



DSE 2155 DATA STRUCTURES
[3 1 0 4]

Linda Varghese
Savitha G

Department of Computer Applications, MIT, Manipal.



Classroom Rules of Engagement.

- 🔔 All laptop computers, mobile phones, tablet computers must be closed during all classroom hours (**offline class**).
- 🔔 Computers distract the most people behind and around the user.
- 🔔 **Maintain social distance and wear masks (correctly) at all times inside and outside class.**
- 🔔 Make your own notes. Slides are not enough.
- 🔔 Self study is paramount.
- 🔔 **All homework** to be **completed** before class commences.

How do you improve your performance?

- ⌚ To transfer information from your short-term memory to your long-term memory, that information must be imposed on your mind ***at least three times.***
- ⌚ You should always try the following:
 - ⌚ Look at the notes/slides (**IF ANY**) before class.
 - ⌚ Attend all lectures (if possible).
 - ⌚ Review the lecture during the evening.
 - ⌚ Rewrite and summarize the slides in your words.
- ⌚ In addition to this, you should:
 - ⌚ Get a reasonable nights sleep (apparently this is when information is transferred to your long-term memory), and
 - ⌚ Eat a good breakfast (also apparently good for the memory)

Douglas Wilhelm Harder, MMath



Agenda

- ▶ Syllabus abstract and Textbook
- ▶ Definitions of basic terms
- ▶ Example
- ▶ Why study Data structure and algorithms?
- ▶ What are the different classifications
- ▶ Some example data structures you study in this course

MAIN TOPICS:-

DSE- 2155 : DATA STRUCTURES [3 1 0 4]

Introduction, Programming fundamentals, Recursion, Stacks, Queues and their applications, Sparse Matrix, Pointers and dynamic memory allocation,

Linked Lists: Singly linked lists, Dynamically Linked Stacks and Queues, Polynomial representation and polynomial operations using singly linked list, Singly Circular Linked List, Doubly Linked Lists,

Trees: Binary trees, Heaps, Binary Search Trees, Threaded binary trees,

Graphs: Terminologies, Depth First Search, Breadth First Search, Sorting and searching Techniques.

Text books

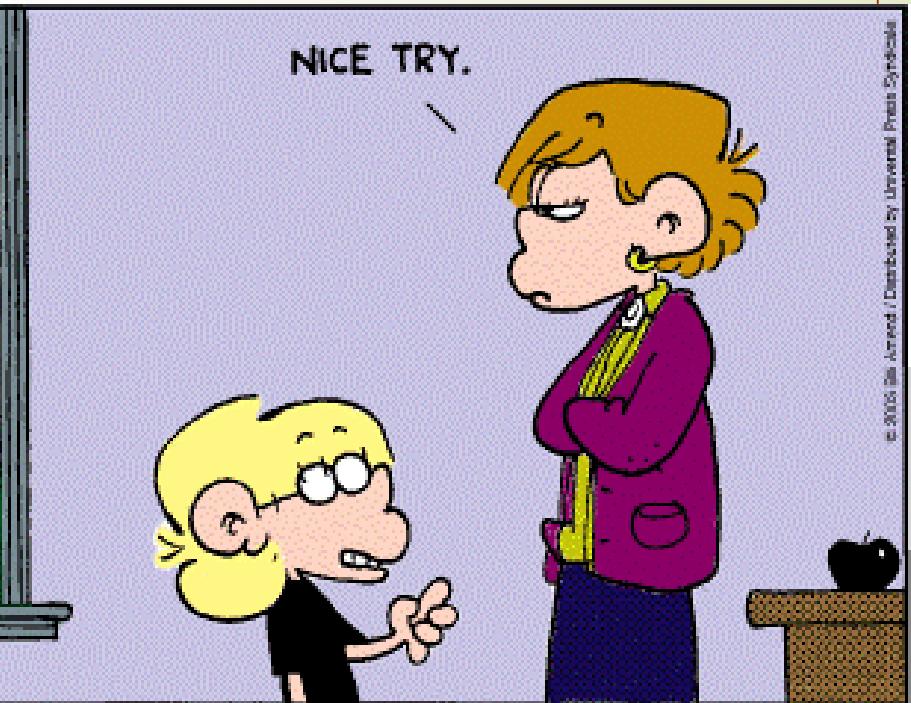
- ▶ Ellis Horowitz, Sartaj Sahni, Dinesh Mehta, Fundamentals of Data Structures in C++, 2nd Edition.
- ▶ Ellis Horowitz, Sartaj Sahni, Dinesh Mehta, Fundamentals of Data Structures in C, 2nd Edition.
- ▶ Behrouz A. Forouzan, Richard F. Gilberg, A Structured Programming Approach Using C, 3e, Cengage, Learning India Pvt.Ltd, India,2007.
- ▶ Behrouz A. Forouzan, Richard F. Gilberg, Data Structures, A Pseudocode approach Using C, 2e, Cengage, learning India Pvt.Ltd, India, 2009.
- ▶ Debasis Samanta, Classic Data structures- 2nd edition, PHI Learning Private Limited , 2010

Language of Implementation.

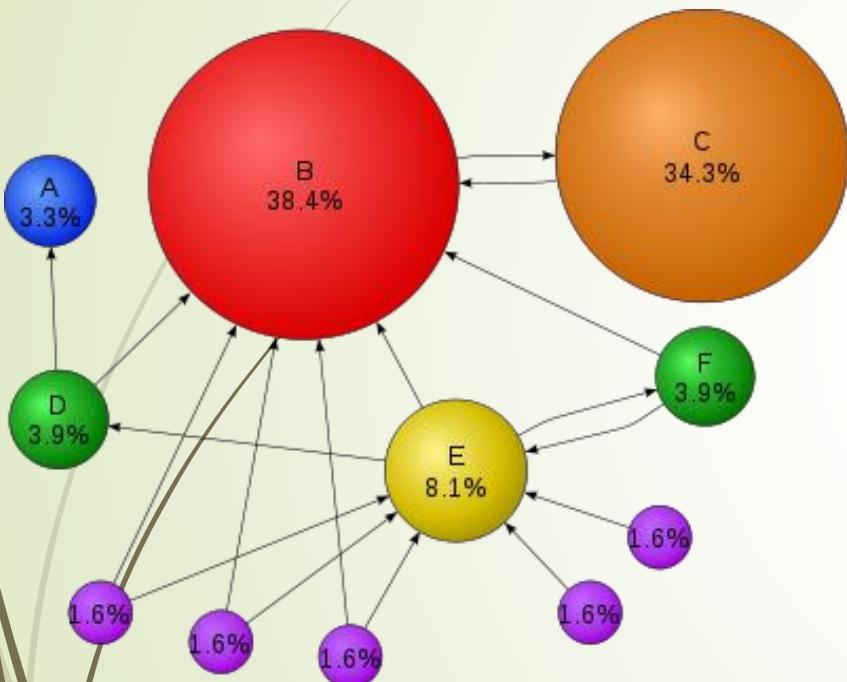
- ➊ You will be using the C++ programming language in this course
 - ➋ Knowledge of syntax of C++ pre-requisite

```
#include <stdio.h>
int main(void)
{
    int count;
    for(count = 1; count <= 500; count++)
        printf("I will not throw paper airplanes in class.");
    return 0;
}
```

AMEND 10-3



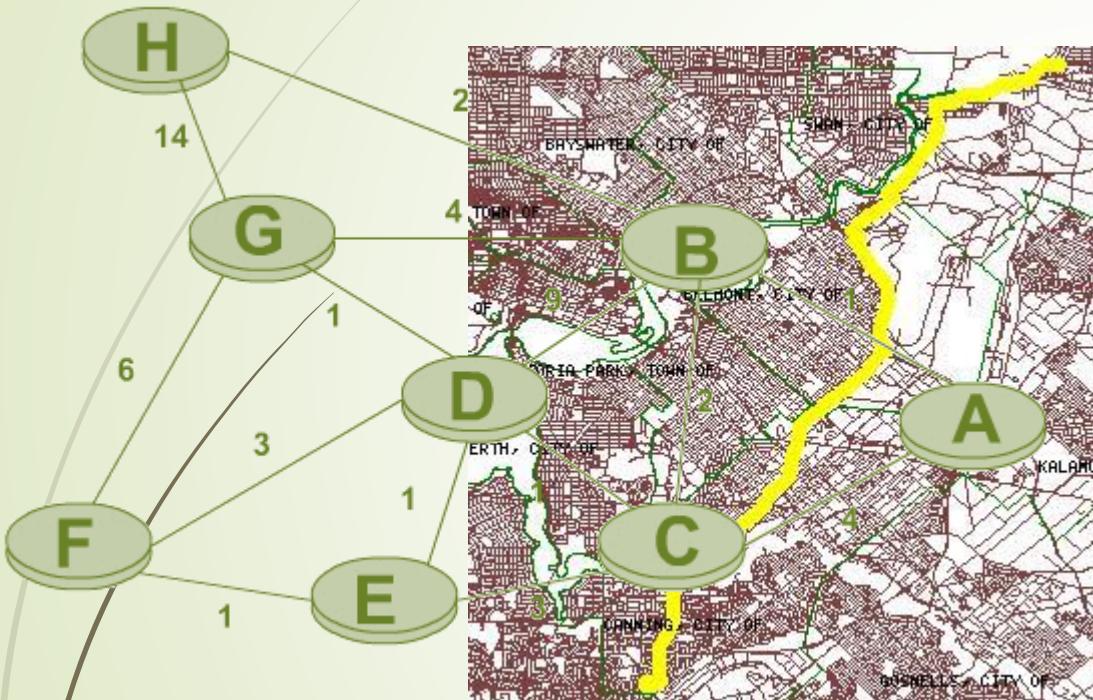
Why you want to study Algorithms?



- Making a lot of money out of a great algorithm
- \$1,000,000,000?
- Example:
 - ▶ PageRank algorithm by Larry Page: The soul of Google search engine.

Google total assets: \$31 billions on 2008

Why you want to study Algorithms?



- Simply to be cool to invent something in computer science
- Example: Shortest Path Problem and Algorithm
 - Used in GPS and Mapquest or Google Maps

Basic Terminologies

- ▶ **Data:** are simply a value or a set of values of different or same type which is called data types like string, integer, char etc.
- ▶ **Structure:** Way of organizing information, so that it is easier to use

Data Structure

- In simple words we can define **data structures** as
 - Its a way of storing and organizing data in such a way that it is easier to use.
 - A data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently in programs or algorithms.
 - A scheme for organizing related pieces of information.

Algorithm

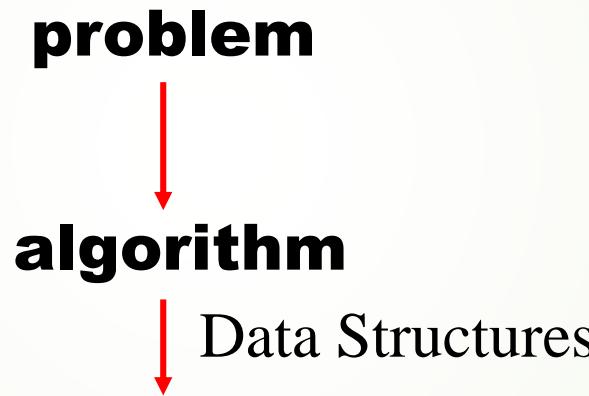
- Set of instructions which work on the data structure to solve a particular problem

Algorithm and Data Structures.

- An algorithm is a sequence of unambiguous instructions/operations for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.

Map Navigation
A → B

input
Graphs



output
Path

How to study algorithms?

- ↳ Problem
- ↳ Representation/data structure in computer
- ↳ Operations on representations



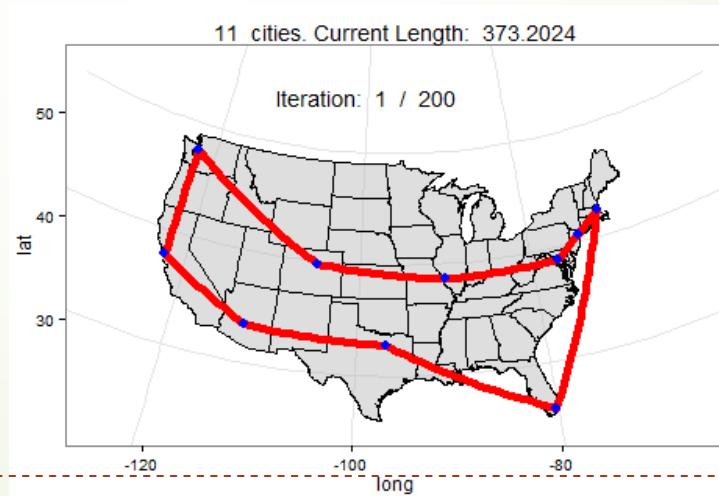
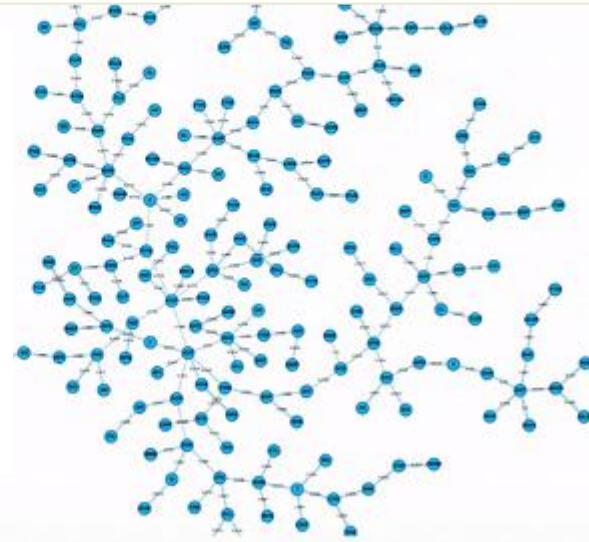
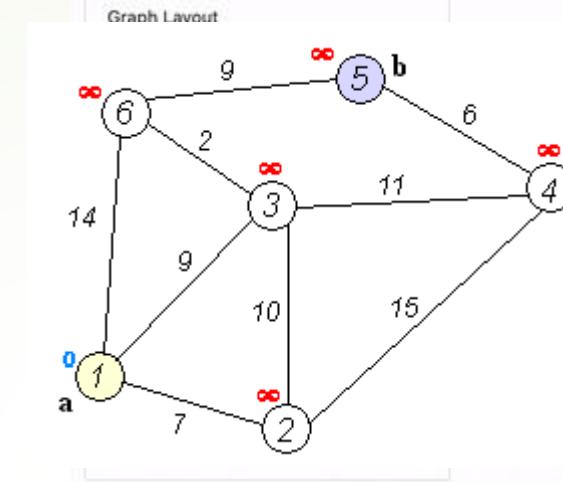
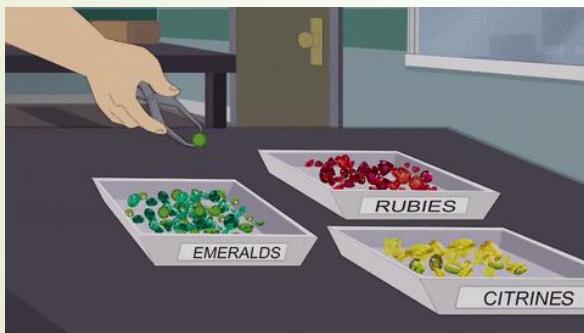
Some Important Points.

- Each step of an algorithm is unambiguous
- The range of inputs has to be specified carefully
- The same algorithm can be represented in different ways
- The same problem may be solved by different algorithms
- Different algorithms may take different time to solve the same problem – we may prefer one to the other

Fundamentals of Algorithmic Problem Solving.

1. Understanding the problem
2. Ascertaining the capabilities of a computational device
3. Choose between exact and approximate problem solving
4. Deciding an appropriate **data structure**
5. **Algorithm** design techniques
6. Methods of specifying an algorithm
 - ▶ **Pseudocode** (for, if, while //, ←, indentation...)
7. Prove an algorithm's correctness – mathematic induction
8. **Analyzing** an algorithm – Simplicity, efficiency, optimality
9. Coding an algorithm

Some Well-known Computational Problems.



11



What is data?

↳ Data

- ↳ A collection of facts from which conclusion may be drawn. Example: data: Temperature 35°C;
Conclusion: It is hot.

↳ Types of data

- ↳ Textual: For example, your name (Muhammad)
- ↳ Numeric: For example, your ID (090254)
- ↳ Audio: For example, your voice
- ↳ Video: For example, your voice and picture

So, what is Data Type then?

- **Data Type:** Collection of objects and a set of operations that act on those objects.
- **Abstract Data Type:**

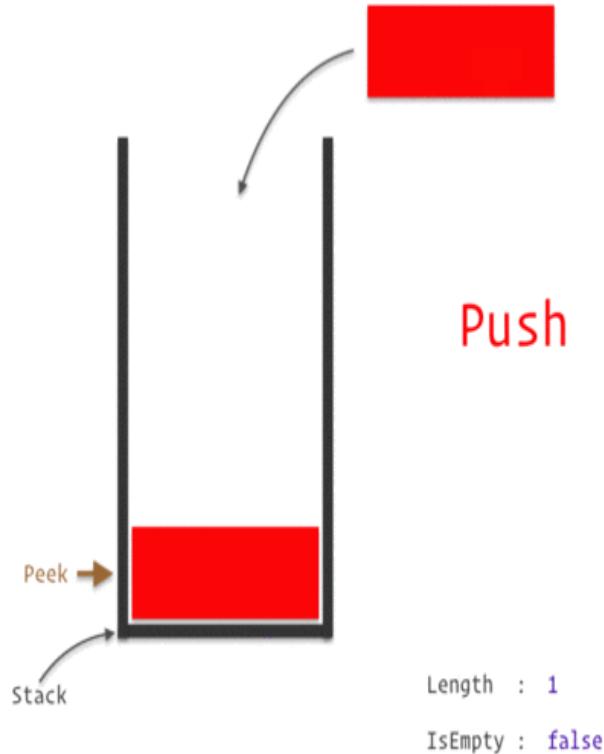
An abstract data type(ADT) is a data type that is organized in such a way that the specification of the objects and the operations on the objects is separated from the representation of the objects and the implementation of the operations.

A logical view of how we view data and operations.

Abstract Data Type (ADT)

- An **abstract data type** is a data declaration packaged together with the operations that are meaningful on the data type (composite types).
- In other words, we encapsulate the data and the operation on data and we hide them from the user.
- ADT has
 1. **Declaration of data** (set of values on which it operates)
 2. **Declaration of operation**(set of functions)and hides the representation and implementation details

And, what is **data structure**?



- A particular way of **storing and organizing data** in a computer so that it can be used efficiently and effectively.
- **Data structure is the logical or mathematical model of a particular organization of data.**
- A group of data elements grouped together under one name.
- For example, an array of integers

Classification of Data Structure ...

- ▶ **Simple Data Structure:** Simple data structure can be constructed with the help of primitive data types. A primitive data structure used to represent the standard data types of any one of the computer languages (integer, Character, float etc.).
- ▶ **Compound Data Structure:** Compound data structure can be constructed with the help of any one of the primitive data structure and it is having a specific functionality. It can be designed by user. It can be classified as **Linear and Non-Linear** Data Structure.

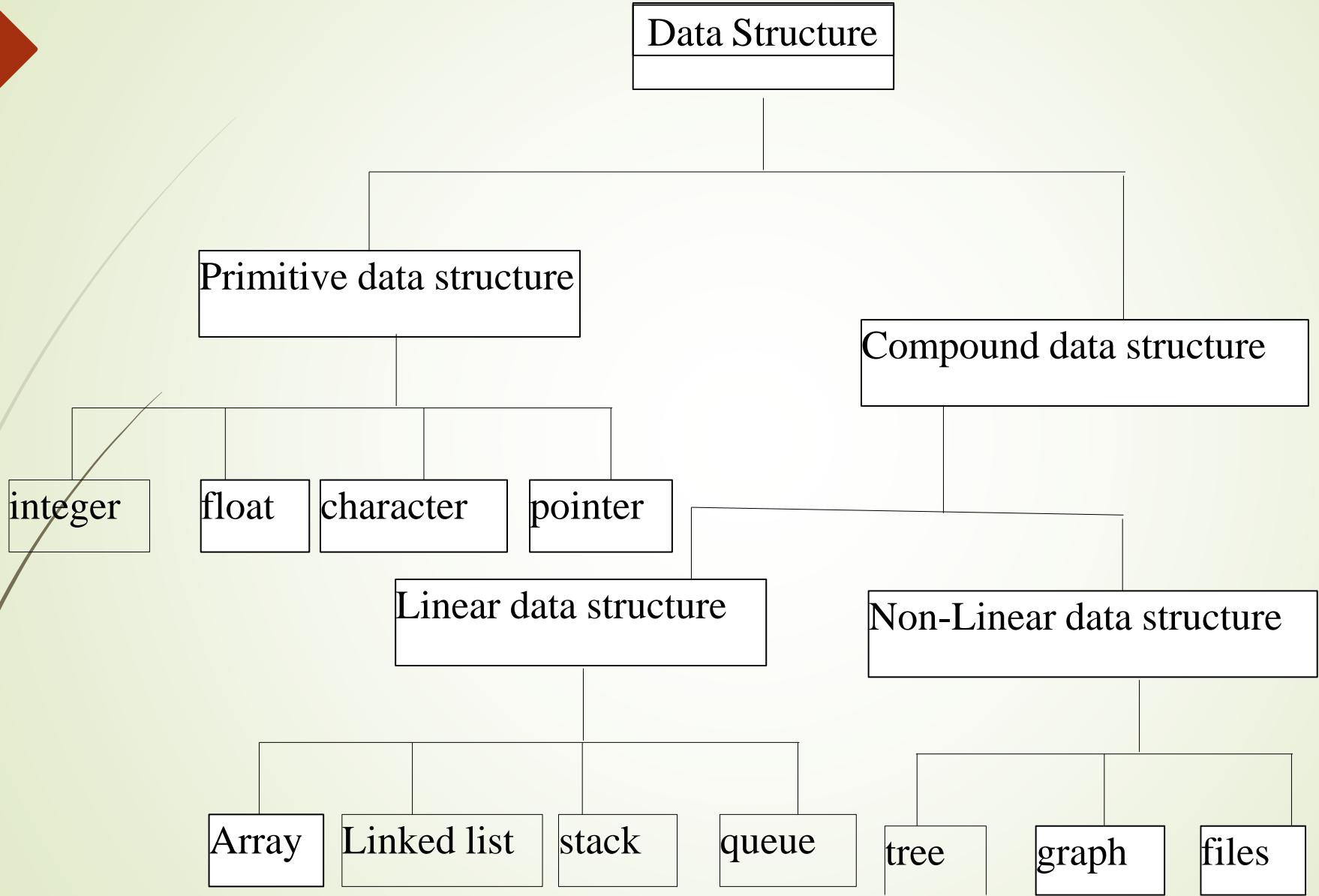
Another way of classification

- ▶ **Linear Data Structures:** A linear data structure traverses the data elements sequentially, in which only one data element can directly be reached.
- ▶ Continuous arrangement of data elements in the memory. **Relationship of adjacency** is maintained between the data elements.

Ex: Arrays, Linked Lists

- ▶ **Non-Linear Data Structures:** Every data item is attached to several other data items in a way that is specific for reflecting relationships. The data items are not arranged in a sequential structure.
- ▶ collection of randomly distributed set of data item joined together by using a special pointer (tag). **Relationship of adjacency** is not maintained between the data elements.

Ex: Trees, Graphs



Types of Data Structure

- ▶ **Array:** is commonly used in computer programming to mean a contiguous block of memory locations, where each memory location stores one fixed-length data item. e.g. Array of Integers int a[10], Array of Character char b[10]

Array of Integers									
0	1	2	3	4	5	6	7	8	9
5	6	4	3	7	8	9	2	1	2

Array of Character									
0	1	y	3	4	5	h	7	8	9
a	6	4	k	7	8	9	q	1	2

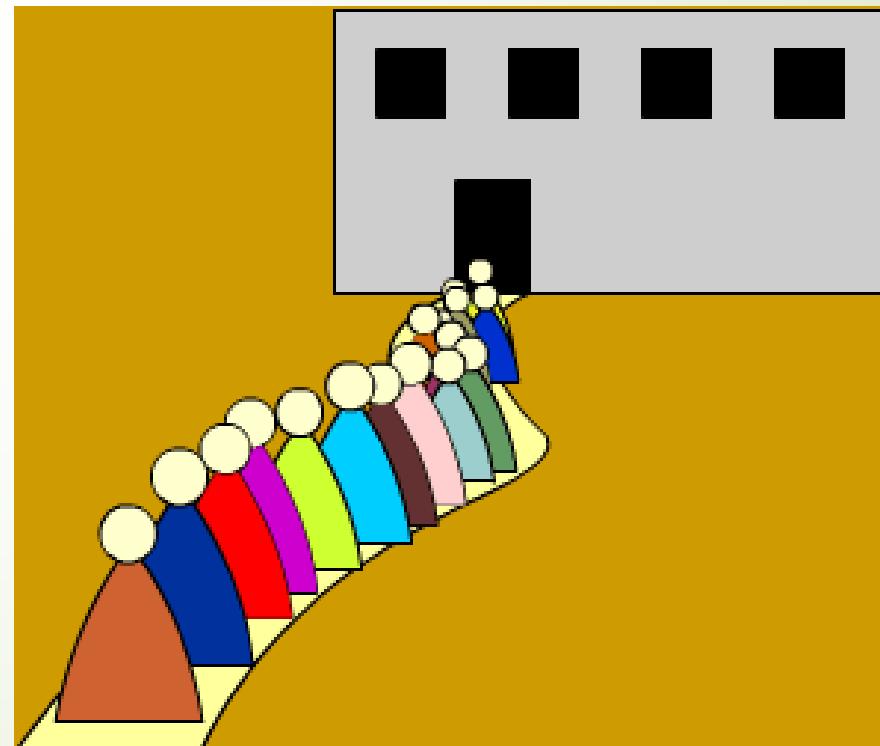
Types of Data Structure ...

- **Stack:** A stack is a data structure in which items can be inserted only from one end and get items back from the same end. There , the last item inserted into stack, is the first item to be taken out from the stack. In short its also called Last in First out [LIFO].



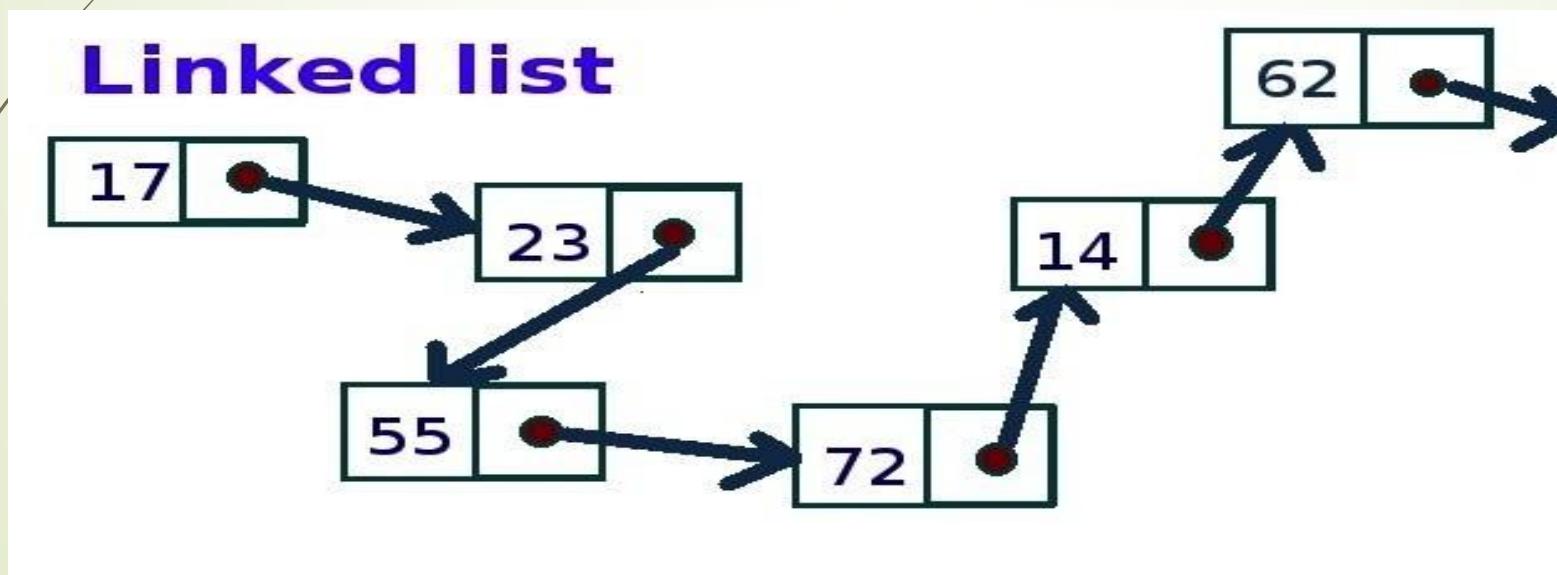
Types of Data Structure ...

- ▶ **Queue:** A queue is two ended data structure in which items can be inserted from one end and taken out from the other end. Therefore ,the first item inserted into queue is the first item to be taken out from the queue. This property is called First in First out [FIFO].



Types of Data Structure ...

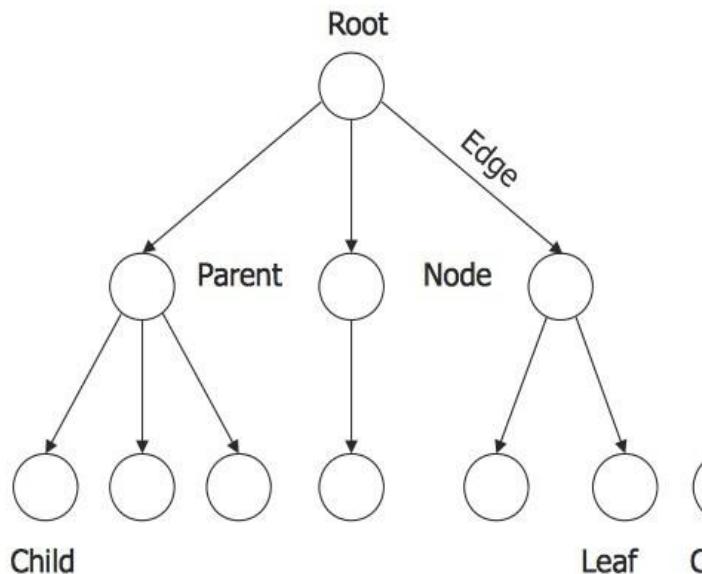
- ▶ **Linked List:** Could alternately used to store items. In linked list space to store items is created as is needed and destroyed when space no longer required to store items. Hence **linked list is a dynamic data structure, space acquired only when need.**



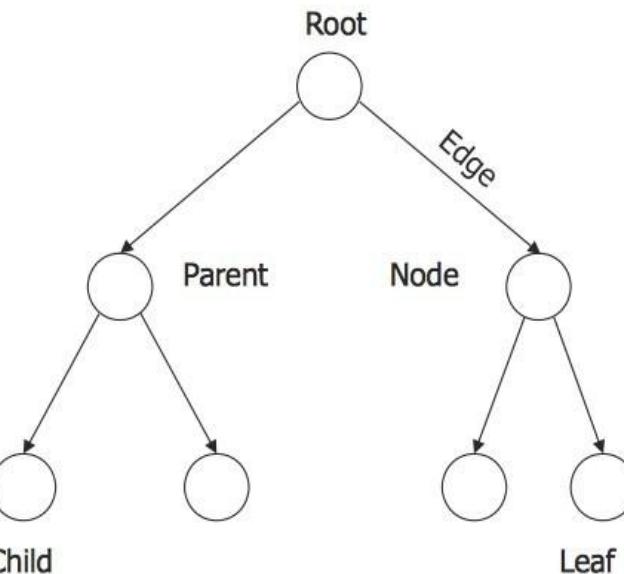
Types of Data Structure ...

- **Tree:** is a non-linear data structure which is mainly used to represent data containing a hierarchical relationship between elements.
- **Binary Tree:** A binary tree is a tree such that every node has at most 2 child and each node is labeled as either left or right child.

■ General tree

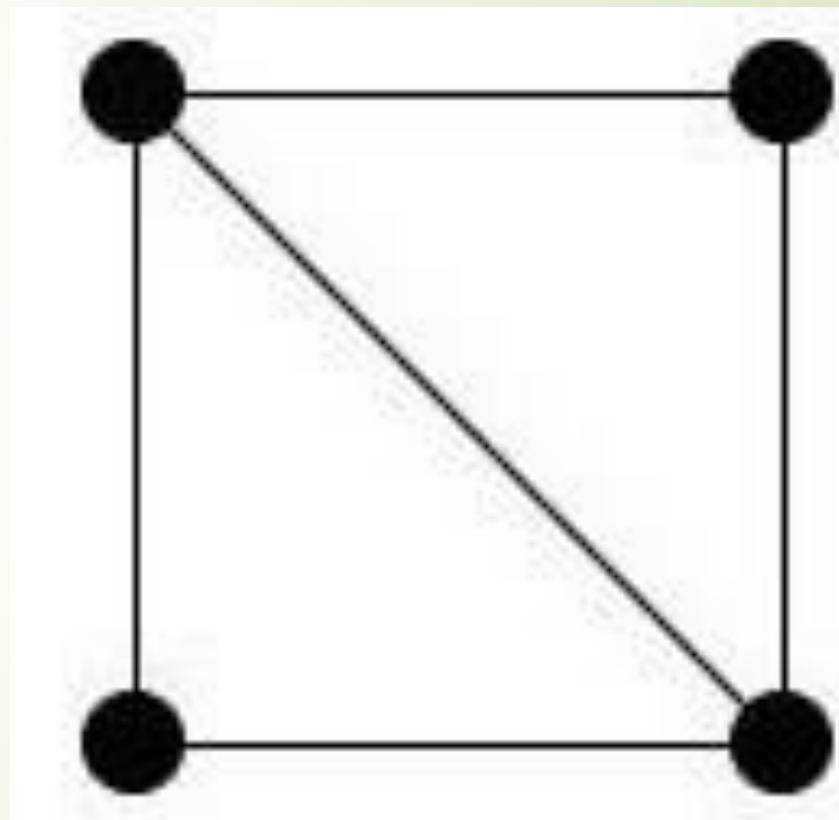


■ Binary Tree



Types of Data Structure ...

- ▶ **Graph:** It is a set of items connected by edges.
- ▶ Each item is called a vertex or node.
- ▶ Trees are just like a special kinds of graphs.
- ▶ Graphs are usually represented by $G = (V, E)$, where V is the set vertices and E is the set of Edges.



Arrays vs. Lists

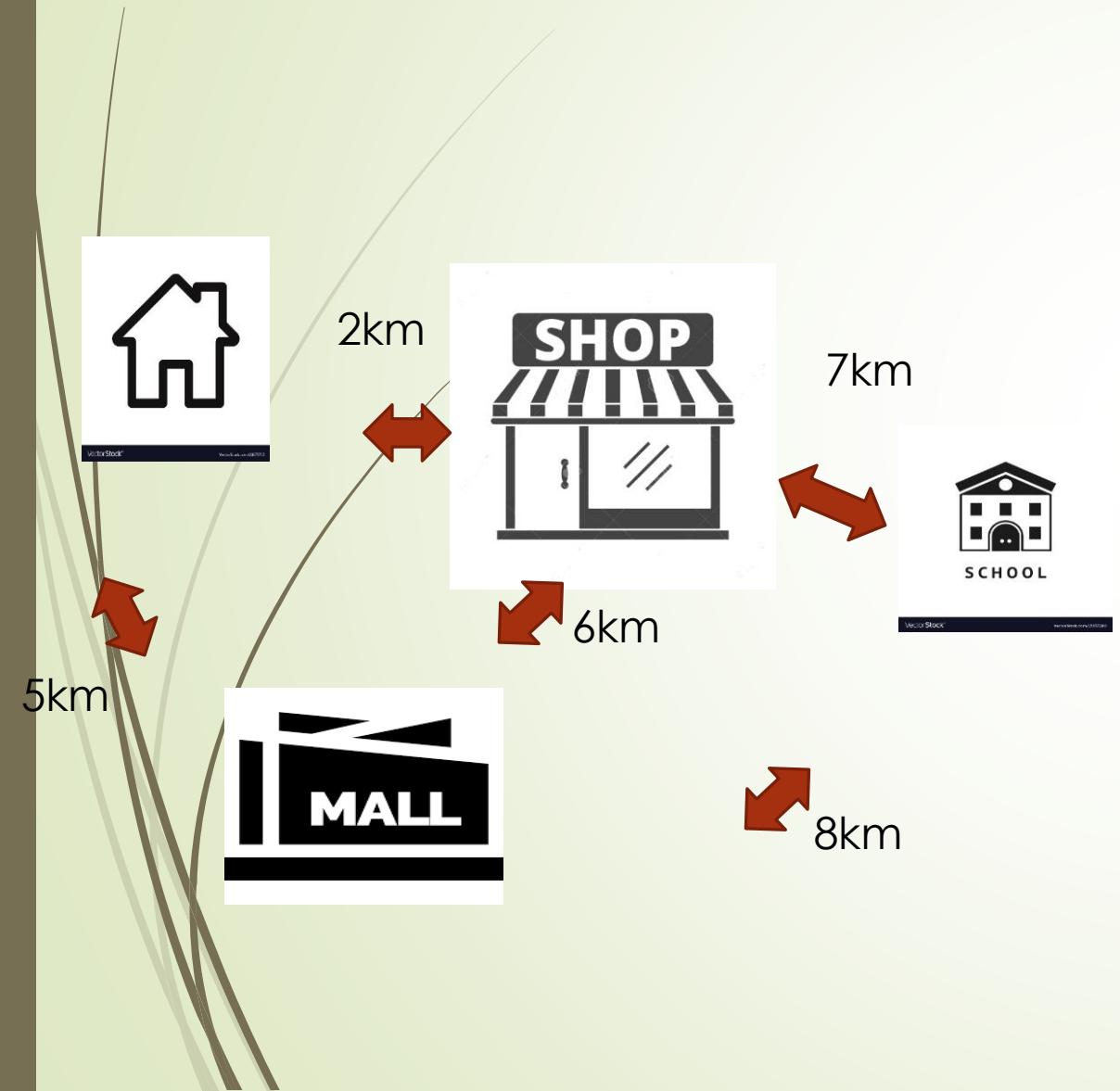
- Array as a Data structure: It is an ordered set which consist of *fixed number of Objects.*
Operations which can be performed on arrays are:
 - Create an array of some fixed size,
 - Store elements, retrieve elements, destroy an array
 - No insertion or deletion possible (fixed size)!!!!
- List: Ordered set consisting of variable number of objects.



Arrays vs. Lists

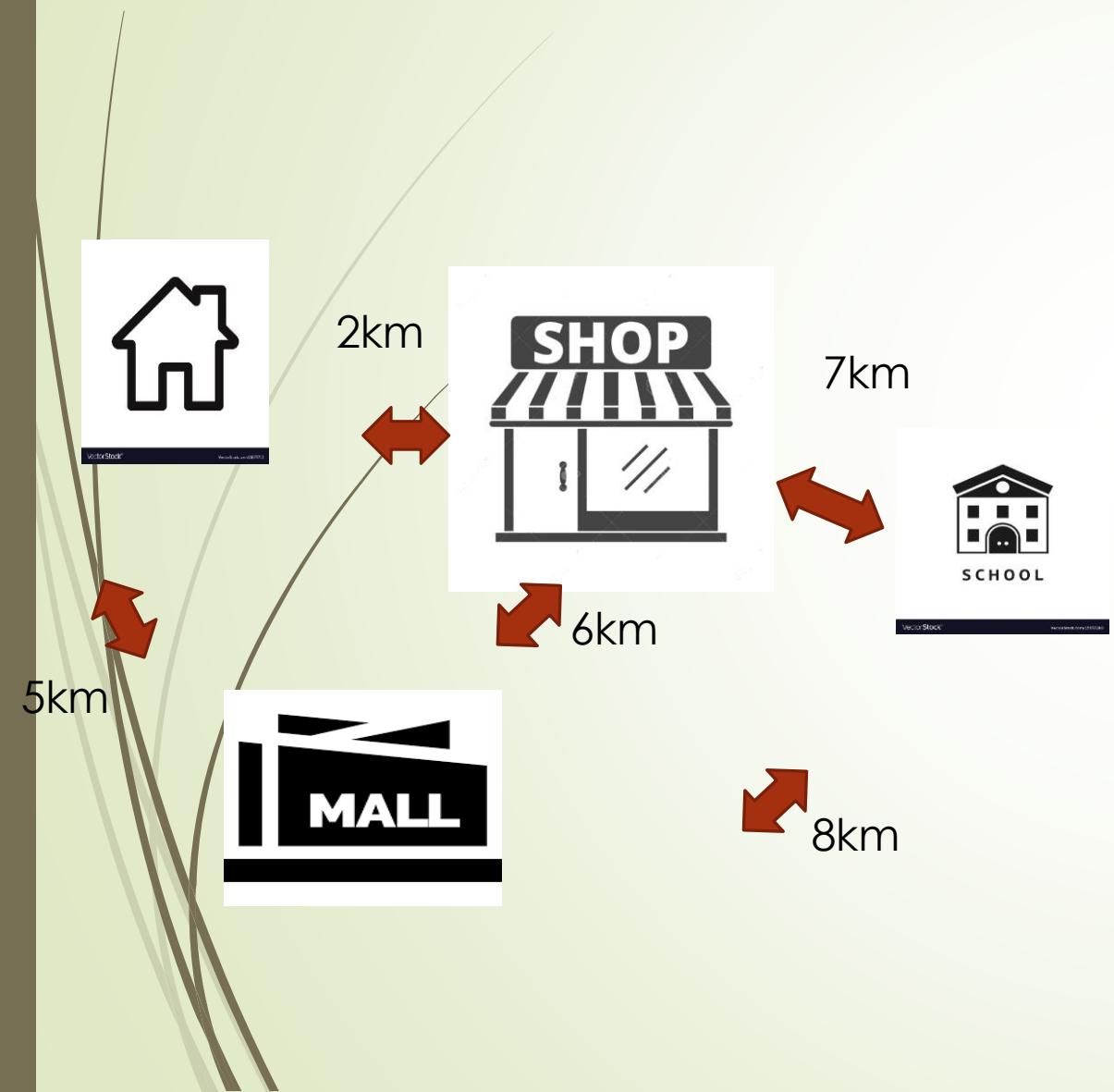
- *Operations which can be performed on Lists are:*
 - *Create a List*
 - *Insert elements, delete elements*
 - *destroy a List*
 - *Size is Not fixed size!!!!*

Example



	home	Mall	shop	school
home	0	5	2	9
Mall	5	0	6	8
shop	2	6	0	7
school	9	8	7	0

Example



	home	Mall	shop	school
home	0	5	2	infinity
Mall	5	0	6	8
shop	2	6	0	7
school	infinity	8	7	0

What is the shortest distance between home and school?

