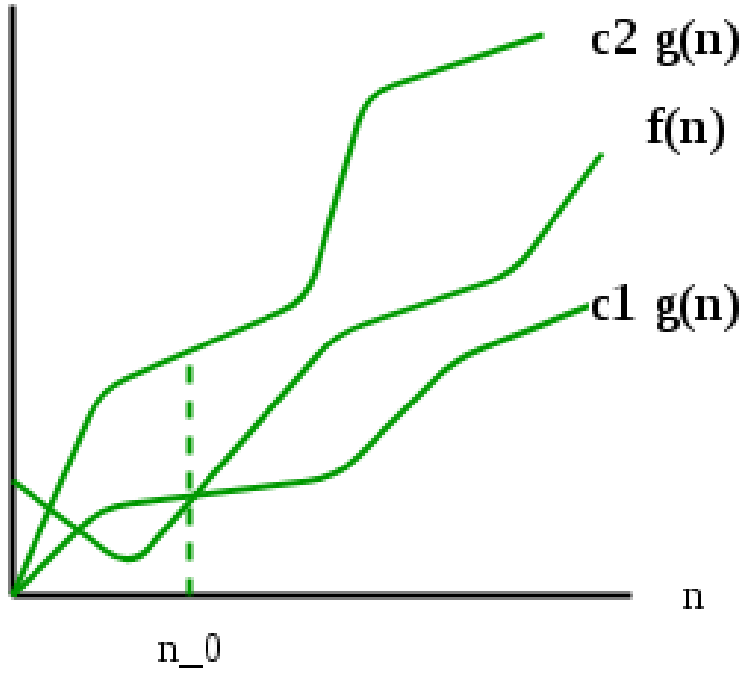
Omega (Ω) Notation

- The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time.
- It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.
- $f(n) = \Omega(g(n))$ iff there exist positive constants c and n_0 such that: $f(n) \geq c * g(n)$ for all $n, n \geq n_0$



Theta (Θ) Notation

- The notation Θ(n) is the formal way to express both, the lower bound and the upper bound of an algorithm's running time..
- $f(n) = \Theta(g(n))$ iff there exist positive constants c_1, c_2 and n_0 such that: $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ for all $n, n \geq n_0$



Space Complexity

- The Space Complexity is the total amount of memory space used by an algorithm/program including the space of input values for execution.

- Memory Usage while Execution:
  - **Instruction Space**: The amount of memory used to save the compiled version of instructions.
  - **Environmental Stack**: When an algorithm (function) is called inside another algorithm (function). In such a situation, the current variables are pushed onto the system stack, where they wait for further execution and then the call to the inside algorithm (function) is made.
  - **Data Space**: The amount of space used by the variables and constants.

Space Complexity

- While calculating the Space Complexity of any algorithm, usually the Data Space is considered and the Instruction Space and Environmental Stack are neglected.

- Space Complexity = Space used by Input Values + Auxiliary Space
  - Auxiliary Space is the extra space or the temporary space used by the algorithm during it's execution.

Calculation of Space Complexity: Examples

```
void main() {
    int a = 5, b = 5, c;
    c = a + b;
    printf("%d", c);
}
```

- Space = 6 Bytes (3 variables -> 2 bytes for each)
- The Space Complexity for the above-given program is O(1), or constant.

Calculation of Space Complexity: Examples

```
void main() {
    int n, i, sum = 0;
    scanf("%d", &n);
    int arr[n];
    for(i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
        sum = sum + arr[i];
    }
    printf("%d", sum);
}
```

- Space = 2n+6 Bytes (3 variables and 1 array of size n)
- The Space Complexity for the above-given program is O(n), or linear.

