



problem solving using computers

CSE 1051

INTRODUCTION TO THE COURSE





Opening remarks

- Greetings
- Importance of the course
- Connection between the course and real-world applications

CSE-1051- PROBLEM SOLVING USING COMPUTERS (PSUC)

Major Course modules

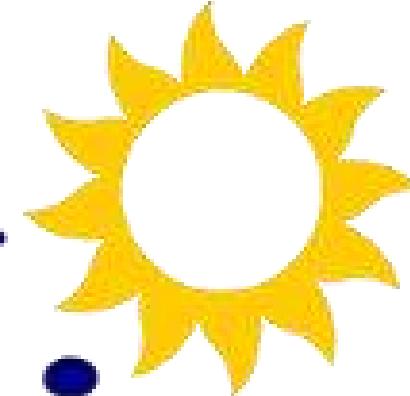
- ✓ Introduction to Computing (5 Hr)
- ✓ C language – Types, operators, expressions and control flow (8 Hrs)
- ✓ Arrays & Strings (8 Hrs)
- ✓ Modular programming and Recursive functions (9 Hrs)
- ✓ Advanced data types in C (Structures and Pointers) (6 Hrs)

Course Facilitation

- Teaching Methodology
 - Power point presentation and Digital scribble pad
- Mode
 - Microsoft Teams
- Syllabus, Course Plan and Assessment scheme & schedule will be shared soon.

Best Practices

- Appeal to move to the higher cognitive levels
- Making reference/notes from prescribed text books/resources
- Preparing own class notes
- Discussion with peers and teacher
- Punctuality and Presence (Attendance) in the class

Happy Learning 



Go to posts/chat box for the link to the question **PQn. S1.0**
submit your solution in next 2 minutes
The session will resume in 3 minutes

Introduction to Computing

s1_1

Objectives

To learn and appreciate the following concepts

- ✓ Problem solving basics
- ✓ Logic and its importance in problem solving
- ✓ Various computational problems and its classification
- ✓ Computer Organization and operating system
- ✓ Different types of languages
- ✓ History of C, Typical C program development environment.

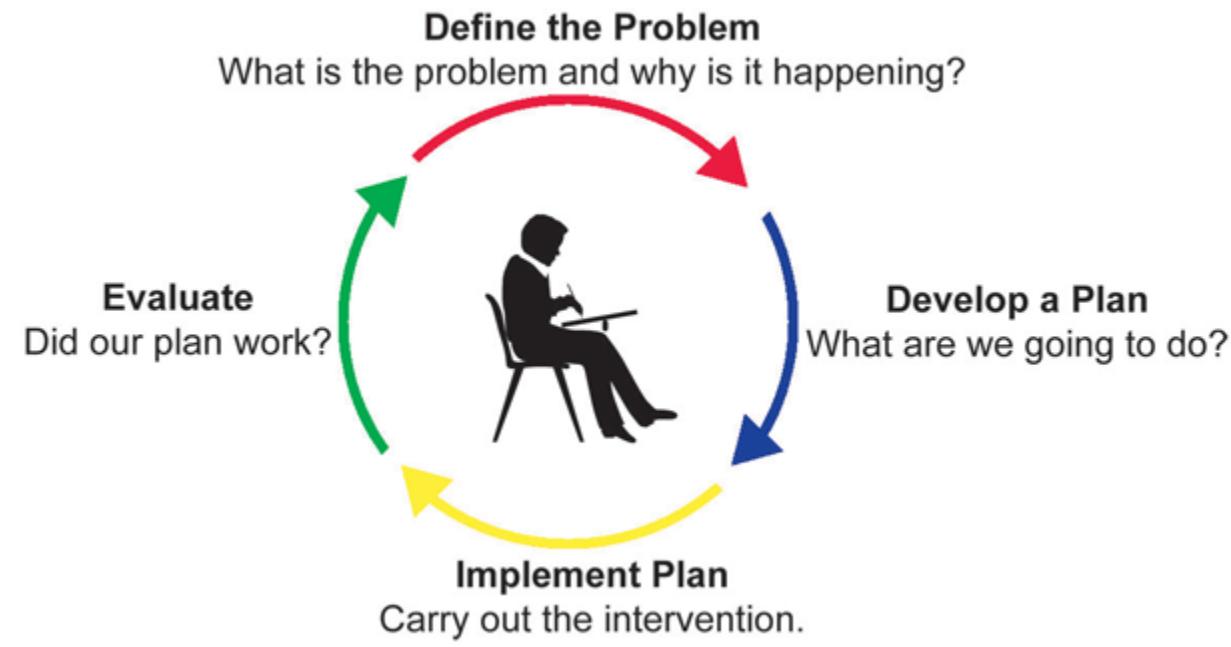
Session outcome

- At the end of session the student will be able to understand
 - Importance of problem solving techniques, Computer organization, Operating system, Types of languages
 - History of C, programming development environment

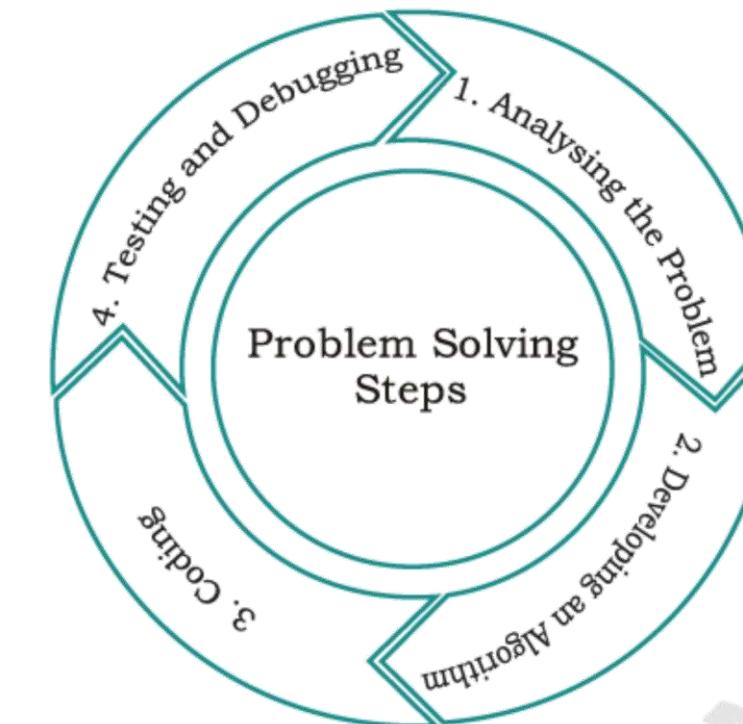


Introduction to problem solving

Problem Solving is the sequential process of analyzing information related to a given situation and generating appropriate response options.



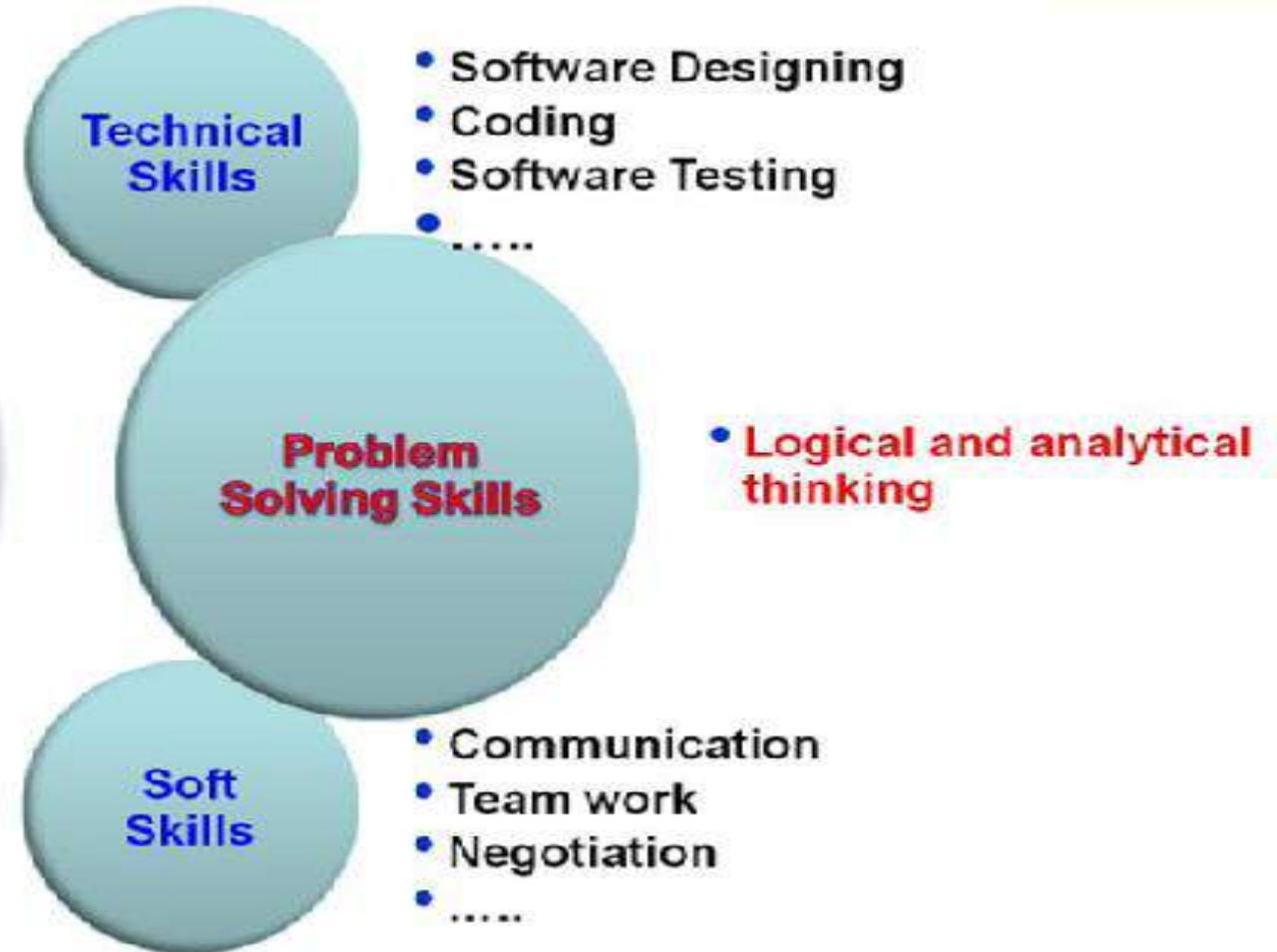
General problem solving steps



Problem solving steps for Computing



Skill set required for Software Engineers



What is a problem?

- ❖ A **problem** is a puzzle that requires **logical thought** or **mathematics** to solve
- ❖ A puzzle could be a set of questions on a scenario which consists of **description of reality** and a set of **constraints** about the scenario.
 - ✓ e.g. **Scenario-** Infosys Mysore campus has a library. The librarian issues book only to Infosys employees.

Description of reality: There is a library in Infosys Mysore campus . There is a librarian in the library

Constraints: librarian issues book only to Infosys employees.

Questions about the scenario:

- How many books are there in the library?
- How many books can be issued to an employee?
- Does the librarian issue book to himself? Etc.

What is the fundamental requirement for answering these questions or in general solving any problem?



Logic

- A method of human thought that involves thinking in a linear, step by step manner about how a problem can be solved.
- **Logic** is a language for reasoning. It is a collection of rules we use when doing reasoning.

e.g. John's mother has four children.

First child is April

Second child is May

Third child is June

What is the name of fourth child?



Importance of logic in problem solving

- Solution for any problem(e.g. summation of two numbers) requires three things.

Input: Input values(e.g. 3 and 2)

Process: Process of summation

Output: Output after process (e.g. sum of numbers, 5)

- The process part (e.g. summation) of the solution requires **logic** (How to sum) or in other words based on the logic, process is developed.

Importance of logic in problem solving

- For solving a problem, there may be multiple valid logics, some may be simple and some may be complex.
e.g. To determine whether the number is prime or not.