Algorithm to answer question

- Home to school direct path does not exist according to our convention
- So calculate the distance to school from home via shop
- Calculate the distance to school from home via mall
- Find smallest distance among these 2

Why study data structure and algorithms

- You know computer is a data processor. Without storing and knowing how to store data in different ways how u will process?
- How u will decide which approach is better ?
- How u will decide what data structure to use?
- Which processing method should be used?
-
- So, It is very very important to know data structures

MCQ's

► The meaning of data is

- A. array B. values C. organization D. Integer

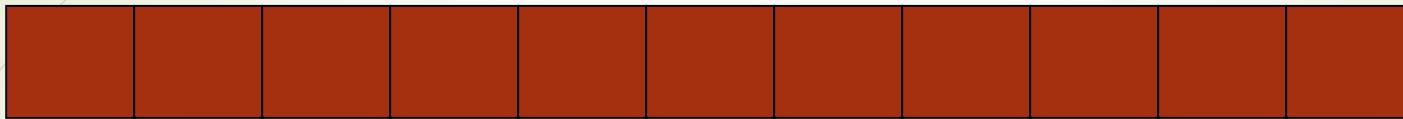
► The meaning of structure is

- A. Storing B. Organizing C. Different ways D. None of these

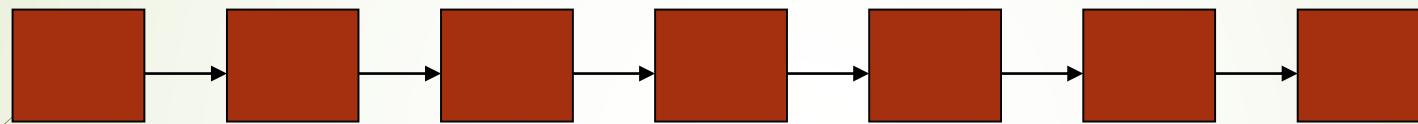
► Float is -----

- A. Value B. data type C. both A and B D. none of these

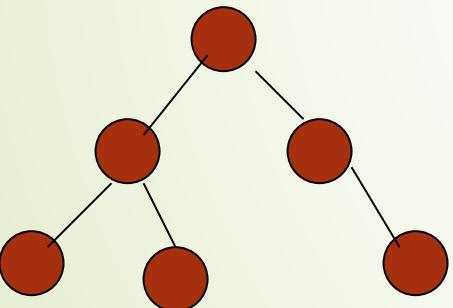
Types of data structures.



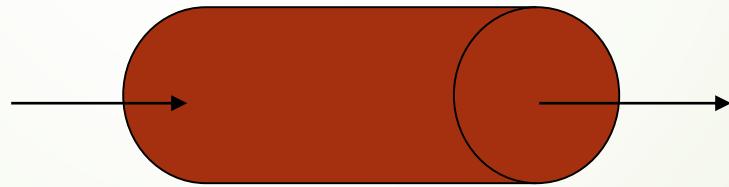
Array



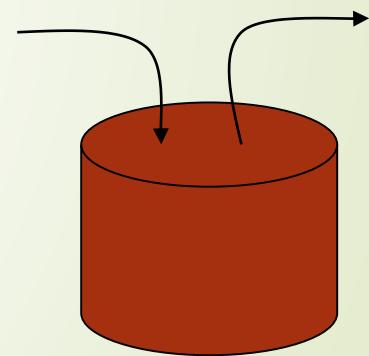
Linked List



Tree



Queue



Stack

The need for data structures.

- 🔔 **Goal:** to organize data
- 🔔 **Criteria:** to facilitate efficient
 - storage of data
 - retrieval of data
 - manipulation of data

This is the main motivation to learn and understand data structures.



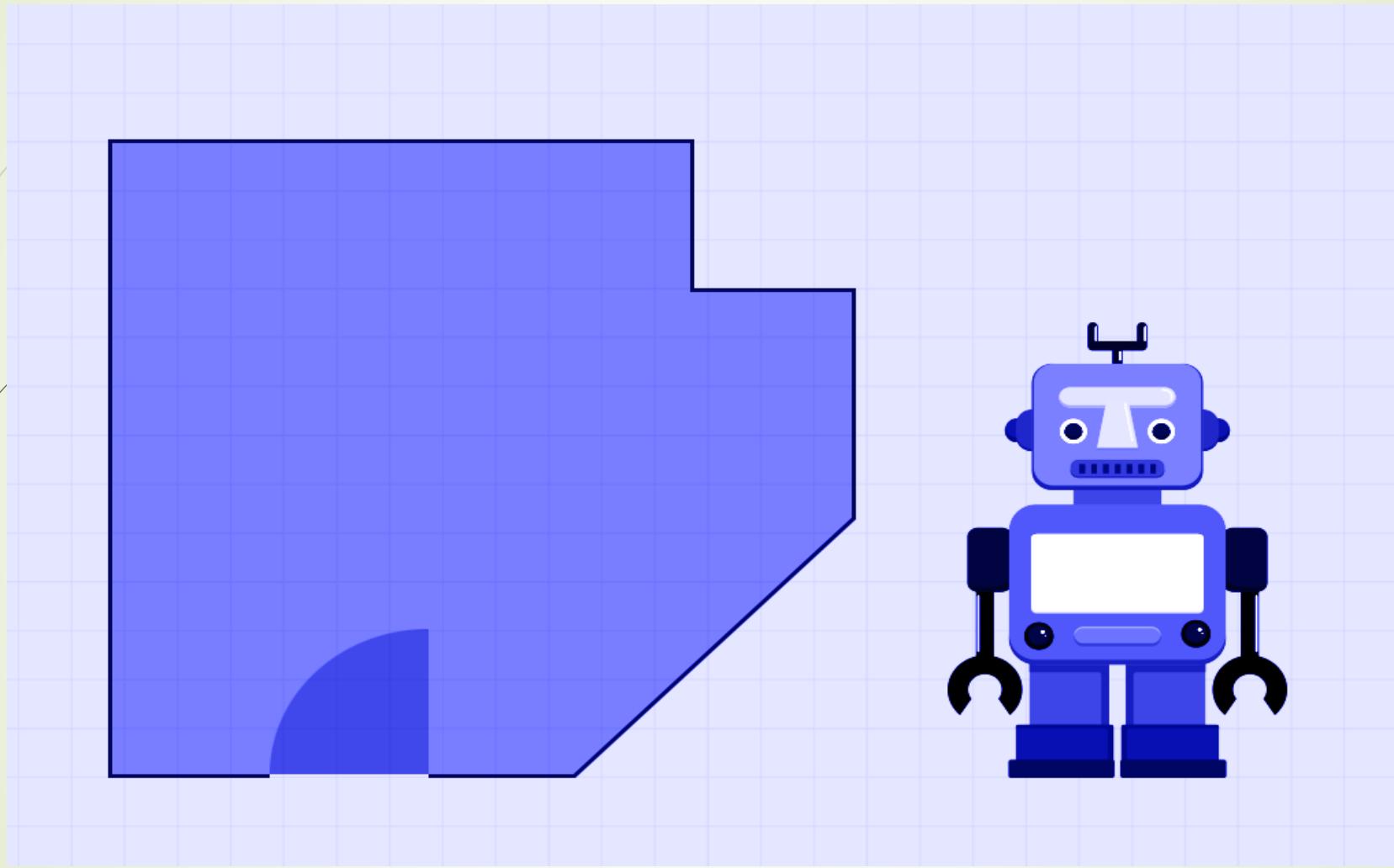
Selecting a Data Structure

- ▶ Analyze the problem to determine the resource constraints a solution must meet.
- ▶ Determine the basic operations that must be supported. Quantify the resource constraints for each operation.
- ▶ Select the data structure that best meets these requirements.

Data Structure Operations.

-  **Traversing:** Accessing each data element exactly once so that certain items in the data may be processed
-  **Searching:** Finding the location of the data element (key) in the structure
-  **Insertion:** Adding a new data element to the structure
-  **Deletion:** Removing a data element from the structure
-  **Sorting:** Arrange the data elements in a logical order (ascending/descending)
-  **Merging:** Combining data elements from two or more data structures into one

What is algorithm?



What is algorithm?

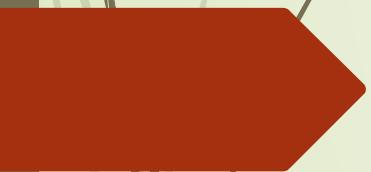
- 🔔 A finite set of instructions which accomplish a particular task
- 🔔 A method or process to solve a problem
- 🔔 Transforms input of a problem to output

Algorithm = Input + Process + Output

- 
- 🔔 Algorithm development is an art – **it needs practice, practice and only practice!**

What is a good algorithm?

- 🔔 It must be correct
- 🔔 It must be finite (in terms of time and size)
- 🔔 It must terminate
- 🔔 It must be unambiguous (Which step is next?)
- 🔔 It must be space and time efficient



A program is an instance of an algorithm, written in some specific programming language

A simple algorithm

🔔 Problem: Find maximum of a, b, c

🔔 Algorithm

- Input = a, b, c
- Output = max
- Process
 - Let max = a
 - If b > max then
 max = b
 - If c > max then
 max = c
 - Display max

Order is very important!!!

Algorithm development: Basics



Clearly identify:

- what output is required?
- what is the input?
- What steps are required to transform input into output
 - The most crucial bit
 - Needs problem solving skills
 - A problem can be solved in many different ways
 - Which solution, amongst the different possible solutions is optimal?

How to express an algorithm?

- 🔔 A sequence of steps to solve a problem
- 🔔 We need a way to express this sequence of steps
 - **Natural Language** (NL) or **Programming language** (PL) is another choice, but again not a good choice. Why?
 - Algorithm should be PL independent
 - We need some balance
 - We need PL independence
 - We need clarity
 - **Pseudo-code** provides the right balance

What is Pseudo-code?

- Pseudo-code is a short hand way of describing a computer program
- Rather than using the specific syntax of a computer language, more general wording is used
- Mixture of NL and PL expressions, in a systematic way
- Using pseudo-code, it is easier for a non-programmer to understand the general workings of the program

Components of Pseudo-code.

1. Expressions

- Standard mathematical symbols are used
 - Left arrow sign (\leftarrow) as the assignment operator in assignment statements (equivalent to the `=` operator in C++)
 - Equal sign ($=$) as the equality relation in Boolean expressions (equivalent to the `==` relation in ++)
 - For example

Sum \leftarrow 0

Sum \leftarrow Sum + 5

What is the final value of sum?

Components of Pseudo-code.

2. Decision structures (if-then-else logic)

- if condition then true-actions [else false-actions]
- Use ***indentation*** to indicate what actions should be included in the true-actions and false-actions. For example:

if marks > 50 then

print “Congratulation, you are passed!”

else

print “Sorry, you are failed!”

end if

What will be the output if marks are equal to 75?

Components of Pseudo-code.

3. Loops (Repetition)

- Pre-condition loops: **While** loops
 - **while** condition **do** actions
 - Use indentation to indicate what actions to include in loop actions
 - For example

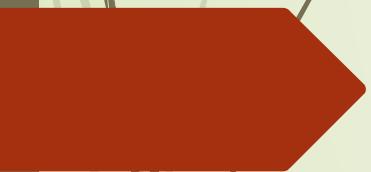
```
while counter < 5 do  
    print "Welcome to MCA4252!"  
    counter ← counter + 1  
end while
```

What will be the output if counter is initialised to 0, 7?

Components of Pseudo-code.

3. Loops (Repetition)

- Pre-condition loops: **For** loops
 - **for** variable-increment-definition **do** actions
 - Use indentation to indicate what actions to include in loop actions
 - For example



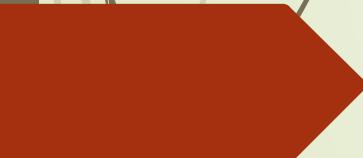
```
for counter ← 0; counter < 5; counter ← counter + 2 do
    print "Welcome to MCA4252!"
end for
```

What will be the output ?

Components of Pseudo-code.

3. Loops (Repetition)

- Post-condition loops: **Do** loops
 - **do** condition **while** actions
 - Use indentation to indicate what actions to include in loop actions
 - For example



do

print "Welcome to CS204!"

counter \leftarrow counter + 1

while counter < 5

What will be the output if counter is initialised to 0, 7?

Components of Pseudo-code.

4. Function declarations

- **return_type method_name (parameter_list)**
method_body

- For example

```
integer sum ( integer num1, integer num2)
```

```
start
```

```
result ← num1 + num2
```

```
end
```

Components of Pseudo-code.

4. Function Calls

- **object.function(arguments)**

- For example

mycalculator.sum(num1, num2)

Components of Pseudo-code.

4.

Function returns

- **return value**

- For example

```
integer sum ( integer num1, integer num2)
```

```
start
```

```
    result ← num1 + num2
```

```
    return result
```

```
end
```

Add Comments.

Algorithm Design: Practice

- ↳ **Example 1: Determining even/odd number**
 - ↳ A number divisible by 2 is considered an even number, while a number which is not divisible by 2 is considered an odd number. Write pseudo-code to display first N odd/even numbers.
- ↳ **Example 2: Computing Weekly Wages**

Gross pay depends on the pay rate and the number of hours worked per week. However, if you work more than 40 hours, you get paid time-and-a-half for all hours worked over 40. Write the pseudo-code to compute gross pay given pay rate and hours worked

Even/ Odd Numbers

Input range

```
for num←0; num<=range; num←num+1 do
    if num % 2 = 0 then
        print num is even
    else
        print num is odd
    endif
endfor
```

Computing weekly wages

```
Input hours_worked, pay_rate  
if hours_worked <= 40 then  
    gross_pay ← pay_rate × hours_worked  
else  
    basic_pay ← pay_rate × 40  
    over_time ← hours_worked - 40  
    over_time_pay ← 1.5 × pay_rate × over_time  
    gross_pay ← basic_pay + over_time_pay  
endfor  
print gross_pay
```

Homework

1. Write an algorithm to find the largest of a set of numbers. You do not know the count of numbers.
2. Write an algorithm in pseudocode that finds the average of (n) numbers.

For example: numbers are [4,5,14,20,3,6]

Analysis of Algorithms



How good is the algorithm?

- Correctness
- Time efficiency
- Space efficiency



Does there exist a better algorithm?

- Lower bounds
- Optimality

Thank You





DSE 2155 DATA STRUCTURES

[3104]

Linda Varghese
Savitha G

Department of Computer Applications, MIT, Manipal.



Classroom Rules of Engagement.

- 🔔 All laptop computers, mobile phones, tablet computers must be closed during all classroom hours **(offline class)**.
- 🔔 Computers distract the most people behind and around the user.
- 🔔 **Maintain social distance and wear masks (correctly) at all times inside and outside class.**
- 🔔 **Make your own notes**. Slides are not enough.
- 🔔 Self study is paramount.
- 🔔 **All homework** to be **completed** before class commences.

How do you improve your performance?

- ⌚ To transfer information from your short-term memory to your long-term memory, that information must be imposed on your mind *at least three times*.
- ⌚ You should always try the following:
 - ⌚ Look at the notes/slides (**IF ANY**) before class.
 - ⌚ Attend all lectures (if possible).
 - ⌚ Review the lecture during the evening.
 - ⌚ Rewrite and summarize the slides in your words.
- ⌚ In addition to this, you should:
 - ⌚ Get a reasonable nights sleep (apparently this is when information is transferred to your long-term memory), and
 - ⌚ Eat a good breakfast (also apparently good for the memory)

Agenda

- Syllabus abstract and Textbook
- Definitions of basic terms
- Example
- Why study Data structure and algorithms?
- What are the different classifications
- Some example data structures you study in this course

MAIN TOPICS:-

DSE- 2155 : DATA STRUCTURES [3 1 0 4]

Introduction, Programming fundamentals, Recursion, Stacks, Queues and their applications, Sparse Matrix, Pointers and dynamic memory allocation,

Linked Lists: Singly linked lists, Dynamically Linked Stacks and Queues, Polynomial representation and polynomial operations using singly linked list, Singly Circular Linked List, Doubly Linked Lists,

Trees: Binary trees, Heaps, Binary Search Trees, Threaded binary trees,

Graphs: Terminologies, Depth First Search, Breadth First Search, Sorting and searching Techniques.

Text books

- Ellis Horowitz, Sartaj Sahni, Dinesh Mehta, Fundamentals of Data Structures in C++, 2nd Edition.
- Ellis Horowitz, Sartaj Sahni, Dinesh Mehta, Fundamentals of Data Structures in C, 2nd Edition.
- Behrouz A. Forouzan, Richard F. Gilberg, A Structured Programming Approach Using C, 3e, Cengage, Learning India Pvt.Ltd, India,2007.
- Behrouz A. Forouzan, Richard F. Gilberg, Data Structures, A Pseudocode approach Using C, 2e, Cengage, learning India Pvt.Ltd, India, 2009.
- Debasis Samanta, Classic Data structures- 2nd edition, PHI Learning Private Limited , 2010

Language of Implementation.

- ⌚ You will be using the C++ programming language in this course
 - ⌚ Knowledge of syntax of C++ pre-requisite

```
#include <csfrio.h>
int main(void)
{
    int count;
    for(count = 1; count <= 500; count++)
        printf("I will not throw paper airplanes in class.");
    return 0;
}
```

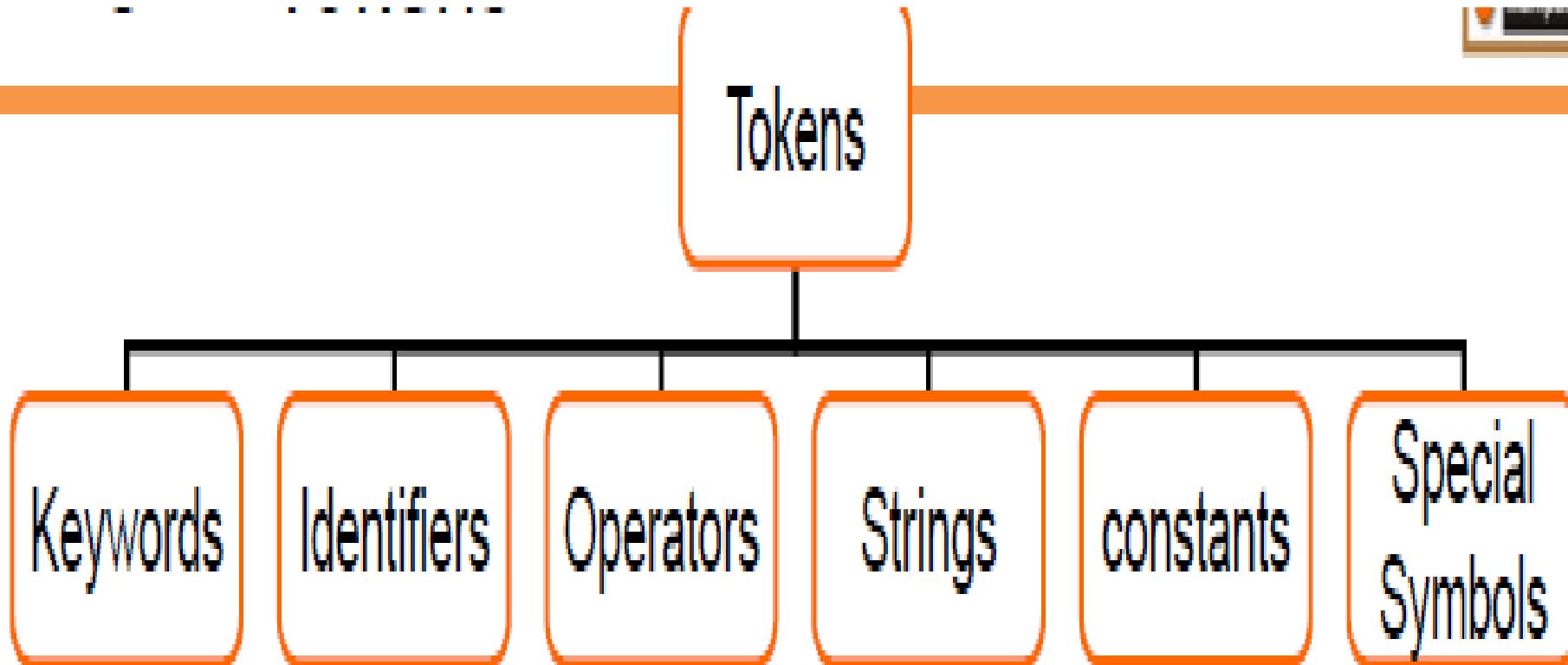
AMEND 10-3





INTRODUCTION TO BASICS OF PROGRAMMING

C++ Tokens



C++ Tokens

- (i) **Keywords**  words that are basically sequence of characters defined by a computer language and have one or more fixed meanings. They are also called as ***reserved words***. Key words cannot be changed. ex. Int, float, do-while, if, else,.....
- (ii) **Identifiers**  words which have to be identified by keywords. user defined names. ex. int amount, float avg,.....
- (iii) **Operators**  +, -, *, %, /,
- (iv) **Strings**  “Manipal”
- (v) **Constants**  -15, 10
- (vi) **Special Symbols**  { } (,.....

C++ Data Types

User Defined Type
Structure, Union, Class
enumeration

Built-in-type

Derived Type
Array ,Function, pointer

Integral type

void

Floating Type

int

float

char

double

```
// my first program in C++
```

Comments

- single line comments: `//.....`
- multiple lines : `/*.....*/`
- All lines beginning with two slash signs (`//` or `/*`) are considered as comments
- They do not have any effect on the behavior of the program
- The programmer can use them to include short explanations or observations within the source code itself.

#include <iostream>

- Lines beginning with a sign (#) are directives for the **preprocessor**.
- They are not regular code lines.
directive `#include <iostream>` tells the preprocessor to include the **iostream** standard header file.
- This specified file **(iostream) includes the declarations of the basic standard input-output library in C++**, and it is included because its functionality is going to be used later.

main()

- The **main function** is the point where all C++ programs start their execution, independently of its location within the source code.
- It is **essential** that all C++ programs have a main function.
- The word main is followed in the code by a pair of **parentheses ()**. That is because it is a **function declaration**.
- Optionally, these parentheses may enclose a list of parameters within them.
- Right after these parentheses we can find the body of the main function enclosed in **braces{ }**.

Simple C++ Program

```
// Display “This is my first C++ program”
// Single line comment
#include <iostream>
Using namespace std; // preprocessor directive
main( ) // Entry point for program execution
{Begin // block of statements:
cout << “This is my first C++ program”; // block of statements:
End}
```

Simple C++ Programs..

```
#include<iostream>
using namespace std;
int main() {
    cout<<"Enter Roll Number and marks of three subjects";
    int RollNo,marks1,marks2,marks3;
    float minimum = 35.0;
    cin>>marks1>>marks2>>marks3;
    avg = (marks1+marks2+marks3)/3;
    if (avg < minimum )
        cout<<RollNo<<"fail";
    else
        cout<<RollNo<<"pass";
    return 0;
}
```

Program to read and display a number

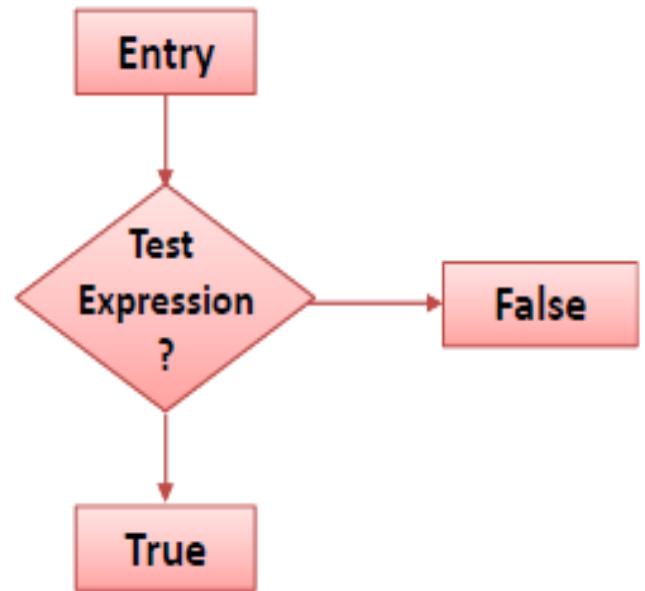
```
#include<iostream>
using namespace std;
int main() {                                //program body begins
    int number;                            //variable declaration
    cout<<"enter number";   // user friendly info display
    cin>>number;                           // reading or input value to the variable
    cout<<"\nthe no. is\n"<<number;  //writing or output variable value
    return 0;
} // end of program
```

C++ decision making and branching statements

- 1. if Statement**
- 2. switch statement**

if statement

- Used to control the *flow of execution* of statements.
- It's a *Two-way decision statement*, used in conjunction with an expression.
- It takes the form: **if(test expression)**
- It allows to *evaluate the expression first* and then, depending on the value; **true** or **false**, it transfer the control to a particular statement.
- i.e. **Two-way branching** as shown in figure



Different forms of if statement

- 1. Simple if statement.**
- 2. if...else statement.**
- 3. Nested if...else statement.**
- 4. else if ladder.**

Simple if Statement

General form of the simplest if statement:

if (*test Expression*)

{

statement-block;

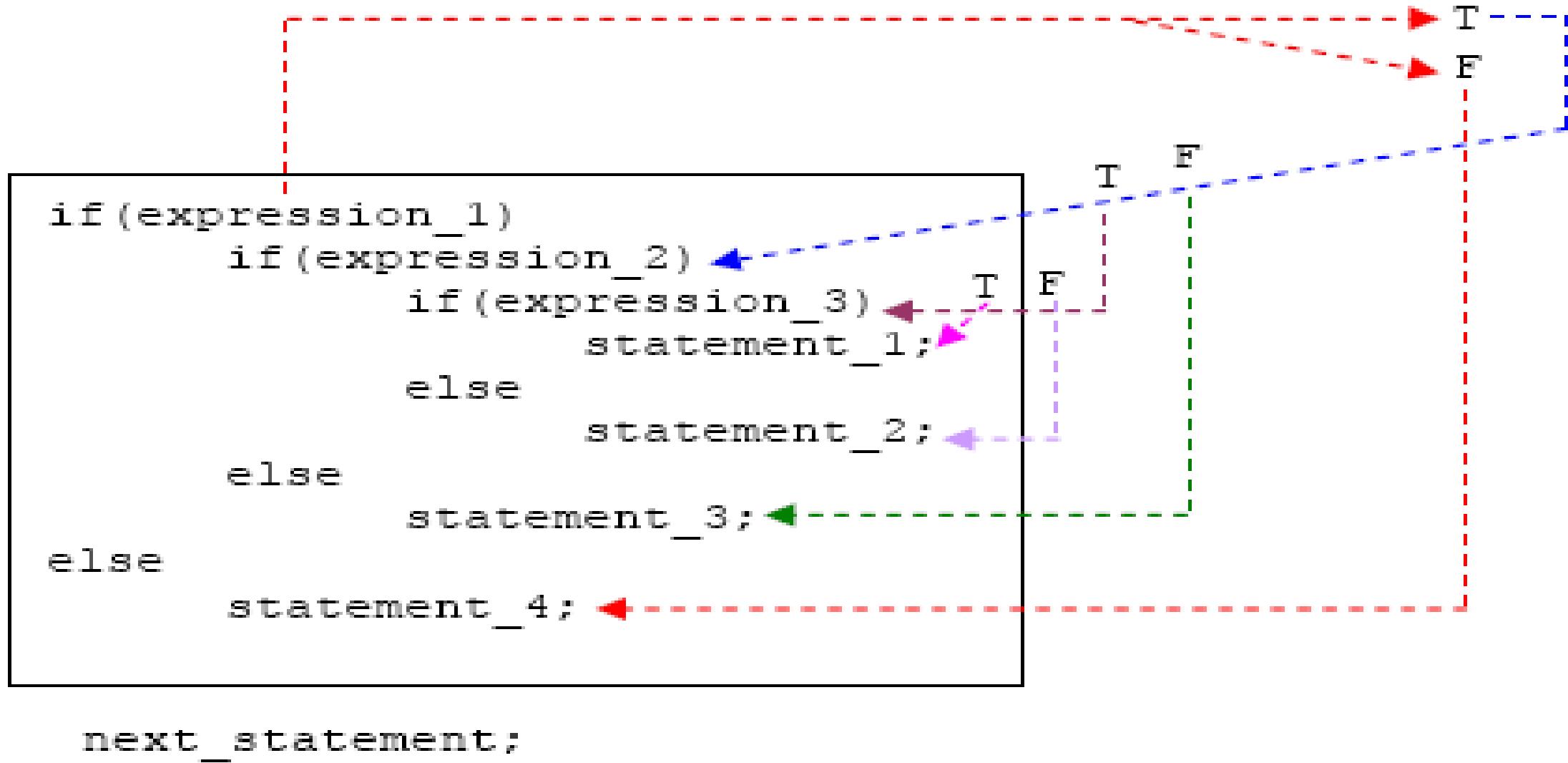
}

statement_x;

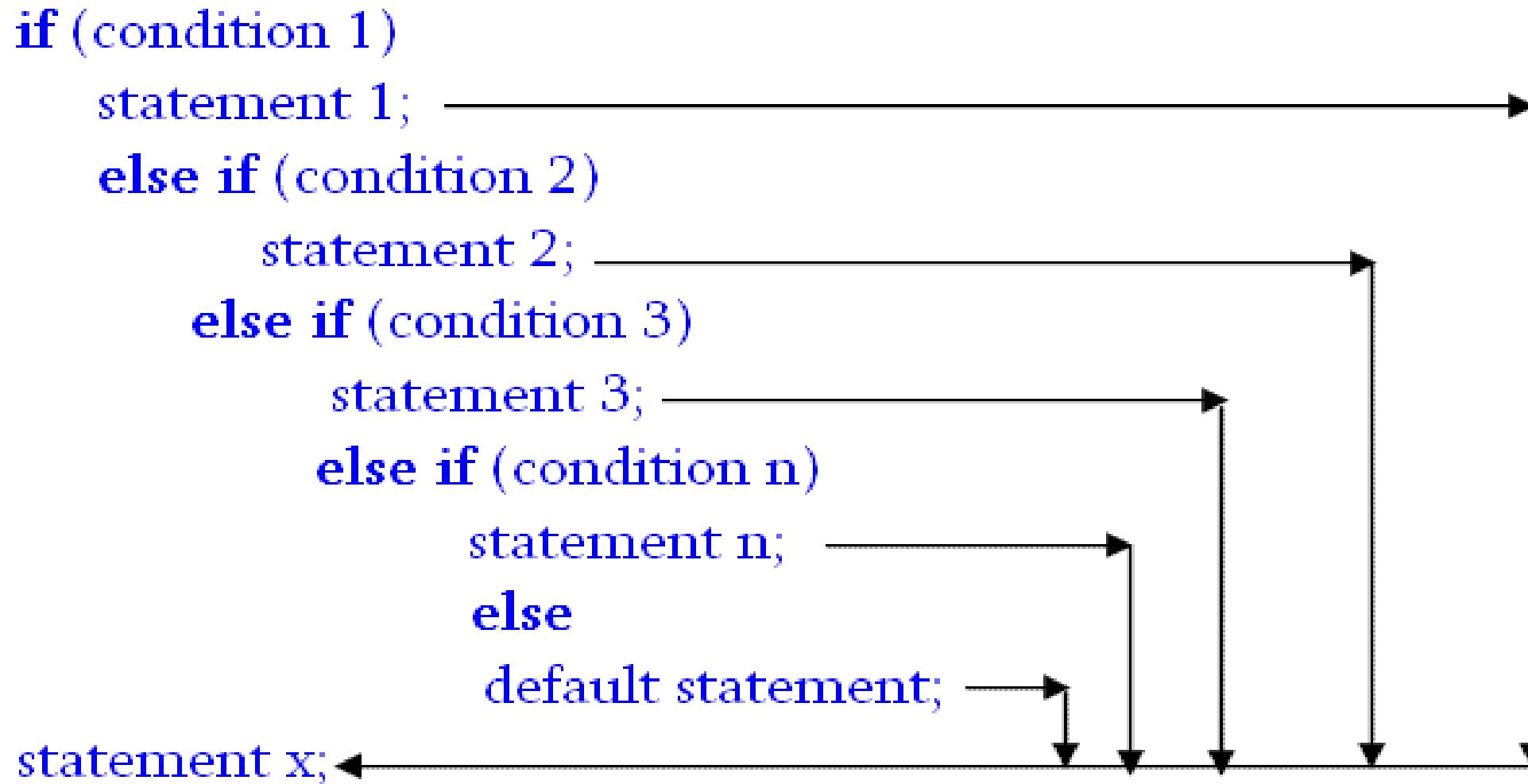
If else statement

```
if (condition)
{
    // block of code to be executed if the condition is true
} else
{
    // block of code to be executed if the condition is false
}
```

Nesting of if-else Statements



The else if Ladder



The switch Statement

- Switch is multiple–branching statement- based on a condition, the control is transferred to one of the many possible points.
- The most flexible control statement in selection structure of program control.
- Enables the program to execute different statements based on an expression that can have more than two values. Also called multiple choice statements.

General form:

```
switch(expression)
{ case value_1 : statement(s);
    break;
  case value_2 : statement(s);
    break; ... case value_n : statement(s);
    break;
  default : statement(s); }
    next_statement;
```

switch- example

```
index=mark/10;
switch (index)
{
case 10:
case 9:
case 8: grade='A';
          break;
case 7:
case 6:
          grade='B';
          break;
case 5:
          grade='C'
          break;
case 4:
          grade='D'
          break;
default: grade='F';
          break;
} cout<<grade;
```

DECISION MAKING AND LOOPING CONTROL STRUCTURES

- Iterative (repetitive) control structures are used to **repeat certain statements for a specified number of times**.
- The statements are executed as long as the **condition is true**
- These kind of control structures are also called as **loop control structures**
- Three kinds of loop control structures:
 - **while**
 - **do while**
 - **for**

While statement

Basic format:

while (test condition)

{

body of the loop

}

- **Entry controlled** loop statement
- Test condition is evaluated & if it is true, then body of the loop is executed.
- After execution, the test condition is again evaluated & if it is true, the body is executed again.
- This is **repeated until the test condition becomes false**, & control transferred out of the loop.
- **Body of loop may not be executed if the condition is false at the very first attempt.**

The do statement

General form:

```
do
{
    body of the loop
}
while (test condition);
```

The for statement

The general form:

for (initialization; test condition; increment)

{

Body of the loop

}

Next statement;

Nesting of for loop

One **for statement** within another **for statement**.

```
for (i=0; i< m; ++i)
```

```
{.....
```

```
....
```

```
for (j=0; j < n; ++j)
```

```
{.....
```

```
.....
```

```
}
```

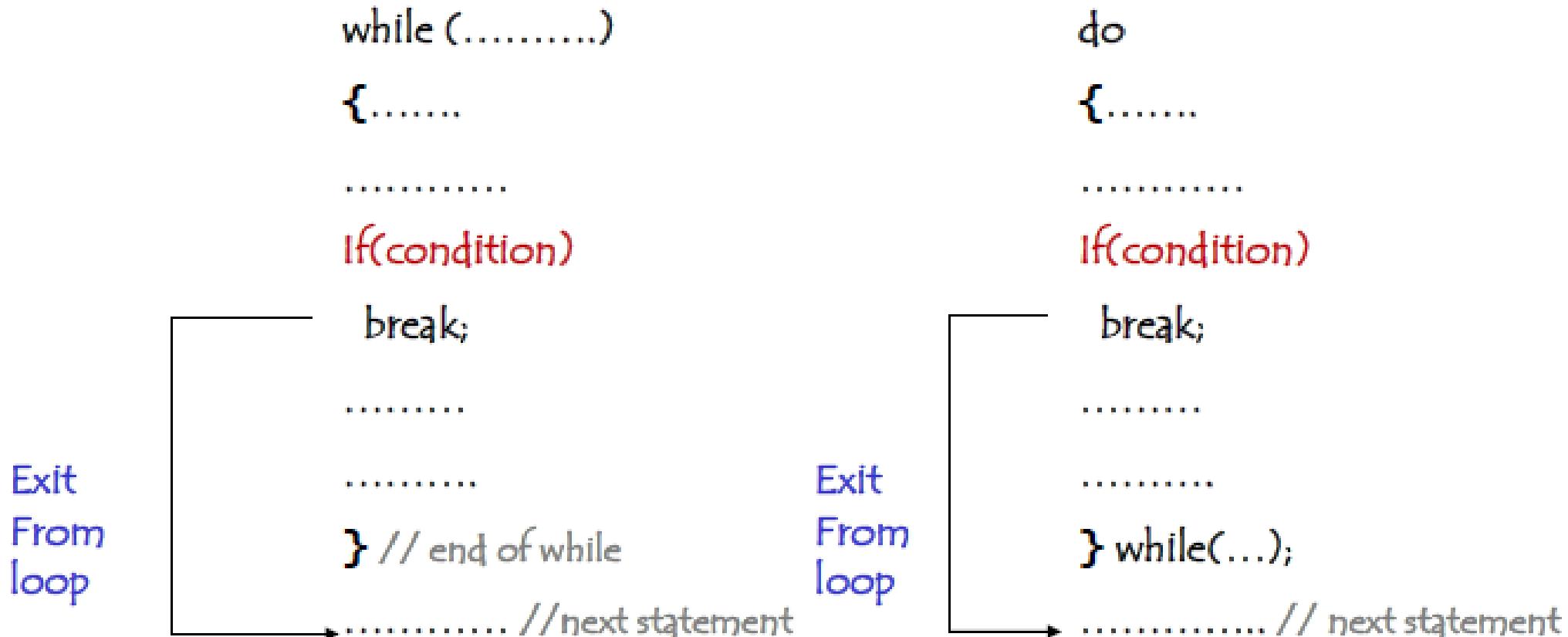
 // end of inner 'for' statement

```
// end of outer 'for' statement
```

Jumping out of a loop

- An early exit from a loop can be accomplished by using the **break** statement.
- When the break statement is encountered inside a loop, the loop is immediately exited & the program continues with the statement immediately following the loop. When the loops are nested , the break would only exit from the loop containing it.
i.e., the break will exit only a single loop.

Exiting a loop with break statement



Skipping a part of loop

- Skip part of the body of loop under certain conditions using continue statement.
- As the name implies, causes the loop to be continued with next iteration, after skipping rest of the body of the loop.

```
→ while (.....)
  {
    .....
    .....
    If(condition)
      continue;
    .....
    .....
  }
```

```
→ do
  {
    .....
    .....
    If(condition)
      continue;
    .....
    .....
  } while(...);
```

Tutorial

- Check if a given number is prime or not
- Factorial of given 10 numbers(do not use arrays)
- Print all odd numbers between m and n
- Menu driven program to sum all elements entered upto -1
- Find $\sin(x)$ using series
- Find $\cos(x)$ using series
- Find e^x using series
- Print triangle in the following form using loops until n.
Ex. If n=6

1		
2	3	
4	5	6

Arrays

- An array is a group of related data items that share a common name.
- The array elements are placed in a contiguous memory locations.
- A particular value in an array is indicated by writing an integer number called index number or subscript in square brackets after the array name.
- The least value that an index can take in array is 0.

Array Declaration:

type name [size];

- where type is a valid data type (like int, float...)
- Name is a valid identifier & size specifies how many elements the array has to contain.
- Size field is always enclosed in square brackets [] and takes static values.

Example:

- **an array salary containing 5 elements is declared as follows**
int salary [5];

One Dimensional Array

- A linear list of fixed number of data items of same type.
- These items are accessed using the same name using a single subscript. E.g. salary [1], salary [4]
- A list of items can be given one variable name using only one subscript and such a variable is called a single-subscripted variable or a one dimensional array.

- Initializing one-dimensional array (compile time)

type array-name [size]={list of values}

Type → basic data type

Array-name → name of the array.

Size → maximum number of elements and may be omitted.

List of values → values separated by commas.

- E.g. `int number[3] = { 0,0,0} or {0}` will declare the variable number as an array of size 3 and will assign 0 to each element

2 – D Arrays

- It is an ordered table of homogeneous elements.
- It can be imagined as a two dimensional table made of elements, all of them of a same uniform data type.
- It is generally referred to as **matrix**, of some rows and some columns.
- It is also called as a **two-subscripted variable**.

- For example

```
int marks[5][3];
float matrix[3][3];
char page[25][80];
```

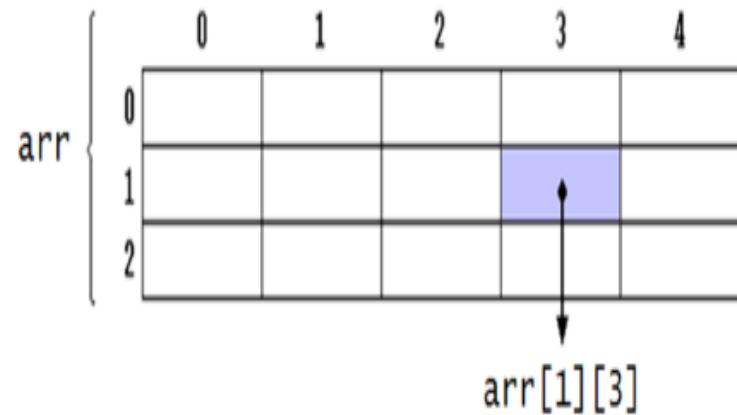
- The first example tells that marks is a 2-D array of 5 rows and 3 columns.
- The second example tells that matrix is a 2-D array of 3 rows and 3 columns.
- Similarly, the third example tells that page is a 2-D array of 25 rows and 80 columns.