Calculation of Space Complexity: Examples

```
void matrixAdd(int a[], int b[], int c[], int n)
{
  for (int i = 0; i < n; ++i)
  {
     c[i] = a[i] + b[j]
  }
}
```

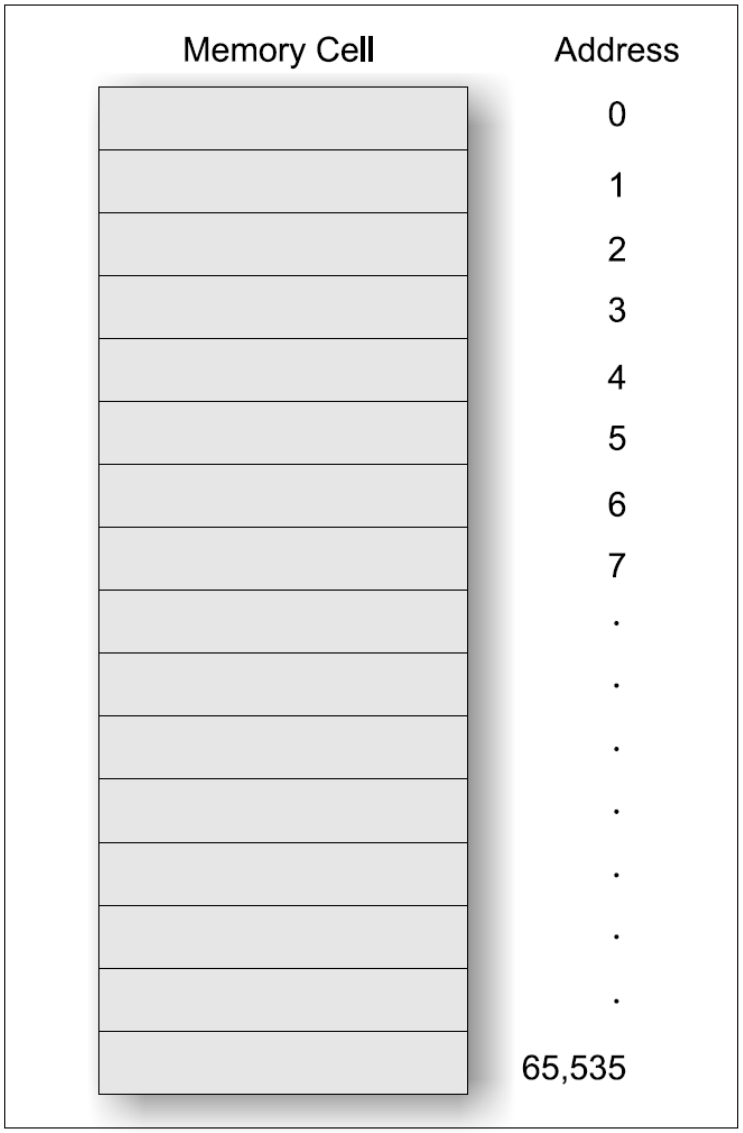
- Space = ?
- Growth Rate = ?

Introduction to Pointer

- A pointer is a derived data type in C.
- It is built from one of the fundamental data types available in C.
- Pointers contain memory addresses as their values.
- Since these memory addresses are the locations in the computer memory, pointers can be used to access and manipulate data stored in the memory.

Computer's Memory Organization

- The computer's memory is a sequential collection of storage cells.
- Each cell, commonly known as a byte, has a number called address associated with it.



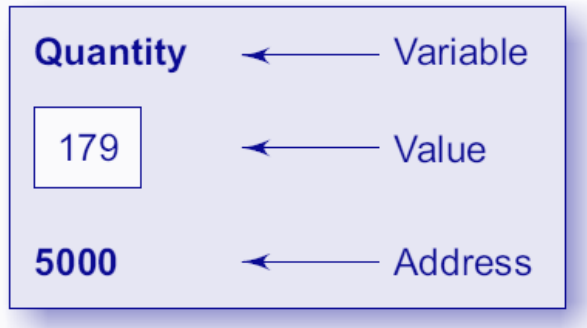
Understanding Pointer

- Whenever we declare a variable, the system allocates, somewhere in the memory, an appropriate location to hold the value of the variable.
- Consider the following statement:

```
int quantity = 179;
```

  - This statement instructs the system to find a location for the integer variable quantity and puts the value 179 in that location.
  - Assume that the system has chosen the address location 5000 for quantity.