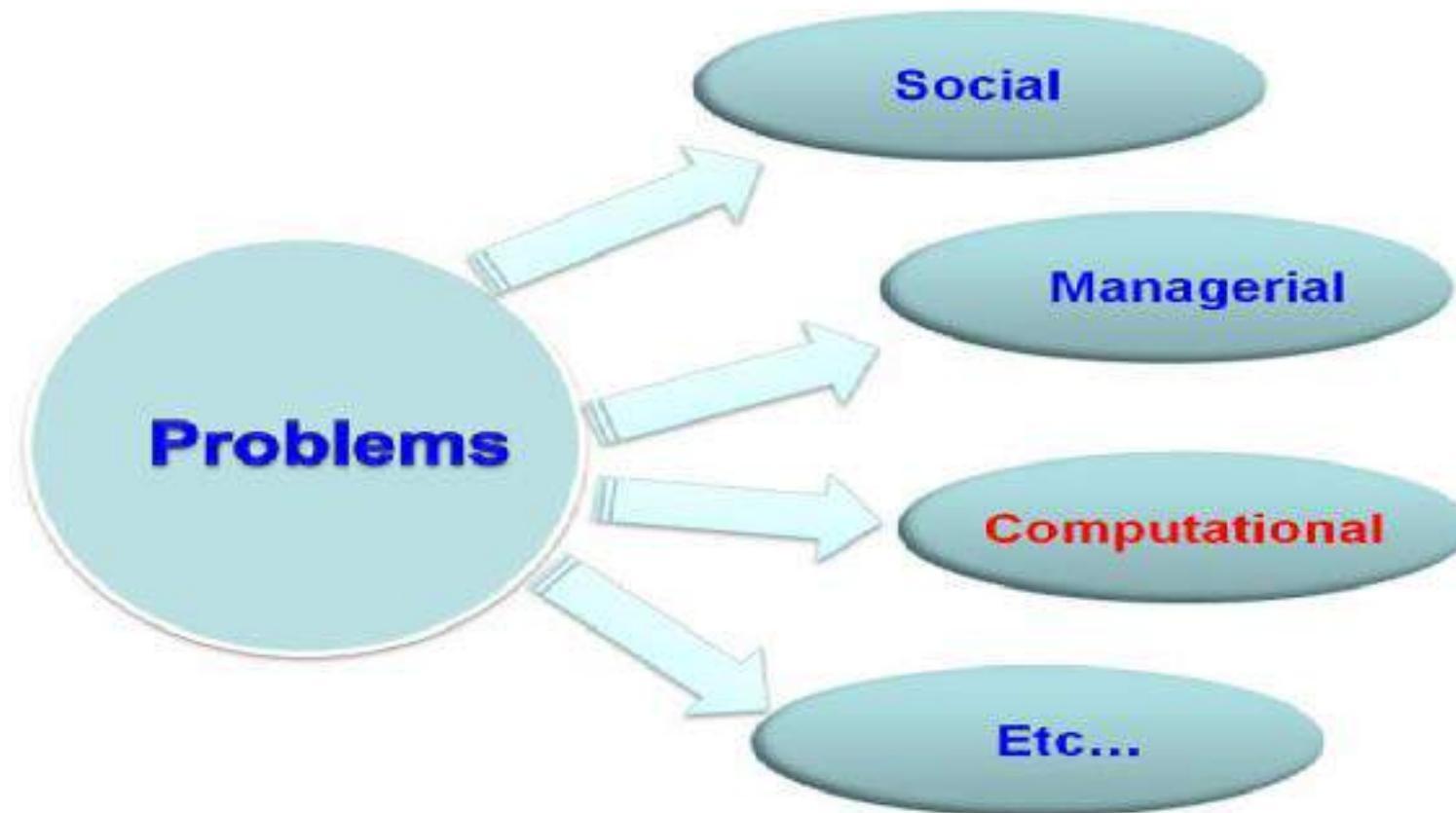
Logic 1- divide the number by all the numbers starting from 2 to one less than the number and if for all the division operations, the remainder is non zero, the number is prime. Else the number is not prime.

Logic 2 – same as logic 1 but divide the number from 2 to $\text{number}/2$

Logic 3 - same as logic 1 but divide the number from 2 to square root of the number



Types of problems



Computational Problems

Definition: Computation is a process of evolution from one state to another in accordance with some rules.



Broad applications of Computational Problem

where the answer for every instance is either yes or no.

Decision Problem

Deciding whether a given number is prime

Searching an element from a given set of elements. Or arranging them in an order

Searching & Sorting Problem

Finding product name for given product ID and arranging products in alphabetical order of names

Counting no. of occurrences of a type of elements in a set of elements

Counting Problem

Counting how many different type of items are available in the store

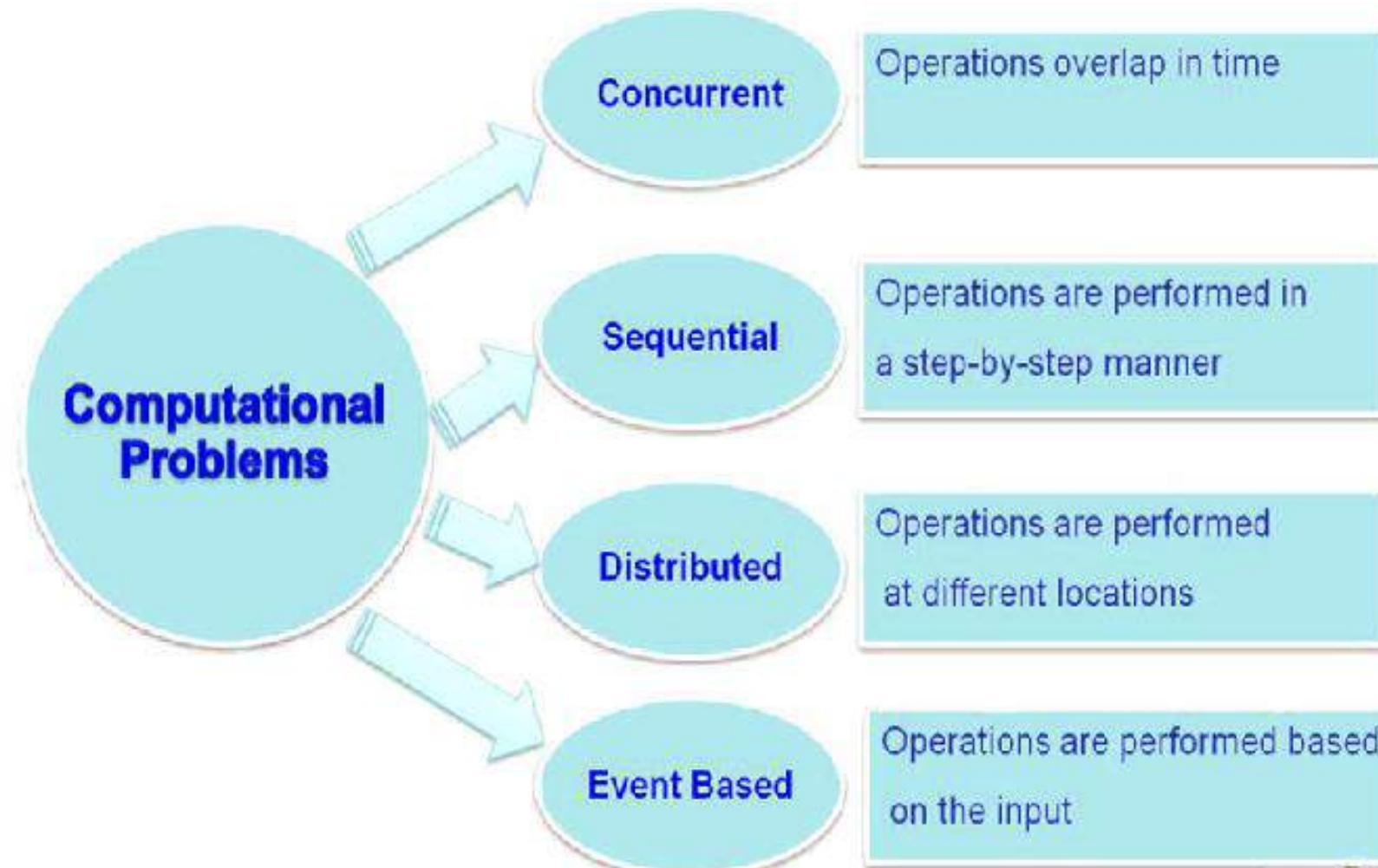
Finding the best solution out of several feasible solutions

Optimization Problem

Finding best combination of products for promotional campaign



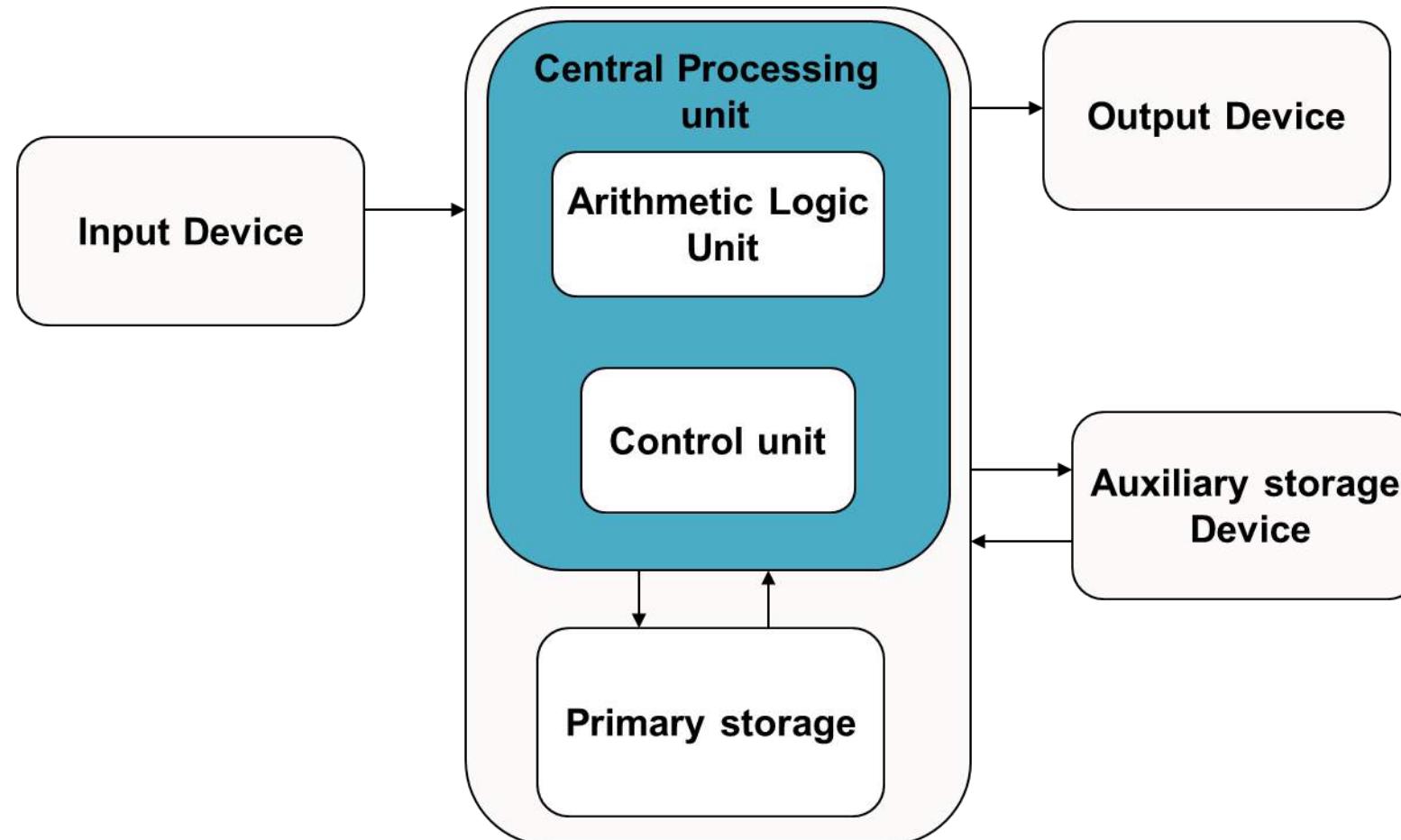
Classification of computational problems