- Declaration

type array_name[row_size][column_size];

- For example,

int arr [3][5];



`arr` represents a two dimensional array or table having 3 rows and 5 columns and it can store 15 integer values.

Read a matrix example

```
void main()
{
int i,j,m,n,a[100][100];
clrscr();
cout<<"enter dimension for a:";
cin>>m>>n;
cout<<"\n enter elements for a \n";
for(i=0;i<m;i++)
{
for(j=0;j<n;j++)
cin>>a[i][j];
}
for(i=0;i<m;i++)
{
for(j=0;j<n;j++)
cout<<"\t"<<a[i][j];
cout<<"\n";
}
getch();
}
```

Declaring a 3D array

- Specify data type, array name, block size, row size and column size.
- Each subscript can be written within its own separate pair of brackets.
- **Syntax: `data_type array_name[block_size][row_size][column_size];`**

Example:

```
int arr[2][3][3]; //array of type integer  
                    //number of blocks of 2D arrays:2 | rows:3 | columns:3  
                    //number of elements:2*3*3=18
```

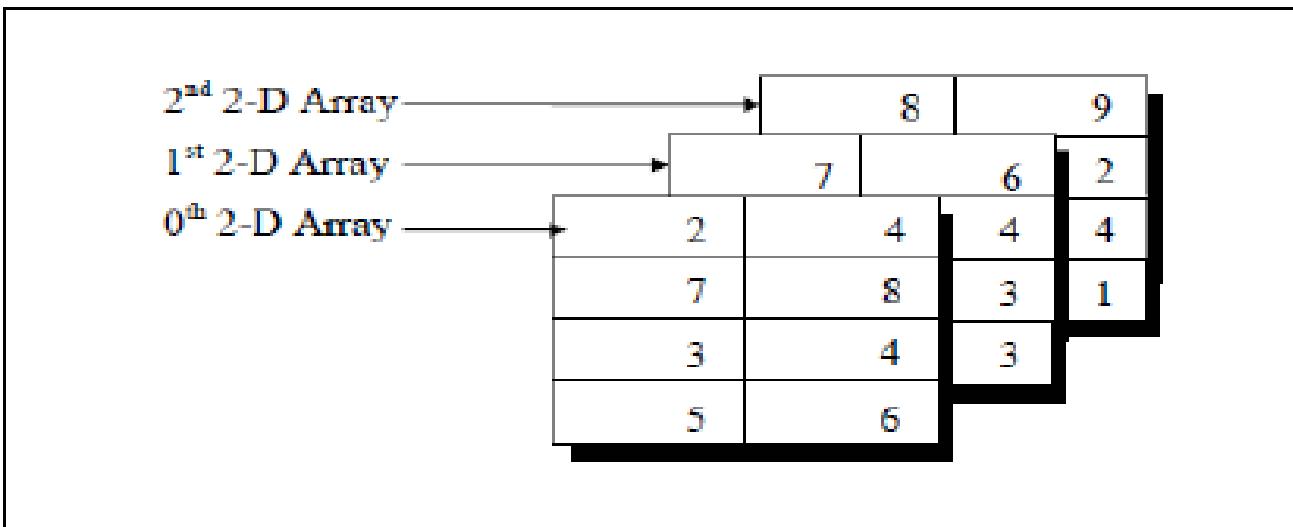
Output:

block(1) 11 22 33	block(2) 12 13 14
44 55 66	21 31 41
77 88 99	12 13 14
3x3	3x3

Multi – dimensional arrays

```
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.



Conceptual
View



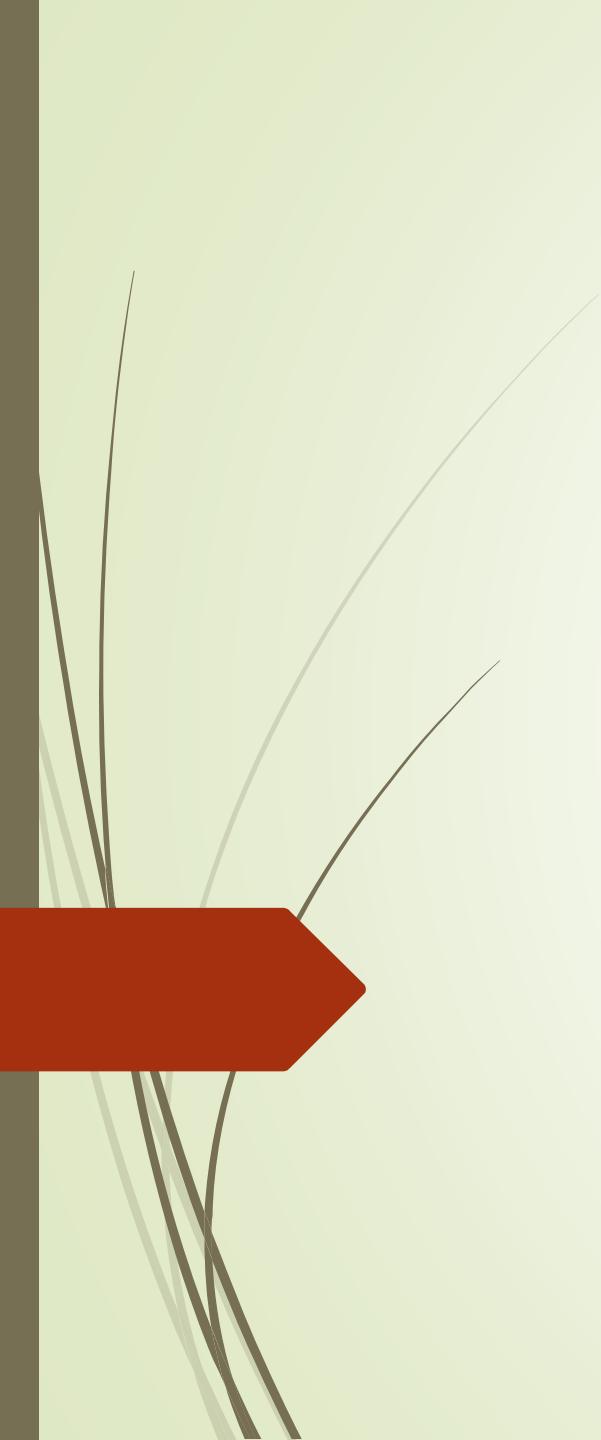
PROGRAMMING AND DATA STRUCTURES



Introduction to three dimensional
Arrays



STRINGS



String Definition

- A string is an array of characters.
- Any group of characters (except double quote sign) defined between double quotation marks is a constant string.
- Character strings are often used to build meaningful and readable programs.
- The common operations performed on strings are
 - Reading and writing strings
 - Combining strings together
 - Copying one string to another
 - Comparing strings to another
 - Extracting a portion of a string ..etc.

- 
- ▶ Declaration and initialization

char string_name[size];

The size determines the number of characters in the string_name.

- ✓ The character sequences "Hello" and "Merry Christmas" represented in an array **name** as follows :

name

H e l l o \o

Merry Christmas

Example

```
#include <iostream.h>
void main ()
{
    char question[] = "Please, enter your first name: ";
    char greeting[] = "Hello, ";
    char yourname [80];
    cout << question;
    cin >> yourname;
    cout << greeting << yourname << "!";
    getch();
}
```

Reading Multiple Lines

- ▶ We use third argument in **cin.get()** function
- ▶ **cin.get(array_name, size, stop_char)**
- ▶ The argument **stop_char** specifies the character that tells the function to stop reading. The default value for this argument is the newline ('`\n`') character, but if you call the function with some other character for this argument, the default will be overridden by the specified character.

Example

```
#include <iostream>
using namespace std;
int main(){
    const int len = 80;           //max characters in string  char str[MAX];
                                //string variable str
    cout << "\nEnter a string:";
    char str[len];
    cin.get(str, len);
    cout << "\n" << str;
    return 0
}
```

Length of string

```
#include <iostream>
using namespace std;
int main(){
    char a[30];
    int i, c=0;
    cout<<"Enter a string:";
    cin.get(a,30);
    for(i=0;a[i]!='\0';i++)
        c++;
    cout<<"\nLength of the string "<<a<<" is "<<c;
    return 0;
}
```

String concatenation

```
#include<iostream>
using namespace std;
int main() {
    char str1[55],str2[25];
    int i=0,j=0;
    cout<<"\nEnter First String:";
    cin>>str1;
    cout<<"\nEnter Second String:";
    cin>>str2;
    while(str1[i]!='\0')
        i++;
```

```
while(str2[j]!='\0')
{
    str1[i]=str2[j];
    j++;
    i++;
}
str1[i]='\0';
cout<<"\nConcatenated String
is\n"<<str1;
return 0;}
```

Library functions: String Handling functions(built-in)

These in-built functions are used to manipulate a given string. These functions are part of **string.h** header file.

- **strlen ()**
✓ gives the length of the string. E.g. **strlen(string)**
- **strcpy ()**
✓ copies one string to other. E.g. **strcpy(Dstr1,Sstr2)**
- **strcmp ()**
✓ compares the two strings. E.g. **strcmp(str1,str2)**
- **strcat ()**
✓ Concatinate the two strings. E.g. **strcat(str1,str2)**

Library function: **strlen()**

- String length can be obtained by using the following function

n=strlen(string);

This function counts and returns the number of characters in a string, where n is an integer variable which receives the value of the length of the string. The argument may be a string constant.

- Copying a String the Hard Way

The best way to understand the true nature of strings is to deal with them character by character. The following program copies one string to another character by character.

Copies a string using a for loop

```
#include <iostream>
#include<string.h>
using namespace std;
int main()
{
    char str1[ ] = "Manipal Institute of Technology";
    char str2[100];
    int j;
    for(j=0 ; j<strlen(str1); j++)
        str2[j] = str1[j];
    str2[j] = '\0';
    cout << str1 << "\n" << str2 << endl;
    return 0;
}
```

Copies a string using a for loop

- ▶ The copying is done one character at a time, in the Statement
`str2[j] = str1[j];`
- ▶ The copied version of the string must be terminated with a null.
- ▶ However, the string length returned by **`strlen()`** does not include the null.
- ▶ We could copy one additional character, but it's safer to insert the null explicitly. We do this with the line **`str2[j] = '\0';`**
- ▶ If you don't insert this character, you'll find that the string printed by the program includes all sorts of weird characters following the string you want.
- ▶ The **<<** just keeps on printing characters, whatever they are, until by chance it encounters a '**`\0`**'.

Library function: **strcpy()**

Copying a String the Easy Way: **strcpy(destination, source)**

- The strcpy function works almost like a string assignment operator and assigns the contents of source to destination.
- ✓ Destination may be a character array variable or a string constant.
e.g., **strcpy(city,"DELHI");**
will assign the string “DELHI” to the string variable city.
- ✓ Similarly, the statement **strcpy(city1,city2);**
will assign the contents of the string variable city2 to the string variable city1.
Note:- The size of the array city1 should be large enough to receive the contents of city2.

strcpy(): Example

```
#include <iostream>
#include<string.h>
using namespace std;
int main()
{
    char str1[ ] = "Manipal Institute of Technology";
    char str2[100];
    strcpy(str2,str1);
    cout << str1 << "\n" << str2 << endl;
    return 0;
}
```

Library function: **strcmp()**

The **strcmp** function compares two strings identified by the arguments and **has a value 0 if they are equal.**

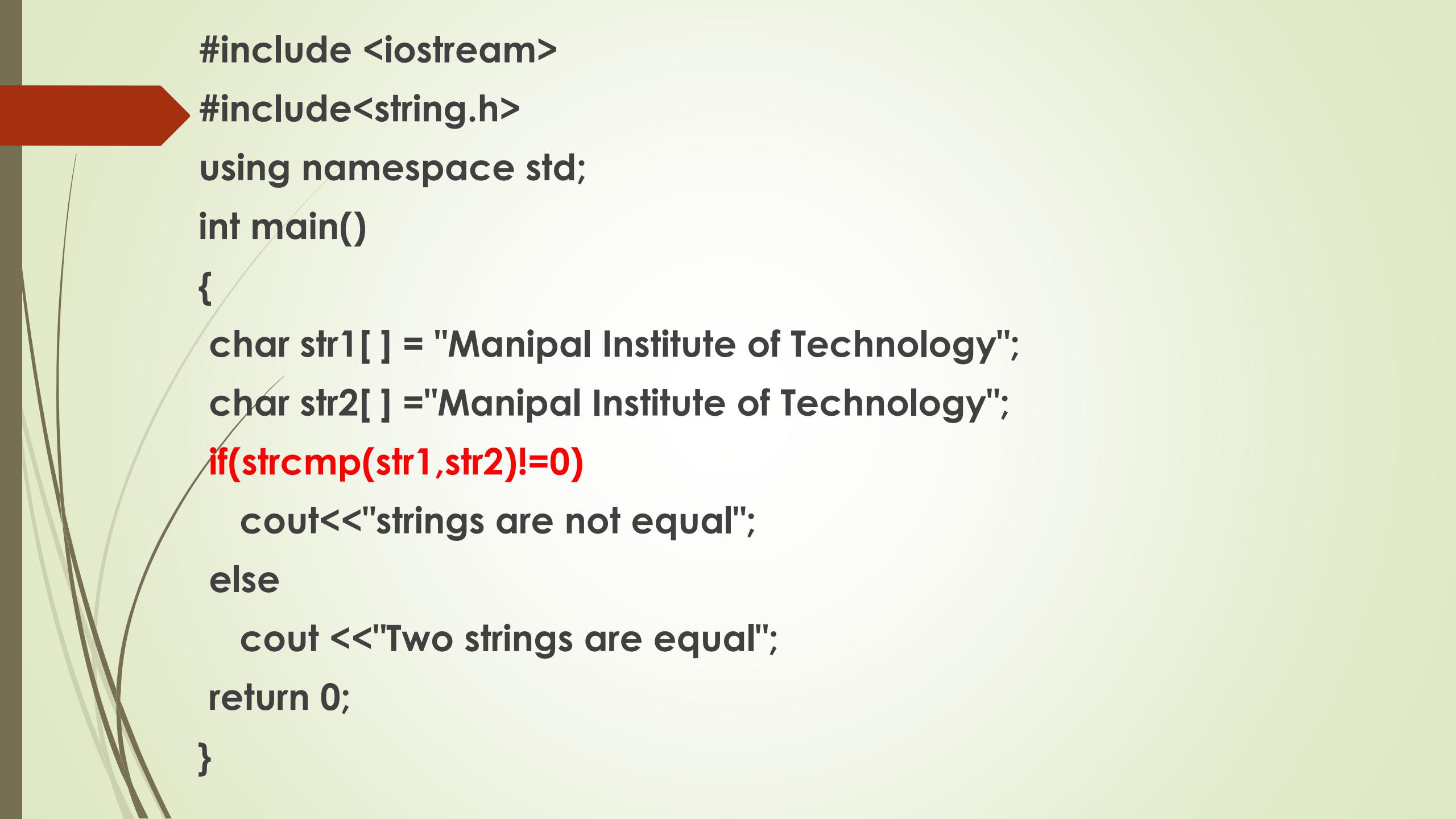
- If they are not, it has the numeric difference between the first non matching characters in the strings.

strcmp(string1,string2);

String1 and string2 may be string variables or string constants.

e.g., **strcmp("their","there");** will return a value of -9 which is the numeric difference between ASCII "i" and ASCII "r". That is, "i" minus "r" with respect to ASCII code is -9.

Note:- If the value is negative, string1 is alphabetically above string2.



```
#include <iostream>
#include<string.h>
using namespace std;
int main()
{
    char str1[ ] = "Manipal Institute of Technology";
    char str2[ ] ="Manipal Institute of Technology";
    if(strcmp(str1,str2)!=0)
        cout<<"strings are not equal";
    else
        cout <<"Two strings are equal";
    return 0;
}
```

Library function: **strcat()**

The **strcat function** joins two strings together.

- It takes the following form:

► **strcat(string1,string2);**

- string1 and string2 are character arrays.

- ✓ When the function **strcat** is executed, string2 is appended to a string1.
- ✓ It does so **by removing the null character at the end of string1 and placing string2 from there.**
- ✓ The string at string2 remains unchanged.



```
#include <iostream>
#include<string.h>
using namespace std;
int main()
{
    char str1[] = "Manipal";
    char str2[] = " Institute of Technology";
    strcat(str1,str2);
    cout<<"concatenated string is \n"<<str1;
    return 0;
}
```

Reading integer followed by sentences

```
#include <iostream>

using namespace std;

//reading integer followed by sentences

int main(){

    int n;

    char s1[10],s2[10];

    cout<<"Enter integer";

    cin>>n;

    fflush(stdin);

    gets(s1);

    fflush(stdin);

    gets(s2);

    cout<<"s1="<<s1<<endl;

    cout<<"s2="<<s2<<endl;

    return 0;
}
```



Reading integer followed by multiline input strings

```
#include <iostream>
using namespace std;

int main(){
    int n;    char s1[10],s2[10];
    cout<<"Enter integer";
    cin>>n;
    fflush(stdin);
    cin.get(s1,10,'$');
    fflush(stdin);
    cin.get(s2,10,'$');
    cout<<"s1="<<s1<<endl;
    cout<<"s2="<<s2<<endl;
    return 0;
}
```

Insert a substring into a given string:

Steps to be followed:

- 1. Take a string, substring and position where the substring is to be inserted in main string as input.
- 2. Copy all the characters from the given position to a temporary array.
- 3. Substring should be inserted at the given position character by character to main string.
- 4. Now to this array copy all characters from the temporary array.

Delete substring

1. Take a string and its substring as input.
2. Take two control variable “i” for main string and “j” for substring
3. Compare $m[i]$ and $s[j]$.
 - If match is found increment both control variables.
 - If match is not found put $m[i]$ into the new array $n[k]$ and increment i .
 - Keep a flag variable to ensure that complete substring is present in the main string
4. If the end of substring is reached update the index j to 0 and again compare $m[i]$ with $s[j]$ and so on....



THANK YOU

Modular Programming

Lengthier programs

- Prone to errors
- tedious to locate and correct the errors

To overcome this

Programs broken into a number of smaller logical components, each of which serves a specific task.

Modularization

- ◆ Process of splitting the lengthier and complex programs into a number of smaller units is called **Modularization**.
- ◆ Programming with such an approach is called **Modular programming**

Advantages of modularization

- Reusability
- Debugging is easier
- Build library

Functions

- ◆ A *function* is a set of instructions to carryout a particular task.
- ◆ Using functions we can structure our programs in a more modular way.

Functions

- ◆ Standard functions
(library functions or built in functions)
- ◆ User-defined functions
Written by the user(programmer)

Defining a Function

✓ Name

- You should give functions descriptive names
- Same rules as variable names, generally

✓ Return type

- Data type of the value returned to the part of the program that activated (called) the function.

Functions

✓ Parameter list

- A list of variables that hold the values being passed to the function

✓ Body

- Statements enclosed in curly braces that perform the function's operations(tasks)

The general form of a function definition

```
return_type function_name(parameter_definition)
{
    variable declaration;
    statement1;
    statement2;
    .
    .
    .
    .
    .
    return(value_computed);
}
```

Functions: understanding

The diagram illustrates the structure of a C++ main function. It shows the declaration `void main (void)` with three parts labeled: "Return type" pointing to `void`, "Function name" pointing to `main`, and "Parameter List" pointing to `(void)`. Below this, the function body is shown as `{ cout << "hello world\n"; }`. A large brace on the right side groups the code block and is labeled "Body".

```
void main (void){    cout << "hello world\n";}
```

Functions

```
Return type      Function name      Parameter List  
void DisplayMessage(void)  
{   cout << "Hello from the function"  
    << "DisplayMessage.\n";  
}  
} Body
```

```
void main()  
{ cout << "Hello from main";  
    DisplayMessage(); // FUNCTION CALL  
    cout << "Back in function main again.\n";  
}
```

Functions

```
Return type      Function name      Parameter List  
void DisplayMessage(void)  
{ cout << "Hello from the function"  
  << "DisplayMessage.\n";  
}
```

} Body

```
void main()  
{ cout << "Hello from main";  
  DisplayMessage(); // FUNCTION CALL  
  cout << "Back in function main again.\n";  
}
```

Functions

```
Return type      Function name      Parameter List  
void DisplayMessage(void)  
{   cout << "Hello from the function"  
    << "DisplayMessage.\n";  
}  
} Body
```

```
void main()  
{ cout << "Hello from main";  
  DisplayMessage(); // FUNCTION CALL  
  cout << "Back in function main again.\n";  
}
```

Return type Function name Parameter List

```
void DisplayMessage(void)
{ cout << "Hello from the function"
"DisplayMessage.\n";
}
```

} Body <<

```
void main()
{ cout << "Hello from main";
    DisplayMessage(); // FUNCTION CALL
    cout << "Back in function main again.\n";
}
```

Return type Function name Parameter List

→ **void DisplayMessage(void)**

{ cout << "Hello from the function"
 << "DisplayMessage.\n";
}

} Body

```
void main()
{ cout << "Hello from main";
  DisplayMessage(); // FUNCTION CALL
  cout << "Back in function main again.\n";
}
```

Return type Function name Parameter List

→ void DisplayMessage(void)

{ cout << "Hello from the function"
 << "DisplayMessage.\n"; }

}

void main()

{ cout << "Hello from main";
 DisplayMessage(); // FUNCTION CALL
 cout << "Back in function main again.\n"; }

The diagram illustrates the structure of a C++ function definition. A blue arrow points from the text "Return type" to the keyword "void" in the function declaration. Another blue arrow points from "Function name" to "DisplayMessage". A third blue arrow points from "Parameter List" to the parentheses "()" in the declaration. A large curly brace on the right side of the code block is labeled "Body", encompassing the entire block of code.

```
→ void DisplayMessage(void)
{ cout << "Hello from the function"
  << "DisplayMessage.\n";
}
```

```
void main()
{ cout << "Hello from main";
  DisplayMessage(); // FUNCTION CALL
  cout << "Back in function main again.\n";
}
```

The diagram illustrates the structure of a C++ function definition. At the top, three labels with arrows point to specific parts of the code: "Return type" points to "void", "Function name" points to "DisplayMessage", and "Parameter List" points to "(void)". Below this, the full function definition is shown:

```
void DisplayMessage(void)
{ cout << "Hello from the function"
  << "DisplayMessage.\n";
}
```

A large curly brace on the right side of the code is labeled "Body". A blue bracket on the left side of the code, starting from the opening brace of the main function and ending at the closing brace of the main function body, encloses the entire function definition.

Below the function definition, the main function is shown:

```
void main()
{ cout << "Hello from main";
  DisplayMessage(); // FUNCTION CALL
  cout << "Back in function main again.\n";
}
```

The line "DisplayMessage(); // FUNCTION CALL" is highlighted with a purple rectangular background.

```
// FUNCTION DEFINITION
```

```
void DisplayMessage(void)
```

```
{   cout << "Hello from the function"
```

```
      << "DisplayMessage.\n";
```

```
}
```

```
void main()
```

```
{ cout << "Hello from main";
```

```
    DisplayMessage(); // FUNCTION CALL
```

```
    cout << "Back in function main again.\n";
```

```
}
```

```
void DisplayMessage(void); //fn declaration
```

```
void main()
{ cout << "Hello from main";
  DisplayMessage(); // FUNCTION CALL
  cout << "Back in function main again.\n";
```

```
}
```

```
// FUNCTION DEFINITION
```

```
void DisplayMessage(void)
{   cout << "Hello from the function"
    << "DisplayMessage.\n";
}
```

```
→void First (void)
{ cout << "I am now inside function First\n"; }
```

```
→void Second (void)
```

```
{ cout << "I am now inside function Second\n";
} // book has now as not
```

```
void main ()
```

```
{ cout << "I am starting in function main\n";
```

```
First ();
```

```
Second ();
```

```
cout << "Back in function main again.\n";
```

Function- definition & call

Return type Function name Parameter List

```
→ void DisplayMessage(void)
{   cout << "Hello from the function"
    << "DisplayMessage.\n";
}
```

}

Body

```
void main()
{ cout << "Hello from main";
    DisplayMessage(); // FUNCTION CALL
    cout << "Back in function main again.\n";
}
```

Arguments (parameters)

- Both arguments (parameters) are variables used in a **program** & **function**.
- Variables used in the *function reference* or *function call* are called as **arguments**. These are written within the parenthesis followed by the name of the function. They are also called **actual parameters**.
- Variables used in *function definition* are called **parameters**, They are also referred to as **formal parameters**.

Home Work: To be solved ...Functions

Write appropriate functions to

1. Find the factorial of a number ‘n’.
2. Reverse a number ‘n’.
3. Check whether the number ‘n’ is a palindrome.
4. Generate the Fibonacci series for given limit ‘n’.
5. Check whether the number ‘n’ is prime.
6. Generate the prime series using the function written for prime check, for a given limit.

To be solved ... Functions

Factorial of a given number ‘n’

```
long factFn(int); //prototype
```

```
void main() {  
    long n, f;  
    cout<<"Enter the number to  
    evaluate its factorial:";  
    cin>>n;  
    f=factFn(n); //function call  
    cout<<"\nFact of "<<n<<" is "<< f;  
}
```

```
long factFn(int num) //factorial calculation  
{  
    int i, fact=1;  
  
    for (i=1; i<=num; i++)  
        fact=fact * i;  
  
    return (fact); //returning the factorial  
}
```

To be solved ... Functions

Try to convert all (non function) programs
into functions...

Function- definition & call

```
Formal parameters → void dispChar(int n, char c); //proto-type  
declaration  
void dispChar(int n, char c) // function  
definition  
{  
    cout<<" You have entered "<< n << "&" <<c;  
}  
void main(){ //calling program  
int no; char ch;  
cout<<"\nEnter a number & a character: \n";  
cin>>no>>ch;  
dispChar( no, ch); //Function reference  
}
```

Actual parameters

Functions- points to note

1. The parameter list must be separated by commas.

dispChar(int n, char c);

2. The parameter names do not need to be the same in the prototype declaration and the function definition.

3. The **types must match the types of parameters** in the function definition, **in number and order**.

void dispChar (int n, char c);//proto-type

void dispChar (int num, char letter) // function definition
{ cout<<" You have entered "<< n<< "&" <<c; }

4. Use of parameter names in the declaration is optional.

void dispChar (int , char);//proto-type

Functions- points to note

4. If the function has no formal parameters, the list is written as (void).
5. The return type is optional, when the function returns **int** type data.
6. The return type must be **void** if no value is returned.
7. When the declared types do not match with the types in the function definition, compiler will produce error.

Functions- Categories

1. Functions with no arguments and no return values.
2. Functions with arguments and no return values.
3. Functions with arguments and one return value.
4. Functions with no arguments but return a value.
5. Functions that return multiple values (later).

Fn with No Arguments/parameters & No return values

```
void dispPattern(void )  
{ int i;  
    for (i=1;i<=20 ; i++)  
        cout << "*";}
```

```
void dispPattern(void); // prototype  
void main()  
{ cout << "fn to display a line of stars\n";  
    dispPattern();  
}
```

Fn with Arguments/parameters & No return values

```
void dispPattern(char c )  
{ int i;  
    for (i=1;i<=20 ; i++)  
        cout << c;  
    cout<<“\n”;}
```

```
void dispPattern(char ch); // prototype  
void main()  
{ cout << “\nfn to display a line of patterns\n”;  
    dispPattern('#');  
    dispPattern('*');  
    dispPattern('@'); }
```

Fn with Arguments/parameters & One return value

```
int fnAdd(int x, int y )  
{ int z;  
    z=x+y  
    return(z);}
```

```
int fnAdd(int a, int b); // prototype  
void main()  
{ int a,b,c;  
cout << “\nEnter numbers to be added\n”;  
cin>>a>>b;  
c=fnAdd(a,b);  
cout<<“Sum is “<< c;}
```

Fn with No Arguments but A return value

```
int readNum()  
{ int z;  
    cin>>z;  
    return(z); }
```

```
int readNum(void); // prototype  
void main(){  
    int c;  
    cout << “\nEnter a number \n”;  
    c=readNum();  
    cout<<“\nThe number read is “<<c;  
}
```

Fn that return multiple values

Will see later...

Passing 2D-Array to Function

Rules to pass a 2D- array to a function

- The function must be called by passing only the array name.
- In the function definition, we must indicate that the array has two-dimensions by including two set of brackets.
- The size of the second dimension must be specified.
- The prototype declaration should be similar to function header.

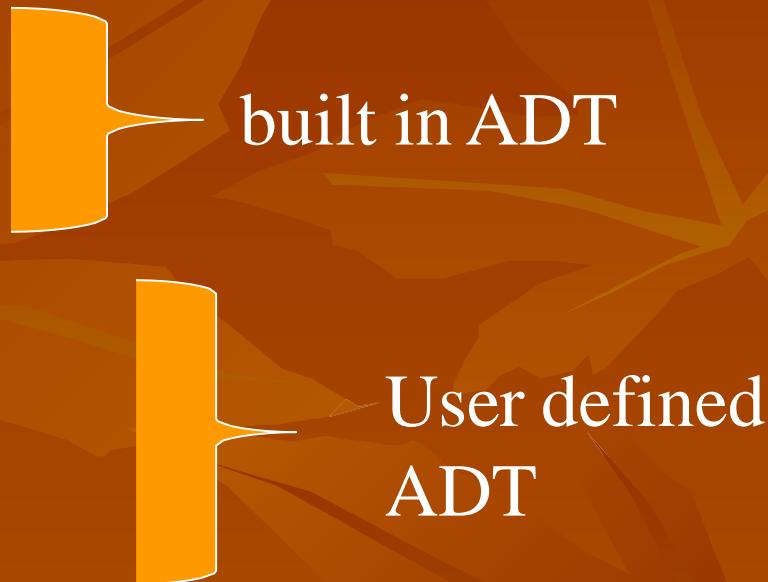
CLASS & OBJECT

- Class: A Class is a user defined data type to implement an abstract object. Abstract means to hide the details. A Class is a combination of data and functions.
- Data is called as data members and functions are called as member functions.

② Abstract data type:-

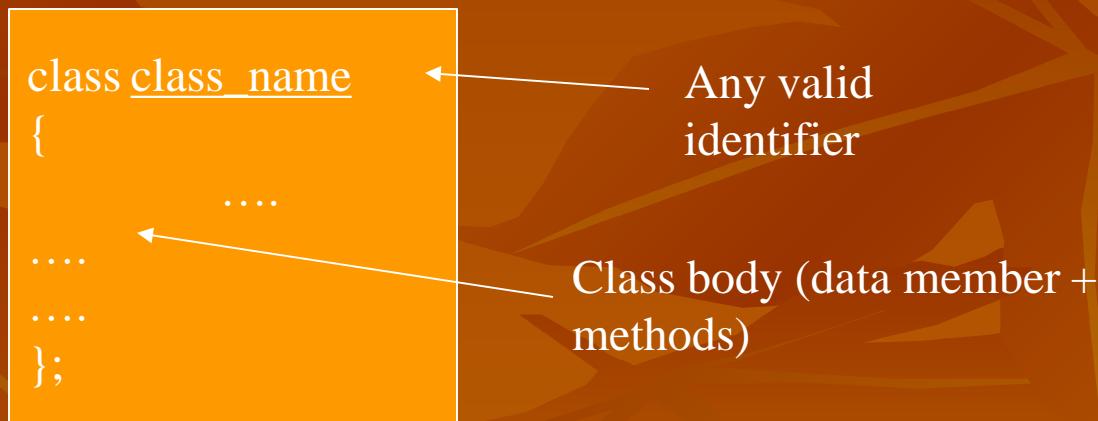
- ② A data type that separates the logical properties from the implementation details called Abstract Data Type(ADT).
- ② An abstract data type is a set of objects and an associated set of operations on those objects.
- ② ADT supports data abstraction, encapsulation and data hiding.

- Examples of ADT are:-
 - Boolean
 - Integer
 - Array
 - Stack
 - Queue
 - Tree search structure
- Boolean {operations are AND,OR,NOT and values are true and false }
- Queues{operations are create , dequeue,inqueue and values are queue elements }



Class definition

- A class definition begins with the keyword *class*.
- The body of the class is contained within a set of braces, { } ; (notice the semi-colon).



The diagram illustrates the structure of a class definition. It shows a code snippet within a yellow box:

```
class class_name
{
    ...
};
```

Annotations with arrows point to specific parts of the code:

- An arrow points to the identifier `class_name` with the label "Any valid identifier".
- An arrow points to the entire block between the braces with the label "Class body (data member + methods)".

- Within the body, the keywords *private*: and *public*: specify the access level of the members of the class.
 - the default is private.
- Usually, the data members of a class are declared in the *private*: section of the class and the member functions are in *public*: section.

- Data member or member functions may be public, private or protected.
- Public means data members or member functions defined inside the class can be used outside the class.(in different class and in main function)
- Member access specifiers
 - **public:** can be accessed outside the class directly.

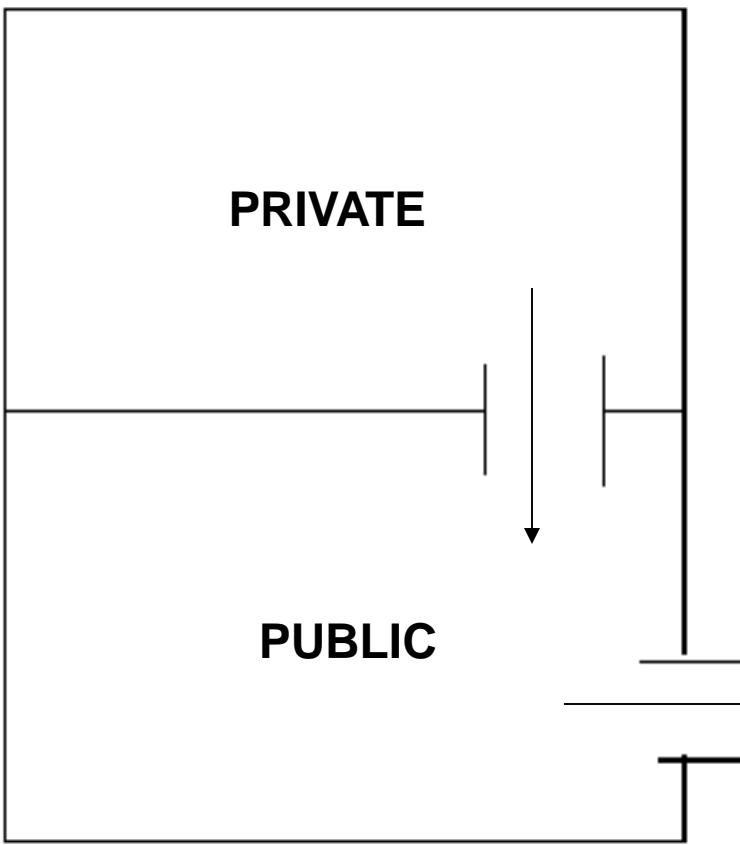
private:

- Accessible only to member functions of class
 - Private members and methods are for internal use only.
-
- Private means data members and member functions can't be used outside the class.

□ Protected means data member and member functions can be used in the same class and its derived class (at one level) (not in main function).

PRIVATE

PUBLIC



```
class class_name  
{  
    private:  
        ...  
        ...  
        ...  
    public:  
        ...  
        ...  
        ...  
};
```

private members or methods

Public members or methods

- This class example shows how we can encapsulate (gather) a circle information into one package (unit or class)

```
class Circle
{
    private:
        double radius;
    public:
        void setRadius(double r);
        double getDiameter();
        double getArea();
        double getCircumference();
};
```

No need for other classes to access and retrieve its value directly. The class methods are responsible for that only.

They are accessible from outside the class, and they can access the member (radius)

Class Example (Problem)

```
#include<iostream>
#include<stdio.h>
class student
{
    int rollno;
    char name[20];
};
```

```
int main()
{
    student s;
    cout<<"enter the rollno.:";
    cin>>s.rollno;
    cout<<"enter the name:";
    gets(s.name);
    cout<<"rollno:"<<s.rollno;
    cout<<"\nname:";
    puts(s.name);
    return 0;
}
```

Class Example (Solution)

```
#include<iostream>
#include<stdio.h>
class student
{
public:
    int rollno;
    char name[20];
};
```

```
int main()
{
    student s;
    cout<<"enter the rollno.:";
    cin>>s.rollno;
    cout<<"enter the name:";
    gets(s.name);
    cout<<"rollno:"<<s.rollno;
    cout<<"\nname:";
    puts(s.name);
    return 0;
}
```

Implementing class methods

- There are two ways:
 1. Member functions defined outside class
 - Using Binary scope resolution operator (::)
 - “Ties” member name to class name
 - Uniquely identify functions of particular class
 - Different classes can have member functions with same name
 - Format for defining member functions

```
ReturnType ClassName :: MemberFunctionName( ) {  
    ...  
}
```

Member Function

Defining Inside the Class

```
class student
{
    int rollno;
    char name[20];
public:
    void getdata()
    {
        cout<<"enter the rollno.:";
        cin>>rollno;
        cout<<"enter the name:";
        gets(name);
    }
    void putdata()
    {
        cout<<"rollno:"<<rollno;
        cout<<"\nname:";
        puts(name);
    }
};
```

Data Members (Private : in this example)

Member Functions (Public: in this example)

Calling member function

```
void main()
{
    student s;
    s.getdata();
    s.putdata();
}
```

Member Function

Defining Outside the Class

```
class student
{
int rollno;
char name[20];
public:
void getdata();
void putdata();
};

void student :: getdata()
{
cout<<"enter the rollno.:";
cin>>rollno;
cout<<"enter the name:";
gets(name);
}
```

```
void student :: putdata()
{
cout<<"rollno:"<<rollno;
cout<<"\nname:";
puts(name);
}

void main()
{
student s;
s.getdata();
s.putdata();
}
```

Characteristics of member function

- Different classes have same function name, the “membership label” will resolve their scope.
- Member functions can access the private data of the class. A non member function cannot do this.(friend function can do this.)
- A member function can call another member function directly, without using the dot operator.

Accessing Class Members

- ❑ Operators to access class members
 - ❑ Identical to those for **structs**
 - ❑ Dot member selection operator (.)
 - ❑ Object
 - ❑ Reference to object
 - ❑ Arrow member selection operator (->)
 - ❑ Pointers

Objects

- An object is an instance of a class.
- An object is a class variable.
- It can be uniquely identified by its name.
- Every object have a state which is represented by the values of its attributes. These state are changed by function which applied on the object.

Creating an object of a Class

- Declaring a variable of a class type creates an **object**. You can have many variables of the same type (class).
 - *Also known as Instantiation*
- Once an object of a certain class is instantiated, a new memory location is created for it to store its data members and code
- You can instantiate many objects from a class type.
 - Ex) Circle c; Circle *c;

Class item

```
{ .....,  
- -----  
}x,y,z;
```

We have to declare objects close to the place where they are needed because it makes easier to identify the objects.

Memory Allocation of Object

```
class student
```

```
{  
    int rollno;  
    char name[20];  
    int marks;  
};
```

```
student s;
```



Array of objects

The array of class type variable is known as array of object.

- ② We can declare array of object as following way:-
Class_name object [length];
Employee manager[3];

- 1. We can use this array when calling a member function
2. Manager[i].put data();
3. The array of object is stored in memory as a multi-dimensional array.

Object as function arguments

- This can be done in two ways:-
- A copy of entire object is passed to the function.
(pass by value)
- Only the address of the object is transferred to the function. (pass by reference)

(pass by value)

- A copy of the object is passed to the function, any changes made to the object inside the function do not affect the object used to call function.

(pass by reference)

- When an address of object is passed, the called function works directly on the actual object used in the call. Means that any change made inside the function will reflect in the actual object.