Note that the address of a variable is the address of the first bye occupied by that variable.

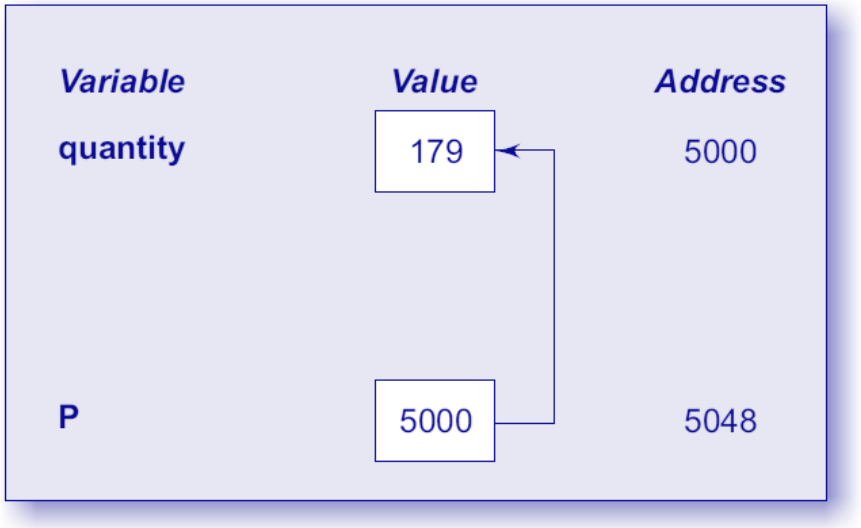


Understanding Pointer (contd…)

- During execution of the program, the system always associates the name quantity with the address 5000.
  - This is something similar to having a house number as well as a house name.
  - We may have access to the value 179 by using either the name quantity or the address 5000.
- Variables that hold memory addresses are called pointer variables.
  - A pointer variable is, therefore, a variable that contains an address, which is a location of another variable in memory.
  - Since a pointer is a variable, its value is also stored in the memory in another location.

Understanding Pointer (contd…)

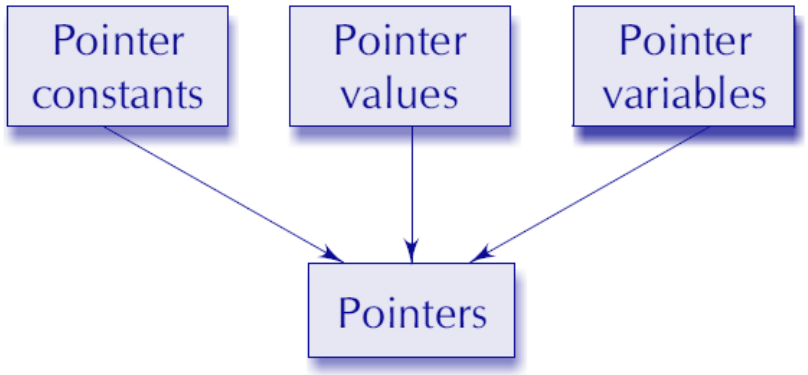
- Suppose, we assign the address of quantity to a variable P:
- The link between the variables P and quantity can be visualized as:



  - Since the value of the variable P is the address of the variable quantity, we may access the value of quantity by using the value of P and therefore, we say that the variable P 'points' to the variable quantity. Thus, P gets the name 'pointer'.

Understanding Pointer (contd…)

- Pointers are built on the three underlying concepts as illustrated below:



  - Memory addresses within a computer are referred to as pointer constants. We cannot change them; we can only use them to store data values.
  - We cannot save the value of a memory address directly. We can only obtain the value through the variable stored there using the address operator (&). The value thus obtained is known as pointer value.
  - Variable that contains a pointer value is called a pointer variable.