Go to posts/chat box for the link to the question **PQn. S1.1**
submit your solution in next 2 minutes
The session will resume in 3 minutes

Computer Organization



Central Processing Unit

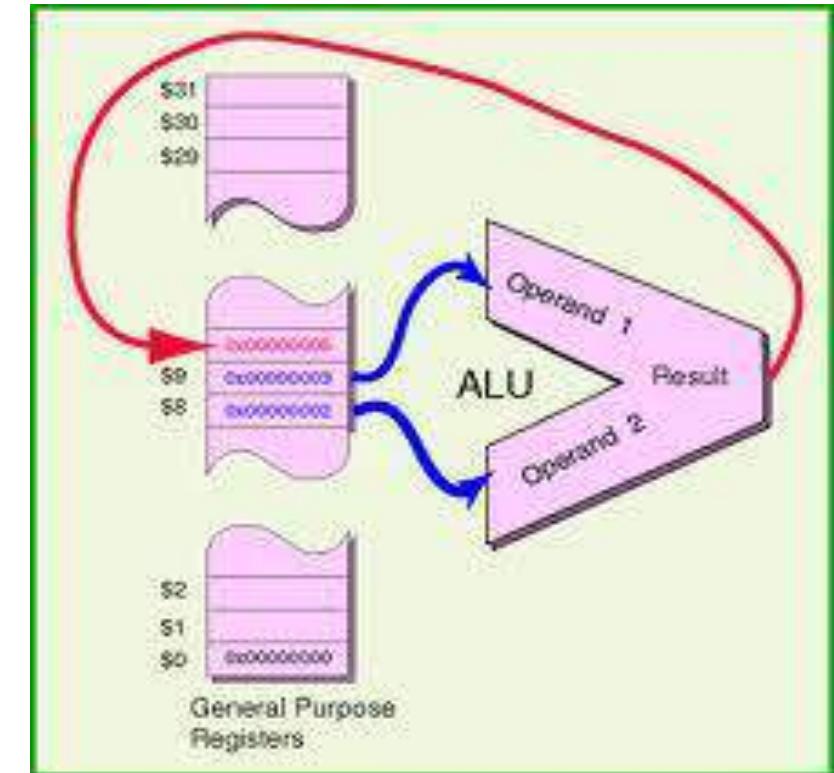
- Data and instructions are processed in the CPU
- Consists of two functional units
 - Control Unit (CU)
 - Arithmetic and Logic Unit (ALU)



Arithmetic and Logical unit

- Performs arithmetic and logical operations:

- Example:
- arithmetic(+,-,*,/ etc..) and
- logical (AND, OR, NOT, <= etc..) operations



Control unit

- Controls the order in which your program instructions are executed.
 - Functions of CU:
 - Fetches data and instructions to main memory
 - Interprets these instructions
 - Controls the transfer of data and instructions to and from main memory
 - Controls input and output devices.
 - Overall supervision of computer system

So what we learned about a computer....



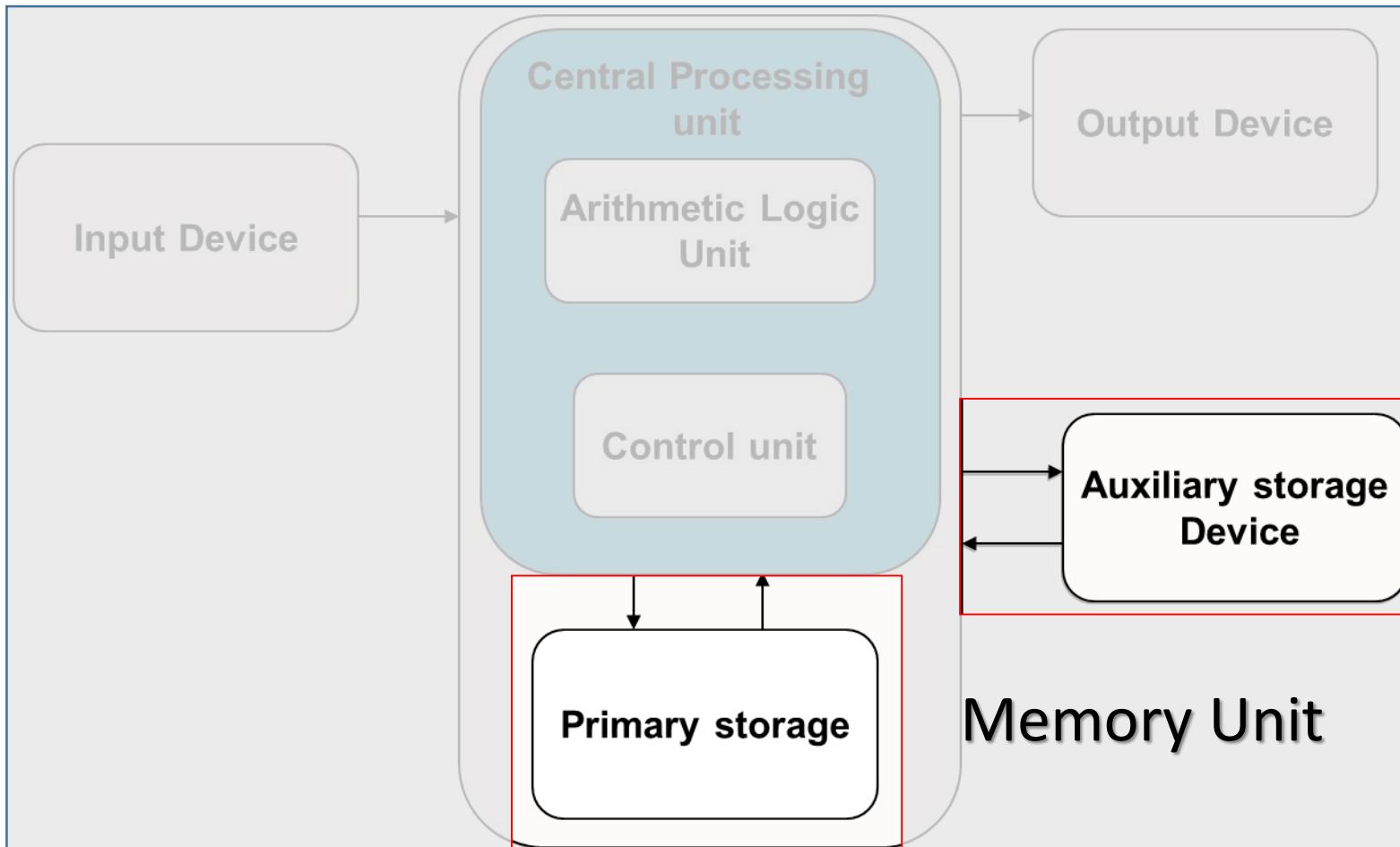
Summary

- ✓ Problem solving
- ✓ Logic and its importance in problem solving
- ✓ Computational problems and its classifications
- ✓ Computer organization

Introduction to Computing

s1_2

Computer Organization



Memory unit

- Storage device where the data and instructions fed by the user are **stored**
- An **ordered sequence of storage cells**, each capable of holding a piece of **information**

- Each cell has its own **unique address**
- The information held can be input data, computed values, or your program instructions.

Address	Contents
00000000	11100011
00000001	10101001
:	:
.	.
11111100	00000000
11111101	11111111
11111110	10101010
11111111	00110011

Memory unit

- The computer memory is measured in terms of **bits, bytes and words.**
- A **bit** is a **binary digit** either 0 or 1.
- A **byte** is unit of memory and is defined as sequence of 8 bits.
- The **word** can be defined as a sequence of 16/32/64 bits or 2/4/8 bytes respectively depending on the machine architecture.

Computer memory classification

- Main memory-Primary storage
- Secondary memory-Auxiliary storage
- Cache memory

Main memory

- Memory where the data and instructions, currently being executed are stored
 - Located outside CPU
 - High speed
 - Data and instructions stored get erased when the power goes off
- Also referred as **primary / temporary** memory
 - Semiconductor memory
 - Measured in terms of megabytes and gigabytes

Primary storage: RAM & ROM

- RAM stands for **Random Access Memory**
 - Read and write memory
 - Information typed by the user are stored in this memory
 - Any memory location can be accessed directly without scanning it sequentially (random access memory)
 - During power failure the information stored in it will be erased → volatile memory
- ROM stands for **Read Only Memory**
 - Permanent memory and non volatile
 - Contents in locations in ROM can not be changed
 - Stores mainly stored program and basic input output system programs

Secondary memory

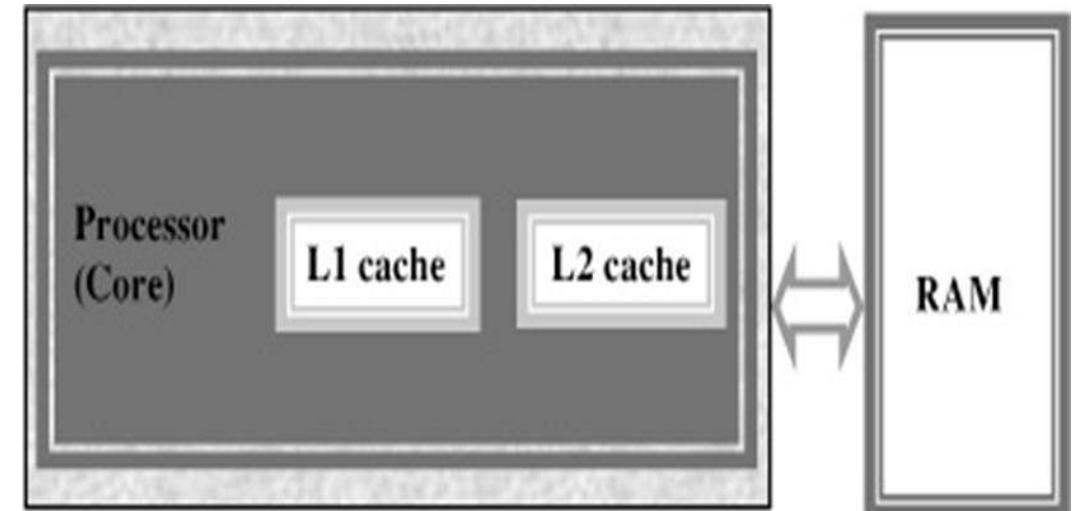
- Main memory is volatile and limited
 - Hence it is essential for other types of storage devices where programs and data can be stored when they are no longer being processed
- Installed within the computer at the factory or added later as needed

Secondary memory

- Non-volatile memory
- Made up of magnetic material
- Stores large amount of information for long time
- Low speed
- Holds programs not currently being executed

Cache memory

- High speed memory placed between CPU and main memory
- Stores data and instructions currently to be executed
- More costlier but less capacity than main memory
- Users can not access this memory



Memory System (Video)



Welcome to

***MAKE IT EASY
EDUCATION***



Operating System

- OS is an **integrated collection** of programs which make the computer operational and help in executing user programs.
 - It acts as an **interface** between the man and machine.
 - It **manages** the system resources like memory, processors, input-output devices and files.
- ✓ e.g. Windows, Linux, DOS