```
class Complex
{
    float real, imag;
public:
    void getdata( );
    void putdata( );
    void sum (Complex A, Complex B);
};

void Complex :: getdata( )
{
    cout<<"enter real part:" ;
    cin>>real;
    cout<<"enter imaginary part:" ;
    cin>>imag;
}

void Complex :: putdata( )
{
    if (imag>=0)
        cout<<real<<"+ "<<imag<<"i";
    else
        cout<<real<<imag<<"i";
}
```

Passing Object

```
void Complex :: sum ( Complex A, Complex B)
{
    real = A.real + B.real;
    imag= A.imag + B.imag;
}

int main( )
{
    Complex X,Y,Z;
    X.getdata( );
    Y.getdata( );
    Z.sum(X,Y);
    Z.putdata( );
    return 0;
}
```

```
#include<iostream.h>
class Complex
{
float real, imag;
public:
void getdata( );
void putdata( );
void sum (Complex A, Complex B);
};
void Complex :: getdata( )
{
cout<<“enter real part:”;
cin>>real;
cout<<“enter imaginary part:”;
cin>>imag;
}
void Complex :: putdata( )
{
if (imag>=0)
cout<<real<<“+”<<imag<<“i”;
else
cout<<real<<imag<<“i”;
}
```

Passing Object

```
void Complex :: sum ( Complex A, Complex B)
{
real = A.real + B.real;
imag= A.imag + B.imag;
}

void main()
{
Complex X,Y,Z;
X.getdata();
Y.getdata();
Z.sum(X,Y);
Z.putdata();
}
```



X

Y

Z

```
#include<iostream.h>
class Complex
{
float real, imag;
public:
void getdata( );
void putdata( );
void sum (Complex A, Complex B);
};
void Complex :: getdata( )
{
cout<<“enter real part:”;
cin>>real;
cout<<“enter imaginary part:”;
cin>>ima g;
}
void Complex :: putdata( )
{
if (imag>=0)
cout<<real<<“+”<<imag<<“i”;
else
cout<<real<<imag<<“i”;
}
```

Passing Object

```
void Complex :: sum ( Complex A, Complex B)
{
real = A.real + B.real;
imag= A.imag + B.imag;
}

void main()
{
Complex X,Y,Z;
X.getdata();
Y.getdata();
Z.sum(X,Y);
Z.putdata();
}
```

5
6

7
8

X

Y

Z

```

#include<iostream.h>
class Complex
{
float real, imag;
public:
void getdata( );
void putdata( );
void sum (Complex A, Complex B);
};
void Complex :: getdata( )
{
cout<<“enter real part:”;
cin>>real;
cout<<“enter imaginary part:”;
cin>>imag;
}
void Complex :: putdata( )
{
if (imag>=0)
cout<<real<<“+”<<imag<<“i”;
else
cout<<real<<imag<<“i”;
}

```

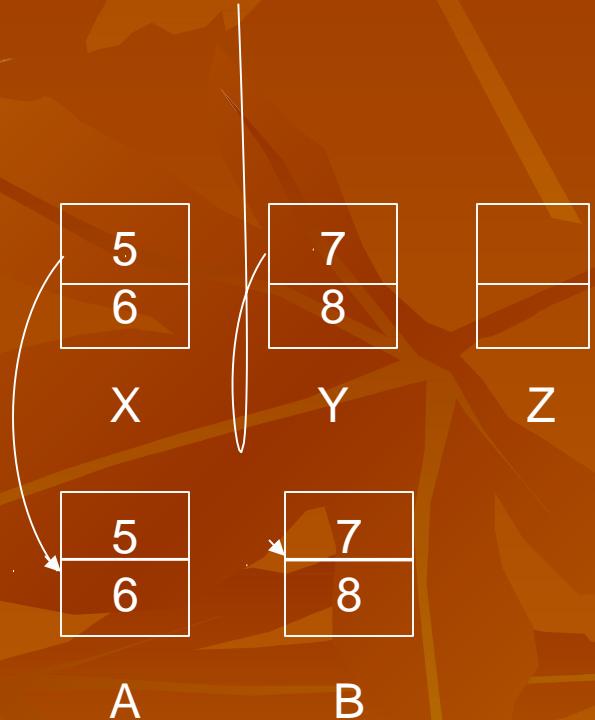
Passing Object

```

void Complex :: sum ( Complex A, Complex B)
{
real = A.real + B.real;
imag= A.imag + B.imag;
}

void main()
{
Complex X,Y,Z;
X.getdata();
Y.getdata();
Z.sum(X,Y);
Z.putdata();
}

```



```

#include<iostream.h>
class Complex
{
float real, imag;
public:
void getdata( );
void putdata( );
void sum (Complex A, Complex B);
};
void Complex :: getdata( )
{
cout<<“enter real part:”;
cin>>real;
cout<<“enter imaginary part:”;
cin>>ima g;
}
void Complex :: putdata( )
{
if (imag>=0)
cout<<real<<“+”<<imag<<“i”;
else
cout<<real<<imag<<“i”;
}

```

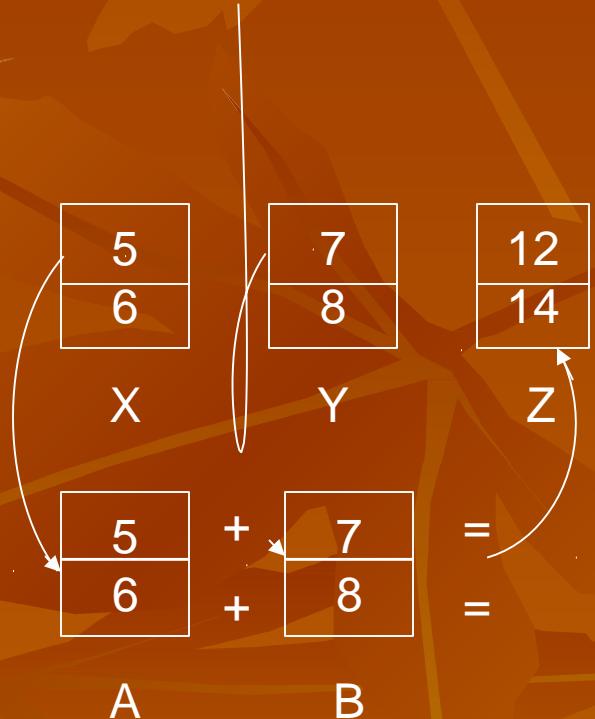
Passing Object

```

void Complex :: sum ( Complex A, Complex B)
{
real = A.real + B.real;
imag= A.imag + B.imag;
}

void main()
{
Complex X,Y,Z;
X.getdata();
Y.getdata();
Z.sum(X,Y);
Z.putdata();
}

```



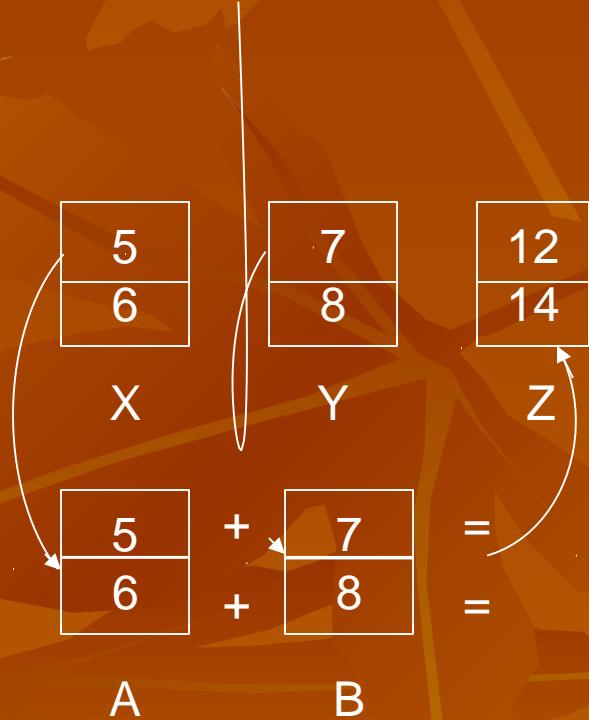
```
#include<iostream.h>
class Complex
{
float real, imag;
public:
void getdata( );
void putdata( );
void sum (Complex A, Complex B);
};
void Complex :: getdata( )
{
cout<<“enter real part:”;
cin>>real;
cout<<“enter imaginary part:”;
cin>>imag;
}
void Complex :: putdata( )
{
if (imag>=0)
cout<<real<<“+”<<imag<<“i”;
else
cout<<real<<imag<<“i”;
}
```

Passing Object

```
void complex :: sum ( Complex A, Complex B)
{
real = A.real + B.real;
imag= A.imag + B.imag;
}

void main()
{
Complex X,Y,Z;
X.getdata();
Y.getdata();
Z.sum(X,Y);
Z.putdata();
}
```

$$12 + 14 i$$



```
#include<iostream.h>
class Complex
{
float real, imag;
public:
void getdata( );
void putdata( );
void sum (Complex A, Complex B);
};
void Complex :: getdata( )
{
cout<<“enter real part:”;
cin>>real;
cout<<“enter imaginary part:”;
cin>>ima g;
}
void Complex :: putdata( )
{
if (imag>=0)
cout<<real<<“+”<<imag<<“i”;
else
cout<<real<<imag<<“i”;
}
```

Returning Object

```
Complex Complex :: sum
(Complex B)
{
Complex temp;
temp.real=real + B.real;
temp.imag= imag + B.imag;
return temp;
}
void main ()
{
Complex X, Y, Z;
X.Getdata();
Y. getdata();
Z= X.sum (Y);
Z.putdata();
}
```

Returning Object

```
class Complex
{
    float real, imag;
public:
    void getdata( );
    void putdata( );
    void sum (Complex A, Complex B);
};
void Complex :: getdata( )
{
    cout<<"enter real part:";
    cin>>real;
    cout<<"enter imaginary part:";
    cin>>imag;
}
void Complex :: putdata( )
{
    if (imag>=0)
        cout<<real<<"+"<<imag<<"i";
    else
        cout<<real<<imag<<"i";
}
```

```
Complex Complex :: sum (Complex B)
{
    Complex temp;
    temp.real=real + B.real;
    temp.imag= imag + B.imag;
    return temp;
}
void main ( )
{
    Complex X, Y, Z;
    X.getdata( );
    Y.getdata( );
    Z= X.sum (Y);
    Z.putdata( );
}
```



X

Y

Z

Returning Object

```
class Complex
{
    float real, imag;
public:
    void getdata( );
    void putdata( );
    void sum (Complex A, Complex B);
};
void Complex :: getdata( )
{
    cout<<“enter real part:”;
    cin>>real;
    cout<<“enter imaginary part:”;
    cin>>imag;
}
void Complex :: putdata( )
{
    if (imag>=0)
        cout<<real<<“+”<<imag<<“i”;
    else
        cout<<real<<imag<<“i”;
}
```

```
Complex Complex :: sum (Complex B)
{
    Complex temp;
    temp.real=real + B.real;
    temp.imag= imag + B.imag;
    return temp;
}
void main ()
{
    Complex X, Y, Z;
    X.Getdata( );
    Y.getdata( );
    Z= X.sum (Y);
    Z.putdata( );
}
```

5
6

7
8

X

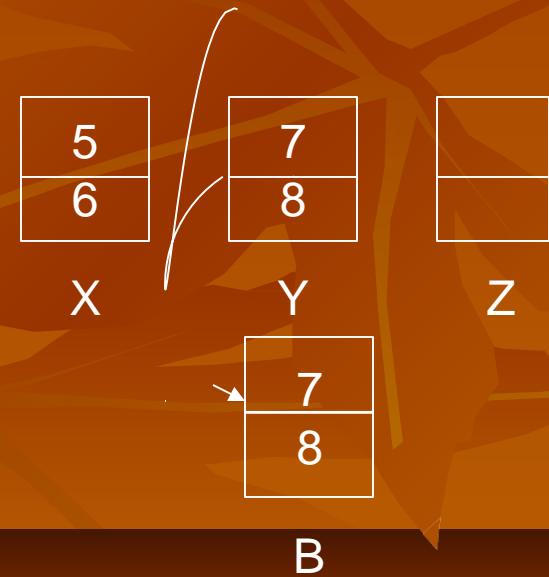
Y

Z

Returning Object

```
class Complex
{
    float real, imag;
public:
    void getdata( );
    void putdata( );
    void sum (Complex A, Complex B);
};
void Complex :: getdata( )
{
    cout<<“enter real part:”;
    cin>>real;
    cout<<“enter imaginary part:”;
    cin>>imag;
}
void Complex :: putdata( )
{
    if (imag>=0)
        cout<<real<<“+”<<imag<<“i”;
    else
        cout<<real<<imag<<“i”;
}
```

```
Complex Complex :: sum (Complex B)
{
    Complex temp;
    temp.real=real + B.real;
    temp.imag= imag + B.imag;
    return temp;
}
void main ()
{
    Complex X, Y, Z;
    X.Getdata();
    Y.getdata();
    Z= X.sum (Y);
    Z.putdata();
}
```



```

class Complex
{
    float real, imag;
public:
    void getdata( );
    void putdata( );
    void sum (Complex A, Complex B);
};
void Complex :: getdata( )
{
    cout<<“enter real part:”;
    cin>>real;
    cout<<“enter imaginary part:”;
    cin>>imag;
}
void Complex :: putdata( )
{
    if (imag>=0)
        cout<<real<<“+”<<imag<<“i”;
    else
        cout<<real<<imag<<“i”;
}

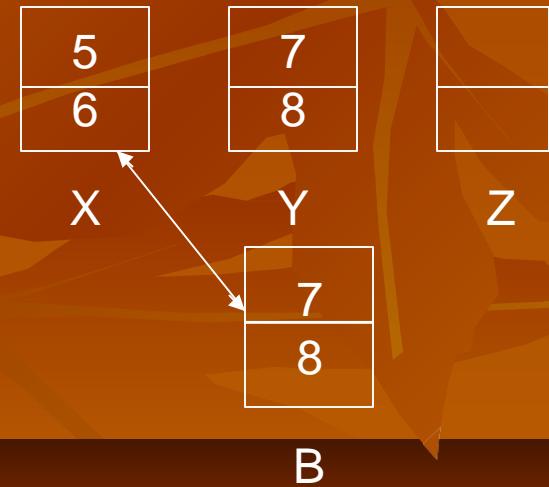
```

Returning Object

```

Complex Complex :: sum (Complex B)
{
    Complex temp;
    temp.real=real + B.real;
    temp.imag= imag + B.imag;
    return temp;
}
void main ()
{
    Complex X, Y, Z;
    X.getdata( );
    Y.getdata( );
    Z= X.sum (Y);
    Z.putdata( );
}

```



```

class Complex
{
    float real, imag;
public:
    void getdata( );
    void putdata( );
    void sum (Complex A, Complex B);
};
void Complex :: getdata( )
{
    cout<<“enter real part:”;
    cin>>real;
    cout<<“enter imaginary part:”;
    cin>>imag;
}
void Complex :: putdata( )
{
    if (imag>=0)
        cout<<real<<“+”<<imag<<“i”;
    else
        cout<<real<<imag<<“i”;
}

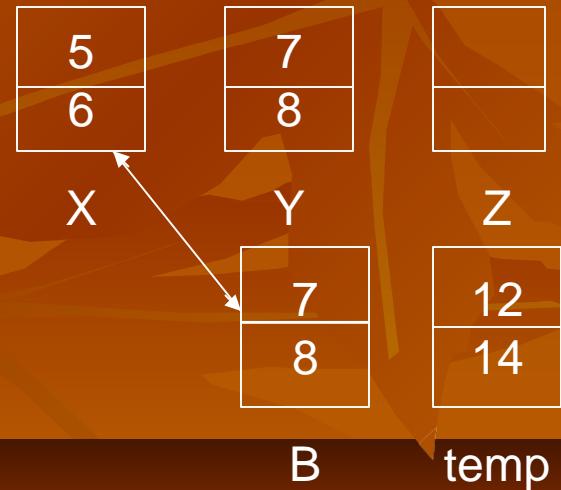
```

Returning Object

```

Complex Complex :: sum (Complex B)
{
    Complex temp;
    temp.real=real + B.real;
    temp.imag= imag + B.imag;
    return temp;
}
void main ()
{
    Complex X, Y, Z;
    X.getdata( );
    Y.getdata( );
    Z= X.sum (Y);
    Z.putdata( );
}

```



```

class Complex
{
    float real, imag;
public:
    void getdata( );
    void putdata( );
    void sum (Complex A, Complex B);
};
void Complex :: getdata( )
{
    cout<<“enter real part:”;
    cin>>real;
    cout<<“enter imaginary part:”;
    cin>>imag;
}
void Complex :: putdata( )
{
    if (imag>=0)
        cout<<real<<“+”<<imag<<“i”;
    else
        cout<<real<<imag<<“i”;
}

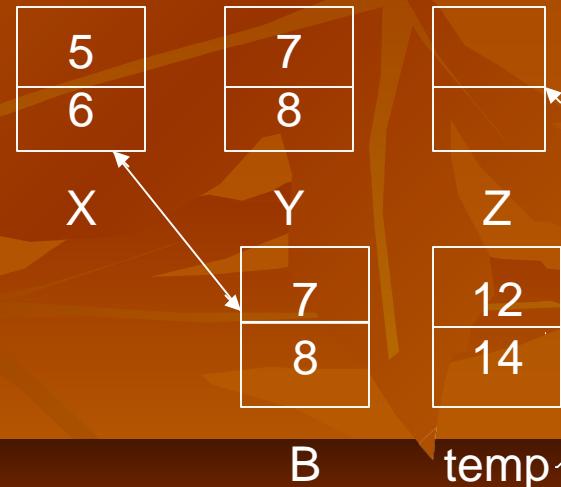
```

Returning Object

```

Complex Complex :: sum (Complex B)
{
    Complex temp;
    temp.real=real + B.real;
    temp.imag= imag + B.imag;
    return temp;
}
void main ()
{
    Complex X, Y, Z;
    X.getdata( );
    Y.getdata( );
    Z= X.sum (Y);
    Z.putdata( );
}

```



```

class Complex
{
    float real, imag;
public:
    void getdata( );
    void putdata( );
    void sum (Complex A, Complex B);
};
void Complex :: getdata( )
{
    cout<<“enter real part:”;
    cin>>real;
    cout<<“enter imaginary part:”;
    cin>>imag;
}
void Complex :: putdata( )
{
    if (imag>=0)
        cout<<real<<“+”<<imag<<“i”;
    else
        cout<<real<<imag<<“i”;
}

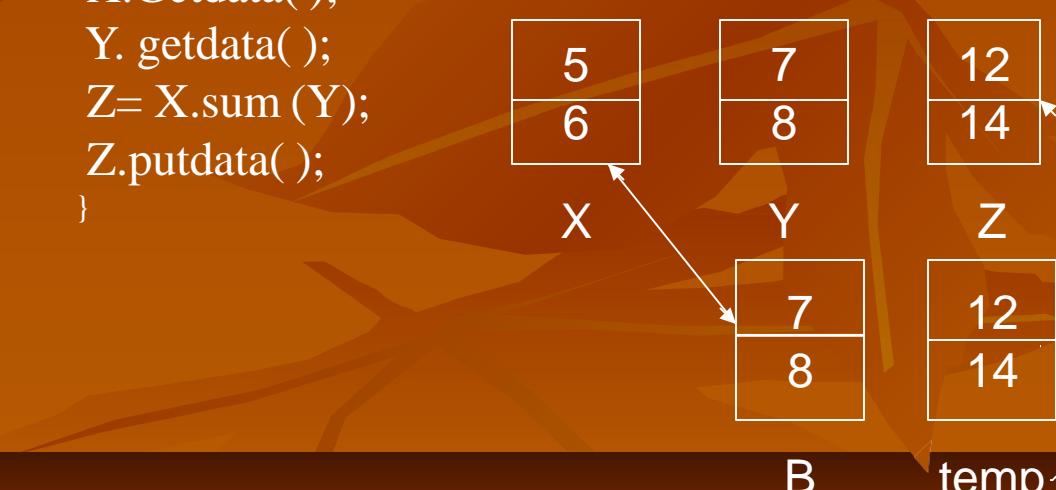
```

Returning Object

```

Complex Complex :: sum (Complex B)
{
    Complex temp;
    temp.real=real + B.real;
    temp.imag= imag + B.imag;
    return temp;
}
void main ()
{
    Complex X, Y, Z;
    X.Getdata( );
    Y.getdata( );
    Z= X.sum (Y);
    Z.putdata( );
}

```



```

class Complex
{
    float real, imag;
public:
    void getdata( );
    void putdata( );
    void sum (Complex A, Complex B);
};
void Complex :: getdata( )
{
    cout<<“enter real part:”;
    cin>>real;
    cout<<“enter imaginary part:”;
    cin>>imag;
}
void Complex :: putdata( )
{
    if (imag>=0)
        cout<<real<<“+”<<imag<<“i”;
    else
        cout<<real<<imag<<“i”;
}

```

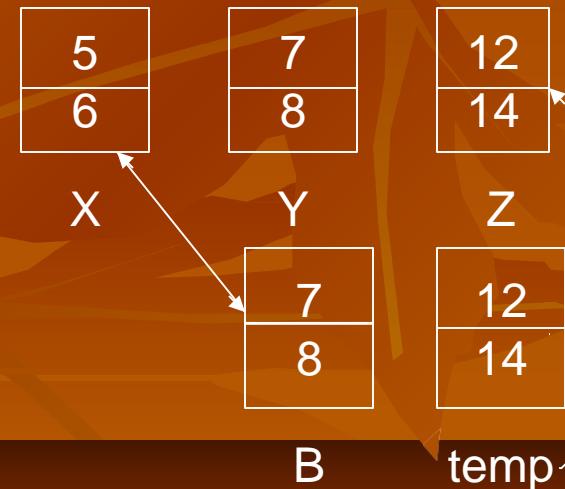
Returning Object

```

Complex Complex :: sum (Complex B)
{
    Complex temp;
    temp.real=real + B.real;
    temp.imag= imag + B.imag;
    return temp;
}
void main ()
{
    Complex X, Y, Z;
    X.Getdata( );
    Y.getdata( );
    Z= X.sum (Y);
    Z.putdata( );
}

```

$12 + 14i$





Constructors

and

Destructors

Constructor

- It is a member function which initializes the objects of its class.
- A constructor has:
 - (i) the same name as the class itself
 - (ii) no return type ,not even void.
- It constructs the values of data member so that it is called constructor.

- A constructor is called automatically whenever a new object of a class is created.
- You must supply the arguments to the constructor when a new object is created.
- If you do not specify a constructor, the compiler generates a default constructor for you (expects no parameters and has an empty body).

```
void main()
{
    rectangle rc(3.0, 2.0);

    rc.posn(100, 100);
    rc.draw();
    rc.move(50, 50);
    rc.draw();
}
```

□ *Warning:* attempting to initialize a data member of a class explicitly in the class definition is a syntax error.

Declaration and definition

class complex

{

 int m, n;

 public:

 complex();

};

complex :: complex ()

{

 m=0; n=0;

}

- A constructor that accepts no parameters is called default constructor.

characteristics

1. They should be declared in public section.
2. Invoked automatically when class objects are created.
3. They do not have return types, not even void and they can't return any value.
4. They cannot be inherited, though a derived class can call the base class constructors.

5. They also can have default arguments like other functions.
6. They are implicitly called when the NEW and DELETE operators execute when memory allocation is required.
7. Constructors can not be virtual.

Parameterized constructors

- The constructors that can take arguments are called parameterized constructors.
- It is used when we assign different value to the data member for different object.
- We must pass the initial values as arguments to the constructors when an object is declared.

- ② This can be done in two ways:-
 - ② By **calling the constructors implicitly**
 - ② Class_name object(arguments);
 - ② Ex:- simple s(3, 67);
 - ② This method also known as shorthand.
 - ② By **calling the constructors explicitly**
 - ② Class_name object =constructor(arguments);
 - ② Ex:- simple s=simple(2,67);
 - ② This statement create object s and passes the values 2 and 67 to it.

Example:-

```
#include<iostream.h>
class integer
{
    int m,n;
public:
    integer(int,int);
    void display()
    {
        cout<<"m"<<m;
        cout<<"n"<<n;
    }
    integer::integer(int x,int y)
    {
        m=x;
        n=y;
    }
}
```

```
int main()
{
    integer i1(10,100);
    integer i2=integer(33,55);
    cout<<"object 1";
    i1.display();
    cout<<"object 2";
    i2.display();
    return 0;
}
```

Notes:-

A constructor function can also be defined as **INLINE** function.

```
Class integer
{
    int m,n;
    public:
        integer (int x,int y)
    {
        m=x;
        n=y;
    };
```

A class can accept a reference of its own class as parameter.

```
Class A  
{
```

```
.....
```

```
.....  
Public:  
A(A&);  
};
```

is valid

In this case the constructor is called as copy constructor.

Copy constructor

- When a class reference is passed as parameters in constructor then that constructor is called Copy constructor.
- A copy constructor is used to declare and initialize an object from another object.

Syntax:-

Constructor _name (class_name & object); Integer (integer
&i);

```
Integer i2(i1); /integer i2=i1;
```

Define object i2 and initialize it with i1.

The process of initialization object through copy constructor is known as copy initialization.

A copy constructor takes a reference to an object of the same class as itself as argument.

```
#include<iostream.h>

Class person
{
    public: int
        age;
    Person(int a)
    { age = a; } Person(person
    & x)
    { age=x.age;
    }
};
```

```
int main()
{
    Person timmy(10); Person
    sally(15);
    Person timmy_clone = timmy;
    cout << timmy.age << " " << sally.age << " "
    << timmy_clone.age << endl;
    timmy.age = 23;
    cout << timmy.age << " " << sally.age << " "
    << timmy_clone.age << endl;
}
```

Destructors

- A destructor is used to destroy the objects that have been created by constructor.
- It is also a member function of class whose name same as class name but preceded by tiled sign(~).
- It never takes any arguments nor return any value
- It will be invoked implicitly by the compiler upon exit from the program to clean up the storage which is allocated

The new operator is used in constructor to allocate memory and delete is used to free in destructors.

Example:-

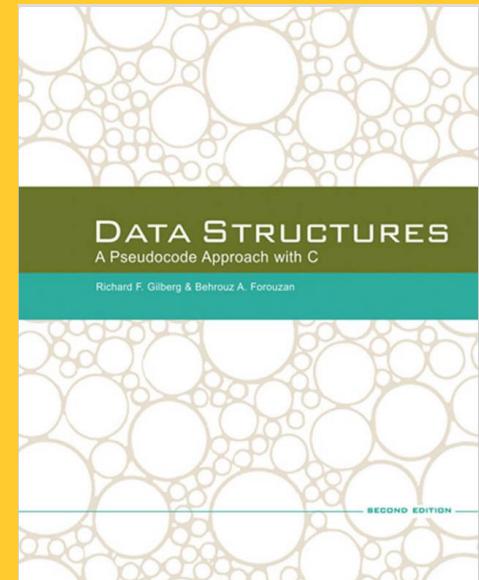
```
~assign()  
{  
    Delete p;  
}
```

```
#include<iostream.h>
Int count=0;
Class try
{
    public:
        try()
        {
            count++;
Cout<<“no of objects created”<<count;
}
~try()
{
    cout<<“no of object destroyed”<<count;
Count- -;
}};
```

```
int main()
{
    cout<<“enter main”;
try t1,t2,t3,t4;
{
    cout<<“block1”;
try t5;
}
{
    cout<<“block 2”;
try t6;
}
cout<<“again in main”;
Return 0;
}
```

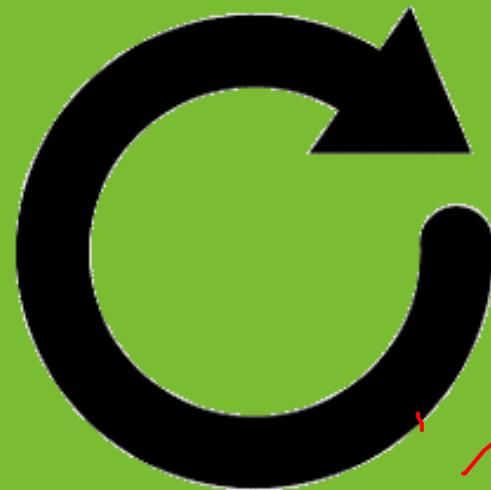
Review of

Recursion



How to write repetitive algorithms?

Two Approaches:



Iteration



Recursion

Recursion is a repetitive process in which an algorithm calls itself.

Recursion: What is it?

Case Study: Factorial Algorithm.

1. Iterative Solution
2. Recursive Definition
3. Recursive Solution

Factorial: Iterative Definition.

The definition involves only the algorithm parameter(s) and not the algorithm itself.

$$\text{Factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1) \times (n-2) \times \dots \times 3 \times 2 \times 1 & \text{if } n > 0 \end{cases}$$

FIGURE 2-1 Iterative Factorial Algorithm Definition

$$5 = 5 \times 4 \times 3 \times 2 \times 1$$

$$\text{factorial}(4) = 4 \times 3 \times 2 \times 1 = 24$$

Factorial: Recursive Definition.

A repetitive algorithm uses recursion whenever the algorithm appears within the definition itself.

$$\text{Factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times (\text{Factorial}(n - 1)) & \text{if } n > 0 \end{cases}$$

FIGURE 2-2 Recursive Factorial Algorithm Definition

Factorial: Recursive Definition.

Recursion is a repetitive process in which an algorithm calls itself.

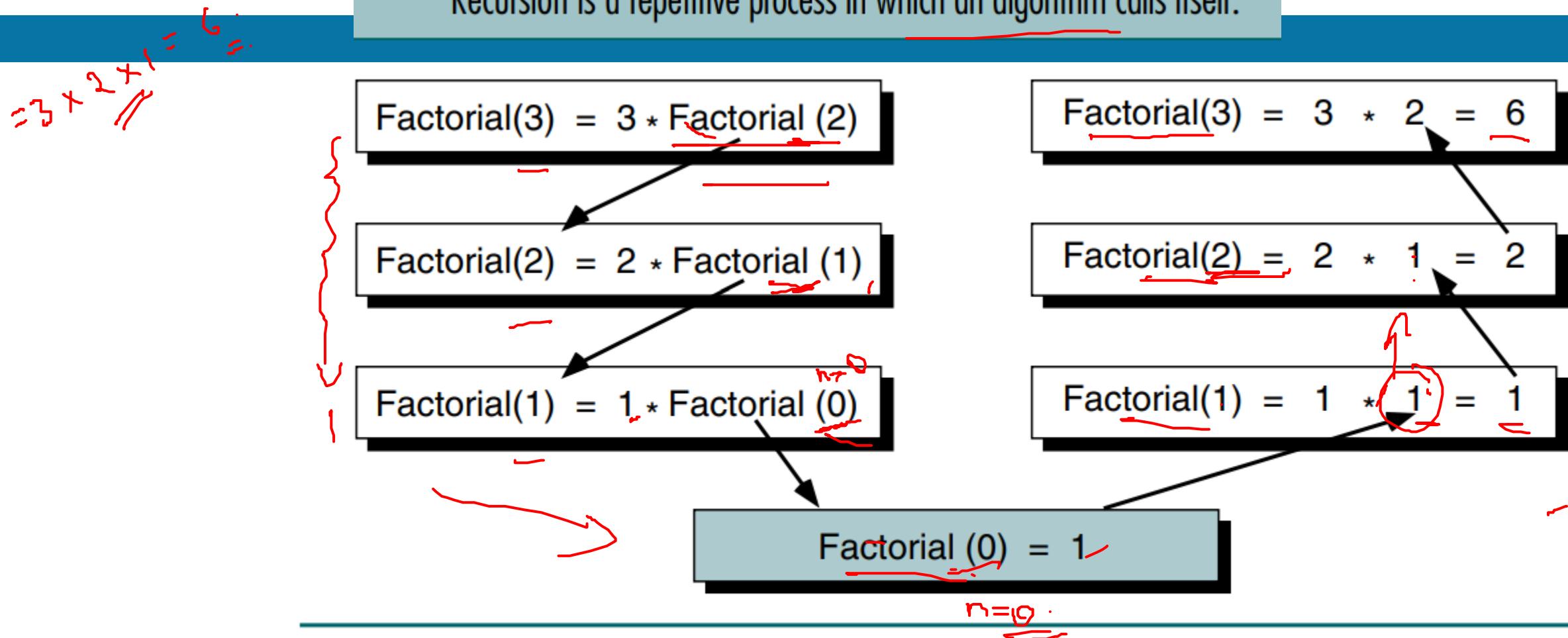


FIGURE 2-3 Factorial (3) Recursively

Iterative Solution

ALGORITHM 2-1 Iterative Factorial Algorithm

```
Algorithm iterativeFactorial (n)
Calculates the factorial of a number using a loop.
  Pre n is the number to be raised factorially
  Post n! is returned
1 set i to 1
2 set factN to 1
3 loop (i <= n)
  1 set factN to factN * i
  2 increment i
4 end loop
5 return factN
end iterativeFactorial
```

*K = 3, 2L ≈ 3
F = (x) - 1
Y*

Recursive Solution

ALGORITHM 2-2 Recursive Factorial

```
Algorithm recursiveFactorial (n)
Calculates factorial of a number using recursion.
    Pre    n  is the number being raised factorially
    Post   n! is returned
1  if (n equals 0)
    1  return 1
2  else
    1  return (n * recursiveFactorial (n - 1))
3 end if
end recursiveFactorial
```

WHICH CODE IS SIMPLER?

Which one does not have a loop?

Recursive Solution Analysis

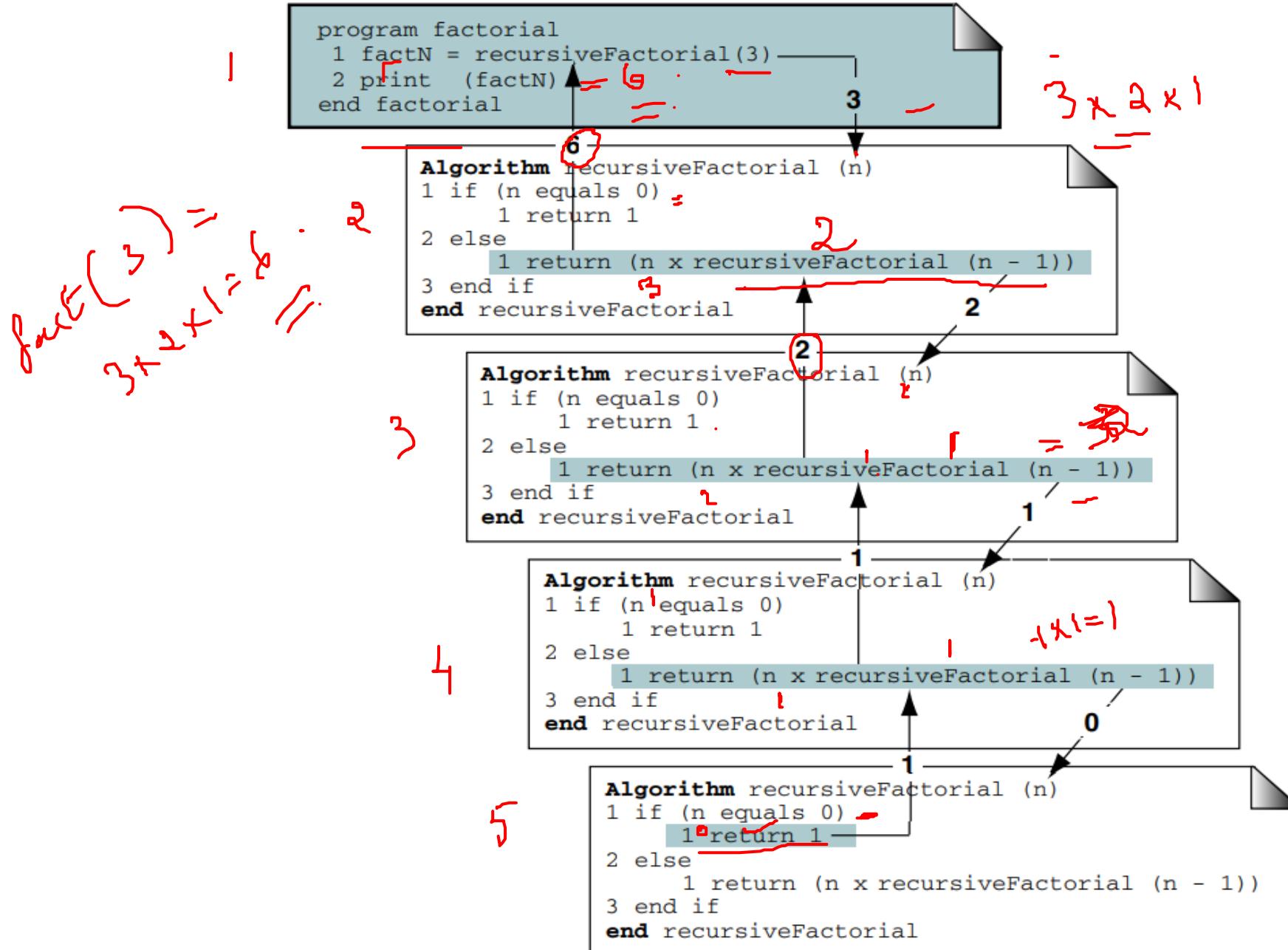


FIGURE 2-4 Calling a Recursive Algorithm

Figure 2-4 traces the recursion and the parameters for each individual call.

Designing Recursive Algorithms.

Analytic Approach:

1. The Design Methodology
2. Limitations of Recursion
3. Design Implementation

THE DESIGN METHODOLOGY.

Every recursive call **either solves a part** of the problem or it **reduce the size** of the problem.

- Base case
 - The statement that “solves” the problem.
 - Every recursive algorithm must have a base case.
- General case
 - The rest of the algorithm
 - Contains the logic needed to reduce the size of the problem.

COMBINE THE BASE CASE AND THE GENERAL CASES INTO AN ALGORITHM

Rules for designing a recursive algorithm:

1. First, determine the base case.
 2. Then determine the general case.
 3. Combine the base case and the general cases into an algorithm
-
- Each call must reduce the size of the problem and move it toward the base case.
 - The base case, when reached, must terminate without a call to the recursive algorithms; that is, it must execute a return.

LIMITATIONS OF RECURSION.

Do not use recursion if answer is NO to any question below:

1. Is the algorithm or data structure naturally suited to recursion?
2. Is the recursive solution shorter and more understandable?
3. Does the recursive solution run within acceptable time and space limits?

DESIGN IMPLEMENTATION.

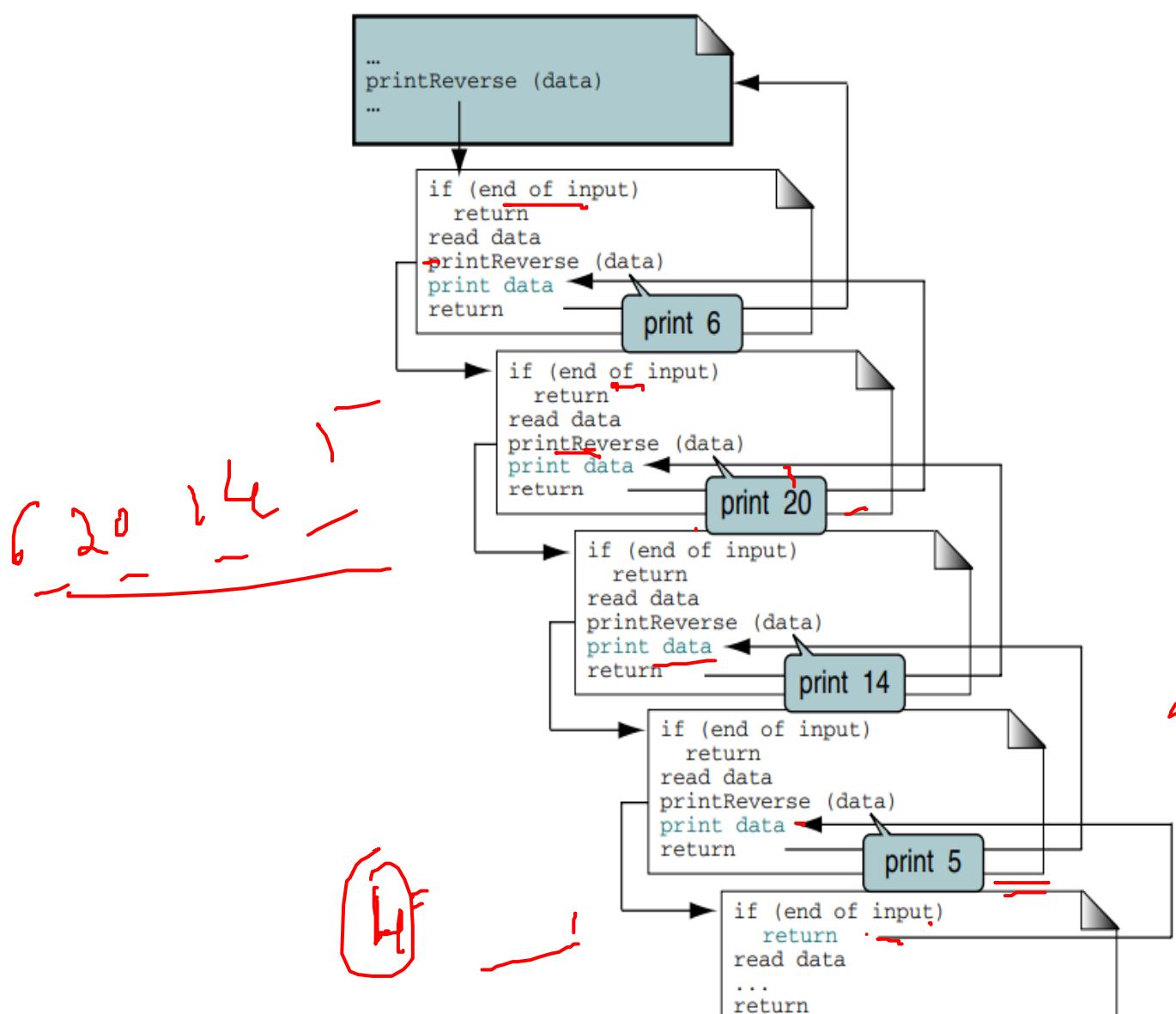
ALGORITHM 2-3 Print Reverse

```
Algorithm printReverse (data)
Print keyboard data in reverse.
Pre nothing
Post data printed in reverse
1 if (end of input) = 4
1 return
2 end if
3 read data →
4 printReverse (data) →
Have reached end of input: print nodes
5 print data
6 return
end printReverse
```

↓ | ↓ | 2 3 4 | → D C B A
O P Q 3 2 1 | → D C B A

↑ 3 2 1

DESIGN IMPLEMENTATION.



Recursive returns (prints)

6
data

20
data

14
data

5
data

FIGURE 2-5 Print Keyboard Input in Reverse

SOME MORE EXAMPLES.

{  Greatest Common Divisor

 Fibonacci Numbers

 Prefix to Postfix ~~Conversion~~

 The Towers of Hanoi

SOME MORE RECURSIVE EXAMPLES: GCD.

$$\text{gcd}(a, b) = \begin{cases} a & \text{if } b = 0 \\ b & \text{if } a = 0 \\ \text{gcd}(b, a \bmod b) & \text{otherwise} \end{cases}$$

Greatest Common Divisor Recursive Definition

We use the **Euclidean algorithm** to determine the greatest common divisor between two nonnegative integers.

Given two integers, a and b , the greatest common divisor is recursively found using the formula given above.

SOME MORE RECURSIVE EXAMPLES: GCD.

Euclidean Algorithm for Greatest Common Divisor

Algorithm gcd (a, b)

Calculates greatest common divisor using the Euclidean algorithm.

Pre a and b are positive integers greater than 0
Post greatest common divisor returned

```
1 if (b equals 0)
  1 return a
2 end if
3 if (a equals 0)
  2 return b
4 end if
5 return gcd (b, a mod b)
end gcd
```

$$\left\{ \begin{array}{l} \text{gcd}(10, 25) \\ \Rightarrow \text{gcd}(25, 10 \bmod 15) = \text{gcd}(25, 10) \\ \Rightarrow \text{gcd}(25, 10, 25 \bmod 10) = \text{gcd}(10, 5) \\ \Rightarrow \text{gcd}(10, 5, 10 \bmod 5) = \text{gcd}(5, 0) \\ b=0 \qquad a=5 \end{array} \right.$$

GCD IMPLEMENTATION.