Accessing the Address of a Variable

- The address of a variable is determined with the help of **&** (address of) operator.

```
P = &quantity;
```

  - The & operator can be used only with a simple variable or an array element.
  - &125 (Pointing at Constant) ----> Correct or Incorrect?
  - &x (Pointing at Array Name x[10]) ----> Correct or Incorrect?
  - &(x+y) (Pointing at Expression) ----> Correct or Incorrect?
  - &x[0] (Pointing at Array Element 0) ----> Correct or Incorrect?
  - &x[i+3] (Pointing at Array Element i+3) ----> Correct or Incorrect?

Program to Print the Addresses of Variables

```
void main()
{
    char a;
    int x;
    float p;

    a = 'A';
    x = 125;
    p = 10.25;

    printf("%c is stored at address %u.\n", a, &a);
    printf("%d is stored at address %u.\n", x, &x);
    printf("%f is stored at address %u.\n", p, &p);
}
```

Declaring a Pointer Variable

- The declaration of a pointer variable takes the following form:

```
dataType *pointerName;
```

  - This tells the compiler three things about the variable `pointerName`:
    - The asterisk (*) tells that the variable `pointerName` is a pointer variable.
    - `pointerName` needs a memory location.
    - `pointerName` points to a variable of type `dataType`.

- Pointer Declaration Style:
  - `int*  p;`
  - `int  *p;`
  - `int  * p;`

Initialization of Pointer Variables

- Once a pointer variable has been declared we can use the assignment operator to initialize the variable. Examples:

```
int quantity;
int *p;              //Declaration
p = *quantity;       //Initialization
```

```
//We can also combine the initialization with the declaration.
int *p = &quantity;
```

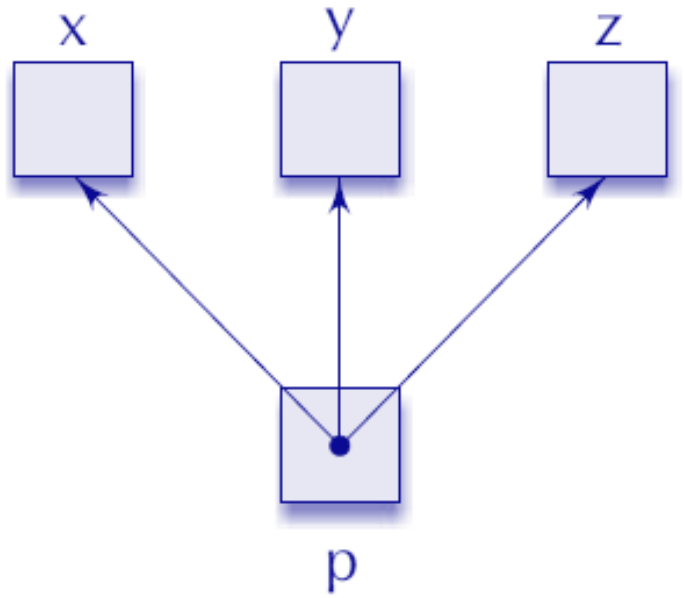
```
//It is also possible to combine the declaration of data variable, the declaration of pointer variable and the initialization of the pointer variable in one step.
int x, *p = &x;  //Three in One
```

- We could also define a pointer variable with an initial value of NULL or 0 (zero).

Pointer Flexibility

- We can make the same pointer to point to different data variables in different statements:

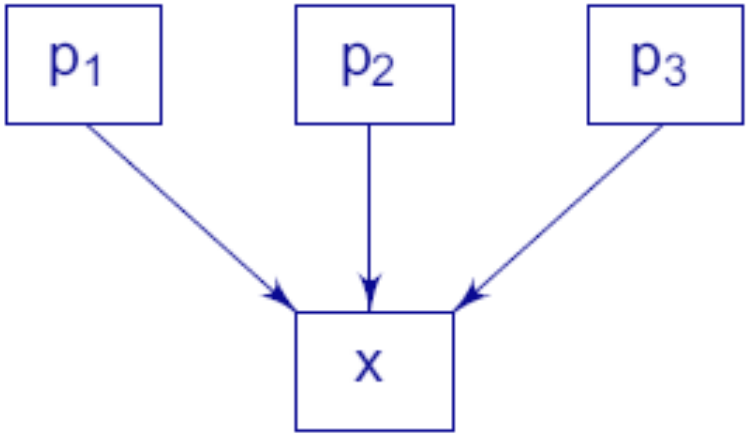
```
int x, y, z, *p;
. . . . .
p = &x;
. . . . .
p = &y;
. . . . .
p = &z;
. . . . .
```