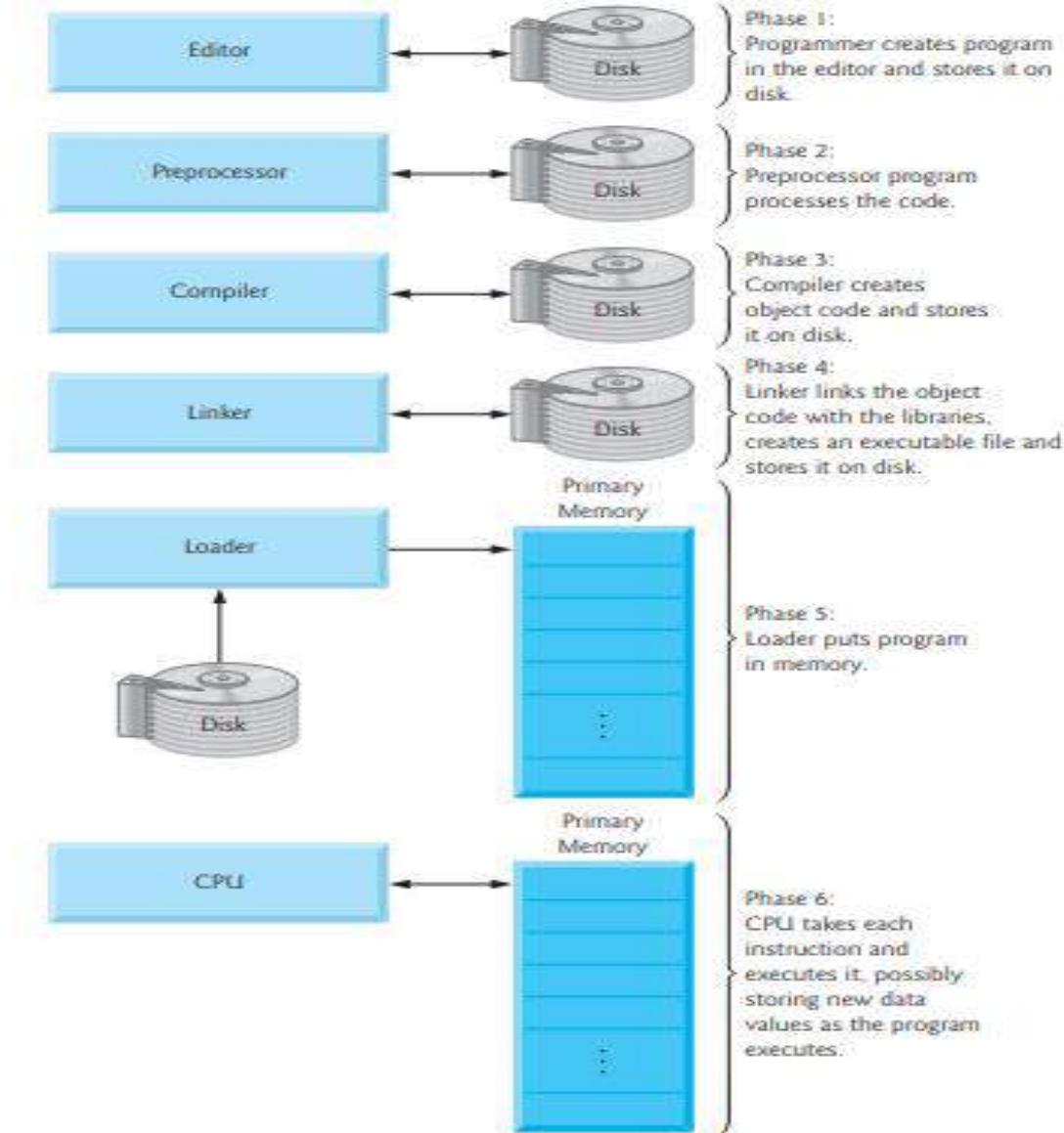
Computer Languages

- Machine Language- The only programming language available in earlier days
 - Consists of only **0's** and **1's**; e.g.: - 10101011
- Symbolic language or Assembly language-
 - **symbols** or **mnemonics** are used to represent instructions
 - hardware specific
 - ✓ e.g. MASM : ADD X,Y; Add the contents of y to x
- High-level languages- **English like** language using which the programmer can write programs to solve a problem.
 - more concerned with the problem specification
 - not oriented towards the details of computer
 - ✓ e.g. C, C++, C#, Fortran, BASIC, Pascal etc.

Language Translator

- **Compiler** : Program that translates entire high-level language program into machine language at a time. e.g. C, C++ compilers.
- **Interpreter** : Program which translates one statement of a high-level language program into machine language at a time and executes it.
e.g. Basic Interpreters, Java Interpreters.
- **Assembler** : Program which translates an assembly language program into machine language.
e.g. TASM(Turbo ASseMbler), MASM(Macro ASseMbler).

Typical C program development environment



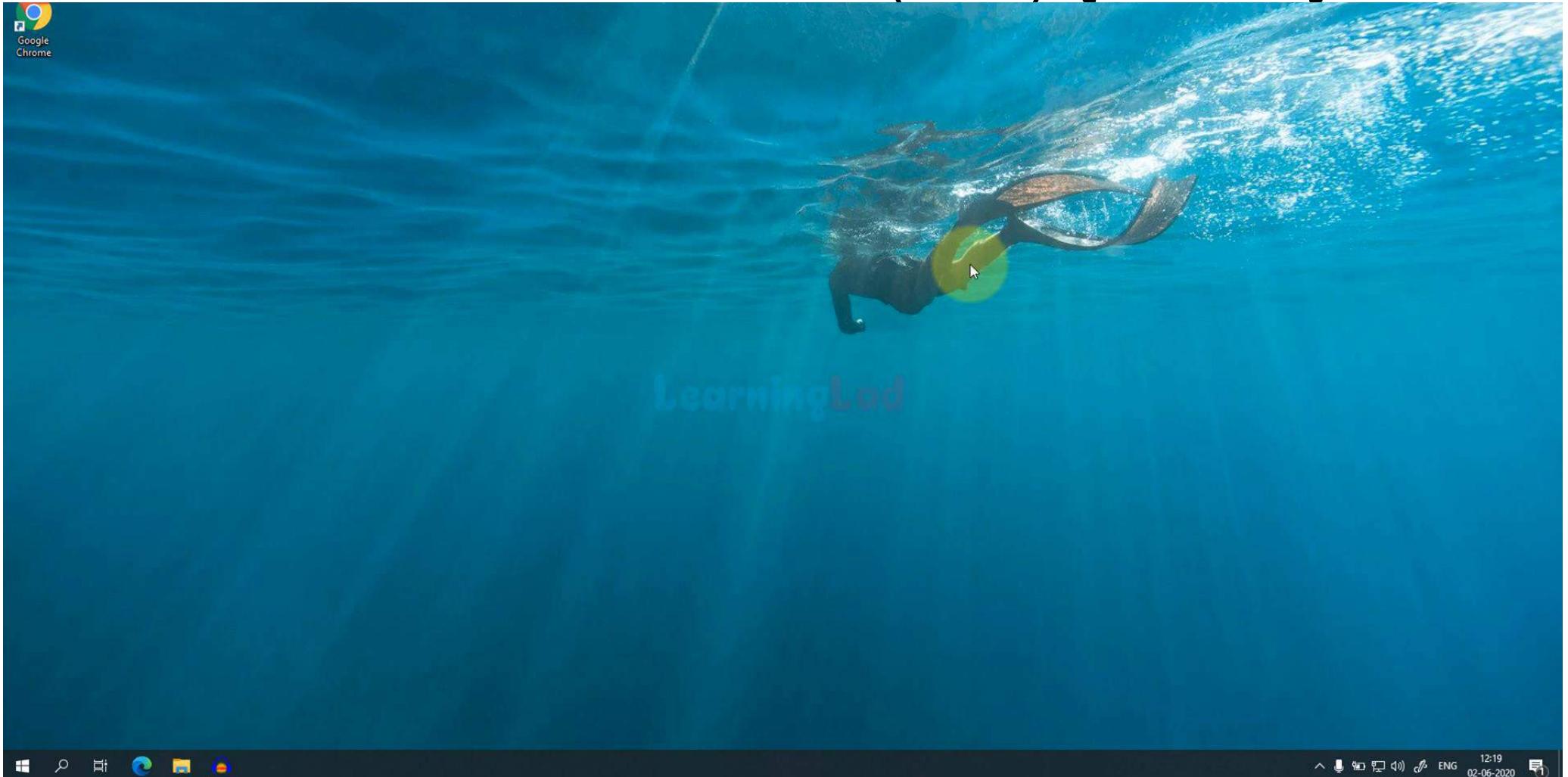
Typical C program development environment

C programs typically go through six phases to be executed.

These are: edit, preprocess, compile, link, load and execute

- ✓ Phase 1 : creating a program
- ✓ Phases 2 and 3: Preprocessing and Compiling a C Program
- ✓ Phase 4: Linking
- ✓ Phase 5: Loading
- ✓ Phase 6: Execution

How to Install Code Blocks (IDE) [Video]



My first C program

#include<stdio.h>  preprocessor directive

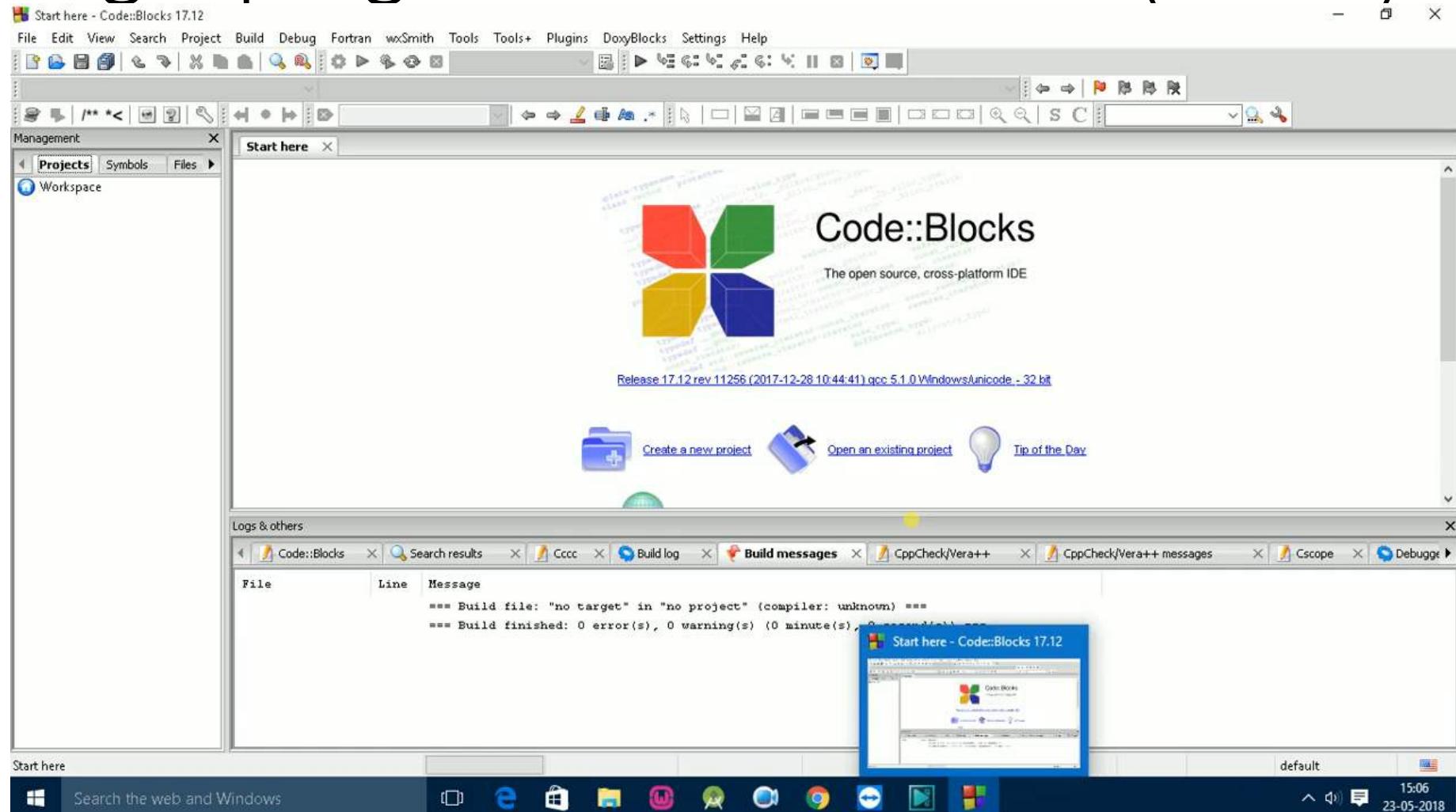
int main()  starting point of execution of the 'C' program
{  Signifies beginning of the 'C' program

printf("Hello World");   Body of the 'C' Program

return 0;
}  Signifies ending of the 'C' program

- **printf**- This statement is used to output any message on the screen or console (message within the double quotes)

Running C program in Code Blocks (Video)