```
9 // Prototype Statements
10 int gcd (int a, int b);
11
12 int main (void)
13 {
14 // Local Declarations
15     int gcdResult;
16
17 // Statements
18     printf("Test GCD Algorithm\n");
19
20     gcdResult = gcd (10, 25);
21     printf("GCD of 10 & 25 is %d", gcdResult);
22     printf("\nEnd of Test\n");
23
24 } // main
```

```
25 /* ===== gcd =====
26     Calculates greatest common divisor using the
27     Euclidean algorithm.
28     Pre a and b are positive integers greater than 0
29     Post greatest common divisor returned
30 */
31 int gcd (int a, int b)
32 {
33     // Statements
34     if (b == 0) {BASE}
35         return a;
36     if (a == 0)
37         return b;
38     return gcd (b, a % b);
39 } // gcd
```

Results:
Test GCD Algorithm
GCD of 10 & 25 is 5
End of Test

SOME MORE RECURSIVE EXAMPLES: FIBONACCI NUMBERS

$$L = 0, 1, 1, 2, 3, \dots$$

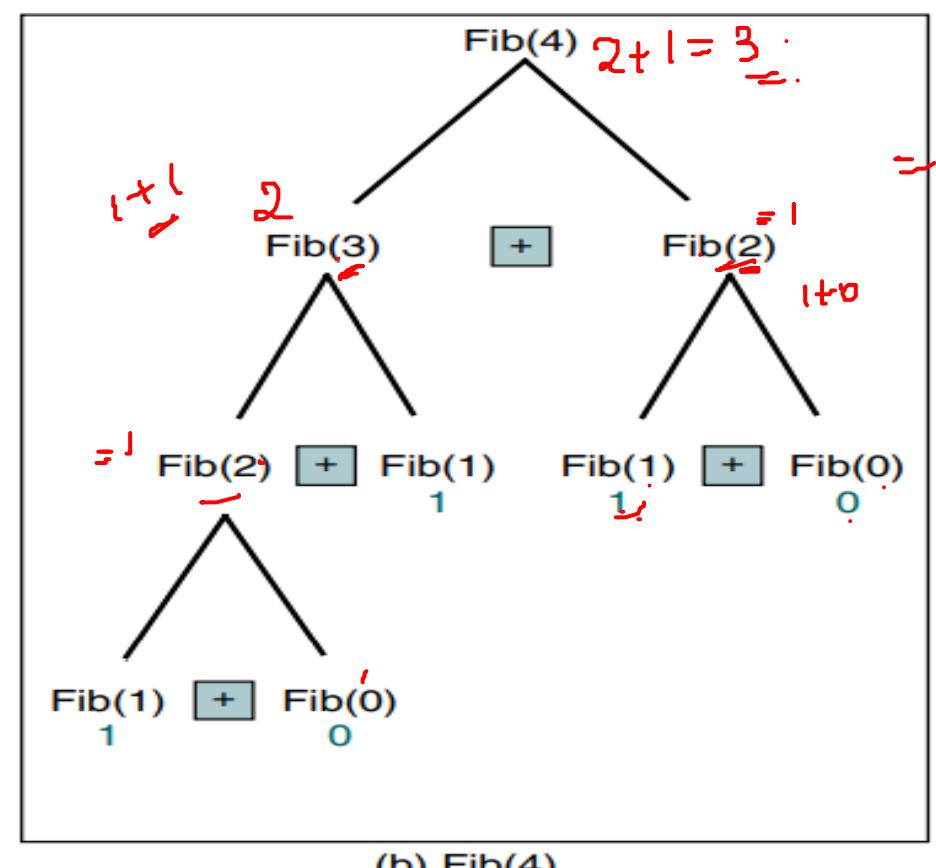
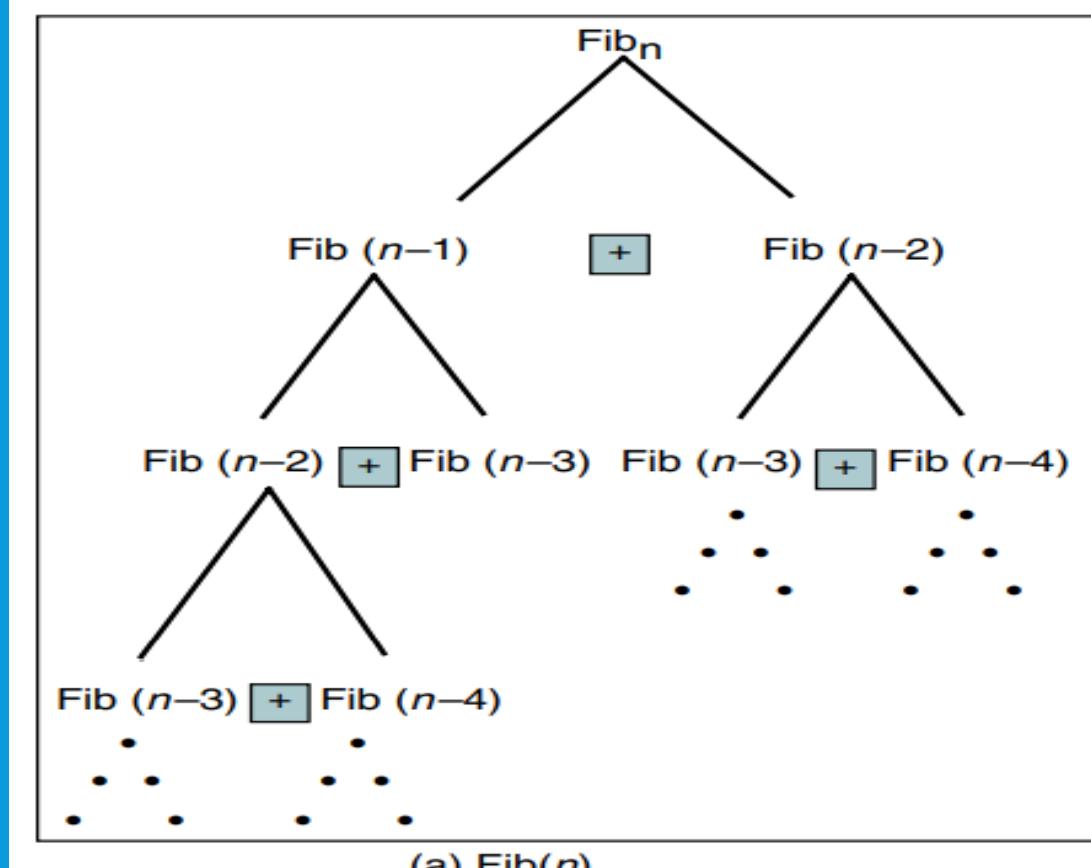
$$\text{Fibonacci } (n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \text{Fibonacci } (n - 1) + \text{Fibonacci } (n - 2) & \text{otherwise} \end{cases}$$

Fibonacci Numbers Recursive Definition $\underbrace{\text{Fibonacci } (n-1) + \text{Fibonacci } (n-2)}_{\text{if } n \geq 2} \quad n$

Fibonacci numbers are named after **Leonardo Fibonacci**, an Italian mathematician who lived in the early thirteenth century. In this series each number is the sum of the previous two numbers. The first few numbers in the Fibonacci series are:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \dots$$

FIBONACCI SERIES: DESIGN.



FIBONACCI NUMBERS IMPLEMENTATION.

```
7 // Prototype Statements
8     long fib (long num);
9
10 int main (void)
11 {
12 // Local Declarations
13     int seriesSize = 10;
14
15 // Statements
16     printf("Print a Fibonacci series.\n");
17
18     for (int looper = 0; looper < seriesSize; looper++)
19     {
20         if (looper % 5)
21             printf(", %8ld", fib(looper));
22         else
23             printf("\n%8ld", fib(looper));
24     } // for
25     printf("\n");
26     return 0;
27 } // main
```

```
29     /* ===== fib ===== */
30     Calculates the nth Fibonacci number
31     Pre  num identifies Fibonacci number
32     Post returns nth Fibonacci number
33 */
34     long fib (long num)
35     {
36 // Statements
37     , if (num == 0 || num == 1)
38         // Base Case
39         return num;
40     } // fib
41     return (fib (num - 1) + fib (num - 2));
```

Results:

Print a Fibonacci series.

0,	1,	1,	2,	3
5,	8,	13,	21,	34

FIBONACCI NUMBERS IMPLEMENTATION CALLS.

$\text{fib}(n)$	Calls	$\text{fib}(n)$	Calls
1		1	287
2		3	465
3		5	753
4		9	1219
5		15	1973
6		25	21,891
7		41	242,785
8		67	2,692,573
9		109	29,860,703
10	177	no=40	331,160,281

Recursive program for multiplying 2 numbers

```
int mul(int m, int n) {  
    int y;  
    if(m==0 || n==0)  
        return 0;  
    if(n==1)  
        return m;  
    else{  
        y=mul(m, n-1);  
        return(y+m);  
    }  
}
```

$$0 + \dots = 0$$

$$1 \times \dots = 1$$

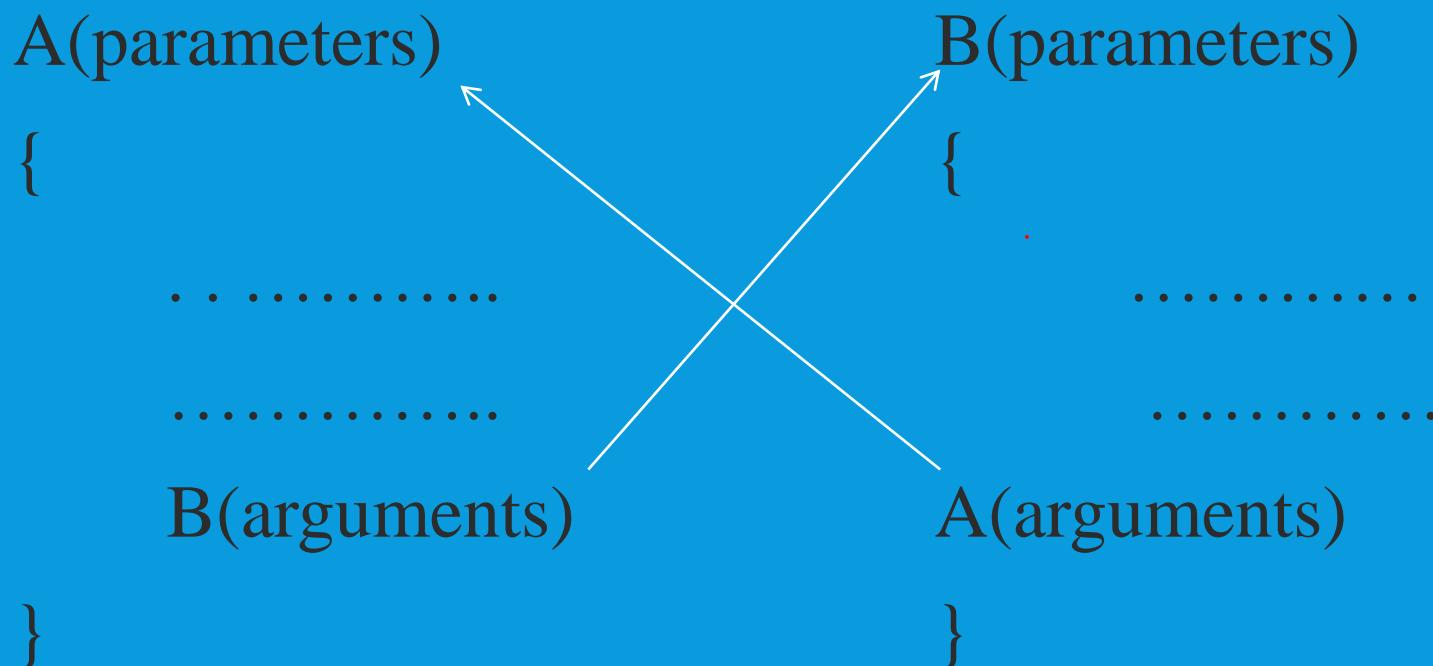


Recursive program to find the nth fibonacci number

```
int fib(int n)
{
    if(n==0)
        return 0;
    if(n==1)
        return 1;
    else
    {
        return (fib(n-1) + fib(n-2));
    }
}
```

Recursive chains

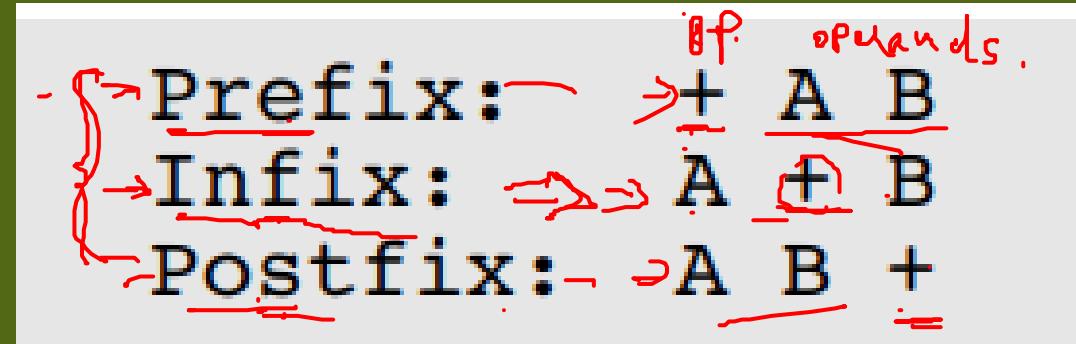
- Recursive function need not call itself directly. It can call itself indirectly as shown



SOME MORE RECURSIVE EXAMPLES: PREFIX TO POSTFIX CONVERSION

An arithmetic expression can be represented in three different formats:

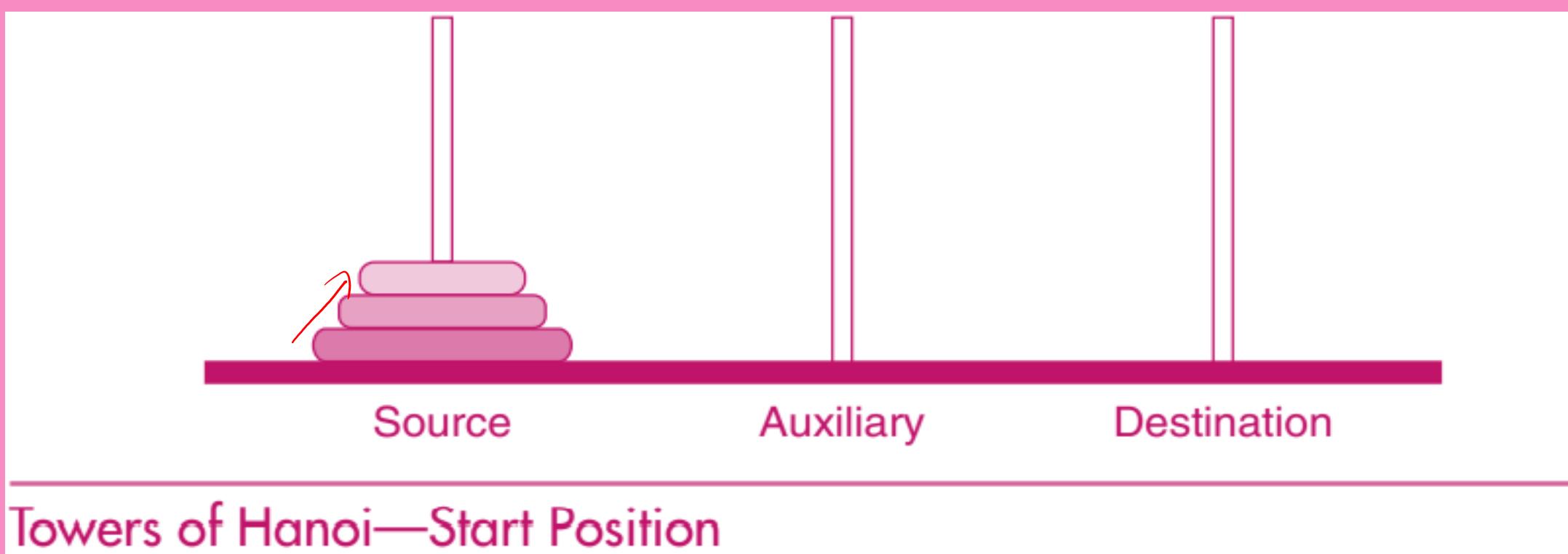
- 1. Infix
- { 2. Postfix
- 3. Prefix



1. **Prefix notation:**
2. **Infix notation:**
3. **Postfix notation:**

Operator **comes before** the operands.
Operator **comes between** the operands.
Operator **comes after** the operands

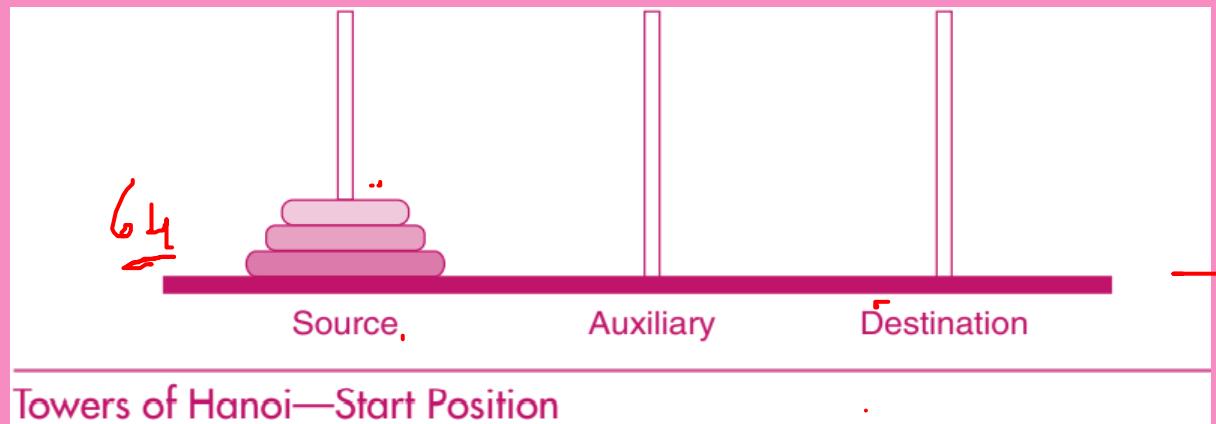
SOME MORE RECURSIVE EXAMPLES: THE TOWERS OF HANOI



According to the legend, the monks in a remote mountain monastery knew how to predict when the world would end.

SOME MORE RECURSIVE EXAMPLES: THE TOWERS OF HANOI

They had a set of **three diamond needles**. Stacked on the **first diamond needle** were **64 gold disks of decreasing size.**



The legend said that when all 64 disks had been transferred to the destination needle, the stars would be extinguished and *the world would end.*

Today we know we need to have $2^64 - 1$ moves to move all the disks.

RECURSIVE TOWERS OF HANOI: RULES.



Case: Hanoi Towers

The monks moved one disk to another needle each hour, subject to the following **rules**:

1. Only one disk could be moved at a time.
2. A larger disk must never be stacked above a smaller one.
3. One and only **one auxiliary needle** could be used for the intermediate storage of disks.

This problem is interesting for two reasons.

1. Recursive solution is much easier to code than the iterative solution would be, as is often the case with good recursive solutions.
2. Its solution pattern is different from the simple examples we have been discussing

RECURSIVE TOWERS OF HANOI: DESIGN.



CASE 1: ONE DISK

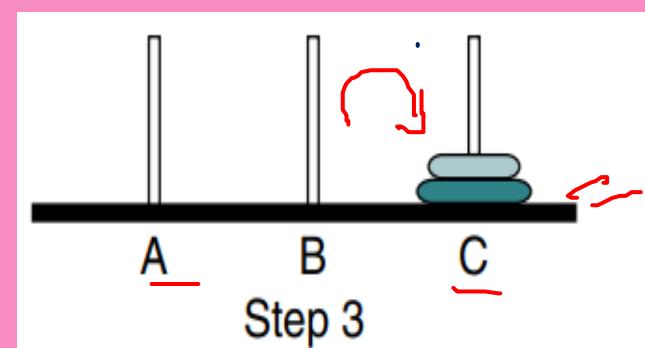
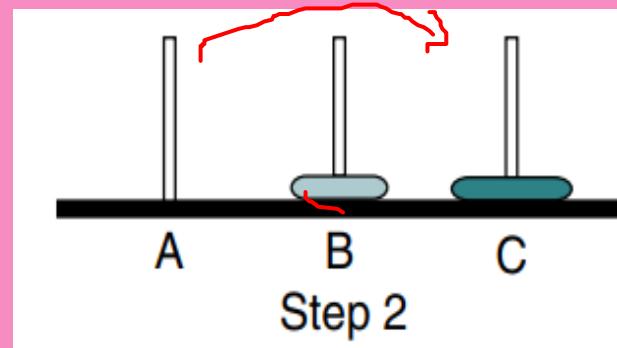
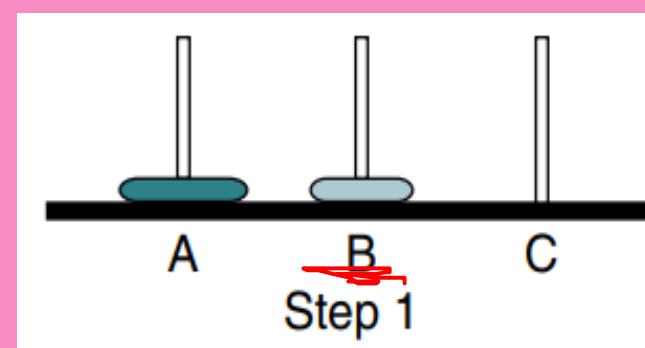
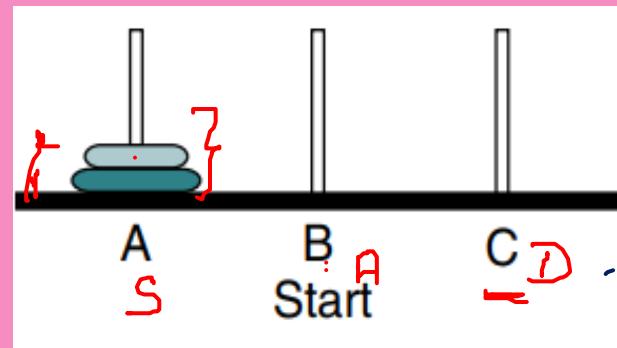
→ Move one disk from source to destination needle

RECURSIVE TOWERS OF HANOI: DESIGN.



CASE 1: TWO DISKS

1. Move one disk to auxiliary needle.
2. Move one disk to destination needle
3. Move one disk from auxiliary to destination needle.



Towers Solution for Two Disks

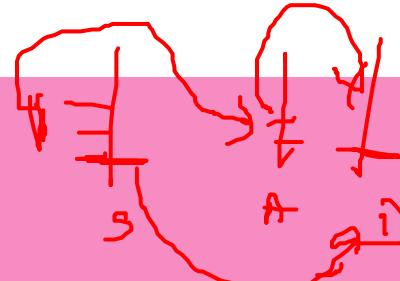
RECURSIVE TOWERS OF HANOI: DESIGN.



CASE 3: THREE DISKS

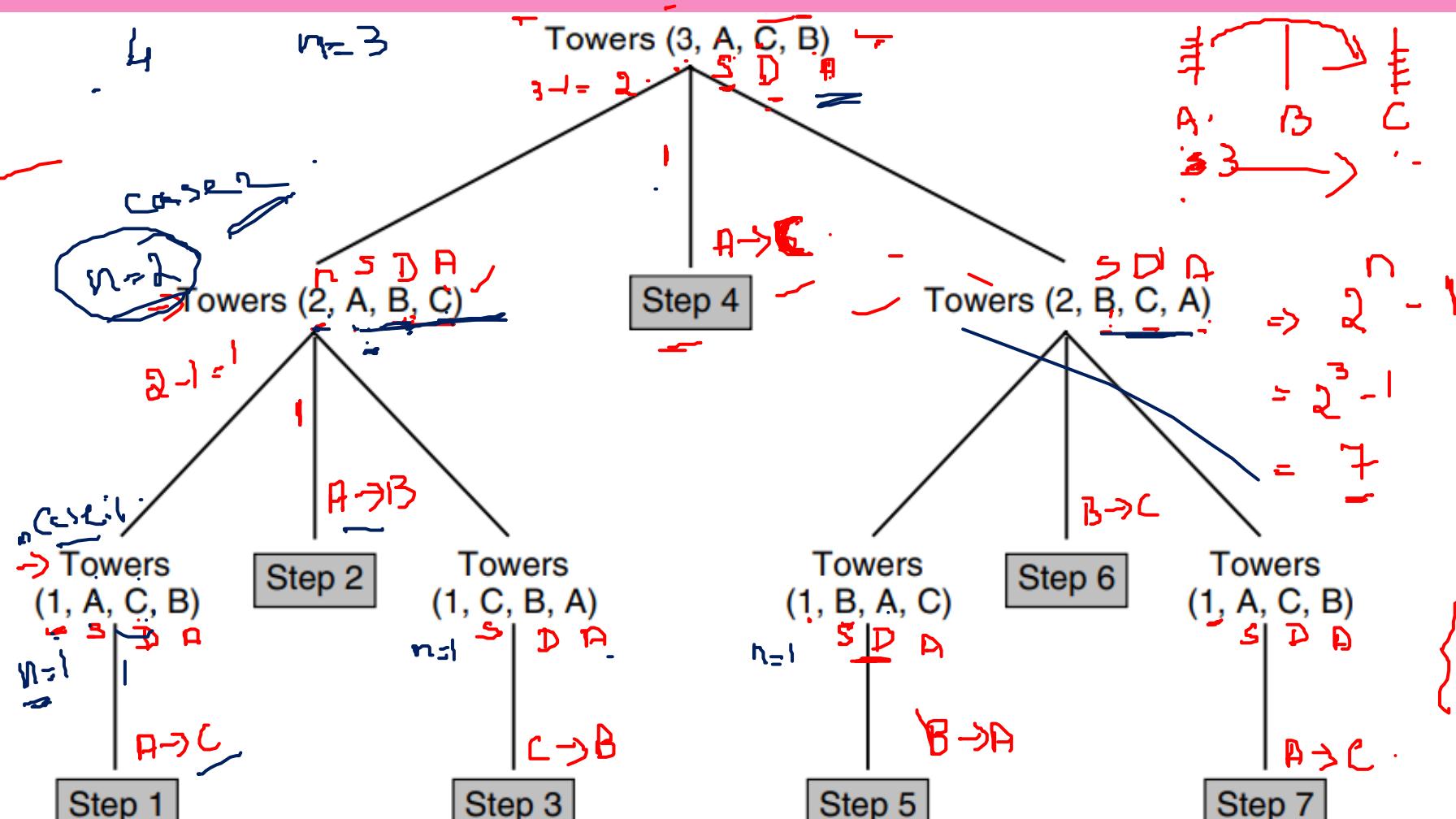
$$n=3 \quad 3 = \frac{3-1}{n-1} = \frac{2}{2}$$

of more (n)



1. Move $n-1$ disks from source to auxiliary. \rightarrow General case
2. Move one disk from source to destination. \rightarrow Base case
3. Move $n-1$ disks from auxiliary to destination. \rightarrow General ?

General case



~~$n=3$~~ Algorithm: ~~Towers of Hanoi~~

$[A \rightarrow C, A \rightarrow B, C \rightarrow B, A \rightarrow C]$
 $[B \rightarrow A, B \rightarrow C, A \rightarrow C]$

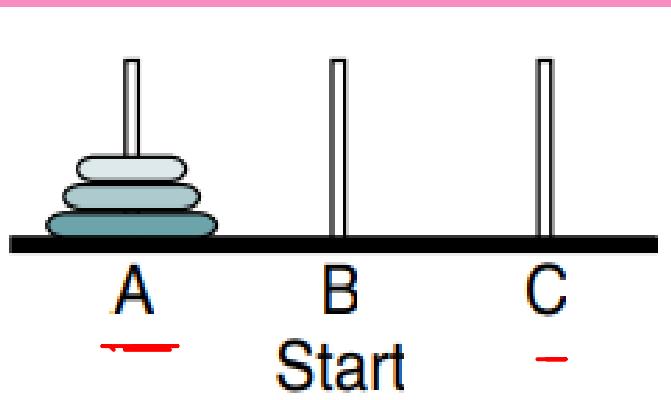
1. Move $n-1$ disks from source to auxiliary.
2. Move one disk from source to destination.
3. Move $n-1$ disks from auxiliary to destination.

Algorithm ~~towers~~ (numDisks, source, dest, auxiliary)

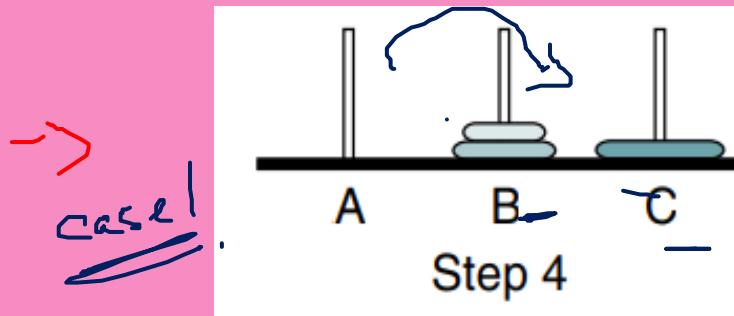
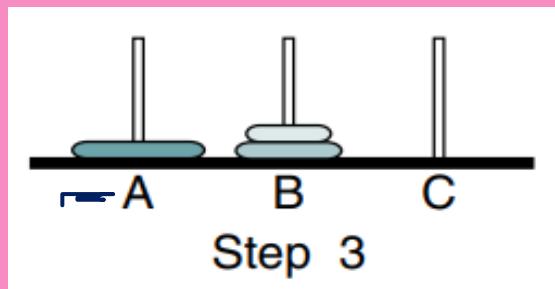
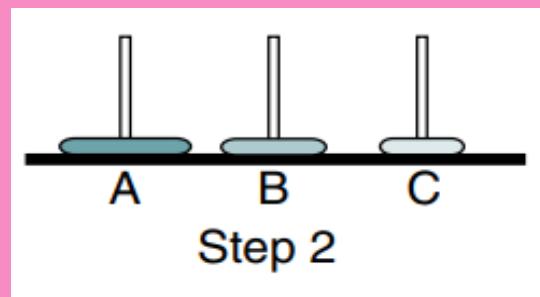
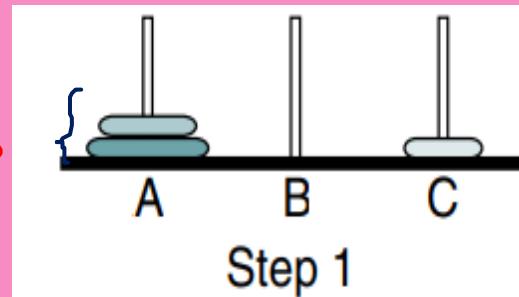
1. { Call Towers ($n - 1$, source, auxiliary, destination)
2. Move one disk from source to destination
3. Call Towers ($n - 1$, auxiliary, destination, source)

RECURSIVE TOWERS OF HANOI: DESIGN.

CASE 3: THREE DISKS



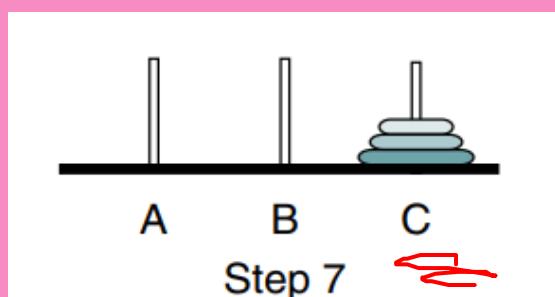
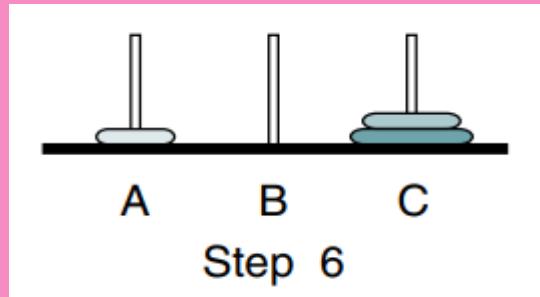
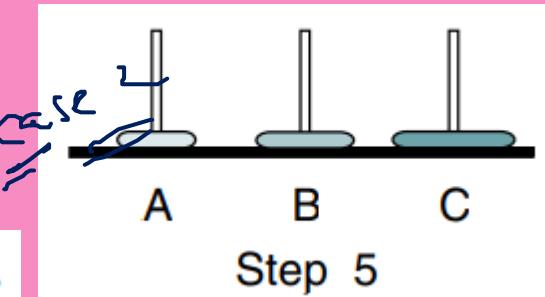
case 2



Move one disk from source to destination.

$n=3$

→



Towers Solution for Three Disks

Towers of Hanoi

```
Algorithm towers (numDisks, source, dest, auxiliary)
Recursively move disks from source to destination.
Pre numDisks is number of disks to be moved
      source, destination, and auxiliary towers given
Post steps for moves printed
1 print("Towers: ", numDisks, source, dest, auxiliary)
2 if (numDisks is 1) →
   1 print ("Move from ", source, " to ", dest)
3 else
   1 towers (numDisks - 1, source, auxiliary, dest, step)
   2 print ("Move from " source " to " dest) →
   3 towers (numDisks - 1, auxiliary, dest, source, step)
4 end if
end towers
```

Algorithm: Towers of Hanoi.

Calls:

Towers (3, A, C, B)
Towers (2, A, B, C)
Towers (1, A, C, B)

Towers (1, C, B, A)

Towers (2, B, C, A)
Towers (1, B, A, C)

Towers (1, A, C, B)

Output:

Move from A to C
Move from A to B

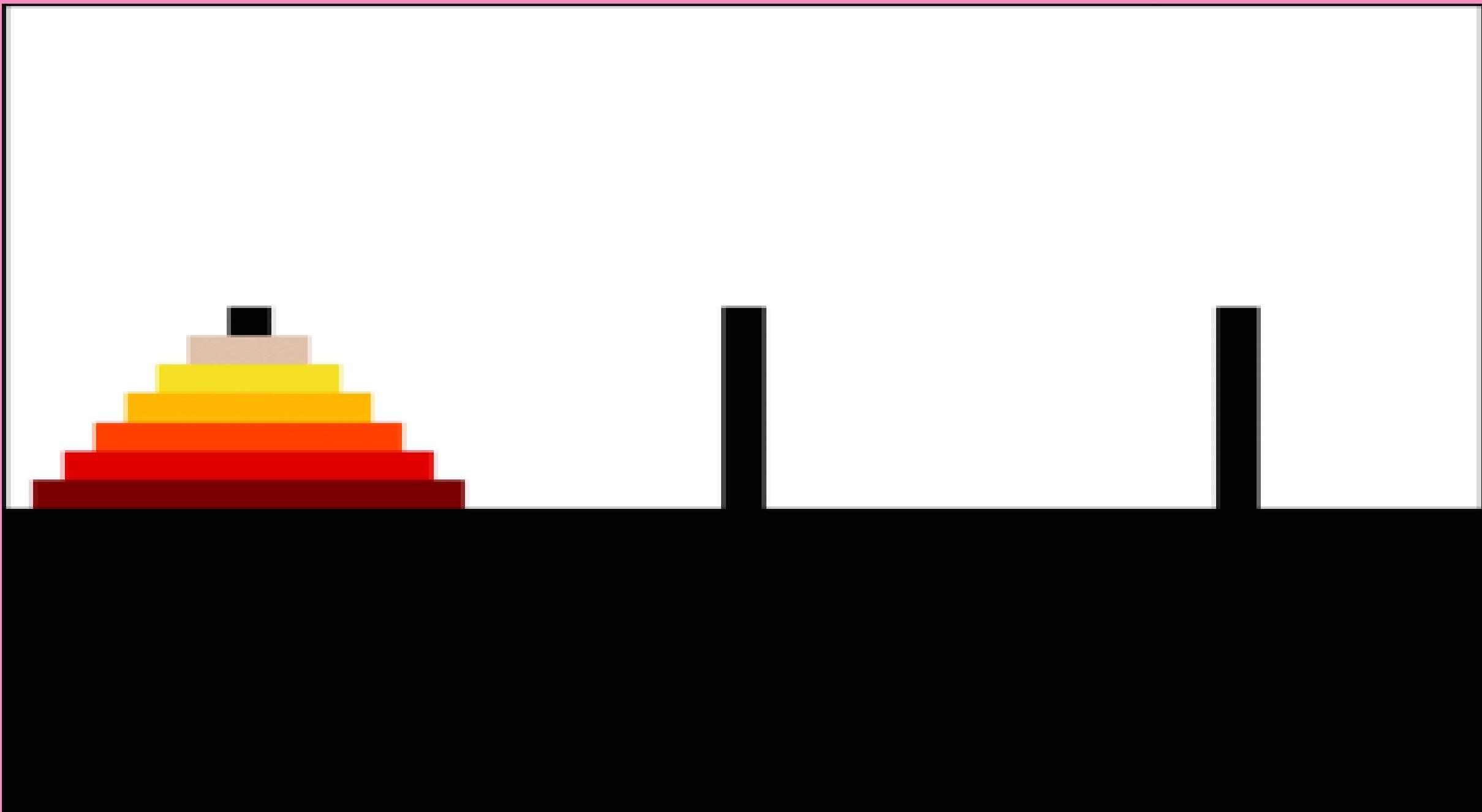
Move from C to B
Move from A to C

Move from B to A
Move from B to C

Move from A to C

Tracing Algorithm 2-7, Towers of Hanoi

```
Algorithm towers (numDisks, source, dest, auxiliary)
    print("Towers: ", numDisks, source, dest, auxiliary)
    if (numDisks is 1)
        1   print ("Move from ", source, " to ", dest)
    else
        1   towers (numDisks - 1, source, auxiliary, dest, s)
        2   print ("Move from " source " to " dest)
        3   towers (numDisks - 1, auxiliary, dest, source, s)
    end if
```



C program for tower of hanoi problem

```
void tower (int n, char source, char temp, char destination) {  
    if(n==1) {  
        cout<<“move disk 1 from “<<source<<“ to “<<destination<<endl;  
        return;  
    }  
  
    /*moving n-1 disks from A to B using C as auxiliary*/  
    tower(n-1, source, destination, temp);  
  
    cout<<“move disk “<<n<<“ from “<<source<<“ to  
    “<<destination<<endl;  
  
    /*moving n-1 disks from B to C using A as auxiliary*/  
    tower(n-1, temp, source, destination);  
}
```

LENGTH OF A STRING USING RECURSION

```
int StrLen(str[], int index)
{
    if (str[index] == '\u0000') return 0;
    return (1 + StrLen(str, index + 1));
}
```

College\0:

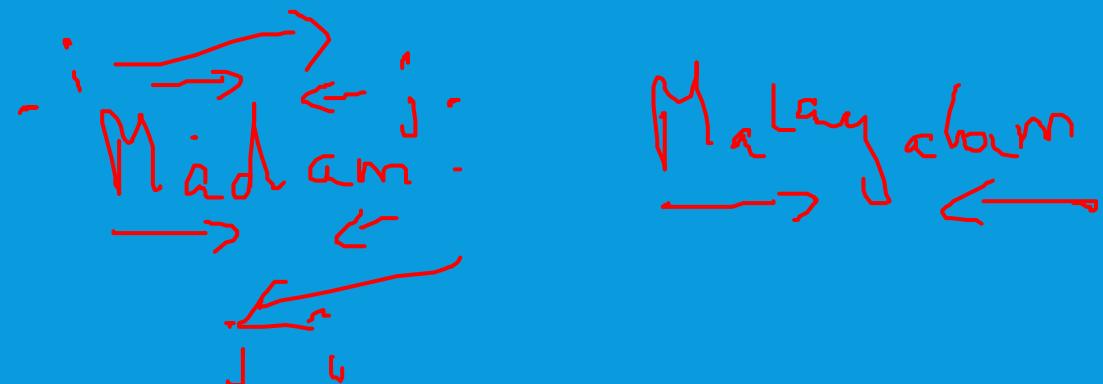
LENGTH OF A STRING USING RECURSION USING STATIC VARIABLE

```
int StrLen(char *str)
{
    static int length=0;
    if(*str != '\0')
    {
        length++; →
        StrLen(++str);
    }
    return length;
}
```

length = 0
length = 1
length = 2

TO CHECK WHETHER A GIVEN STRING IS PALINDROME OR NOT USING RECURSION

```
int isPalindrome(char *inputString, int leftIndex, int rightIndex) {  
    /* Recursion termination condition */  
    if(leftIndex >= rightIndex) return 1;  
    if(inputString[leftIndex] == inputString[rightIndex]) {  
        return isPalindrome(inputString, leftIndex + 1, rightIndex - 1);  
    }  
    return 0;  
}
```



```
int main(){
    char inputString[100];
    printf("Enter a string for palindrome check\n");
    scanf("%s", inputString);
    if(isPalindrome(inputString, 0, strlen(inputString) - 1))
        printf("%s is a Palindrome \n", inputString);
    else
        printf("%s is not a Palindrome \n", inputString);
    getch();
    return 0;
}
```

TO COPY ONE STRING TO ANOTHER USING RECURSION

```
void copy(char str1[], char str2[], int index)
{
    str2[index] = str1[index];
    if (str1[index] == '\0') return;
    copy(str1, str2, index + 1);
}
```

Advantages of Recursion

Clearer and simpler versions of algorithms can be created using recursion.

2. Recursive definition of a problem can be easily translated into a recursive function.
3. Lot of book keeping activities such as initialization etc. required in iterative solution is avoided.

Disadvantages

1. When a function is called, the function saves formal parameters, local variables and return address and hence consumes a lot of memory.
2. Lot of time is spent in pushing and popping and hence consumes more time to compute result.

Iteration

Uses loops

Counter controlled and body of loop terminates when the termination condition fails.

- Execution is faster and takes less space.
- Difficult to design for some problems.

Recursion

uses if-else and repetitive function calls

Terminates when base condition is reached.

Consumes time and space because of push and pop.

Best suited for some problems and easy to design.

Exercise

1. Consider the following algorithm:

```
algorithm fun1 (x)  
1 if (x < 5)  
    1 return (3 * x)  
2 else  
    1 return (2 * fun1 (x - 5) + 7)  
3 end if  
end fun1
```

What would be returned if `fun1` is called as

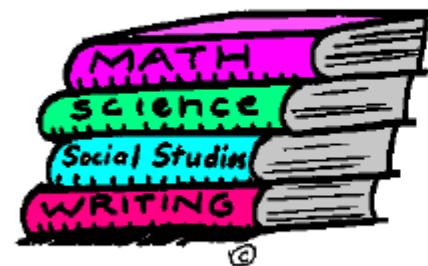
- a. `fun1 (4)?`
- b. `fun1 (10)?`
- c. `fun1 (12)?`

The End.
OF RECURSION.



STACKS- DSE

- The linked list only allows for sequential traversal
- Sequential traversal is present in stacks and queues
- Linked lists are used to implement these

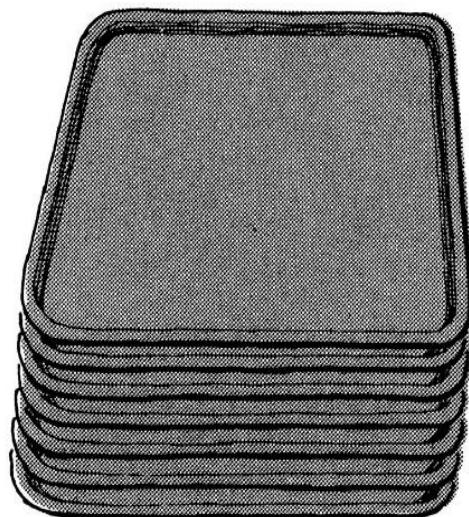


Stack



Queue

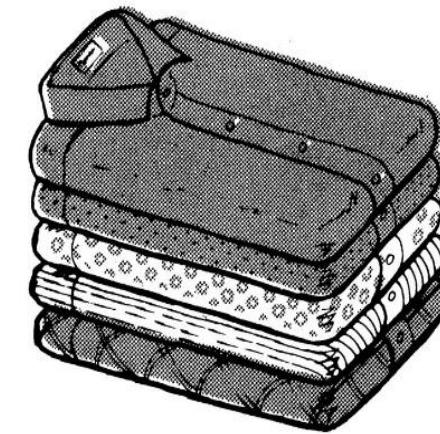
A stack of
cafeteria trays



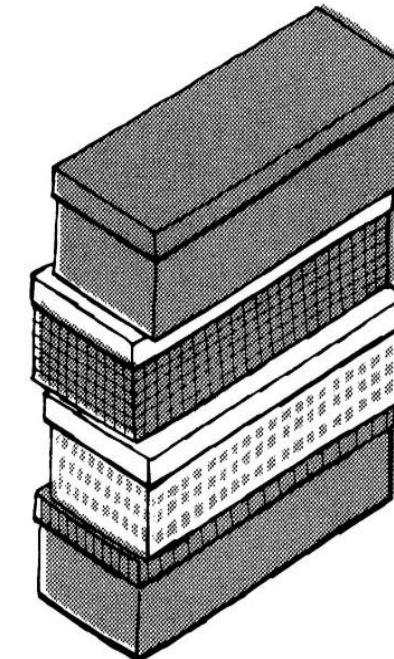
A stack
of pennies



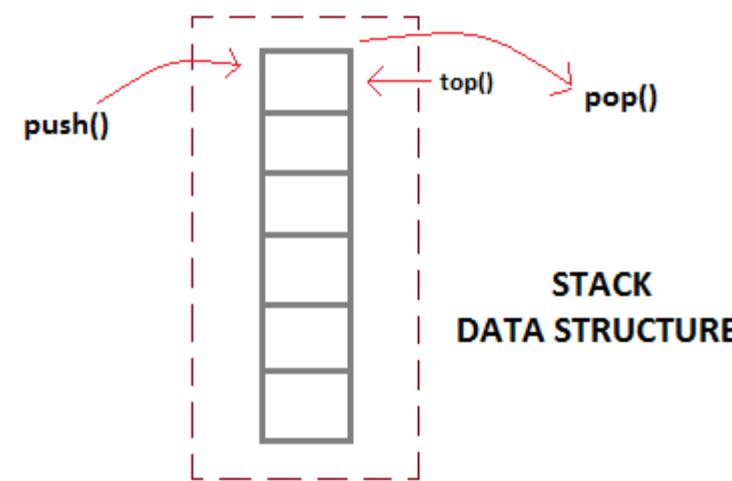
A stack of
neatly folded shirts



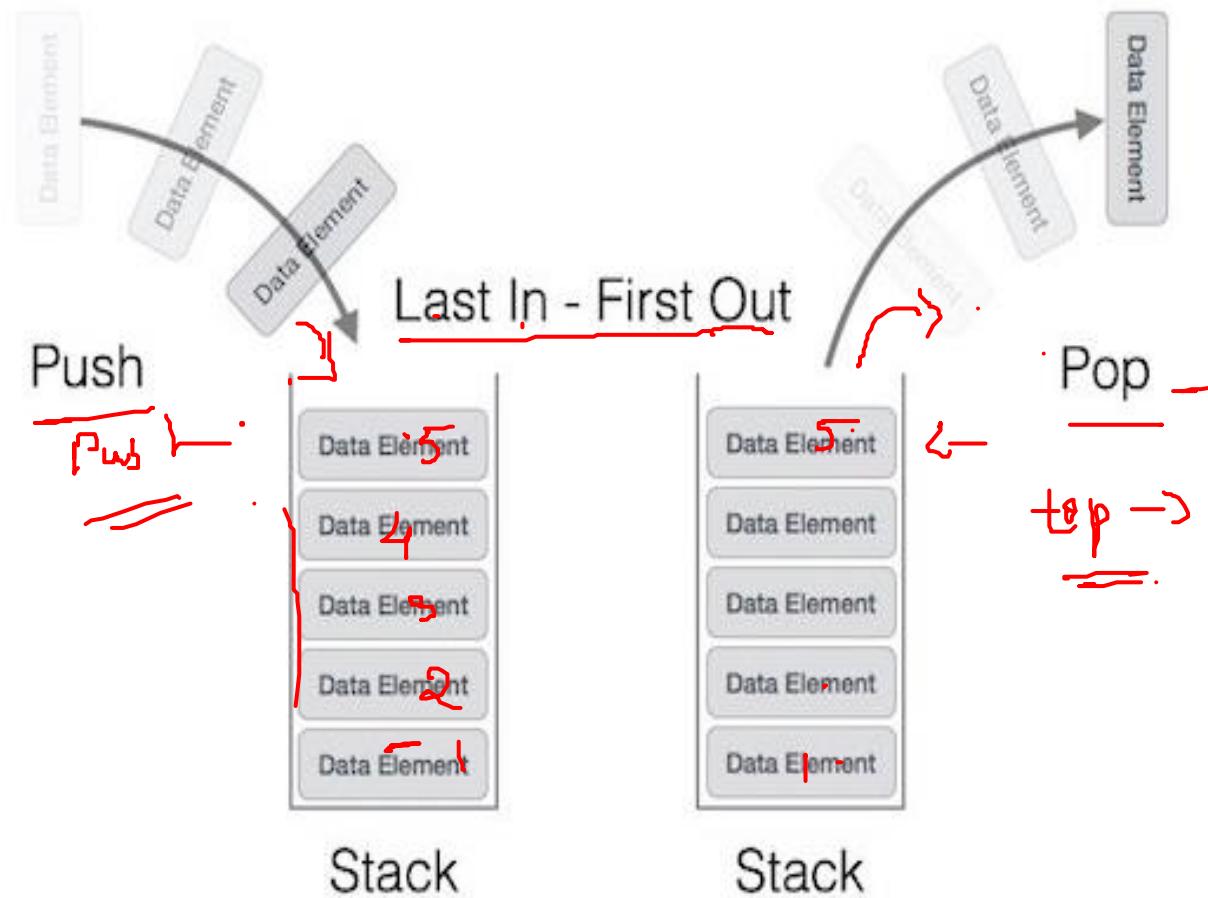
A stack of shoe boxes



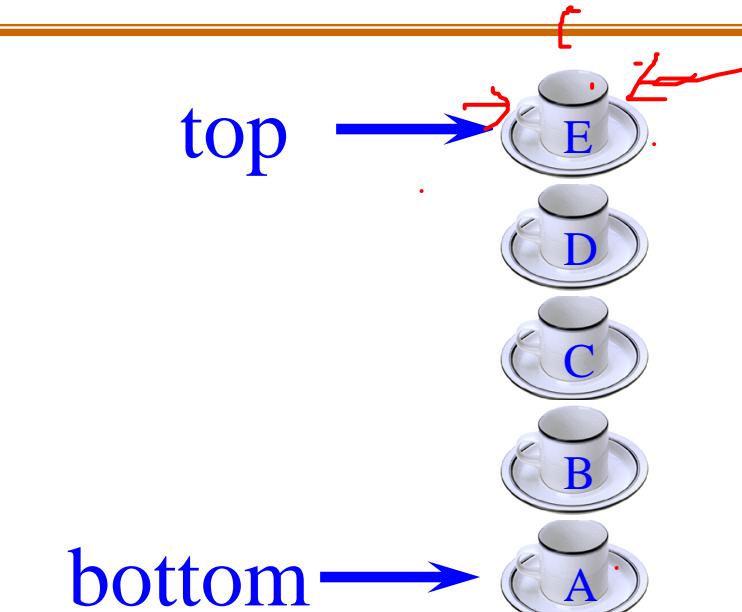
Stack is an abstract data type with a bounded(predefined) capacity. It is a simple data structure that allows adding and removing elements in a particular order. Every time an element is added, it goes on the **top** of the stack and the only element that can be removed is the element that is at the top of the stack.



Stack Representation



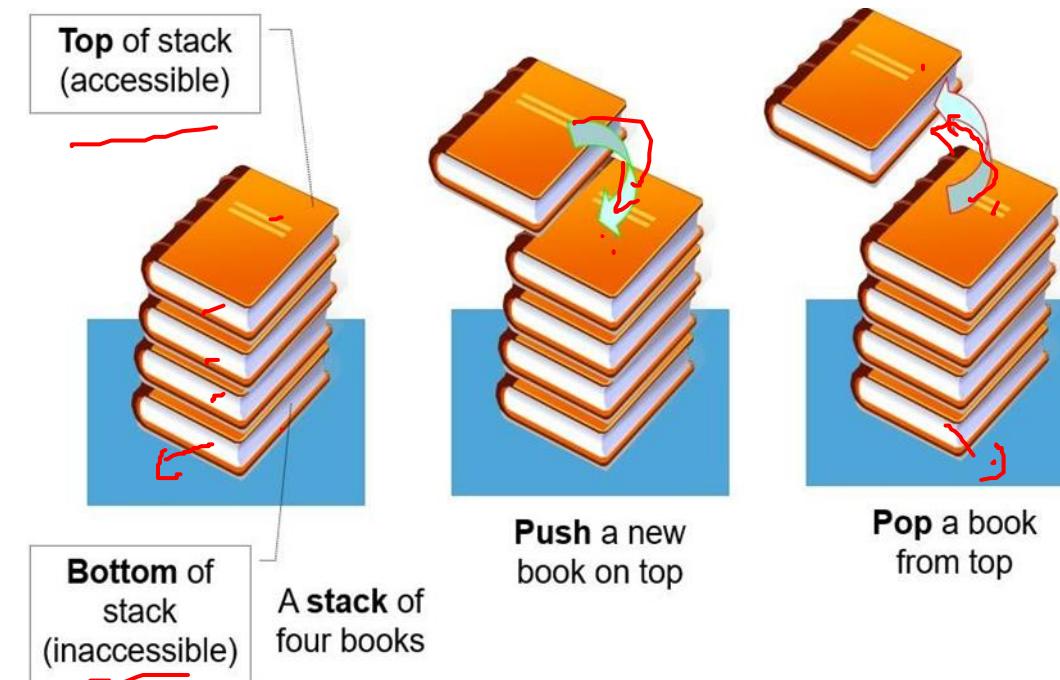
- Linear list.
- One end is called top.
- Other end is called bottom.
- Additions to and removals from
the top end only.



- Insert at top of stack and remove from top of stack
- Stack operations also called Last-In First-Out (LIFO)

Stack Operations: Push and Pop

- **Push:** insert at the top/beginning of stack
- **Pop:** delete from the top/beginning of stack

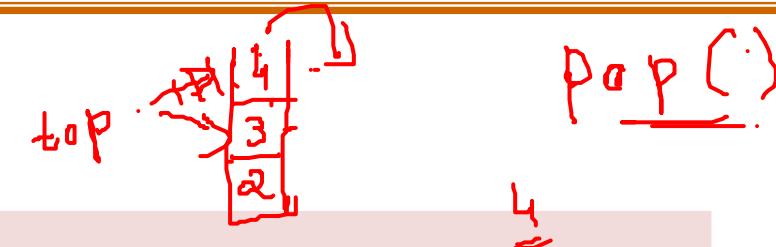


Stack Operations: Push and Pop

Syntax :

stackname.push(value)

Push [4]



Parameters : The value of the element to be inserted is passed as the parameter.

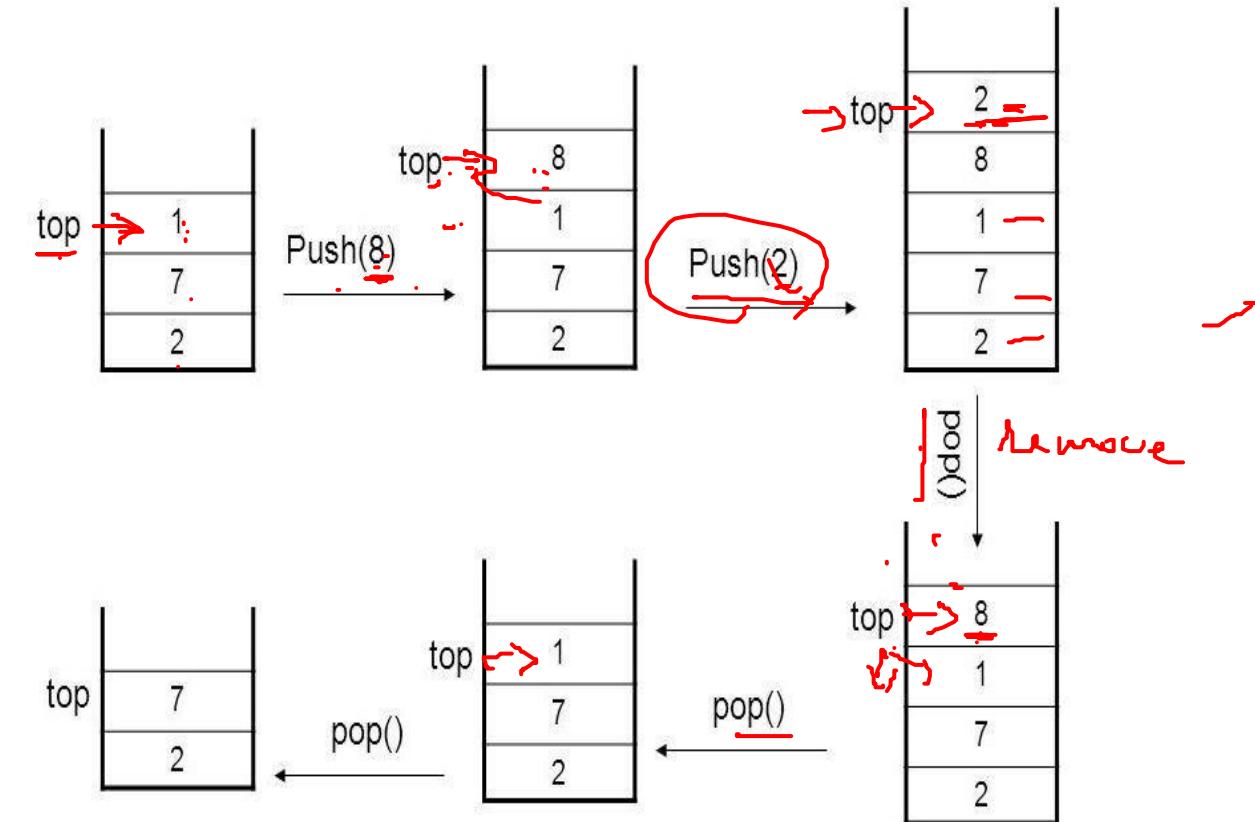
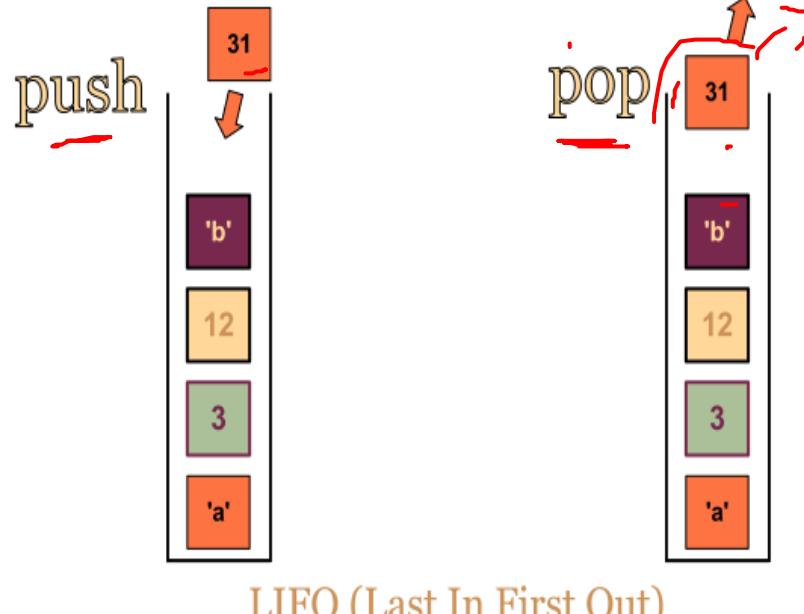
Result : Adds an element of value same as that of the parameter passed at the top of the stack.

stackname.pop()

Parameters : No parameters are passed.

Result : Removes the newest element in the stack or basically the top element.

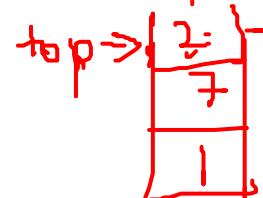
STACK



Stack Operations: Push and Pop

check the status of stack

- peek() – get the ^{top} top data element of the stack, without removing it.



~~2 → value~~

- isFull() – check if stack is full.



~~stack[20] =~~

- isEmpty() – check if stack is empty.



Algorithm of peek() function

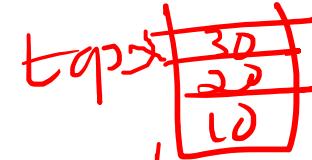
```
begin procedure peek →  

    return stack[top]  

end procedure
```

int peek()

return stack[top];



Stack [10]

Algorithm of isfull() function

```
begin procedure isfull →  

if top equals to MAXSIZE  

    return true  

else  

    return false  

endif  

end procedure
```

Yes
No

bool isfull()

if(top == MAXSIZE)
 return true;
else
 return false;

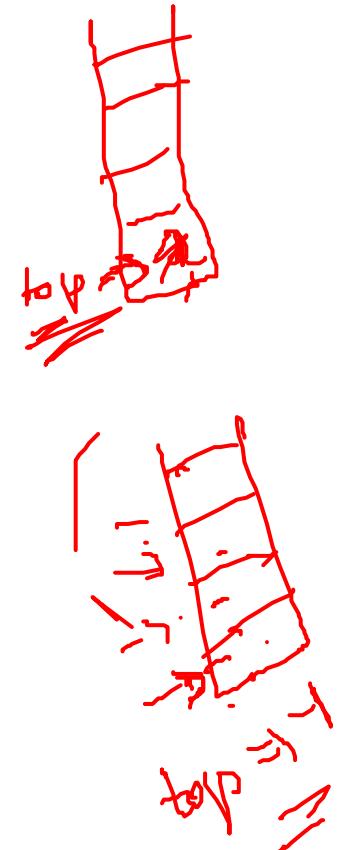
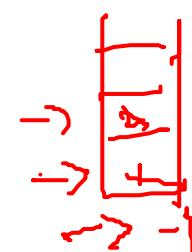


Stack Operations: Push and Pop

isempty()

```
begin procedure isempty
  if top less than 1
    return true
  else return false
  endif
end procedure
```

```
bool isempty()
{
  if (top == -1)
    return true;
  else
    return false;
}
```



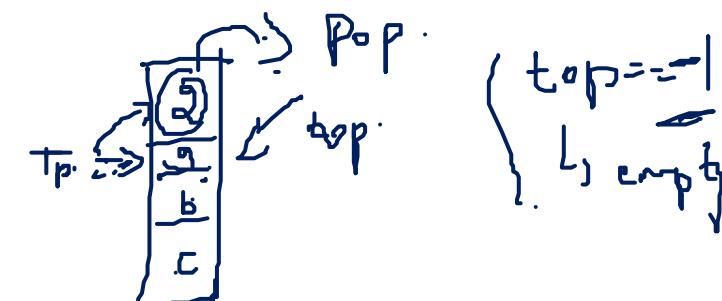
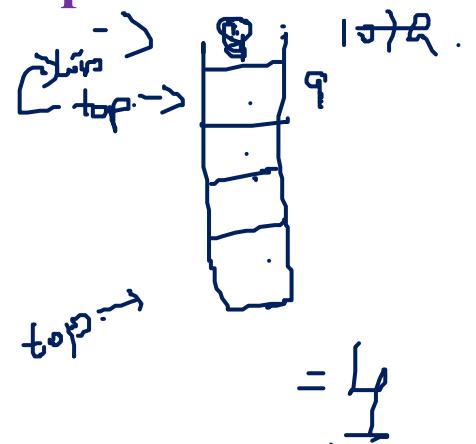
Stack Operations: Push and Pop

Push Operation

The process of putting a new data element onto stack is known as a **Push Operation**.

Push operation involves a series of steps –

- **Step 1** – Checks if the stack is full.
- **Step 2** – If the stack is full, produces an error and exit.
- **Step 3** – If the stack is not full, increments top to point next empty space.
- **Step 4** – Adds data element to the stack location, where top is pointing.
- **Step 5** – Returns success.



{
 1) Push → empty → exit
 2) Decrement top, delete element

Stack Operations: Push and Pop

Algorithm for PUSH and POP Operation

```

begin procedure push: stack, data
    if stack is full
        return null
    endif
    → top ← top + 1
    → stack[top] ← data
end procedure

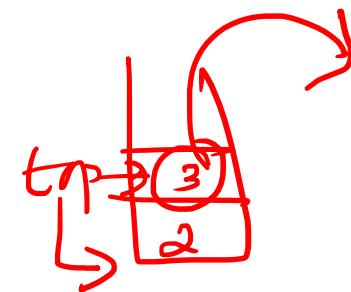
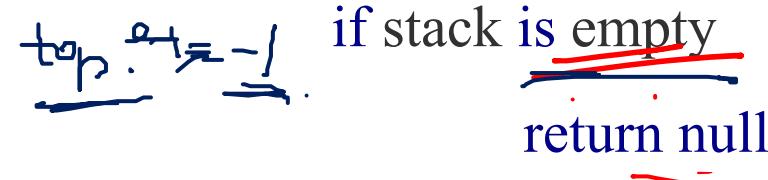
```