Pointer Flexibility (contd…)

- We can also use different pointers to point to the same data variable:

```
int x;
int *p1 = &x;
int *p2 = &x;
int *p3 = &x;
. . . . .
. . . . .
```



Accessing a Variable through its Pointer

- To access the value of the variable using the pointer, we use unary operator * (asterisk), usually known as the indirection operator.

```
int quantity, *p, n;
quantity = 179;
p = &quantity;
n = *p;
//p = &quantity; n = *p; are equivalent to n = *&quantity; which in turn is equivalent to n = quantity;
```

Chain of Pointers

- It is possible to make a pointer to point to another pointer, thus creating a chain of pointers as shown:

  - Here, the pointer variable p2 contains the address of the pointer variable p1, which points to the location that contains the desired value. This is known as multiple indirections.
  - A variable that is a pointer to a pointer must be declared using additional indirection operator symbols in front of the name. Example:

```
int **p2;
```

Pointer Expressions

- Like other variables, pointer variables can be used in expressions.
- For example, if p1 and p2 are properly declared and initialized pointers, then the following statements are valid:

Valid Expressions

```
y = *p1 * *p2;
sum = sum + *p1;
z = 5* - *p2/ *p1;
*p2 = *p2 + 10;
p1 + 4;
p2 - 2;
p1 - p2;
p1++;
p1 > p2; p1 = = p2; p1 != p2;
```

Invalid Expressions

```
p1 / p2;
p1 * p2;
p1 / 3;
p1  +  p2;
```

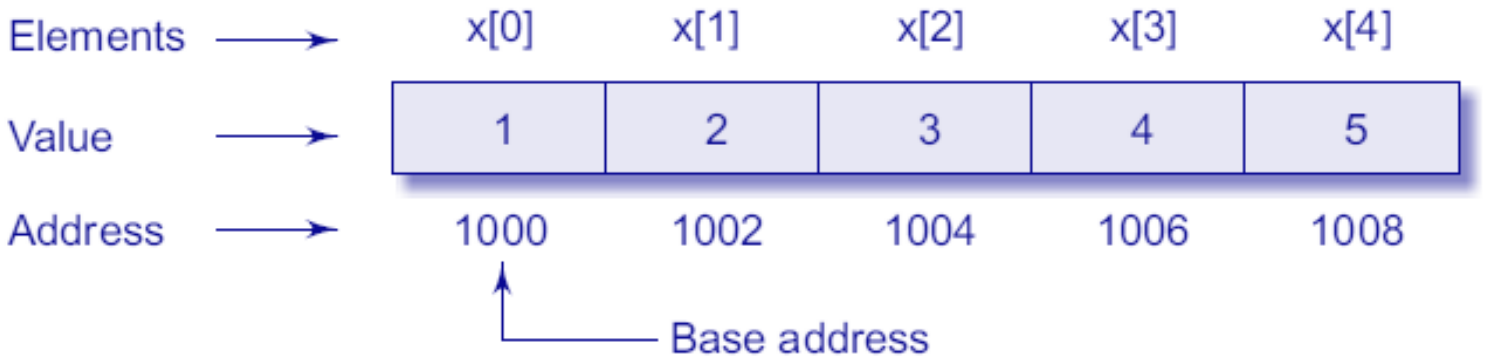
//Division, Multiplication, and Addition of two pointers variables is illegal.

Rules of Pointer Operations

- A pointer variable can be assigned the address of another variable.
- A pointer variable can be assigned the values of another pointer variable.
- A pointer variable can be initialized with NULL or zero value.
- A pointer variable can be pre-fixed or post-fixed with increment or decrement operators.
- An integer value may be added or subtracted from a pointer variable.
- When two pointers point to the same array, one pointer variable can be subtracted from another.
- When two pointers point to the objects of the same data types, they can be compared using relational operators.
- A pointer variable cannot be multiplied by a constant.
- Two pointer variables cannot be added.