Summary

- ✓ Memory System
- ✓ Operating system
- ✓ Different computer languages
- ✓ Typical C program development environment
- ✓ Code::Blocks Integrated Development Environment (IDE)

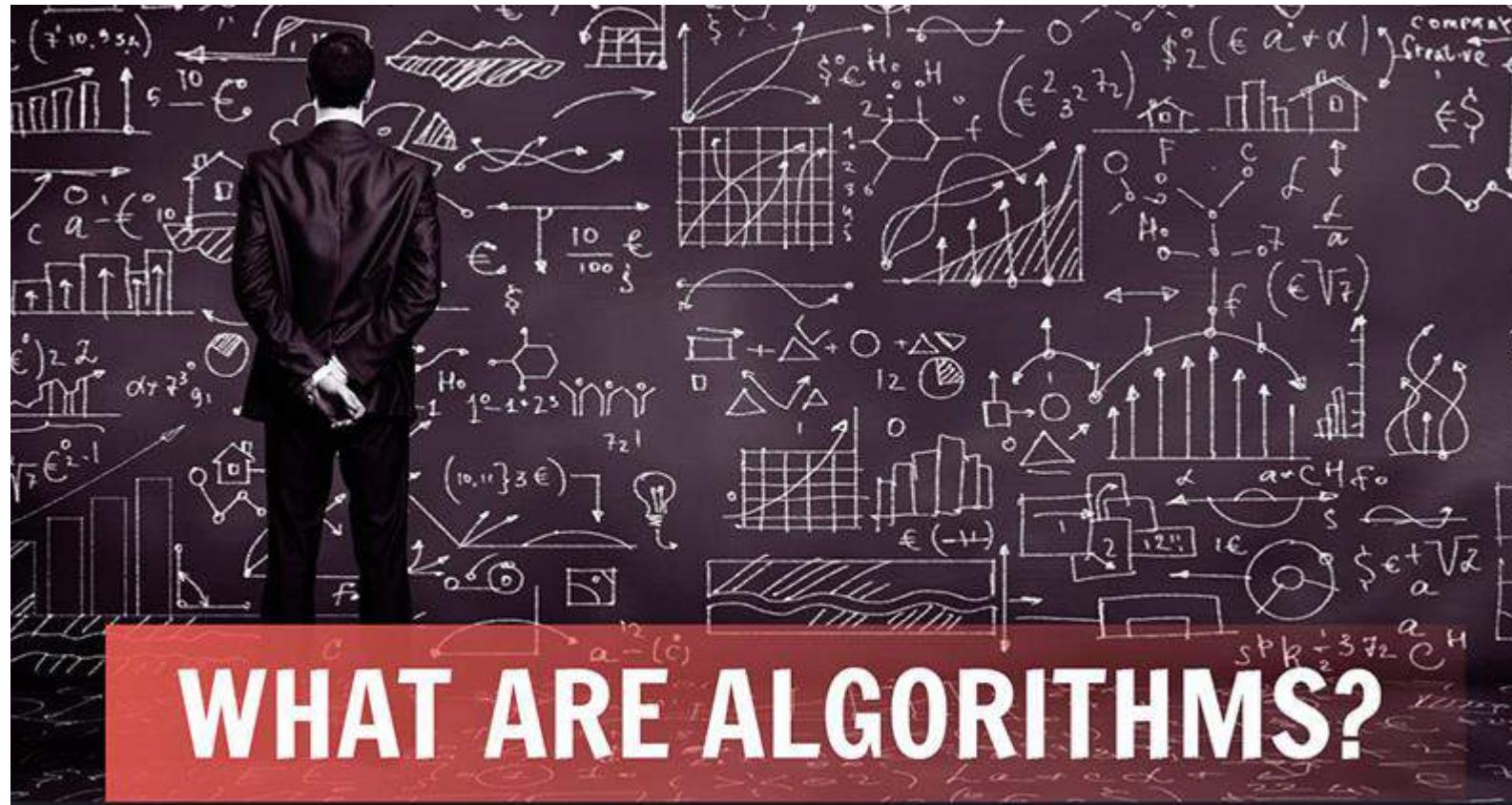


Go to posts/chat box for the link to the question **PQn. S1.2**
submit your solution in next 2 minutes
The session will resume in 3 minutes



Session 2_1

Algorithms



Learning objectives

To learn and appreciate the following concepts

- ✓ Introduction to algorithms
- ✓ Algorithms for simple problems

Session outcome

- ✓ At the end of session the student will be able to write
- ✓ Algorithms for simple problems

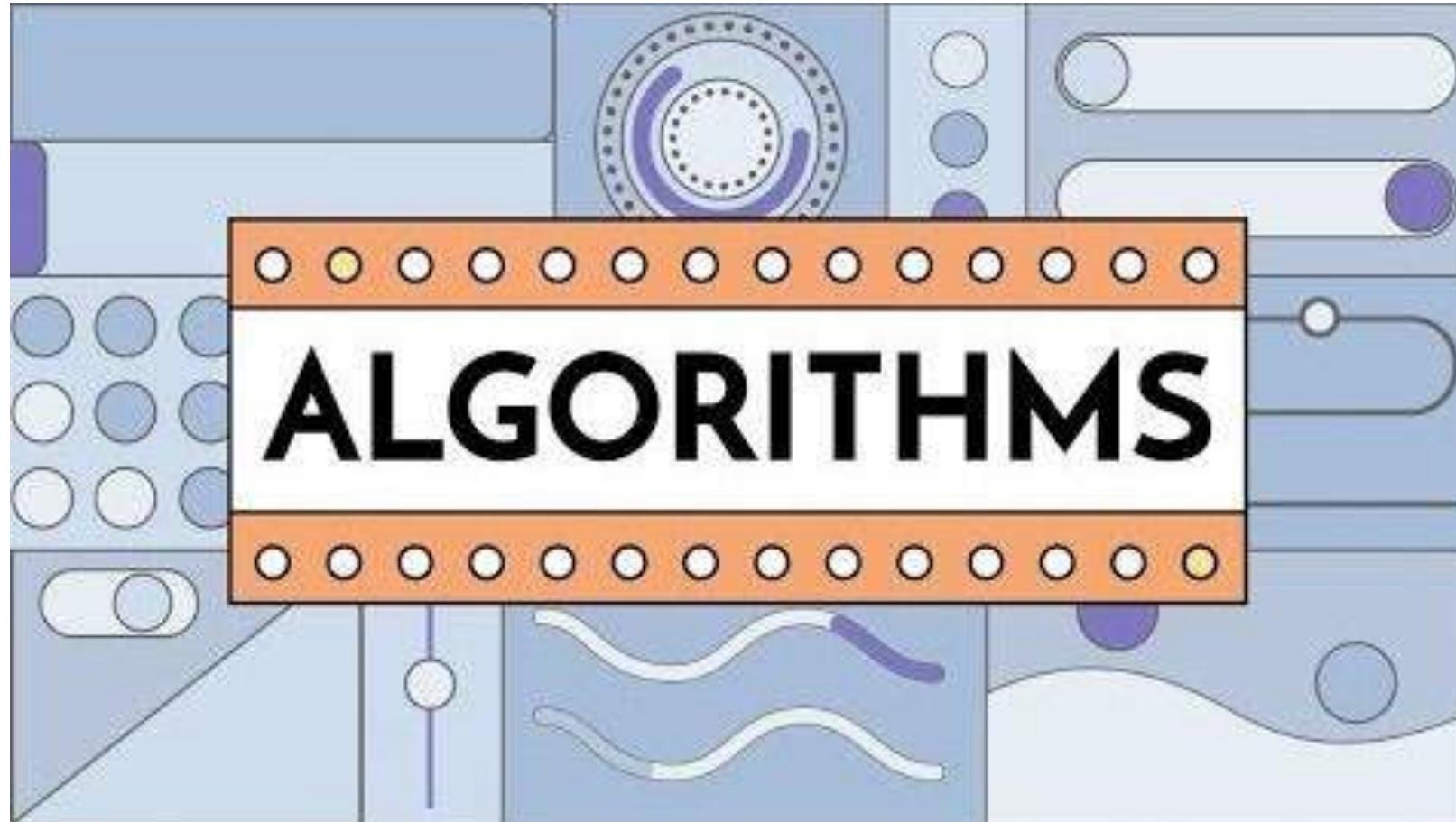
Algorithm

- ✓ A step by step procedure to solve a particular problem
- ✓ Named after Arabic Mathematician Abu Jafar Mohammed Ibn Musa

Al Khowarizmi



Relevance of an algorithm to Computer Science



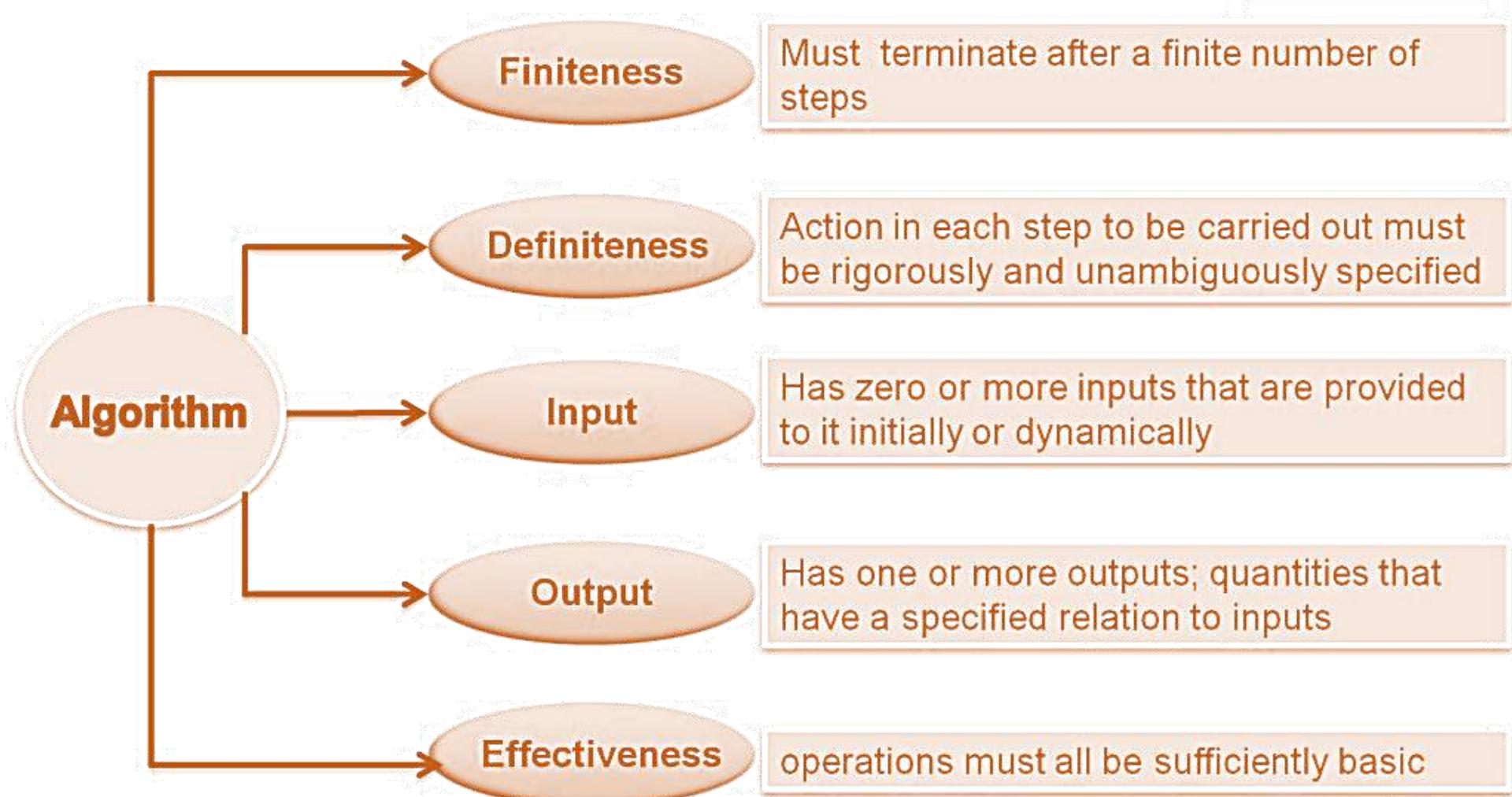
URL: https://www.youtube.com/watch?v=kM9ASKAni_s