```

begin procedure pop: stack
    if stack is empty
        return null
    endif
    {data} ← stack[top]
    top ← top - 1
    return data
end procedure

```



Implementation of Stack Data Structure

Stack can be easily implemented using an Array or a Linked List.

- Arrays are quick, but are limited in size.



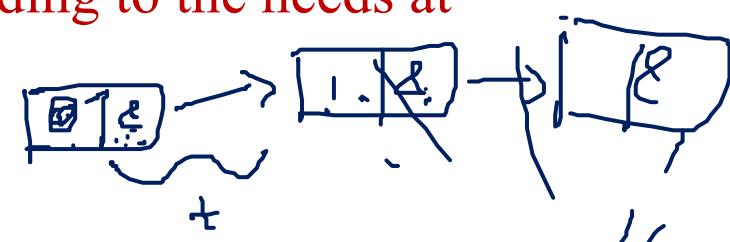
Pros: Easy to implement. Memory saved as pointers are not involved.

Cons: It is not dynamic. It doesn't grow and shrink depending on needs at runtime.

- Linked List requires overhead to allocate, link, unlink, and deallocate, but is not limited in size.

Pros: The linked list implementation of stack can grow and shrink according to the needs at runtime.

Cons: Requires extra memory due to involvement of pointers.



Stack - Relavance

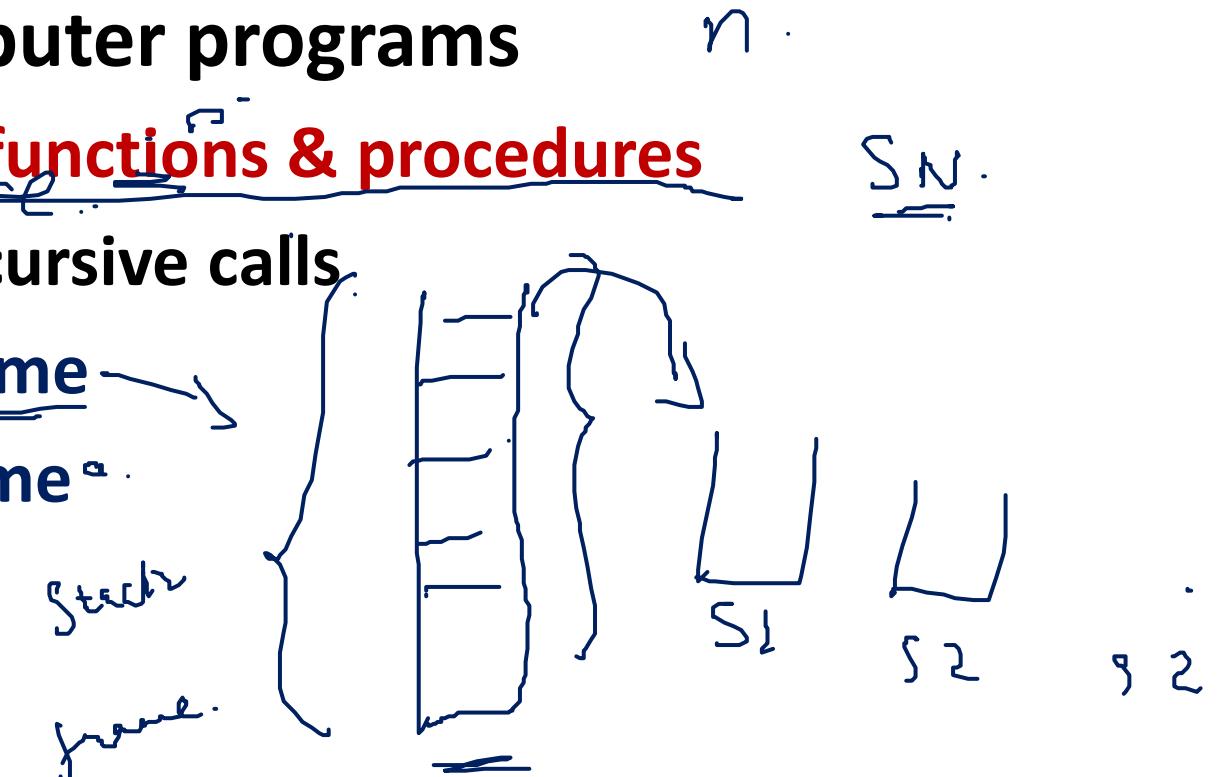
- Stacks appear in computer programs

- Key to call / return in functions & procedures

- Stack frame allows recursive calls

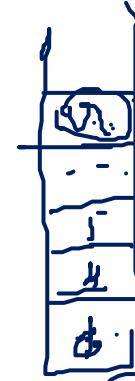
- Call: push stack frame

- Return: pop stack frame



Stack - Relavance

- Stacks appear in computer programs
 - Key to call / return in functions & procedures
 - Stack frame allows recursive calls
 - Call: push stack frame
 - Return: pop stack frame
- Stack frame
 - Function arguments
 - Return address
 - Local variables ↗



Use of Stacks in Function call – System Stack



- Whenever a function is invoked program creates a structure called activation record or a stack frame and places it on top of system stack.
- Initially, the activation record for the invoked functions contains only a pointer to the previous frame and return address.
- The previous stack frame pointer points to the stack frame of invoking function.



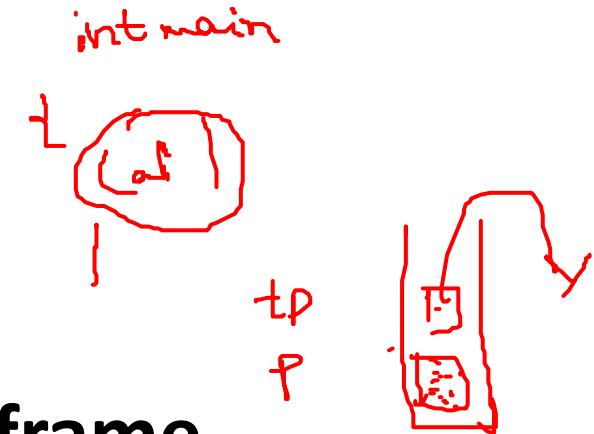


Use of Stacks in Function call – System Stack

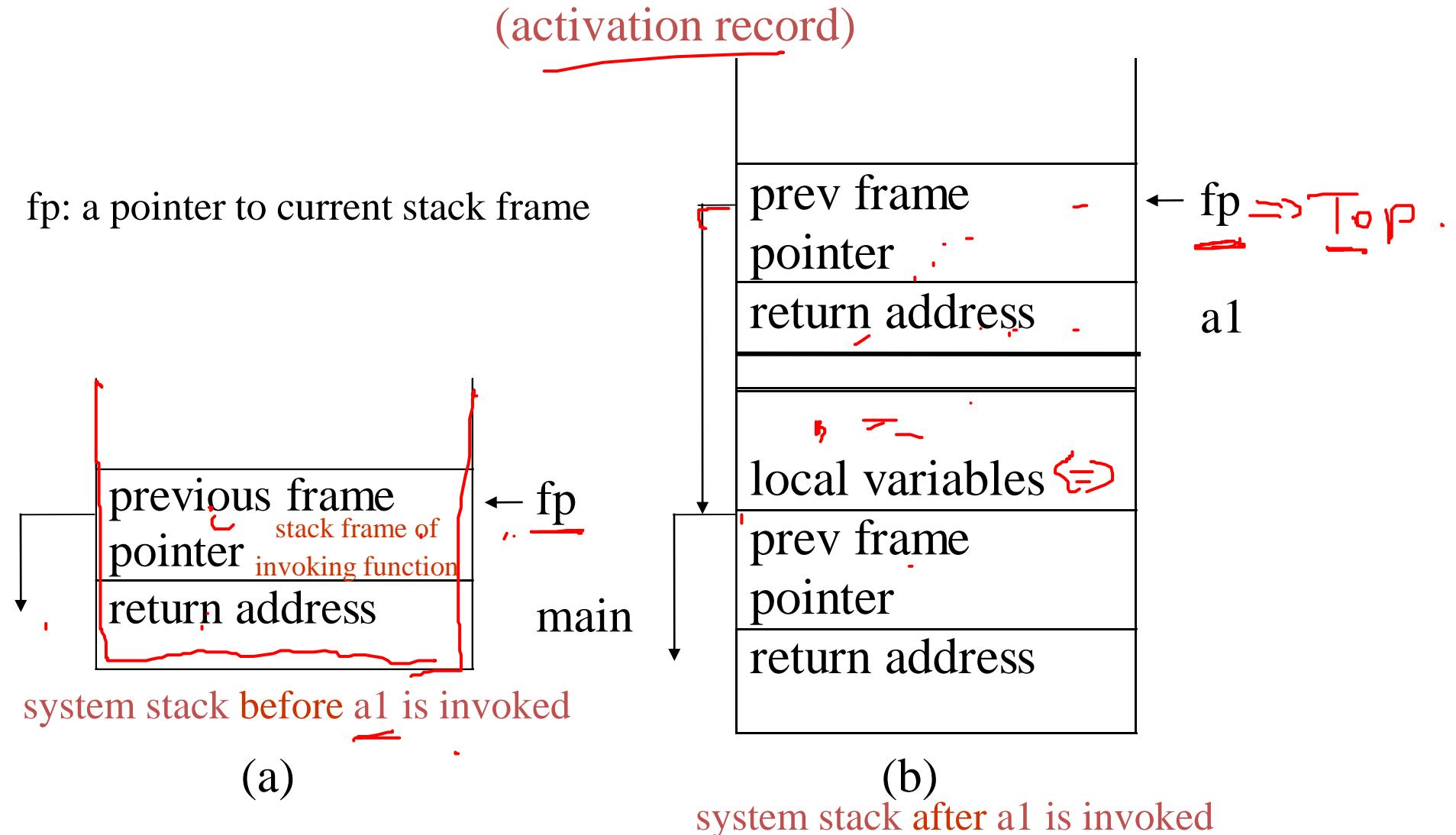
- Return address contains the location of the statement to be executed after the function terminates.
- Only one function executes at given time which is the top of stack.
- If this function invokes another, the non-static local variables parameters of invoking function are added to stack frame.

Stacks in Function call – System Stack

- Assume that main() invokes function a1.
- It creates stack frame for a1.
- Frame pointer is a pointer to the current stack frame
- Also system maintains a stack pointer separately
- When a function terminates its stack frame is removed
- Process of invoking which is on top of stack continues

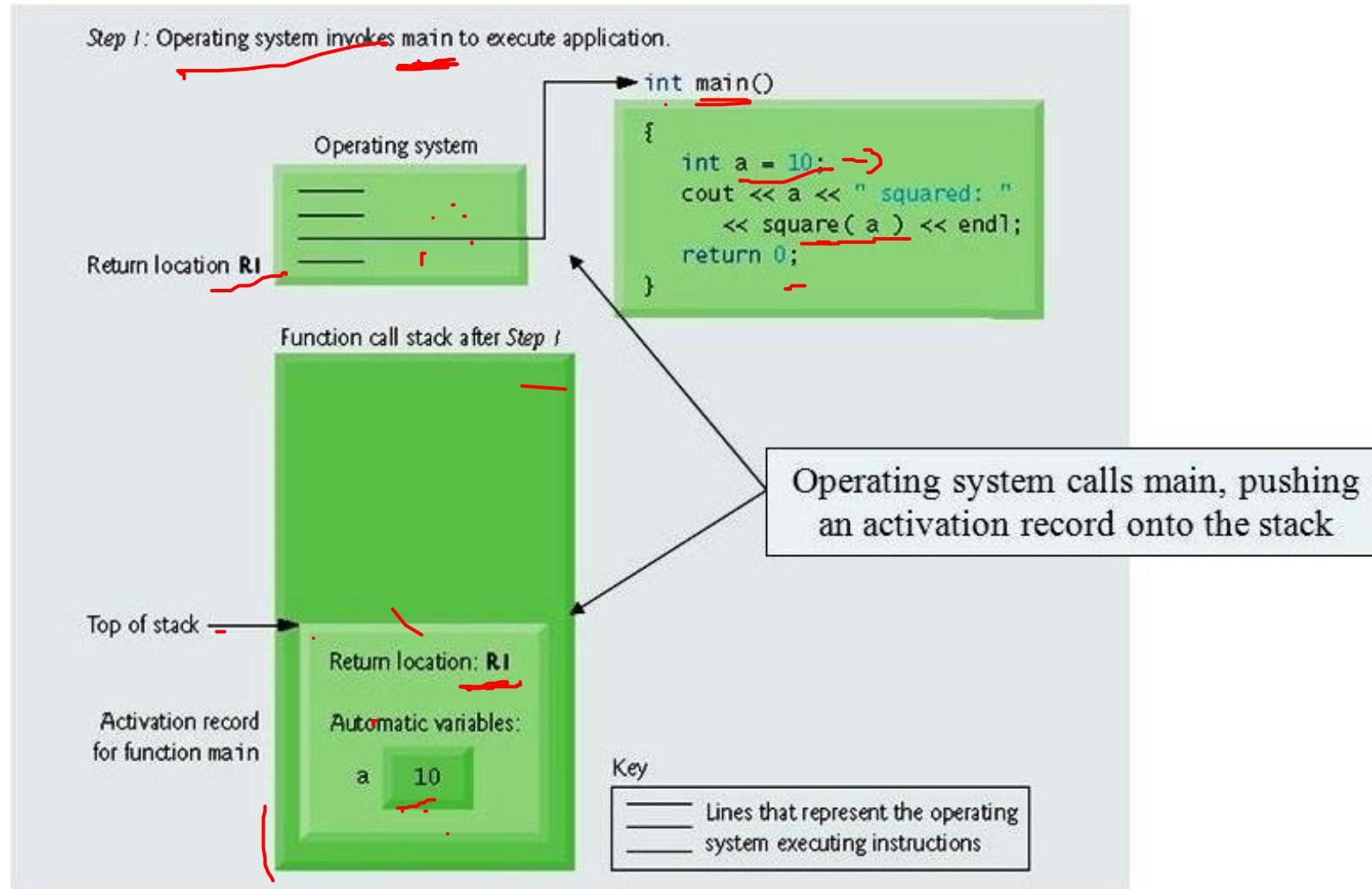


An application of stack: stack frame of function call



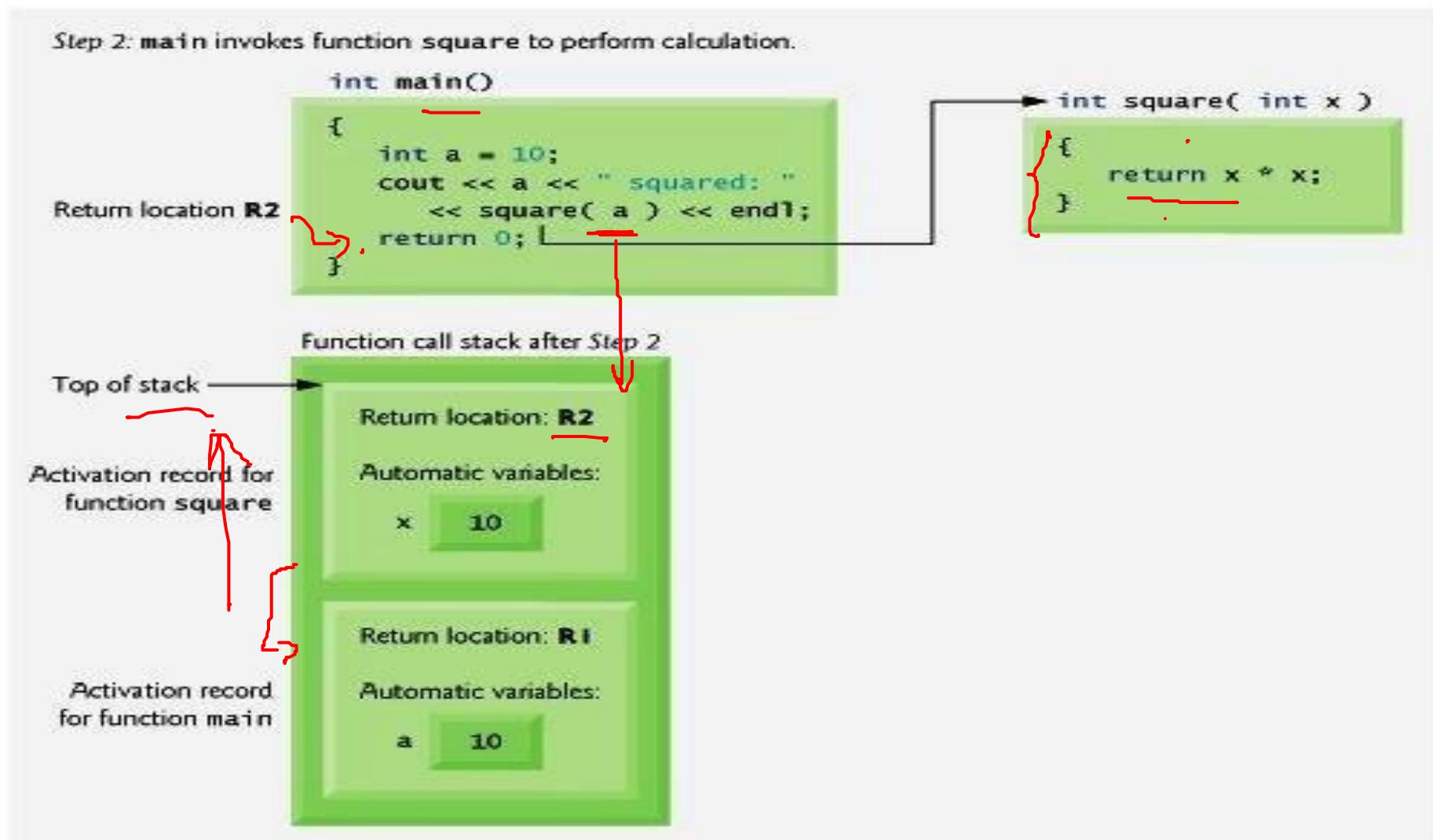
Applications of Stack

■ Function Call (Continued ...)



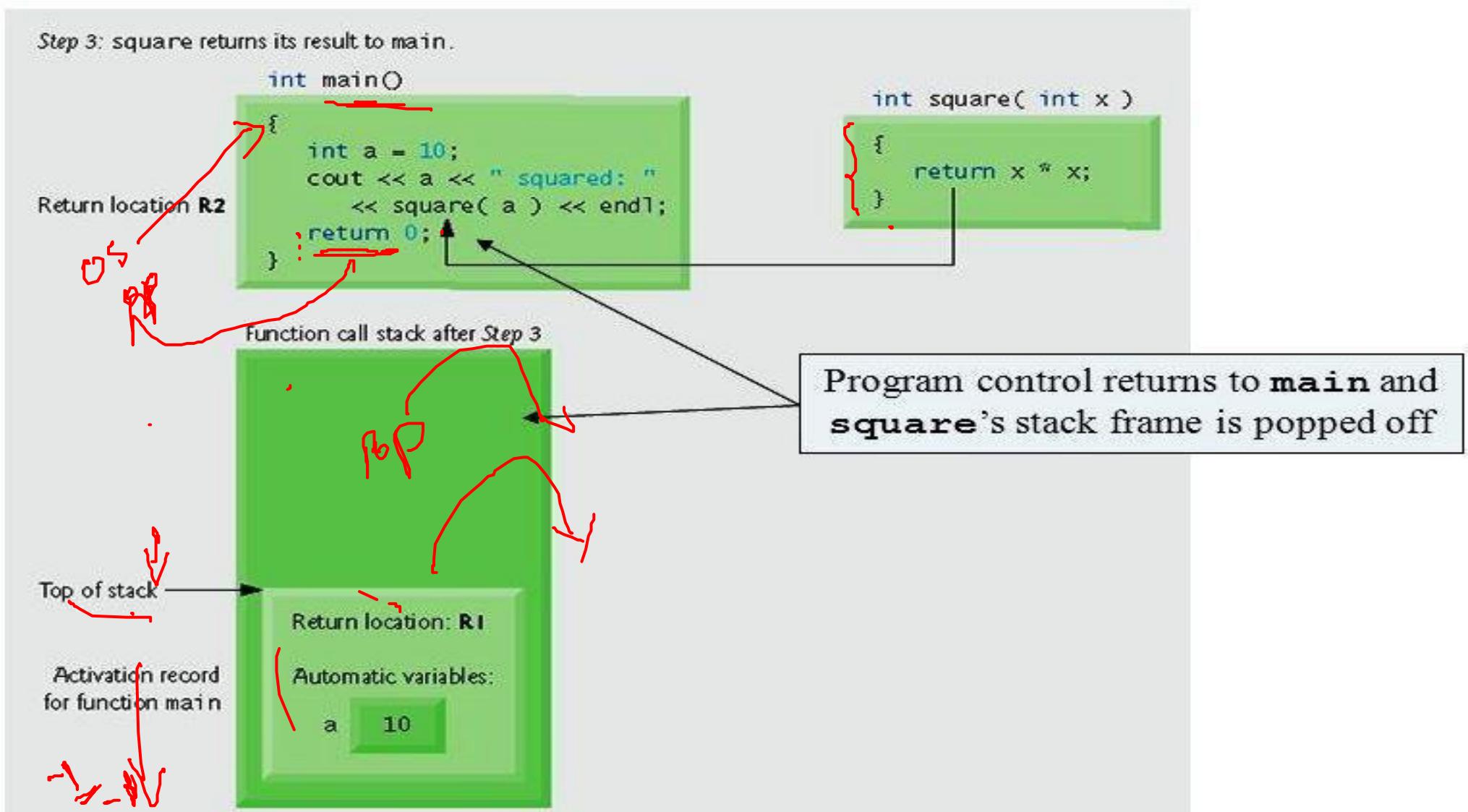
Applications of Stack

■ Function Call (Continued ...)



Applications of Stack

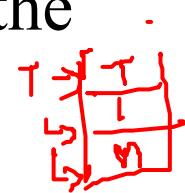
■ Function Call (Continued ...)



Applications of Stack

✓ The simplest application of a stack is to reverse a word. You push a given word to stack - letter by letter - and then pop letters from the stack.

M L T
—
T U M



T | M

There are other uses also like:

$a+b$. $a b +$ $+ a b$.

{ 1. Expression Conversion(Infix to Postfix, Postfix to Prefix etc)

2. Parsing ✓

3. It can be used to process function calls.

4. Implementing recursive functions in high level languages

Application of Stack

Parentheses Matching



$(a+b)$
Valid

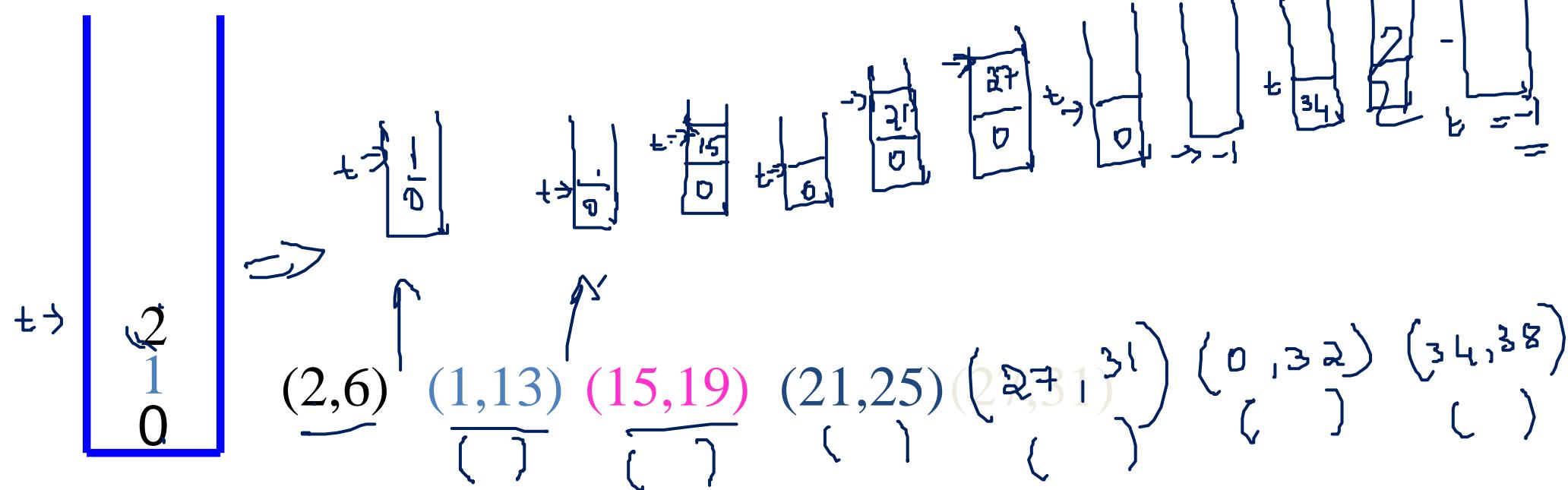
$\underline{a+b}$ $\underline{(a+b)+c}$ Not valid

- scan expression from left to right
- when a left parenthesis is encountered, add its position to the stack
- when a right parenthesis is encountered, remove matching position from stack



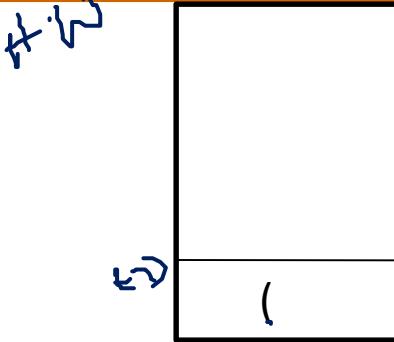
Example

$$\bullet \frac{((a+b)^*c+d-e)/(f+g)-(h+j)^*(k-l))}{(m-n)}$$

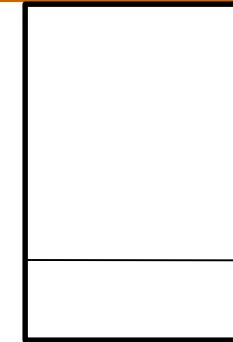




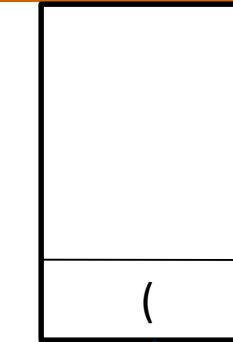
Ex1: $(a+b)^* (c+d)$



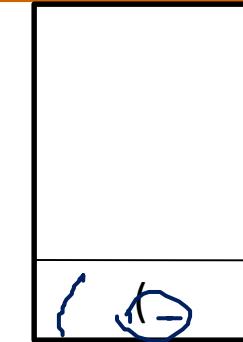
(
push '('



(a+b)
Pop '(' since
there is ')' &
continue

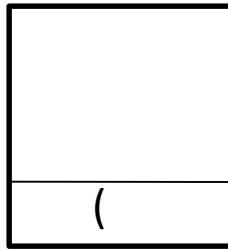


(a+b) *(
Push '('



(a+b) *(c+d)
End of string reached
but stack not empty.
Hence invalid

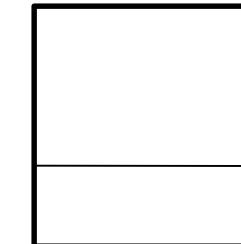
Ex2: $(a+b)^*c+d$



(
push '('



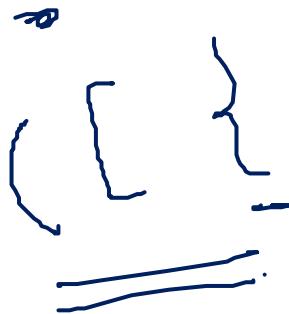
(a+b)
pop '(' and
continue

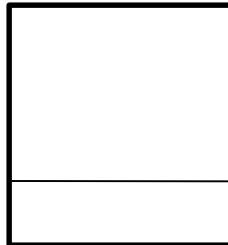
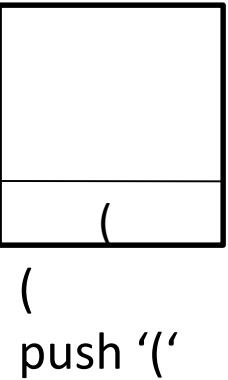


(a+b)*c+d)
Stack empty when ')'
encountered. Hence invalid

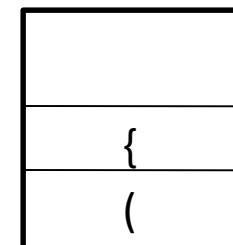
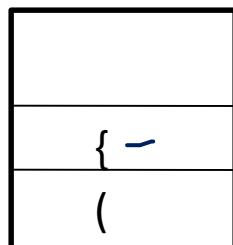
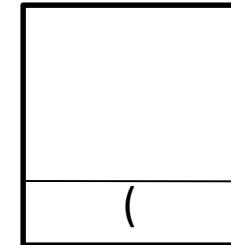


Ex3: $(a+b)^*(\{c^*d)$


{} -

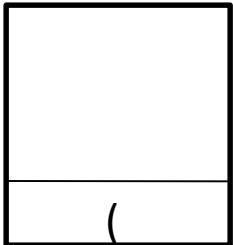


$(a+b)$
pop '(' and
continue

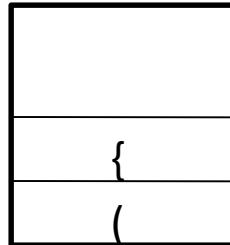




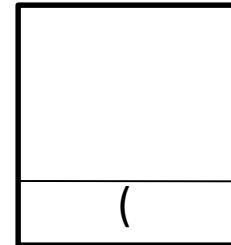
Ex4: $(a+\{b*c\}+(c*d))$



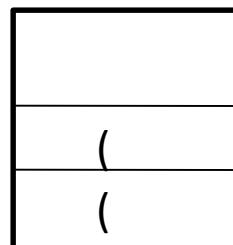
(
push '(' and
continue



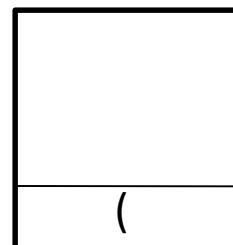
(a+{
push '{' and
continue



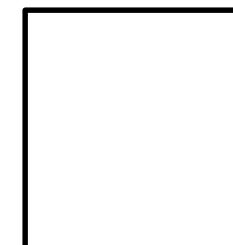
(a+{b*c}
pop '{' and
continue



(a+{b*c}+(
push '(' and
continue'



(a+{b*c}+(c*d)
Pop '('



(a+{b*c}+(c*d))
Pop '('.
End of string and stack
empty. Hence valid



Evaluation of Expressions

$$X = a / b - c + d * e - a * c$$

$$a = 4, b = c = 2, d = e = 3$$

Interpretation 1:

$$((4/2)-2)+(3*3)-(4*2)=0 + 9-8=1$$

Interpretation 2:

$$(4/(2-2+3))*(3-4)*2=(4/3)*(-1)*2=-2.6666\cdots$$

How to generate the machine instructions corresponding to a given expression?

precedence rule + associative rule

$a + (b - c)$

$\stackrel{15}{+}$ $\stackrel{15}{-}$

inner

Scan
=

Token	Operator	Precedence ¹	Associativity
()	function call	17	left-to-right
[]	array element		
->.	struct or union member		
- ++	increment, decrement ²	16	left-to-right
- -++	decrement, increment ³	15	right-to-left
!	logical not		
-	one's complement		
<u>-</u> +	unary <u>minus</u> or plus		
& *	address or indirection		
sizeof	size (in bytes)		
(type)	type cast	14	right-to-left
* / %	multiplicative	13	Left-to-right

$a + b - c$
 $\rightarrow (a + b) - c$

$+ -$	binary add or subtract	12	left-to-right
$<< >>$	shift	11	left-to-right
$> >=$ $< <=$	relational	10	left-to-right
$== !=$	equality	9	left-to-right
$\&$	bitwise and	8	left-to-right
$^$	bitwise exclusive or	7	left-to-right
$ $	bitwise or	6	left-to-right
$\&\&$	logical and	5	left-to-right
$\ $	logical or	4	left-to-right

?:	conditional	3	right-to-left
= += -= /= *= %= <<= >>= &= ^= =	assignment	2	right-to-left
,	comma	1 <i>↖</i>	left-to-right



user

compiler

Infix

$2+3*4$ →

$a*b+5$

$(1+2)*7$

Postfix

$234*$ + →

$ab*$ 5 +

$12+7*$

Postfix: no parentheses, no precedence



Infix to Postfix Conversion

(Intuitive Algorithm/ Manual method)

- (1) Fully parenthesize expression

$$\begin{array}{c} \text{---t} \\ | \nearrow \searrow \\ L \rightarrow R \end{array} \quad \begin{array}{l} \xrightarrow{\quad} \{a / b - c\} + d * e - a * c \rightarrow \\ \xrightarrow{\quad} (((a / b) - c) + (d * e)) - (a * c) \end{array}$$

- { (2) All operators replace their corresponding right parentheses.

$$\begin{array}{c} ab \} \leftarrow - \quad de \} \leftarrow + \quad ac \} \leftarrow * \\ (((a / b) - c) + (d * e)) - (a * c) \\ ab \} \leftarrow - \quad de \} \leftarrow + \quad ac \} \leftarrow * \quad - \end{array}$$

12 op.

- (3) Delete all parentheses.

$$ab/c-de^*+ac^*-$$

two passes



Infix expression

- In an expression if the binary operator, which performs an operation , is written in between the operands it is called an **infix expression.** Ex: $a+b*c$



Infix, Prefix and Postfix expression

- In an expression if the binary operator, which performs an operation , is written in between the operands it is called an **infix expression**.

Ex: a+b*c

- If the operator is written before the operands , it is called **prefix expression**

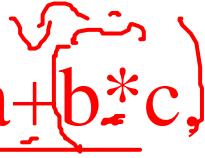
Ex: +a*bc

- If the operator is written after the operands, it is called **postfix expression**

Ex: abc*+.



Infix, Prefix, and Postfix expression

- An expression in infix form is dependent of precedence during evaluation
- Ex: to evaluate $a+b*c$, sub expression $a+b$ can be evaluated only after evaluating $b*c$. 
- As soon as we get an operator we cannot perform the operation specified on the operands.
- So it takes more time for compilers to check precedence to evaluate sub expression.



Infix, Prefix, and Postfix expression

- Both prefix and postfix representations are independent of precedence of operators.
- In a single scan an entire expression can be evaluated
- Takes less time to evaluate.
 - However infix expressions have to be converted to postfix or prefix.



Conversion and evaluation of expressions

- Infix Notation
- Prefix (Polish) Notation
- Postfix (Reverse-Polish) Notation

Infix Notation: operators are used **in**-between operands

e.g. $a \underline{-} b \underline{+} c$

- Advantage: easy to read, write, and speak for humans
- Difficult and costly in terms of time and space consumption



Postfix Notation:

The operator is written after the operands.

For example, $ab+$

Also, known as **Reversed Polish Notation**

Prefix Notation : operators are followed by operands i.e the operators are fixed before the operands.

For example, $+ab$

Also, known as **Polish Notation**

All the infix expression will be converted into post fix notation with the help of stack in any program



Parsing Expressions

To parse any arithmetic expression, we need to take care of
operator precedence and associativity also.

Precedence

$$a + b * c \rightarrow a + (b * c)$$

Associativity

Rule where operators with the same precedence appear in an
expression — L \rightarrow R.

$$a \underline{+} b \underline{-} c$$

$$\underline{(a + b)} - c$$



Table shows the default behavior of operators

Sr.No.	Operator	Precedence	Associativity
1	Exponentiation \wedge	Highest	Right Associative
2	Multiplication (*) & Division (/)	Second Highest	Left Associative
3	Addition (+) & Subtraction (-)	Lowest	Left Associative

Altered by using parenthesis

$a + b*c$

$(a + b)*c$





■ Conversion of infix form to postfix

INFIX_POSTFIX (Q, P)

1. Push '(' onto stack, and add ')' to the end of Q.
2. Scan Q **from left to right** and repeat Steps 3 through 6 for each element of Q until the stack is empty.
3. If an **operand** is encountered, add it to the right of P.
4. If a **left parenthesis** is encountered, push it on to the stack.
5. If an **operator** op is encountered, then⁴⁵
 - a) Repeatedly pop from the stack and add to the right of P each operator (on the top of the stack) which has the same precedence as or higher precedence than op .
 - b) Add op to the stack.

a + b

■ Conversion of infix form to postfix (Continued...)

6. If a **right parenthesis** is encountered, then
 - a) Repeatedly pop from the stack and add to the right of P each operator (on the top of the stack) until a left parenthesis is encountered.
 - b) Remove the left parenthesis. [Do not add the left parenthesis to P]
7. Return



P = []
46



Infix to Postfix.

	$a - (b + c * d) / e$	
1.) .		<u>Stack</u>
2. a	a	[]
3. -	(-	-
4. ((-	(
5. b	b	ab
6. *	*	ab
7. c	c	abc
8. *	*	abc

9. /		<u>Stack</u>	$(- [+ *$	abcd
10.))		-	abcd*
11. /	/		- /	abcd*
12. e	e		(- / →	abcd*+e
13.))		-	abcd*+e)-

Postfix.



Postfix

$a =$	$(A + B) \cdot C$	\rightarrow	$S \rightarrow$
	$\boxed{A \quad + \quad B}$	\rightarrow	$\boxed{+ \quad \cdot \quad C}$
1.)	(
2.)	((
3.)	A		
4.)	+ ((
5.)	B / ((+		
6.)) (:		
7.)	*		
8.)	[
9.))		

$\underline{\underline{AB+C}}$ = Post

Ex) $(a+b \cdot c - d) / (e+f)$

Postfix

\rightarrow $a b c * + d - e f *$

S
~~+~~

{ equal
higher }

$+$ $\overline{+}$



Convert infix to post fix expression

$$a - (b + c * d)/e$$

Ch	Stack(bottom to top)	PostfixExp
a		a
-	-	a
(-()	a
b	-()	ab
+	-(+	ab
c	-(+	abc
*	-(+*	abc
d	-(+*	abcd
)	-(+	abcd*
/	-()	abcd*+
e	-	abcd*+
	-/	abcd*+
	-/	abcd*+e
		abcd*+e/-

Infix $a+b \rightarrow \underline{ab} +$ Postfix

$a+b$ $\Rightarrow + \underline{ab} \rightarrow$ Prefix

Mandal

①

$$A + B * C$$

1) $(A + (B * C))$

2) $(A + (B + C))$

3) $(\underline{\underline{A}} + (\underline{\underline{B}} * \underline{\underline{C}}))$

4) $(A (\underline{\underline{B}} \underline{\underline{C}} *)) +$

5) $A \underline{\underline{B}} \underline{\underline{C}} * +$

Infix to Postfix

②

$$(A + B) * C + D + E * F - G$$



Infix to postfix conversion:

Sample Exercises

Convert the following infix expression to postfix expression

- $a+b*c+d*e$
- $a*b+5$
- $(a/(b-c+d))*((e-a)*c)$
- $a/b-c+d*e-a*c$



Applications of Stack

$$a^b \cdot t \rightarrow + \underbrace{a^b}_{\leftarrow}$$

■ Conversion of infix form to prefix

INFIX_PREFIX (Q, P)



1

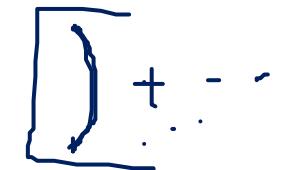
- Push ')' onto stack, and add '(' at the beginning of Q.
 - Scan Q from right to left and repeat Steps 3 through 6 for each element of Q until the stack is empty.
 - If an operand is encountered, add it to the left of P. 
 - If a right parenthesis is encountered, push it onto the stack.
 - If an operator op is encountered, then
 - Repeatedly pop from the stack and add to the left of P each operator (on the top of the stack) which has higher precedence than op. 
 - Add op to the stack.



Applications of Stack

■ Conversion of infix form to prefix (Continued...)

6. If a **left parenthesis** is encountered, then
 - a) Repeatedly pop from the stack and add to the left of P each operator (on the top of the stack) until a right parenthesis is encountered.
 - b) Remove the right parenthesis. [Do not add the right parenthesis to P]
7. Return



(



Infix → Prefix

$$Q = (a + b * c)$$

$\leftarrow \eta \rightarrow L$

char	S	P
i.) .	{ } \rightarrow	
ii.) C	}	$\rightarrow C$
iii.) *) *	C
iv.) b) *	bC
v.) +) *	bC
vi.) a) +	*bC
vii.)) +	a *bC

Q.) (-
 + a * b c



Pactix

$$Q_1 = \frac{(a+b - (c+d \wedge e)) * \cancel{z+4}}{\cancel{4} \cancel{z}}$$

char	S	P	11.) C) + *) *	Acronym
1.)	y	12.) () + * _	*CAdency
2. y)	y	13.) -) + ? -	*?CAdency
3. +) +	y	14.) b	<u>) + =</u>	b + * CAdency
4. x) +	xy	15.) +) + - +	b - -
5. *) + *	xy	16.) a	<u>) + - +</u>	ab + * CAdency
6.)) + *)	xy	17.) (<u> </u>	+ - + ab + * CAdency
7. e) + *)	exy			
8. A) + *) A	exy			
9. d	<u>) + *) A</u>	deсы			
10. *) + *) *	Adency			



Infix to Prefix

—

$$1) \ a + b * c + d$$

$$2) \ (a+b) * c$$



Convert Infix To Prefix Notation (Alternative method)

- Step 1: Reverse the infix expression

Ex. A+B*C will become C*B+A.

Note: While reversing each '(' will become ')' and each ')' becomes '('.

- Step 2: Obtain the postfix expression of the modified expression

i.e.: CB*A+

- Step 3: Reverse the postfix expression. Hence in our example prefix is
+A*BC.

Complexity: O(n)



Infix - Prefix

(1)

$$A = a + b * c$$

1) Reverse $\rightarrow (c * b + a)$

2) Postfix C S P
1) (-

2) C (C

3) * (* C

4) b (* cb

5) + (+ cb*

6) - (+ cb*a

7)) - cb*a+
 =====

3.) Reverse.

Prefix = + a * b c
=====

(2) (a+b) * (c+d)



Infix to Prefix

e.g., Infix: $A * \underline{B+C} / D$

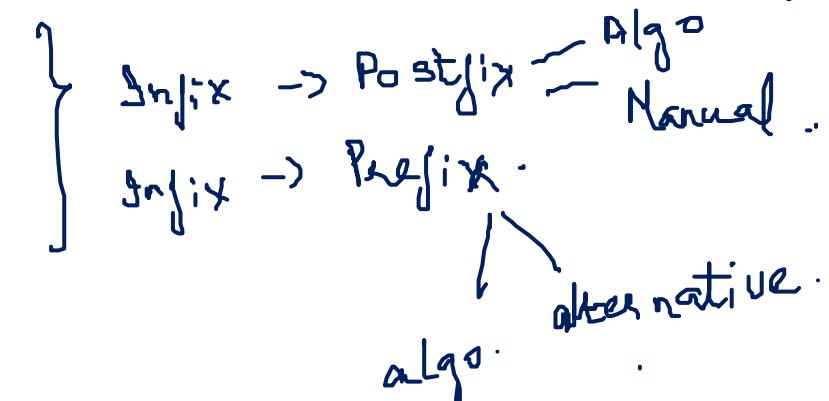
Infix: $(A-B/C)^*(A/K-L)$

Prefix: $+ * AB / CD$

Prefix: $* - \overline{A/B C} - / A K L$

1) $(300+23) * (43-21) / (84+7)$

2) $(4+9) * (6-5) / ((3-2)*(2+1))$





Applications of Stack

Evaluation of a postfix expression

POST_EVALUATE (P, VALUE)

1. Scan P from left to right and repeat Steps 2 through 3 for each element of P until the end of the expression is encountered.

2. If an operand is encountered, push it to stack.

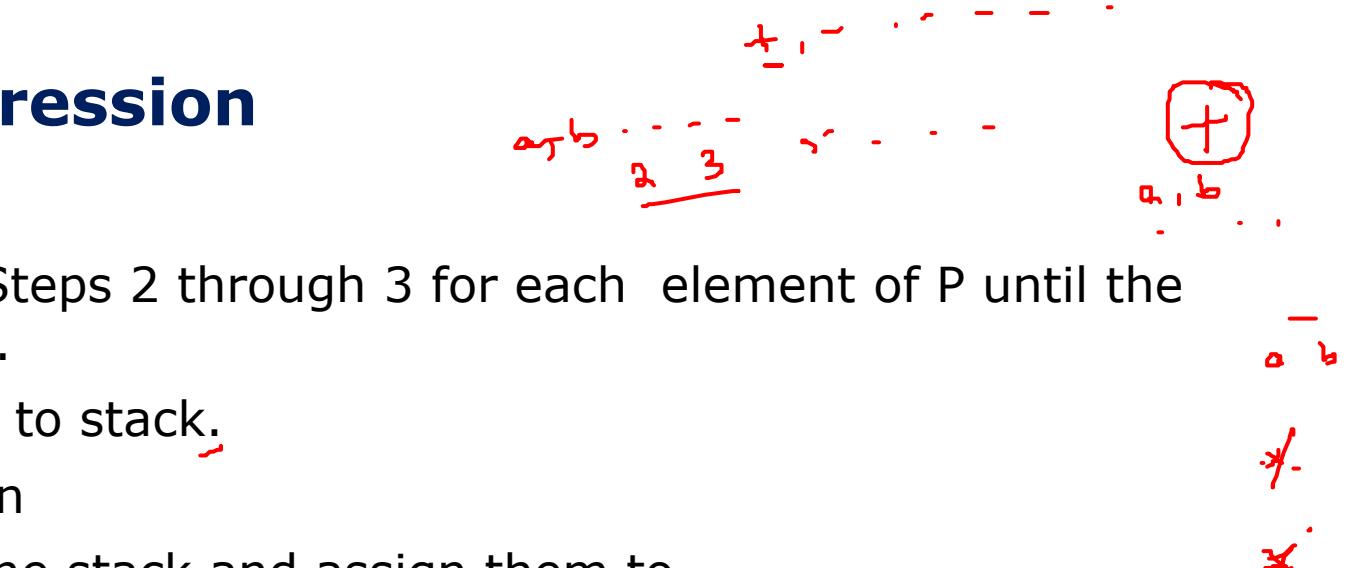
3. If an operator op is encountered, then

a) Pop the two top elements from the stack and assign them to $opnd_2$ and $opnd_1$ respectively.

b) Evaluate $opnd_1 op opnd_2$ and push the result onto the stack.

1. Set VALUE equal to the top element on the stack.

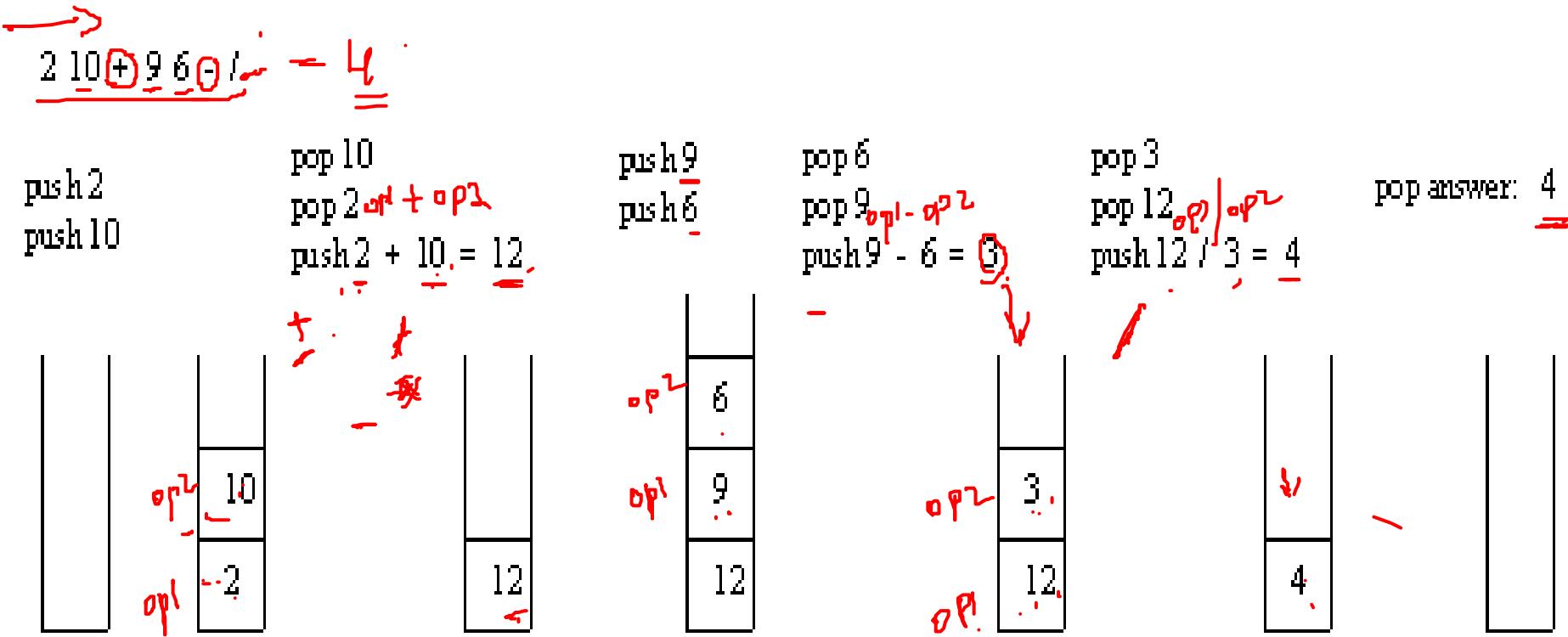
2. Return



$$\begin{matrix} + \\ op_1 + op_2 = op_3 \\ \hline \end{matrix}$$



Example for evaluation of Postfix expression



$$P = 4 \cdot 8 + 6 - 5 - * 3 \cdot 2 - 2 \cdot 2 + * /$$

Steps

stack

OP

- 1. 4
- 2. 8
- 3. +
- 4. 6
- 5. -
- 6. *
- 7. 3



$$4 + 8 = 12$$



$$OP1 \quad OP2 \\ 6 - 5 = 1$$

$$12 * 1 = 12$$

$$OP1 \quad OP2 \\ 9) \quad 2$$



$$OP1 \quad OP2 \\ 10) \quad -$$



$$OP1 \quad OP2 \\ 11) \quad 2$$



$$OP1 \quad OP2 \\ 12) \quad 2$$



$$OP1 \quad OP2 \\ 13) \quad * +$$



$$OP1 \quad OP2 \\ 14) \quad *$$



$$OP1 \quad OP2 \\ 15) \quad /$$



$$Ans = \underline{\underline{3}}$$

$$OP1 \quad OP2 \\ 3 - 2 = 1$$

$$2 * 2 = 4$$

$$1 * 4 = 4$$

$$12 / 4 = \underline{\underline{3}}$$



$$\rightarrow \underline{360} \quad \underline{23} \quad + \quad \underline{43} \quad \underline{21} \quad - \quad * \quad \underline{84} \quad 7 \quad + \quad /$$



Evaluation of Postfix Using Stack

Token	Stack			Top
	[0]	[1]	[2]	
6	6			0
2	6	2		1
/	6/2			0
3	6/2	3		1
-	6/2-3			0
4	6/2-3	4		1
2	6/2-3	4	2	2
*	6/2-3	4*2		1
+	6/2-3+4*2			0



Applications of Stack

■ Evaluation of a prefix expression

PRE_EVALUATE (P, VALUE)



1. Scan P from right to left and repeat Steps 2 through 3 for each element of P until the beginning of the expression is encountered.
2. If an operand is encountered, push it to stack.
3. If an operator op is encountered, then
 - a) Pop the two top elements from the stack and assign them to $opnd1$ and $opnd2$ respectively.
 - b) Evaluate $opnd1 \ op \ opnd2$ and push the result onto the stack.
1. Set VALUE equal to the top element on the stack.
2. Return



Prefix

Ans $\rightarrow 3$

$$/ * + 4 \ 8 = 6 \ 5 * - 3 \ 2 + 2 \ 2$$

$\leftarrow \text{right}$

Step 9.

stack

op:

*



+



+



*



-



*



*



$$\begin{matrix} \text{op2} & \text{op1} \\ 2 + 2 & = 4 \end{matrix}$$

$$\begin{matrix} \text{op2} & \text{op1} \\ 3 - 2 & = 1 \end{matrix}$$

$$\begin{matrix} \text{op2} & \text{op1} \\ 1 * 4 & = 4 \end{matrix}$$

5



6



-



8



4



+



*



/



$$\begin{matrix} \text{op2} & \text{op1} \\ 6 - 5 & = 1 \end{matrix}$$

$$4 + 8 = 12$$

$$12 + 1 = 13$$

$$13 / 4 = 3$$



Prefix \Rightarrow / * + 300 23 - 43 21 + 84 7

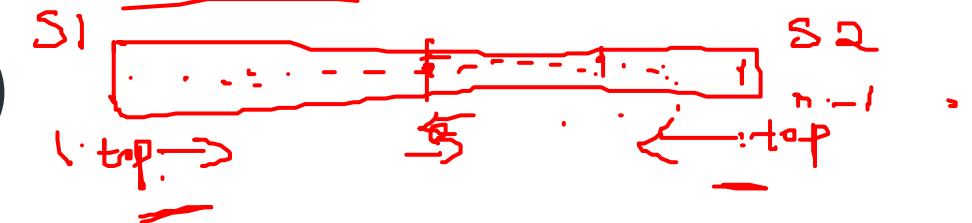
Multiple Stacks



Method 1 (Divide the space into two halves)

- Divide the array into two halves and assign each of the half space to two stacks, i.e., use $\text{arr}[0]$ to $\text{arr}[\lfloor n/2 \rfloor]$ for stack1, and $\text{arr}[\lfloor (n/2) + 1 \rfloor]$ to $\text{arr}[n-1]$ for stack2 where $\text{arr}[]$ is the array to be used to implement two stacks and size of array be n .
- The problem with this method is inefficient use of array space. A stack push operation may result in stack overflow even if there is space available in $\text{arr}[]$.

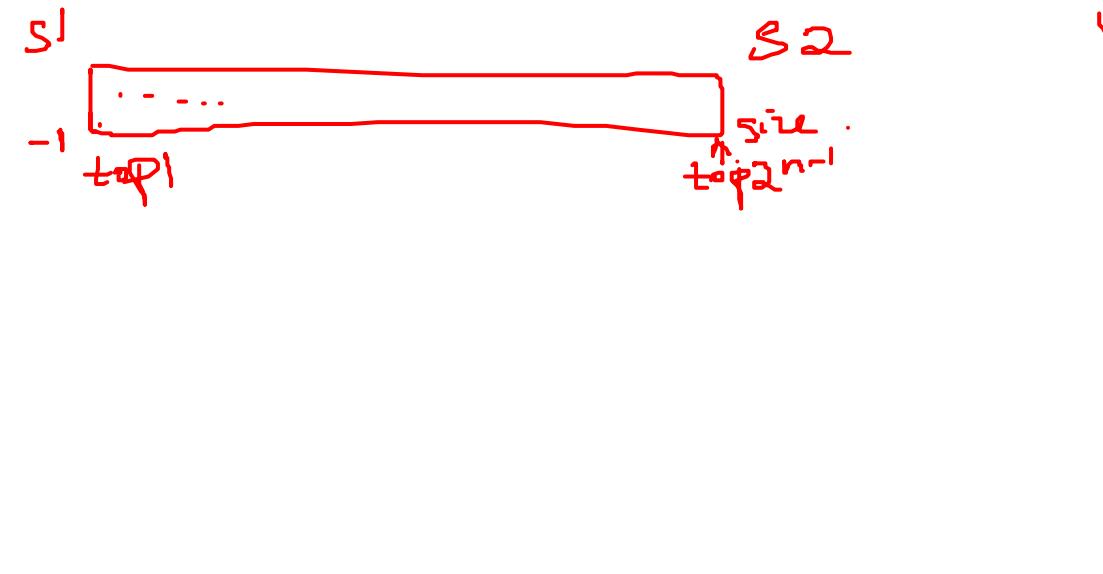
Method 2 (A space efficient implementation)



- Stack1 starts from the leftmost element, the first element in stack1 is pushed at index 0. The stack2 starts from the rightmost corner, the first element in stack2 is pushed at index ($n-1$). Both stacks grow (or shrink) in opposite direction.
- To check for overflow, all we need to check is for space between top elements of both stacks.

Multiple Stacks

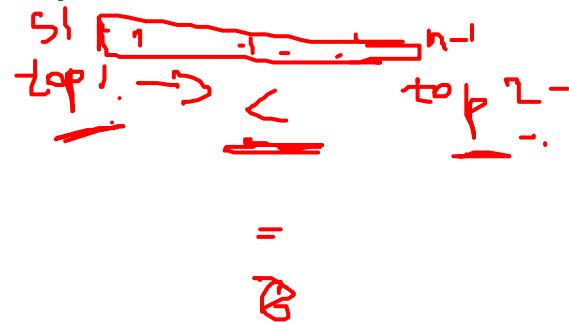
```
class twoStacks
{
    int arr[size];
    int top1, top2;
public:
    twoStacks() // constructor
    {
        top1 = -1;    }
        top2 = size;
    }
```



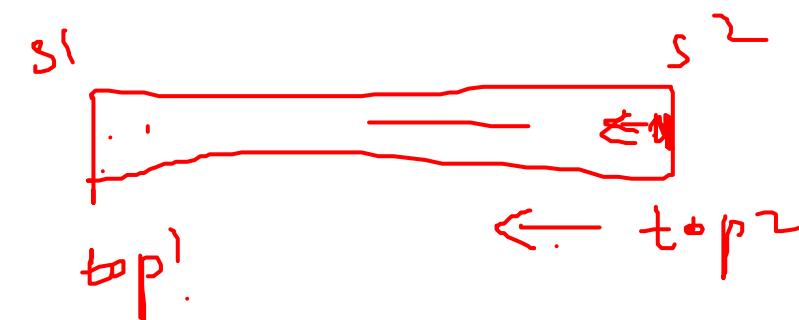


Multiple Stack implementation in single array

```
// Method to push an element x to stack1
void push1(int x)
{
    // There is at least one empty space for new element
    if(top1 < top2 - 1)
    {
        top1++;
        arr[top1] = x;
    }
    else
    {
        cout << "Stack Overflow";
    }
}
```

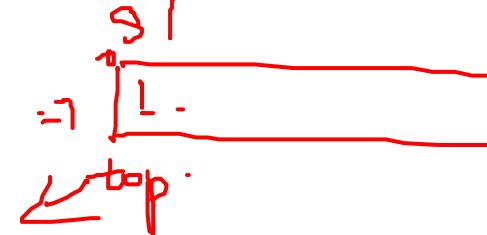


```
// Method to push an element x to stack2
void push2(int x)
{
    // There is at least one empty space for new element
    if (top1 < top2 - 1)
    {
        top2--;✓
        arr[top2] = x;
    }
    else
    {
        cout << "Stack Overflow";
    }
}
```



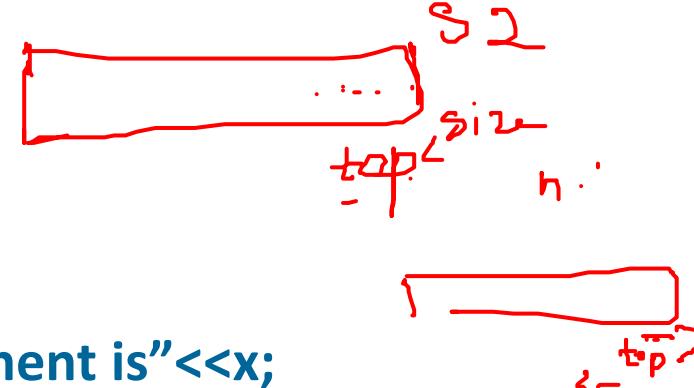