Pointers and Arrays

- When an array is declared, the compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory locations.
- The base address is the location of the first element (index 0) of the array.
- Suppose we declare an array x as follows: `int x[5] = {1, 2, 3, 4, 5};`



- If we declare p as an integer pointer, then we can make the pointer p to point to the array x by the following assignment: `p = x;`

Pointers and Arrays (contd…)

- `p = x;` is equivalent to `p = &x[0];`
- Now, we can access every value of x using p++ to move from one element to another. The relationship between p and x is shown as:

```
p = &x[0] (= 1000)
p+1 = &x[1] (= 1002)
p+2 = &x[2] (= 1004)
p+3 = &x[3] (= 1006)
```

When handling arrays, instead of using array indexing, we can use pointers to access array elements. Note that *(p+3) gives the value of x[3].

The address of an element is calculated using its index and the scale factor of the data type.

```
address of x[3] = base address + (3 x scale factor of int)
                = 1000 + (3 x 2) = 1006
```

Pointers and Arrays (contd…)

- Pointers can be used to manipulate two-dimensional arrays as well.
- As we know that in a one-dimensional array `x`, the expression `*(p+i)` represents the element `x[i]`.
- Similarly, an element in a two-dimensional array can be represented by the pointer expression as `*(*(p+i)j)`.

Pointers as Function Arguments

```
void main()
{
    int x;
    x = 20;
    change(&x);     //Call by Reference or Addres
    printf("%d\n",x);
}
void change(int *p)
{
    *p = *p + 10;
}
```

Functions Returning Pointers

```
void main ( )
{
    int a = 10, b = 20, *p;
    p = larger(&a, &b);  //Function Call
    printf (“%d”, *p);
}

int *larger (int *x, int *y)
{
    if (*x>*y)
        return (x);  // Address of a
    else
        return (y);  //Address of b
}
```

Structure

- Structure is a mechanism for packing data of different types.
- Structure is a convenient tool for handling a group of logically related data items.
- Examples:
  - **Time**: Seconds, Minutes, Hours
  - **Date**: Day, Month, Year
  - **Book**: Author, Title, Price, Year
  - **Address**: Door No, Street, City, State
  - **Student**: Name, Roll, Course, Year

Structure

- Defining a Structure:

```
struct Tag_Name
{
    dataType member1;
    dataType member2;
    … … … .
};
```

- Example:

```
struct Book
{
    char title[20];
    int year;
    float price;
};
```

Structure

- Declaring Structure Variable:

```
struct Book
{
    char title[20];
    int year;
    float price;
};
struct Book b1, b2;
```

OR

```
struct
{
    char title[20];
    int year;
    float price;
} Book b1, b2;
```

Structure

- Type-Defined Structure:

```
typedef struct
{
    char title[20];
    int year;
    float price;
}Book;
Book b1, b2;
```

Structure

- Structure Initialization:

```
struct book
{
    char title[20];
    int year;
    float price;
};
struct book b1 = {“Data Structure”, 2015, 199.90};
struct book b2 = {“Java Programming”, 2017, 395.50};
struct book b3 = {“Software Engineering”, 2017};
```

Structure

- Copying and Comparing Structure Variables:
  - If **b1** and **b2** belong to same structure, then the following is valid:

```
b1 = b2;
```

  - However, the following statements are not permitted.

```
b1 == b2;

b1 != b2
```

Structure

- Accessing Structure Members (Using Dot Notation):
  - The line between a structure Member and a Variable is established using the member operator '.' which is also known as 'dot operator' or 'period operator'.
  - Examples:

```
struct book b;
b.title = "C Programming";
b.page = 160;
scanf("%d",&b.page);
printf("%d",b.page);
```

Structure

- Accessing Structure Members (Using Selection Notation):
  - When the structure variables are declared as Pointer Variables, the line between a structure Member and a Variable is established using the '->' operator.
  - Examples:

```
struct book b1, *b;
b = &b1;
b->title = "C Programming";
b->page = 160;
scanf("%d",&b->page);
printf("%d",b->page);
```