Algorithmic Notations

- **Name of the algorithm** [mandatory]
[gives a meaningful name to the algorithm based on the problem]
- **Start** [Begin of algorithm]
- **Step Number** [mandatory]
[indicate each individual simple task]
- **Explanatory comment** [optional]
[gives an explanation for each step, if needed]
- **Termination** [mandatory]
[tells the end of algorithm]

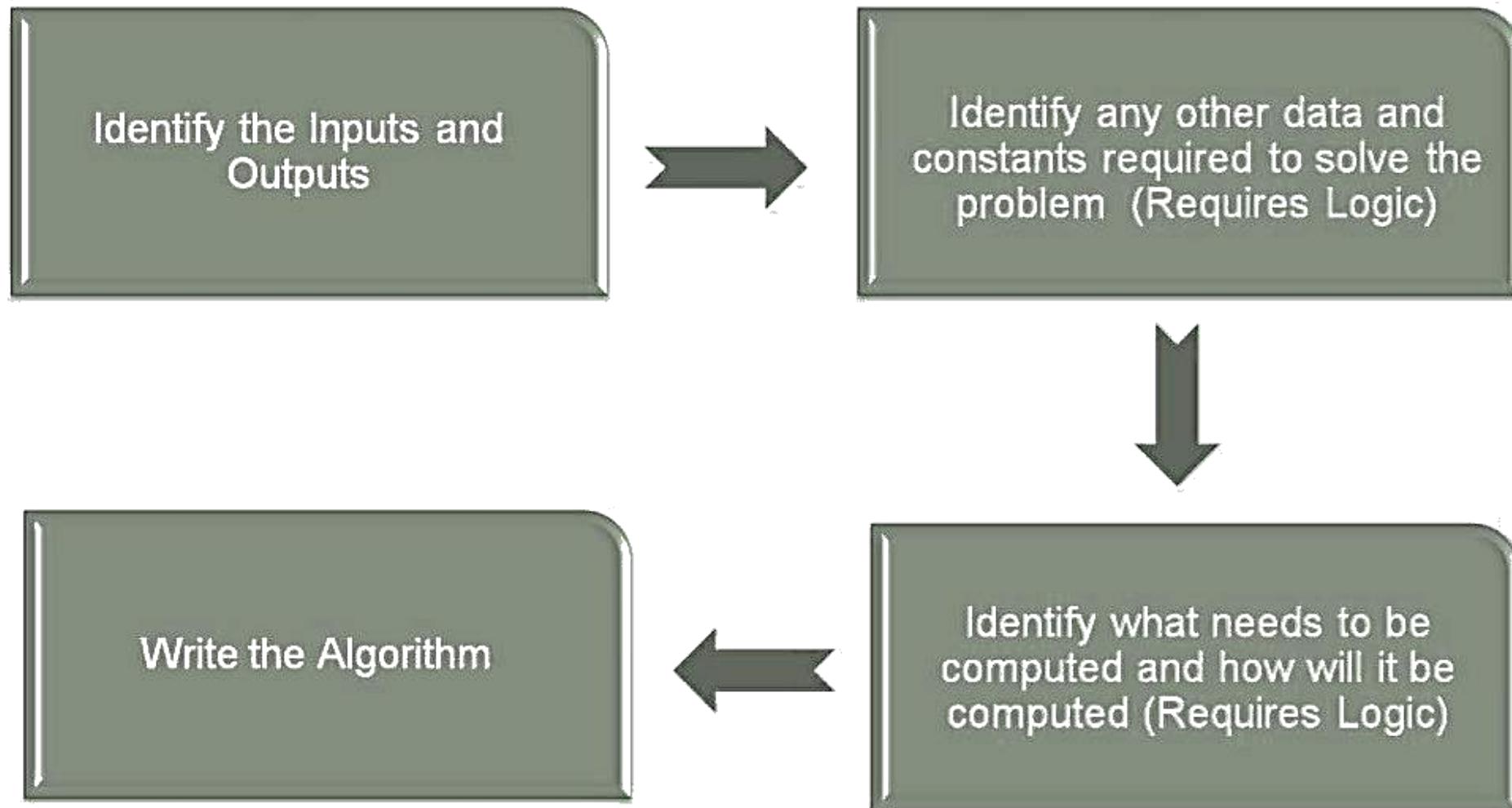


Properties of an algorithm





Steps to develop an algorithm



Algorithm to compute the area of circle!!!

Name of the algorithm : Compute the area of a circle

Step1: Start

Step 2: Input radius

Step 3: [Compute the area]

Area $\leftarrow 3.1416 * \text{radius} * \text{radius}$

Step 4: [Print the Area]

Print 'Area of a circle =', Area

Step 5: [End of algorithm]

Stop

Algorithm to Interchange values of two variables!!!

Name of the algorithm: Interchange values of 2 variables

Step 1: Start

Step 2: Input A,B

Step 3: temp \leftarrow A

Step 4: A \leftarrow B

Step 5: B \leftarrow temp

Step 6: Print 'A=' , A

Print 'B=' , B

Step 7: [End of Algorithm]

Stop



Go to posts/chat box for the link to the question

submit your solution in next 2 minutes

The session will resume in 3 minutes

Algorithm to find largest of 3 numbers!!!

Name of the algorithm: Find largest of 3 numbers

Step 1: Start

Step 2: [Read the values of A, B and C]

 Read A, B, C

Step 3: [Compare A and B]

 if A>B Go to step 5

Step 4: [Otherwise compare B with C]

 if B>C then

 Print 'B' is largest'

 else

 Print 'C' is largest'

 Go to Step 6

Step 5: [Compare A and C for largest]

 if A>C then

 Print 'A' is largest'

 else

 Print 'C' is largest'

Step 6: [End of the algorithm]

 Stop

What's great about algorithm!!! Think

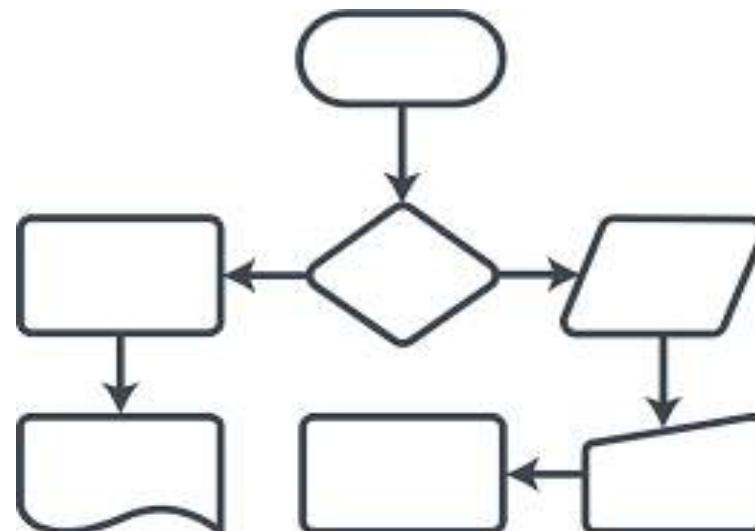
- By developing a good understanding of a large range of algorithms, you will be able to choose the right one for a problem and apply it properly.

Tutorial on Algorithms

- Write an algorithm to add, subtract, multiply and divide two integers
- Write an algorithm to swap values of two variables without using a third variable

S2_2

Flow charts



Learning objectives

To learn and appreciate the following concepts

- ✓ Introduction to flowcharts
- ✓ Installation of RAPTOR

Session outcome

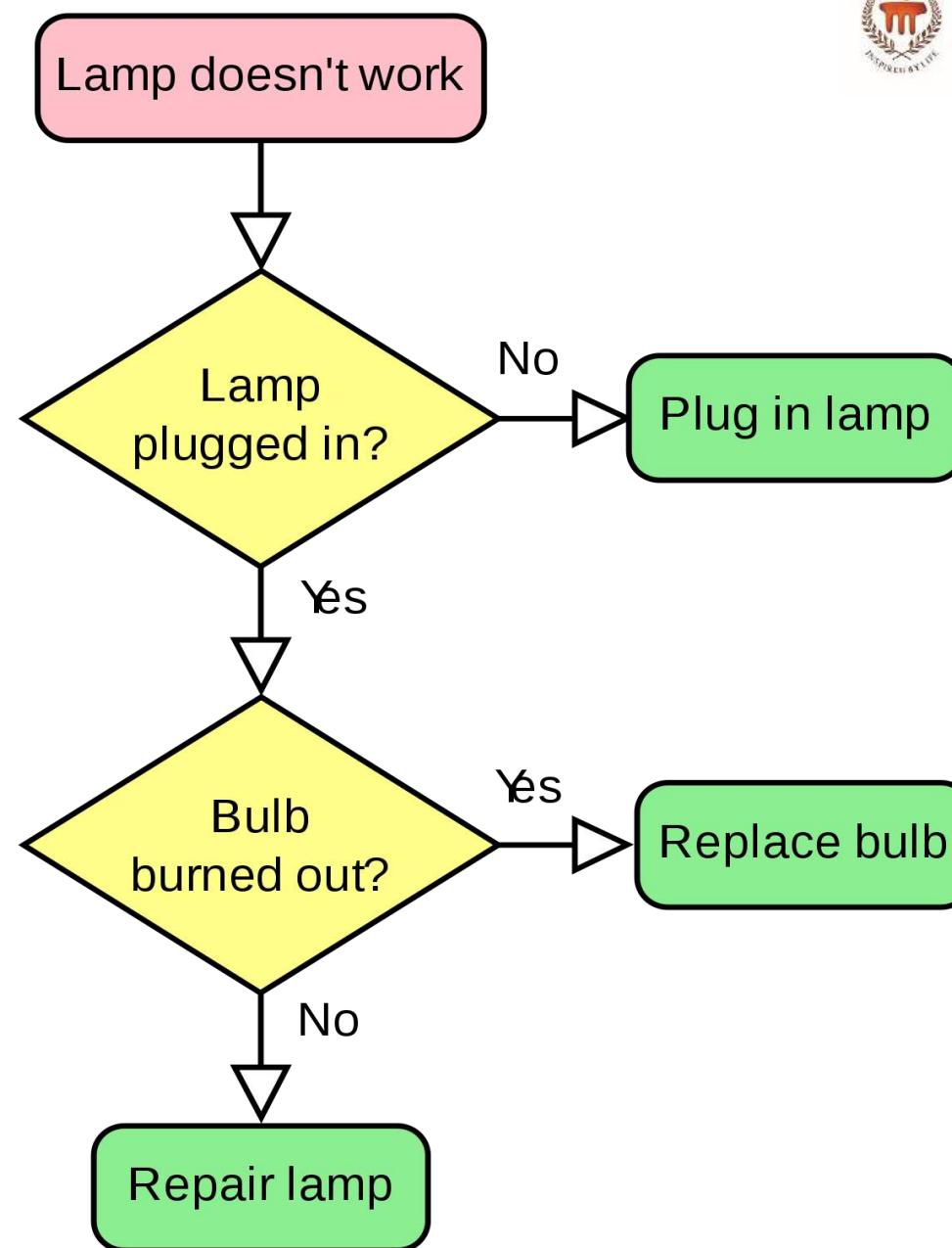
- ✓ At the end of session the student will be able to
 - ✓ Understand importance of flowchart
 - ✓ Install RAPTOR and appreciate how it works

Flowcharts

- ✓ In Computer Science, **Flow chart** is used to represent algorithm which basically provides a solution to any computational problem.
- **Flowchart:** A graphical/pictorial representation of computation

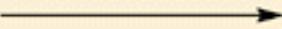
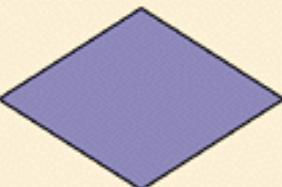
Key features of flowchart

- ✓ Diagrammatic / visual / graphical representation of computation of an algorithm/pseudo code
- ✓ Easier to understand and analyze the problem and it's solution before programming
- ✓ Machine independent
- ✓ Well suited for any type of logic



Simple
Flowchart!!!