```
// Method to pop an element from first stack
void pop1() {
    if (top1 >= 0 )
    {
        int x = arr[top1];
        top1--;
        cout<<"popped element is"<<x;
    }
    else
    {
        cout << "Stack UnderFlow";
    }
}
```

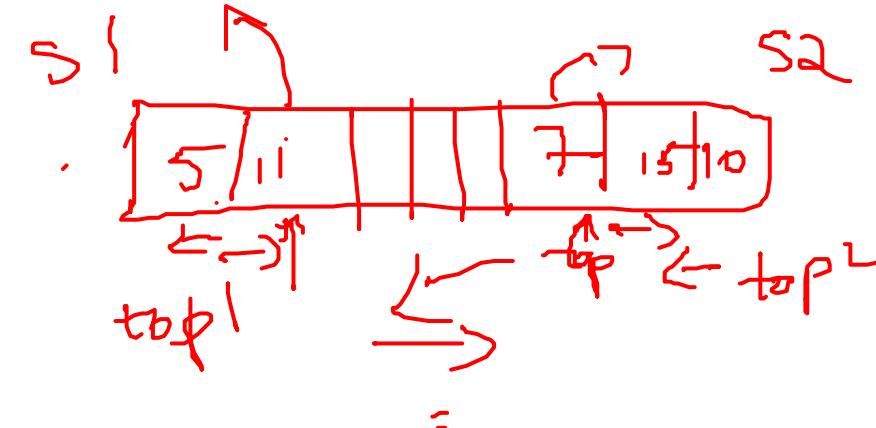




```
// Method to pop an element from second stack
void pop2()
{
    if (top2 < size)
    {
        int x = arr[top2];
        top2++;
        cout<<"popped element is"<<x;
    }
    else
    {
        cout << "Stack UnderFlow";
    }
}
```



```
int main()
{
    twoStacks ts;
    ts.push1(5); ✓
    ts.push2(10); ✓
    ts.push2(15);
    ts.push1(11);
    ts.push2(7);
    cout << "Popped element from stack1 is " << ts.pop1();
    ts.push2(40);
    cout << "\nPopped element from stack2 is " << ts.pop2();
    return 0;
}
```





Multiple stacks(case n \geq 3)

```
#include<iostream>
using namespace std;
#define max_size 20
class stack
{
    int top[10];
    int a[50];
    int boundary[10];
    int n; //no.of stacks entered by user
public:
    stack(int);           //
    void push(int ,int);
    void pop(int);
    void display(int);
};
```

To divide the array into roughly equal segments, the code:-

Refer page 139; Sartaj

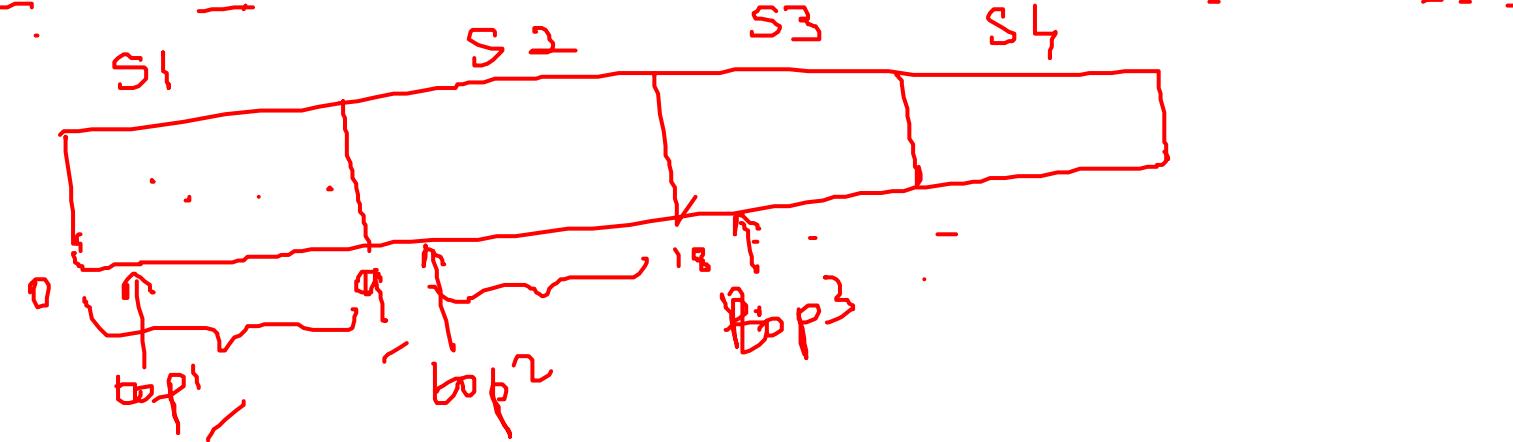
```
stack::stack(int n)
```

```
{
```

```
    for(int i=0; i<n; i++)
```

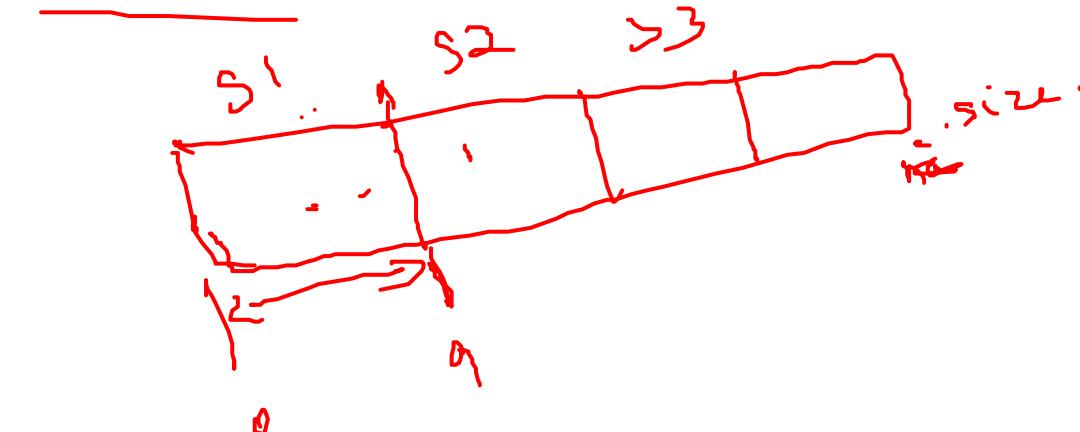
```
        boundary[i]=top[i]=(max_size/n)*i-1;
```

```
}
```



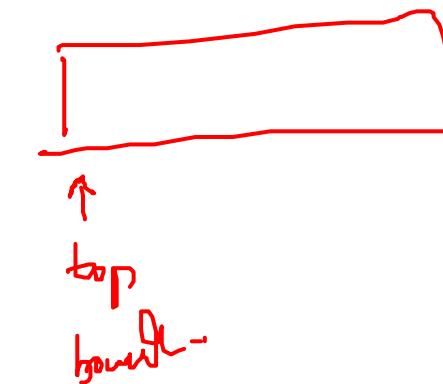
To add an item to the i th stack

```
void stack::push(int i,int x)
{
    // add an item to the  $i$ th stack
    if((top[i]==boundary[i+1])||(top[i]==(max_size-1)))
        cout<<"Stack is full \n";
    else
        a[++top[i]]=x;
}
```



To delete an item from the i th stack

```
void stack::pop(int i)
{
    //remove top element from the ith stack
    if(top[i]==boundary[i])
        cout<<"stack is empty\n";
    else
        cout<<"deleted element is "<<(a[top[i]--]);}
```





To display the elements

```
void stack::display(int i)
{
    if(top[i]==boundary[i])
        cout<<"stack is empty\n";
    else
        for(int j=top[i];j>boundary[i];j--)
            cout<<"\nThe elements of stack are "<<"\n"<<a[j];
}
```

s₁ *s₂* *s₃*
; ; ;
; ; ;
; ; ;



```
int main()
{
    stack s(5);
    s.push(1,10);
    s.push(2,30);
    s.push(3,40);
    s.push(3,28);
    s.pop(3);
    s.display(2);
    return 0;
}
```



END OF STACKS



Queue



Bus Stop Queue





Bus Stop Queue



front

rear





Bus Stop Queue



front

rear





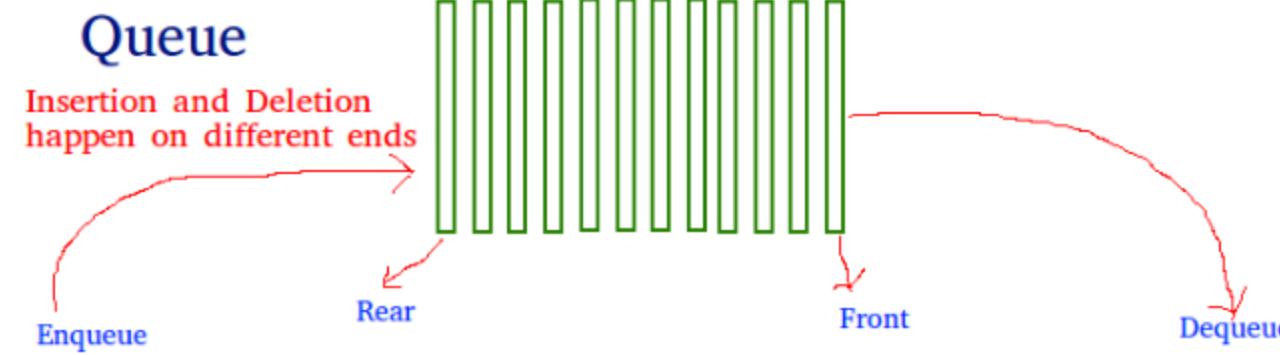
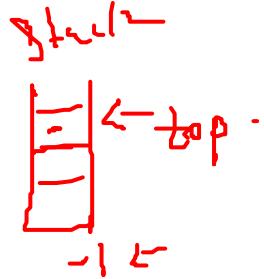
Bus Stop Queue



front

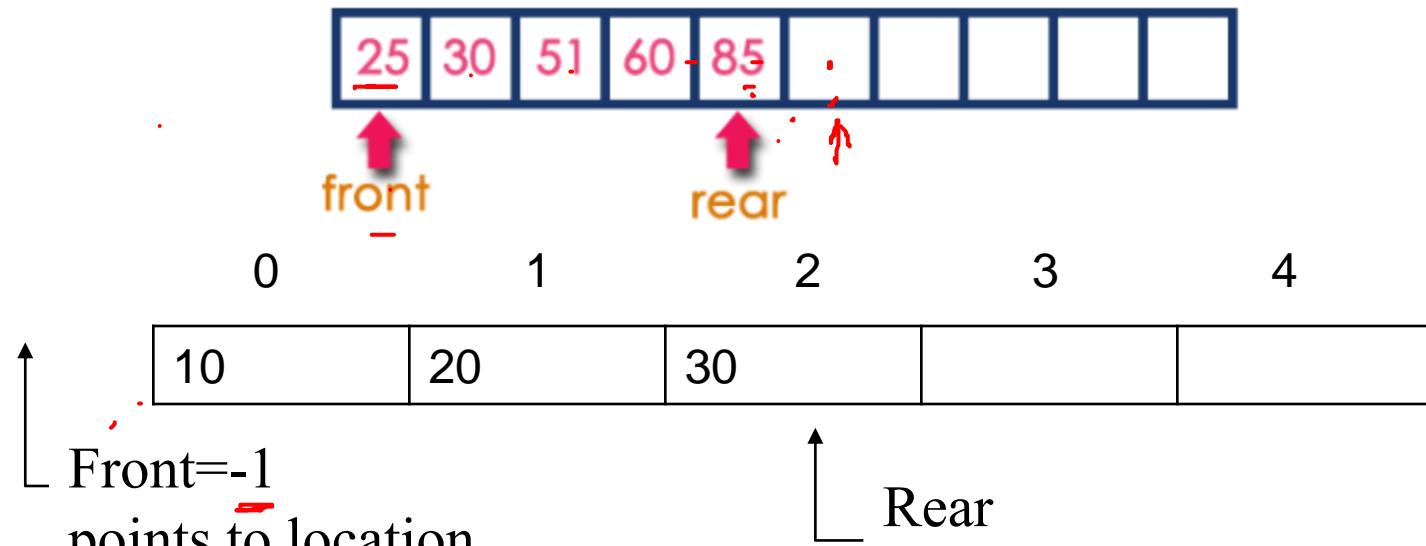
rear





First in, first out

FIFO



Front = -1
points to location
one before the 1st element

Rear
always points to last data

Basic features of Queue

- Linear list.
- One end is called front. head
- Other end is called rear. tail
- Additions are done at the rear only.
- Removals are made from the front only.



Basic features of Queue

- Queue is an abstract data structure
- A queue is open at both its ends
- One end (rear) is always used to insert data (enqueue) and
- the other (front) is used to remove data (dequeue).
- Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.
- Real-world examples can be seen as queues at the ticket windows and bus-stops.

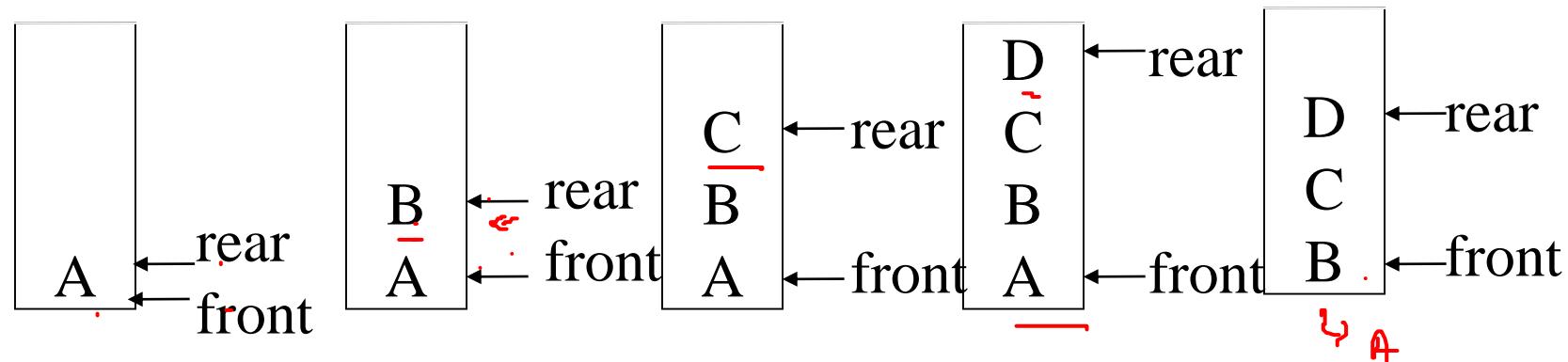


Basic features of Queue

Queue data structure is a linear data structure in which the operations are performed based on FIFO principle.

Queue data structure is a collection of similar data items in which insertion and deletion operations are performed based on FIFO principle

Queue: a First-In-First-Out (FIFO) list



*Figure 3.4: Inserting and deleting elements in a queue (p.106)

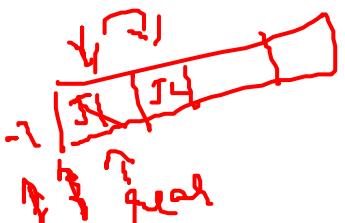


Queues - Application

- Used by operating system (OS) to create job queues.
- If OS does not use priorities then the jobs are processed in the order they enter the system

QUESTION

Application: Job scheduling



front	rear	Q1	Q2	Q3	Contents
-1	-1	-	-	-	queisempty
-1	0	J1	-	-	Jblisadded
-1	1	J1	J2	-	Jb2isadded
-1	2	J1	J2	B	Jb3isadded
0	2	J2	J2	B	Jblisadded
1	2	-	J2	B	Jb2isadded

*Figure 3.5: Insertion and deletion from a sequential queue (p.108)

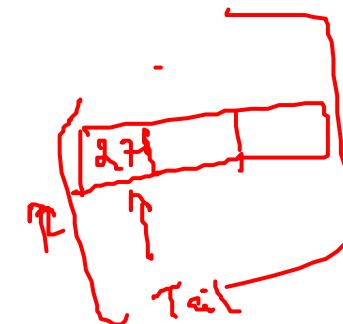
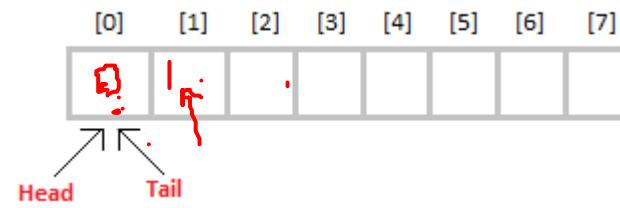


Applications of Queue

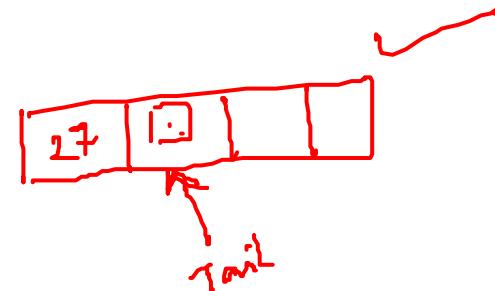
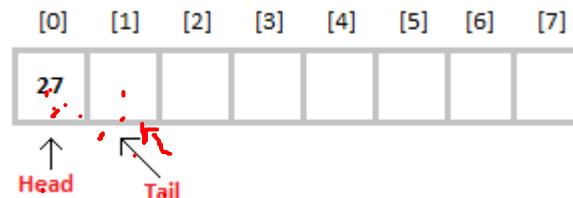
1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
2. In real life scenario, Call Center phone systems uses Queues to hold people calling them in an order, until a service representative is free.
3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive i.e First come first served.

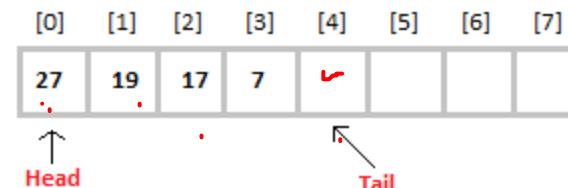
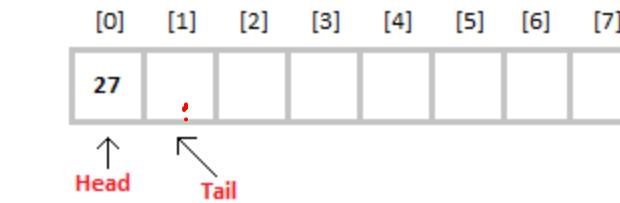
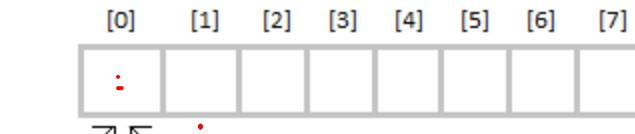
Implementation of Queue Data Structure

Initially the head(FRONT) and the tail(REAR) of the queue points at the first index of the array



As we add elements to the queue, the tail keeps on moving ahead, always pointing to the position where the next element will be inserted, while the **head** remains at the first index.





Adding elements to Queue

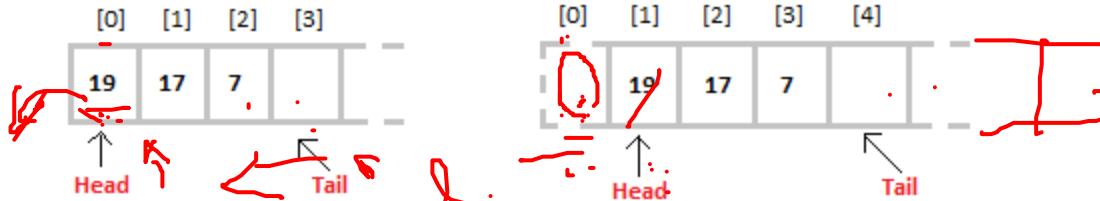
Enqueue

removing element from Queue

Dequeue



27



[B]

overflow



In [A] approach, we remove the element at **head** position, and then one by one shift all the other elements in **forward** position.

In approach [B] we remove the element from **head** position and then move **head** to the next position.

In approach [A] there is an **overhead of shifting the elements one position forward** every time we remove the first element.

In approach [B] there is no such overhead, but whenever we move head one position ahead, after removal of first element, the **size of Queue is reduced by one space** each time.



Operations on a Queue

1. enQueue(value) - (To insert an element into the queue)
2. deQueue() - (To delete an element from the queue)
3. display() - (To display the elements of the queue)



Algorithm for ENQUEUE operation

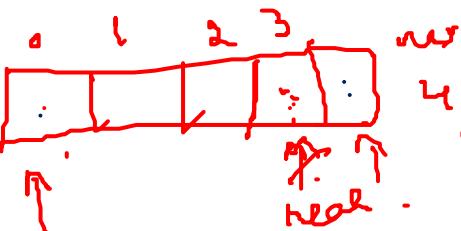
1. Check if the queue is full or not.
2. If the queue is full, then print overflow error and exit the program.
3. If the queue is not full, then increment the tail and add the element.

Algorithm for DEQUEUE operation

1. Check if the queue is empty or not.
2. If the queue is empty, then print underflow error and exit the program.
3. If the queue is not empty, then print the element at the head and increment the head.

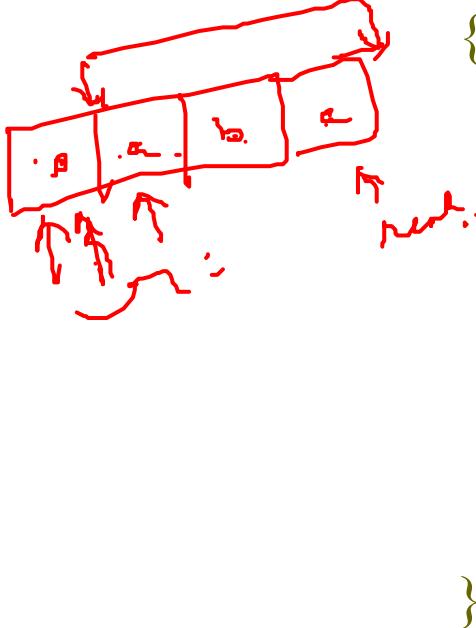
Enqueue: If the queue is not full, this function adds an element to the back of the queue, else it prints “OverFlow”.

```
void enqueue(int queue[], int element, int& rear, int arraySize)
{
    if(rear == arraySize) // Queue is full
        printf("OverFlow\n");
    else
        queue[rear] = element; // Add the element to the back
        rear++;
}
```





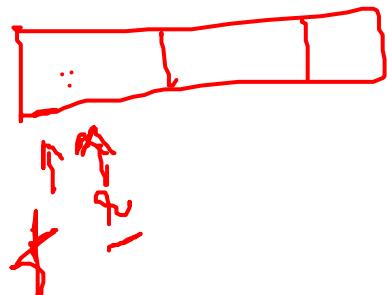
Dequeue: If the queue is not empty, this function removes the element from the front of the queue, else it prints “UnderFlow”.



```
void dequeue(int queue[], int& front, int rear)
{
    if(front == rear) // Queue is empty
        printf("UnderFlow\n");
    else
    {
        queue[front] = 0; // Delete the front element
        front++;
    }
}
```

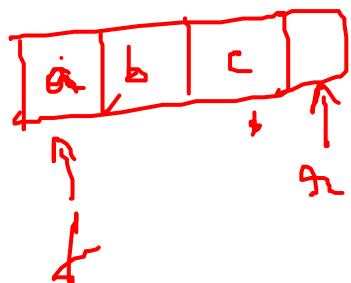


IsEmpty: If a queue is empty, this function returns 'true', else it returns 'false'.



```
bool isEmpty(int front, int rear)  
{  
    return (front == rear);  
}
```

Front: This function returns the front element of the queue.

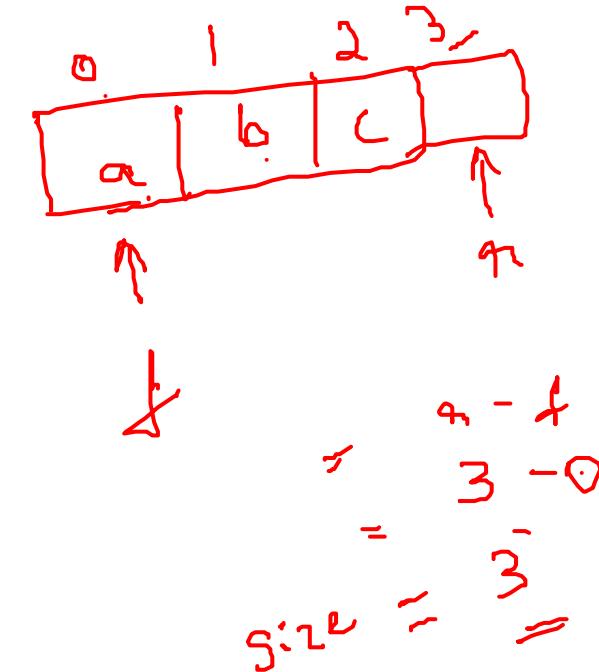


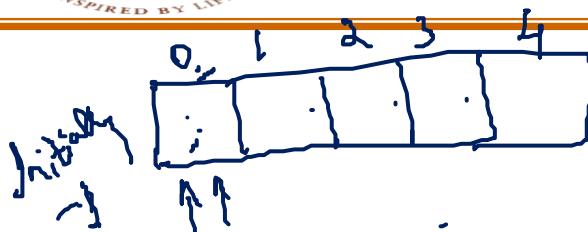
```
int Front(int queue[], int front)  
{  
    return queue[front];  
}
```



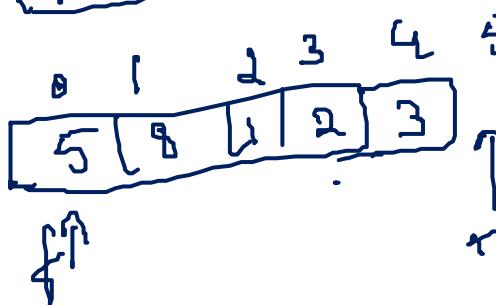
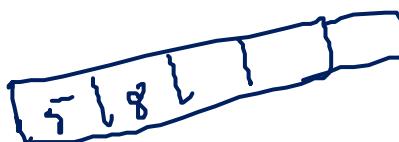
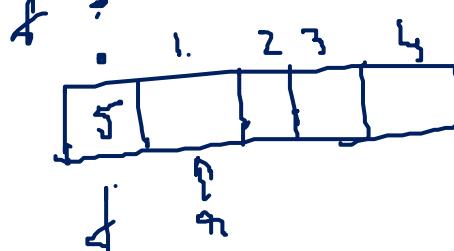
Size: This function returns the size of a queue or the number of elements in a queue.

```
int size(int front, int rear)  
{  
    return (rear - front);  
}
```



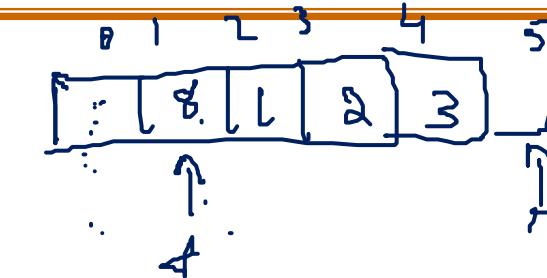


$$\text{maxsize} = 5$$

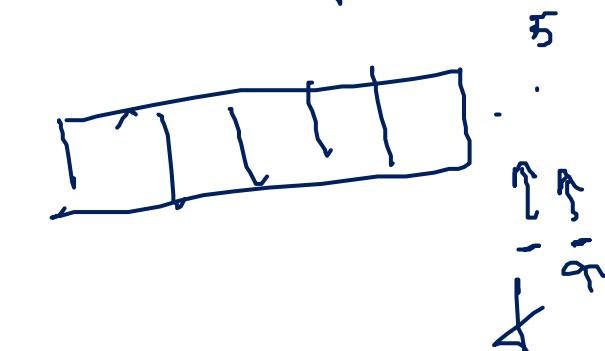
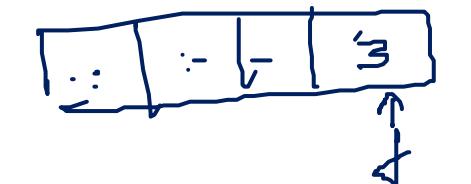


$$S = 5 - 0$$

$$\Rightarrow 5$$



$$S = \frac{n-1}{2} \\ 5 - 1 = 4$$



Count el

 { → count --
 { → count ++

$$S = \frac{n-1}{2}$$

$$= 5 - 5$$

$$= 0$$



Complexity Analysis of Queue Operations

- Enqueue: **O(1)**
- Dequeue: **O(1)**
- Size: **O(1)**





Circular queues

- A circular queue is an improvement over the standard queue structure.
- In a standard queue, when an element is deleted, the vacant space is not reutilized. However, in a circular queue, vacant spaces are reutilized.
 -
- While inserting elements, when you reach the end of an array and you need to insert another element, you must insert that element at the beginning (given that the first element has been deleted and the space is vacant).



front
rear

insert 5



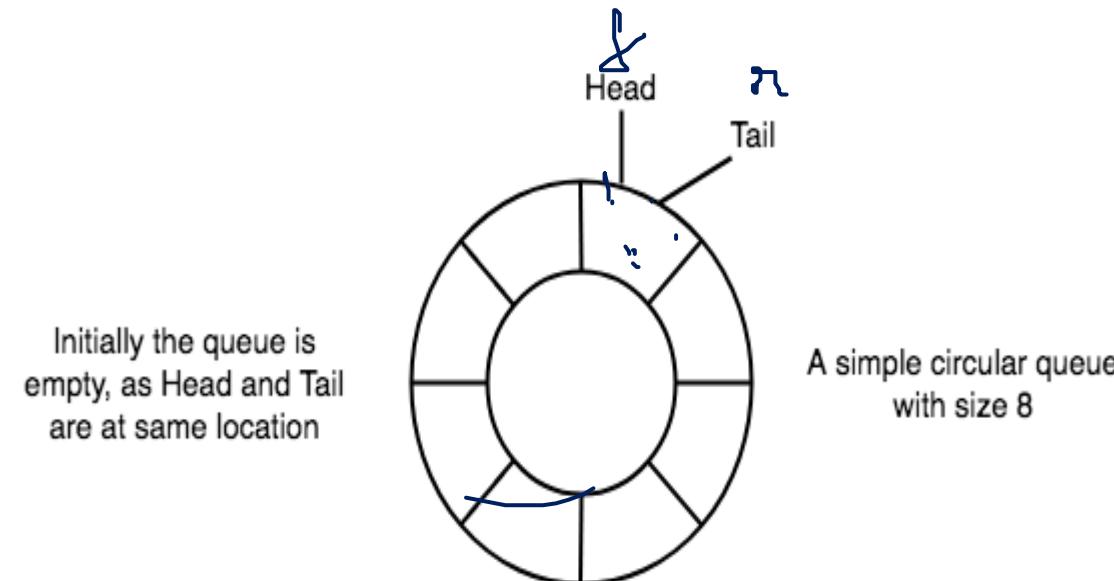
rear
front

Circles

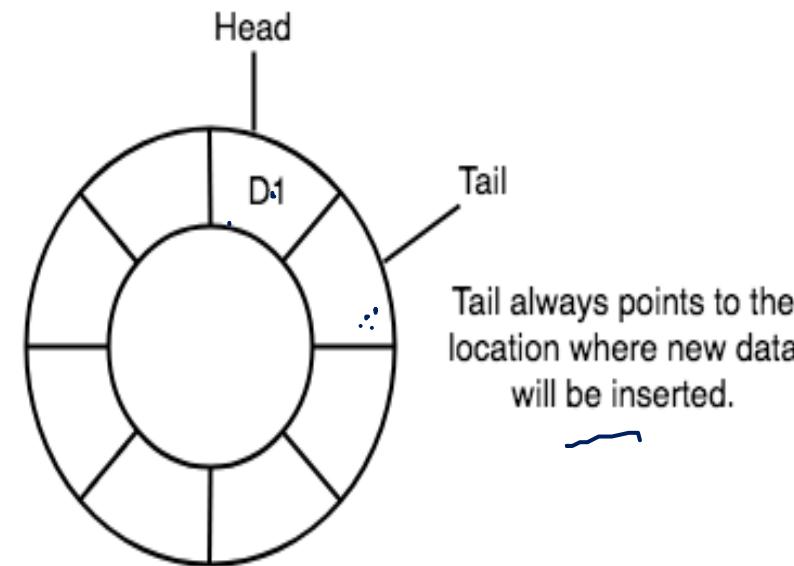
✓

Basic features of Circular Queue

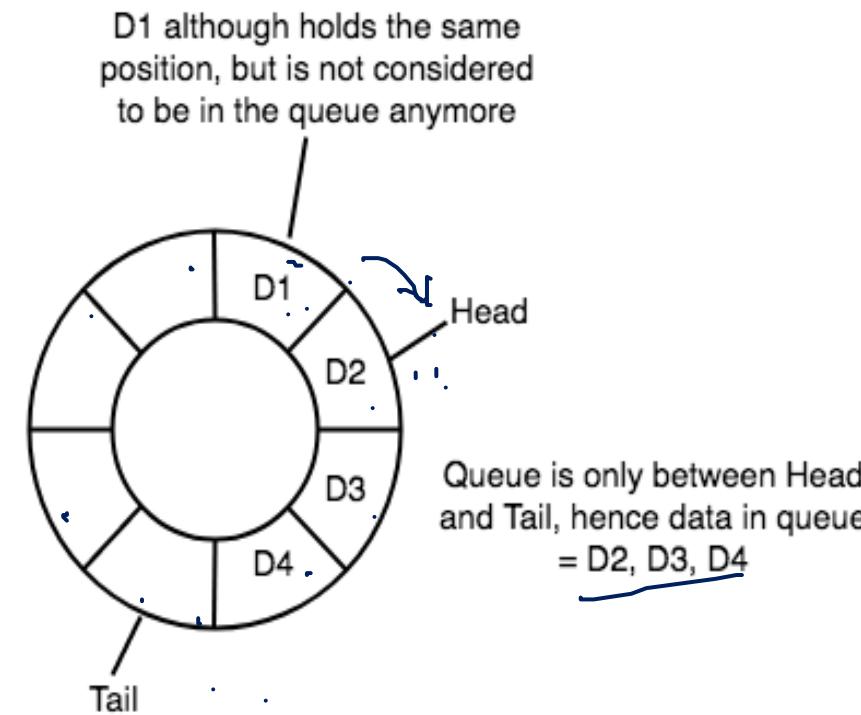
1. Head pointer will always point to the front of the queue, and
2. Tail pointer will always point to the end of the queue.
3. Initially, the head and the tail pointers will be pointing to the same location, this would mean that the queue is empty.



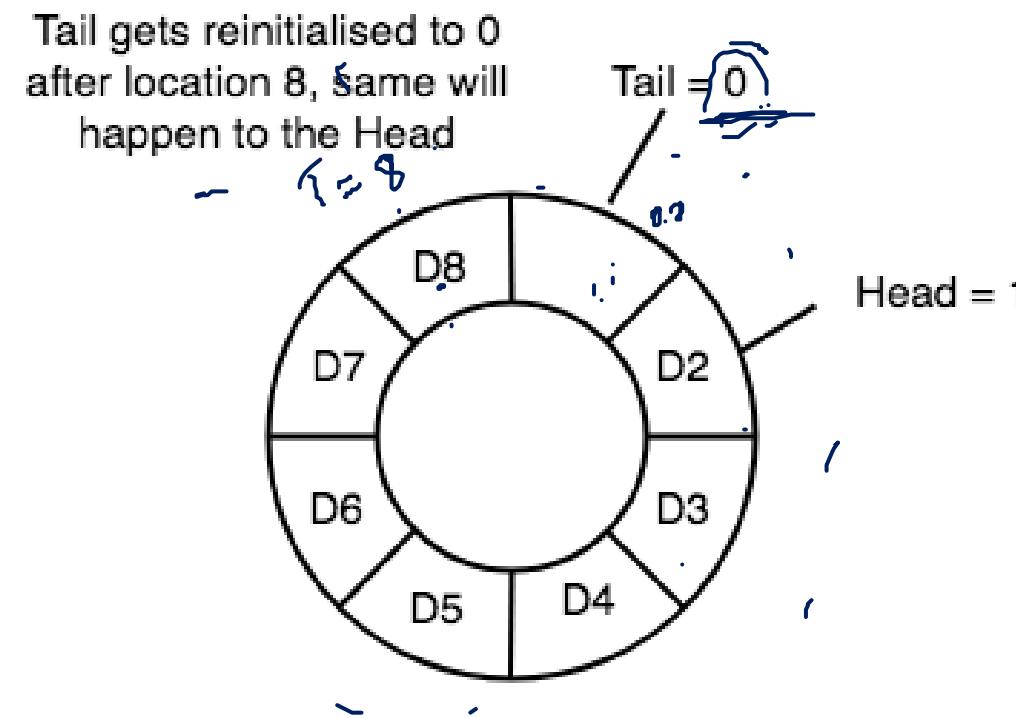
New data is always added to the location pointed by the **tail** pointer, and once the data is added, **tail** pointer is incremented to point to the next available location.



- In a circular queue, data is not actually removed from the queue.
- Only the **head** pointer is incremented by one position when **dequeue** is executed.
- As the queue data is only the data between **head** and **tail**, hence the data left outside is not a part of the queue anymore, hence removed.



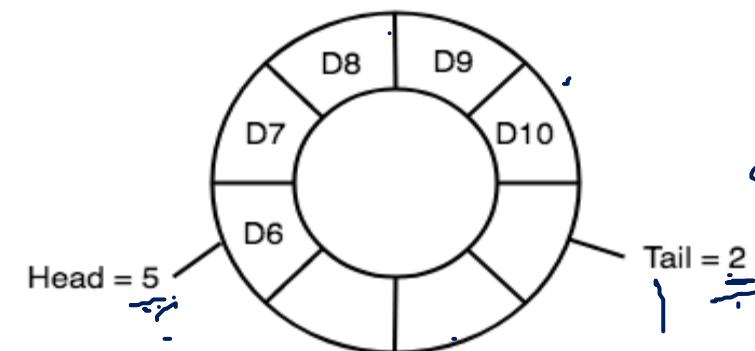
The **head** and the **tail** pointer will get reinitialized to **0** every time they reach the end of the queue.



Also, the **head** and the **tail** pointers can cross each other.

In other words, **head** pointer can be greater than the **tail**. Sounds odd?

This will happen when we dequeue the queue a couple of times and the **tail** pointer gets reinitialized upon reaching the end of the queue.



In such a situation the value of the Head pointer will be greater than the Tail pointer



Going Round and Round

Another very important point is keeping the value of the **tail** and the **head** pointer **within the maximum queue size**.

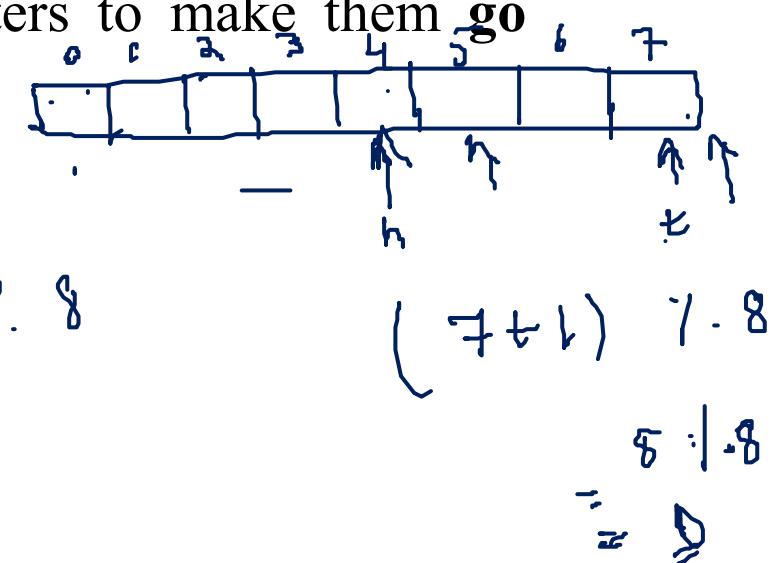
In the diagrams above the queue has a size of **8**, hence, the value of **tail** and **head** pointers will always be between **0** and **7**.

- i. This can be controlled either by checking everytime whether **tail** or **head** have reached the **maxSize** and then setting the **value 0**
- ii. Or, we have a better way, which is, for a value **x** if we divide it by **8**, the remainder will never be greater than **8**, it will always be between **0** and **8**, which is exactly what we want.

So the formula to increment the head and tail pointers to make them go round and round over and again will be,

$$\left. \begin{array}{l} \text{head} = (\text{head} + 1) \% \text{maxSize} \\ \text{tail} = (\text{tail} + 1) \% \text{maxSize} \end{array} \right\}$$

$\text{max} = 8$



$$\begin{aligned} \text{new head} &= (4+1) \% 8 \\ &= 5 \% 8 \end{aligned}$$

$$\begin{aligned} \text{new tail} &= (7+1) \% 8 \\ &= 8 \% 8 \\ &= 0 \end{aligned}$$

Application of Circular Queue

Below we have some common real-world examples where circular queues are used:

1. Computer controlled Traffic Signal System uses circular queue.

2. CPU scheduling and Memory management.



Implementation of Circular Queue

Below we have the implementation of a circular queue:

1. Initialize the queue, with size of the queue defined (**maxSize**), **head** and **tail** pointers.
2. **enqueue**: Check if the number of elements is equal to **maxSize - 1**:
 - If **Yes**, then return **Queue is full**.
 - If **No**, then add the new data element to the location of **tail** pointer and increment the **tail** pointer.
3. **dequeue**: Check if the number of elements in the queue is **zero**:
 - If **Yes**, then return **Queue is empty**.
 - If **No**, then increment the **head** pointer.



Enqueue

```
void enqueue(int queue[], int element, int& rear, int arraySize, int& count)
```

```
{
```

```
    if(count == arraySize) // Queue is full
```

```
        printf("OverFlow\n");
```

```
    else
```

```
{
```

```
        queue[rear] = element;
```

```
        rear = (rear + 1)%arraySize;
```

```
        count = count + 1;
```

```
}
```

```
}
```



Dequeue

```
void dequeue(int queue[], int& front, int rear, int& count)
```

```
{
```

```
    if(count == 0) // Queue is empty
```

```
        printf("UnderFlow\n");
```

```
    else
```

```
{
```

```
    queue[front] = 0; // Delete the front element
```

```
    front = (front + 1)%arraySize;
```

```
    count = count - 1;
```

```
}
```

```
}
```



Front

```
int Front(int queue[], int front)  
{  
    return queue[front];  
}
```

Size

```
int size(int count)  
{  
    return count;  
}
```

IsEmpty

```
bool isEmpty(int count)  
{  
    return (count == 0);  
}
```



END OF QUEUE