Structure

- Arrays of Structure:
  - Examples:

```
struct book b[2];
b[1].page = 160;
scanf("%d", &b[1].page);
printf("%d", b[1].page);
b[2].page = 170;
scanf("%d", &b[2].page);
printf("%d", b[2].page);
```

Structure

- Self-Referential Structure:
  - Structure that include an element that is a pointer to another structure of the same type.
  - This type of structure is useful in creating complex data structure such as linked list, tree, graph.
  - Example:

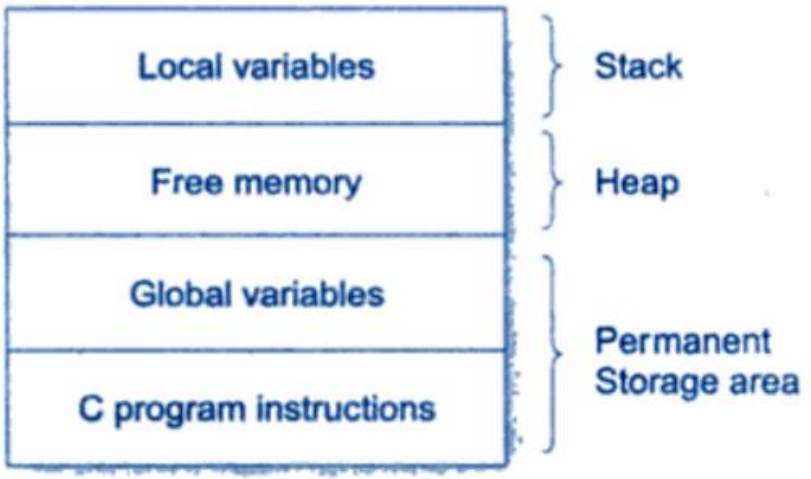
```
typedef struct node
{
    int info;
    struct node *next;
}NODE;

NODE *start;
```

Structure

- Structure and Function:
  - Structure variables can be passed as arguments to function.
  - Example:

```
struct book update(struct book b1)
{
    b1.price = b1.price + 10.0;
    return (b1);
}
```

Dynamic Memory Management



- The program instructions and global and static variables are stored in a region known as ***permanent storage area***.
- Local Variables are stored in another area, called ***Stack***.
- The memory space available between **stack** and **permanent storage area** is available for dynamic allocation during execution of program.
- This free memory area is called ***Heap***.
- The size of **heap** keeps changing due to creation and death of variables that are local to function and blocks.

Memory Management Functions

- **malloc() ← stdlib.h**
  - The name malloc stands for "memory allocation".
  - The function malloc() reserves a block of memory of specified size and return a pointer of type **void** which can be casted into pointer of any form.
  - **Syntax:**

```
ptr = (cast_type *) malloc (byte_size);
```

  - **Example:**

```
ptr = (int *) malloc (100*sizeof(int));
b = (Book *) malloc (sizeof(Book));
```

  - If the space is insufficient, allocation fails and returns NULL pointer.

Memory Management Functions

- **calloc()**
  - The name calloc stands for "contiguous allocation".
  - calloc() allocates multiple blocks of memory each of same size.
  - **Syntax:**

```
ptr = (cast_type *) calloc(n,element_size)
```

  - **Example:**

```
ptr = (int *) calloc(20, sizeof(int));
```

Memory Management Functions

- **free()**
  - Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on its own.
  - free() releases the space.
  - **Syntax:**

```
free(ptr);
```

Memory Management Functions

- **realloc()**
  - If the previously allocated memory is insufficient or more than required, we can change the previously allocated memory size using realloc().
  - **Syntax:**

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

Other Readings / Suggestions

- For basic understanding of array, structure, pointer, and file handling concepts in “C Programming”, students are advised to go through following course website:
  - https://spoken-tutorial.org/tutorial-search/?search_foss=C+and+Cpp&search_language=English