Basic Flowchart Symbols

Name	Symbol	Use in flowchart
Oval		Denotes the beginning or end of a program.
Flow line		Denotes the direction of logic flow in a program.
Parallelogram		Denotes either an input operation (e.g., INPUT) or an output operation (e.g., PRINT).
Rectangle		Denotes a process to be carried out (e.g., an addition).
Diamond		Denotes a decision (or branch) to be made. The program should continue along one of two routes (e.g., IF/THEN/ELSE).

Area of the circle

Name of the algorithm:
Compute the area of a circle

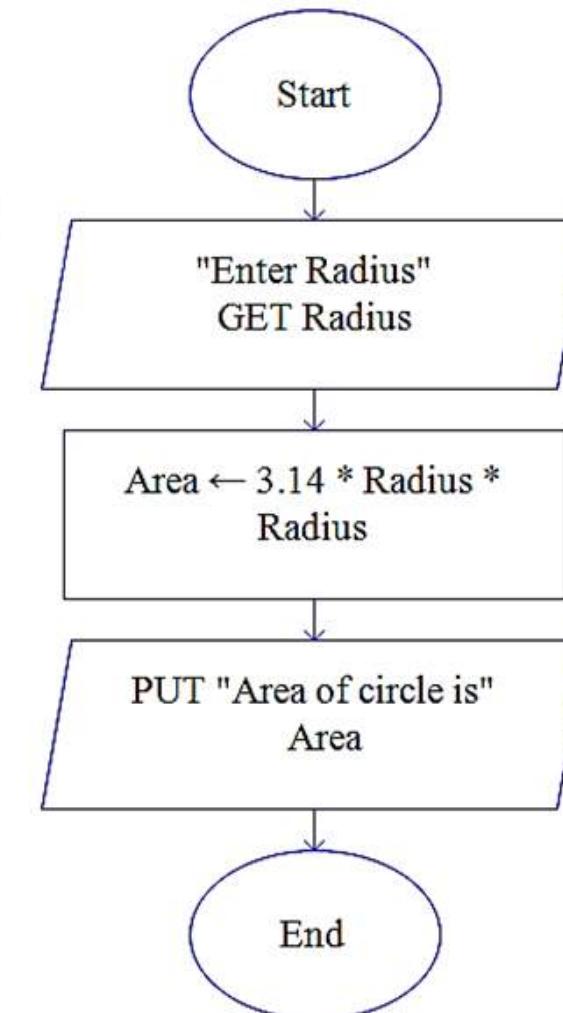
Step1: Input radius

Step 2: [Compute the area]
 $\text{Area} \leftarrow 3.1416 * \text{radius} * \text{radius}$

Step 3: [Print the Area]
Print 'Area of a circle =', Area

Step 4: [End of algorithm]
Stop

Flowchart



Comparing two numbers

Name of the algorithm: Comparing 2 numbers

Step 1: Start

Step 2: Input num1, num2

Step 3: if num1 > num2 then

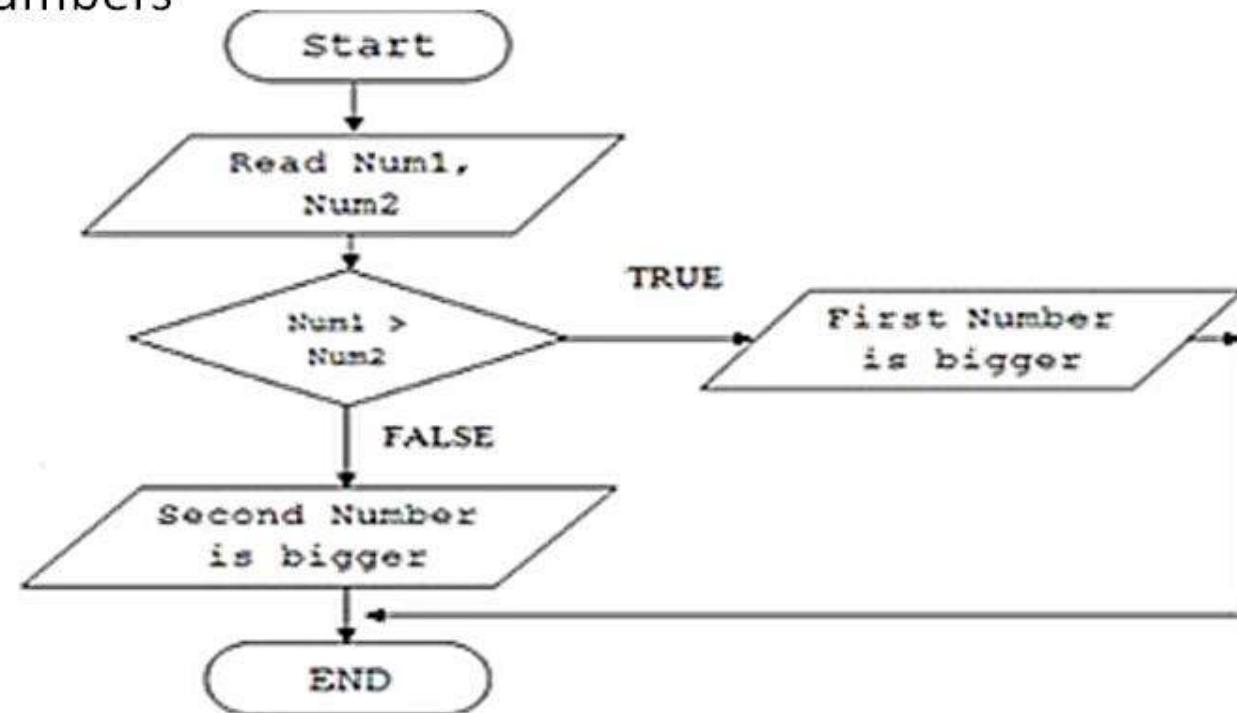
 Print num1 is bigger

else

 Print num2 is bigger

Step 4: end

Flowchart



Swapping two numbers

Name of the algorithm: Swapping 2 numbers

Step1: Input two numbers

Step 2: [swapping]

temp=a

a=b

b=temp

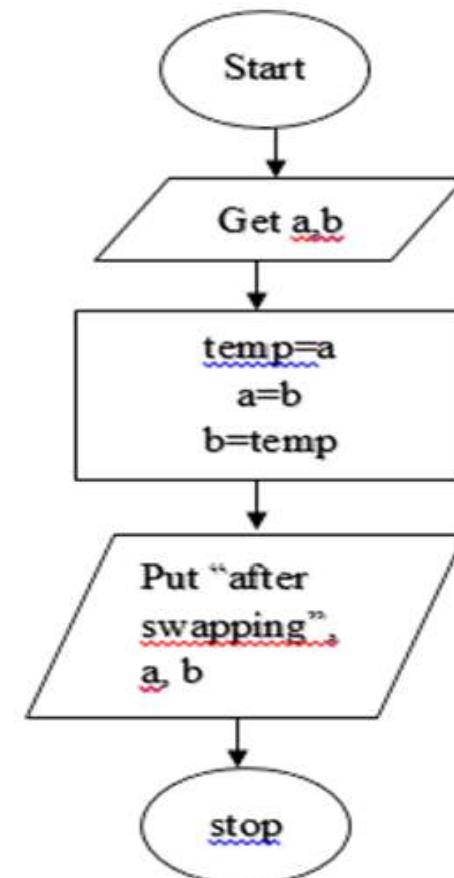
Step 3: [Print]

Print 'after swapping=', a, b

Step 4: [End of algorithm]

Stop

Flowchart





Go to posts/chat box for the link to the question
submit your solution in next 2 minutes
The session will resume in 3 minutes



RAPTOR – Rapid Algorithmic Programming Tool for Ordered Reasoning. Flowchart Interpreter!!!

- RAPTOR is a flowchart-based programming environment, designed specifically to help students visualize their algorithms and avoid syntactic baggage.





URL and setup - RAPTOR

- <https://raptor.martincarlisle.com/>

A screenshot of a web browser window titled "RAPTOR - Flowchart Interpreter". The address bar shows the URL "raptor.martincarlisle.com". The main content area displays an advertisement for "Whitehat Jr - Coding Platform" with a link to "code.whitehatjr.com" and a large "OPEN" button. Below the ad, there is a section titled "Download RAPTOR using button below" with a "Download latest version" button. A hand-drawn style arrow points from the text "DOWNLOAD RAPTOR HERE!!" at the bottom right towards the "Download latest version" button.

Advertisement (may include download button for something else)

code.whitehatjr.com

Whitehat Jr - Coding Platform OPEN

Download RAPTOR using button below

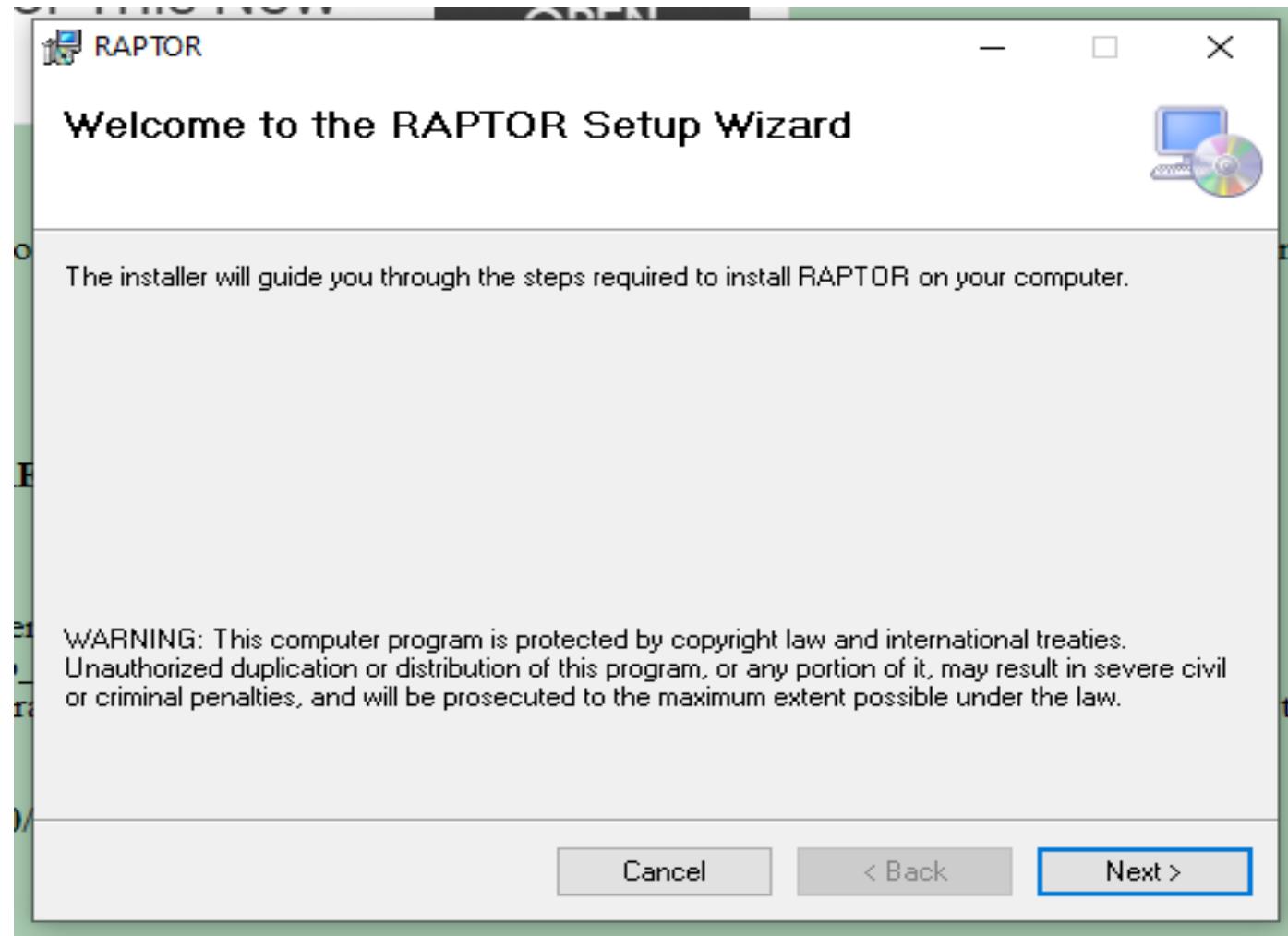
Where/how are you using RAPTOR? I keep a list of what schools and universities are using RAPTOR and for what class.

Newest Installer (11/19/2019)

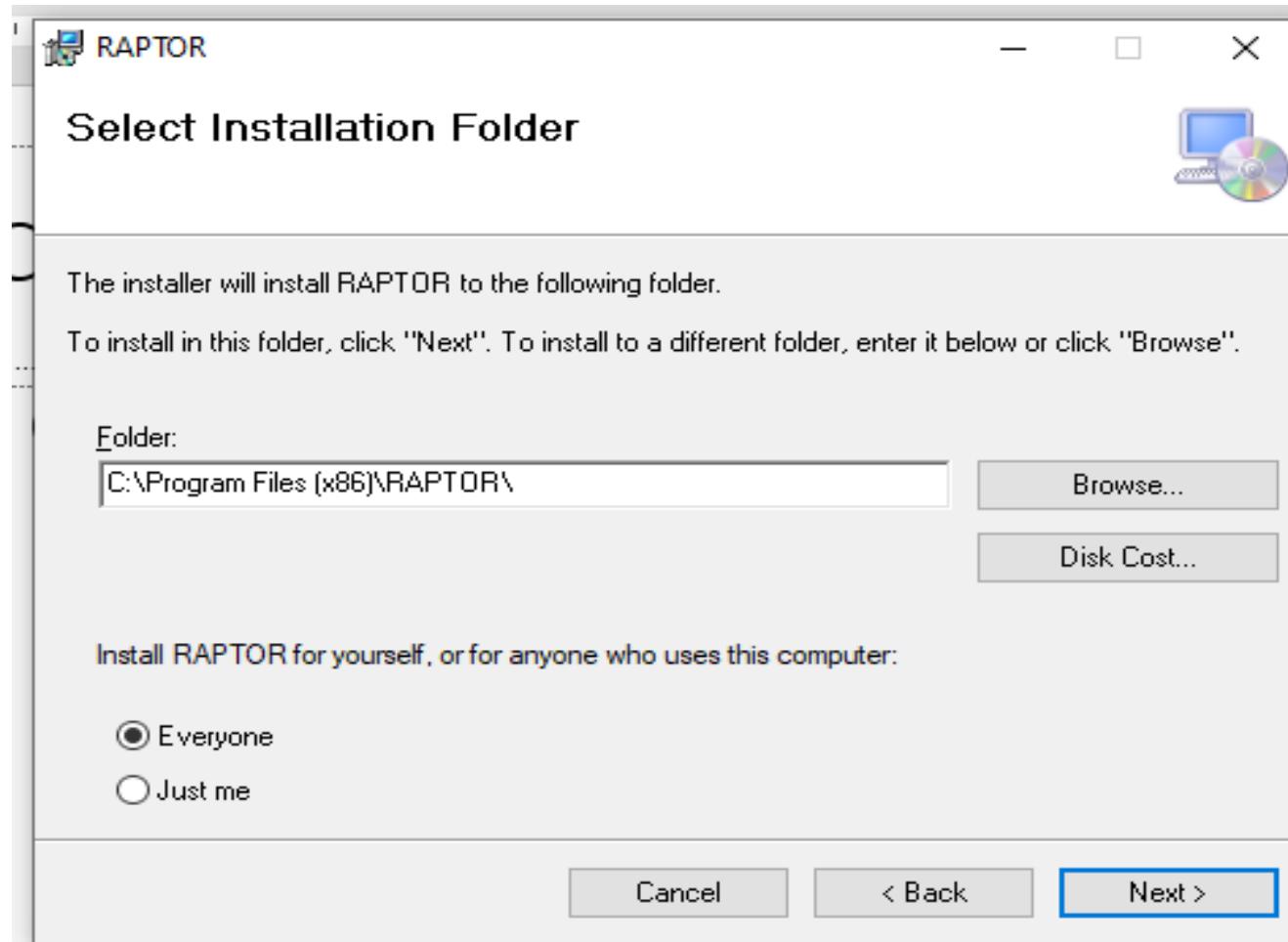
Download latest version

DOWNLOAD RAPTOR HERE!!

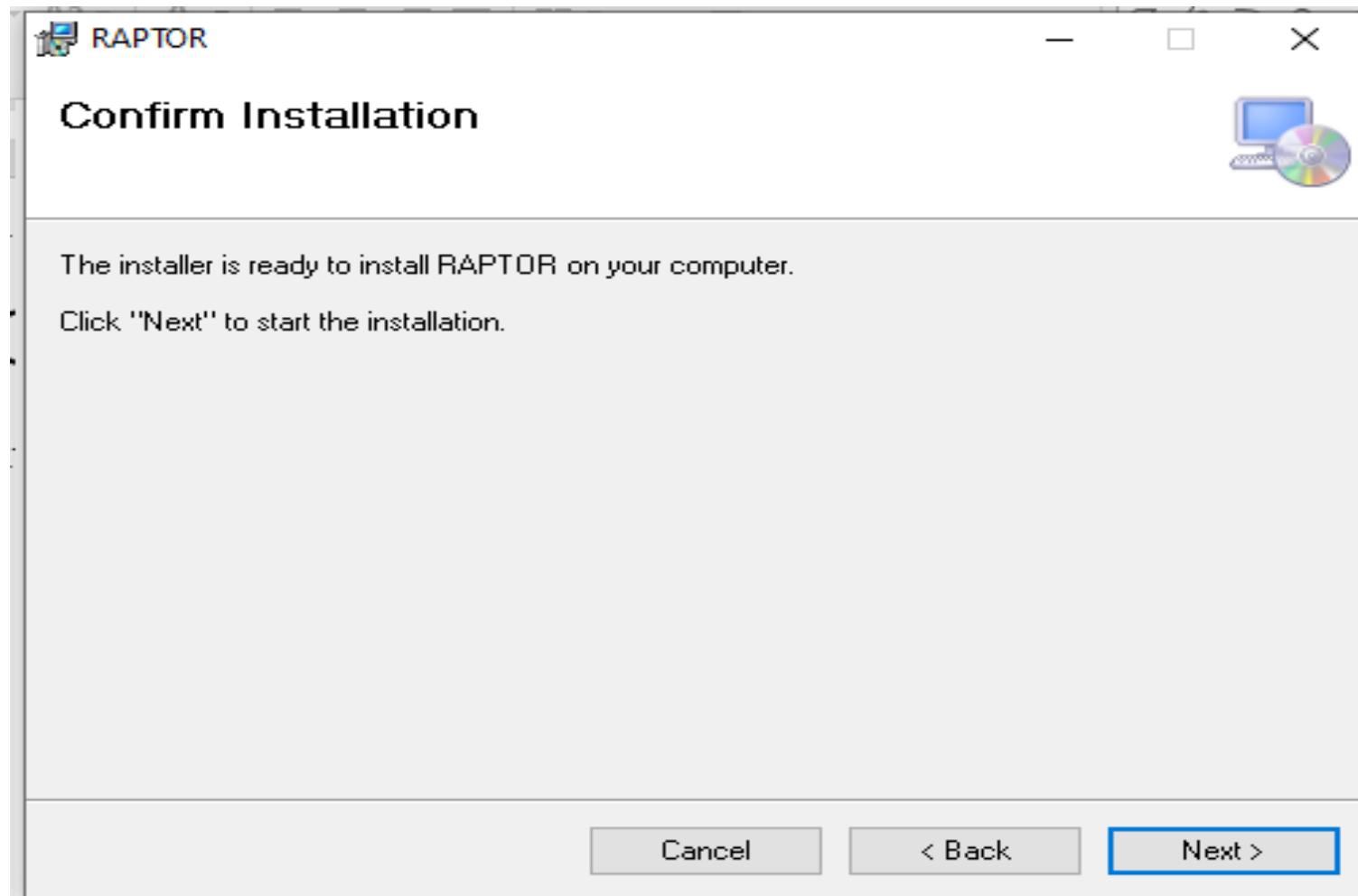
Installation process!!!



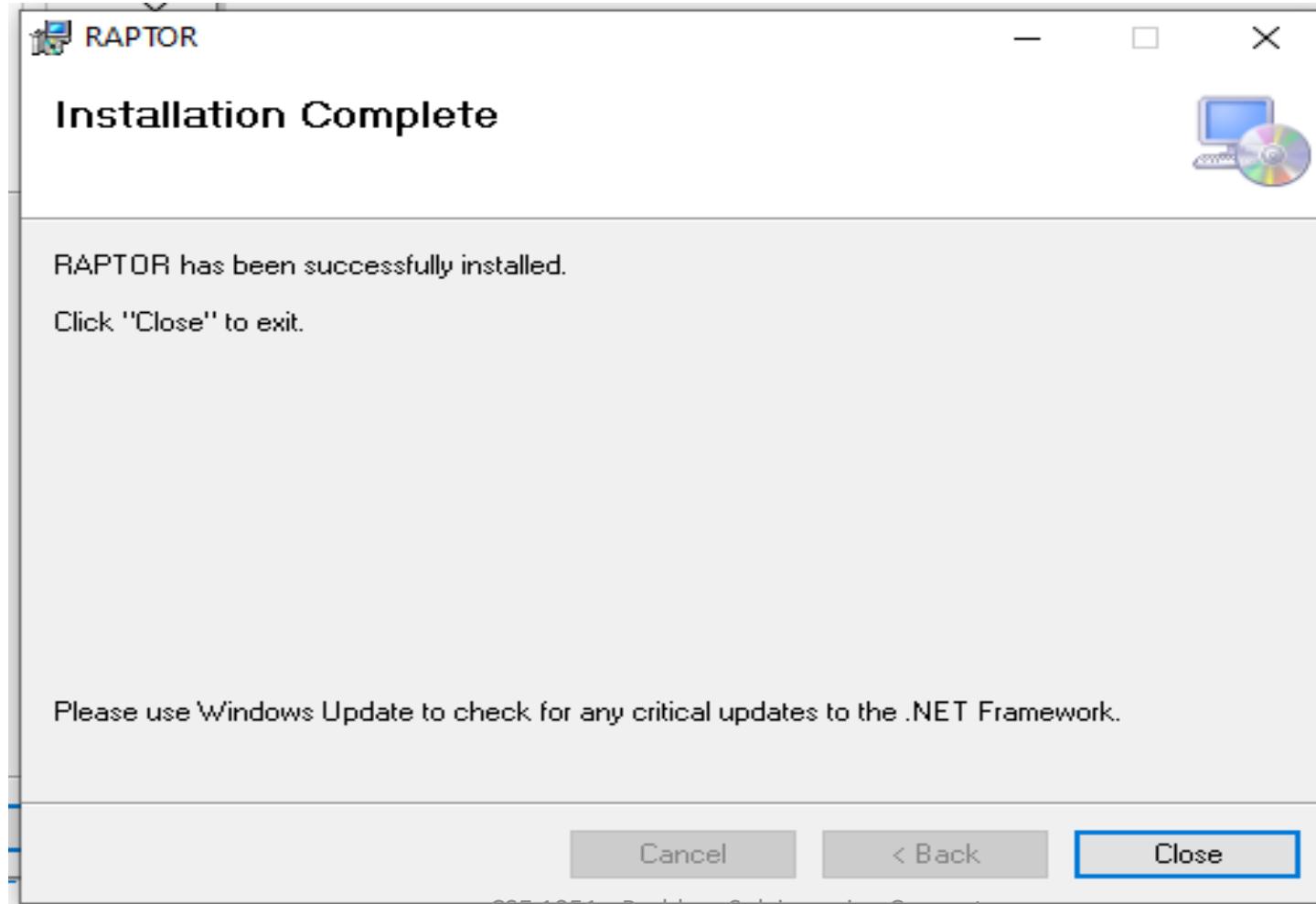
Installation process!!!



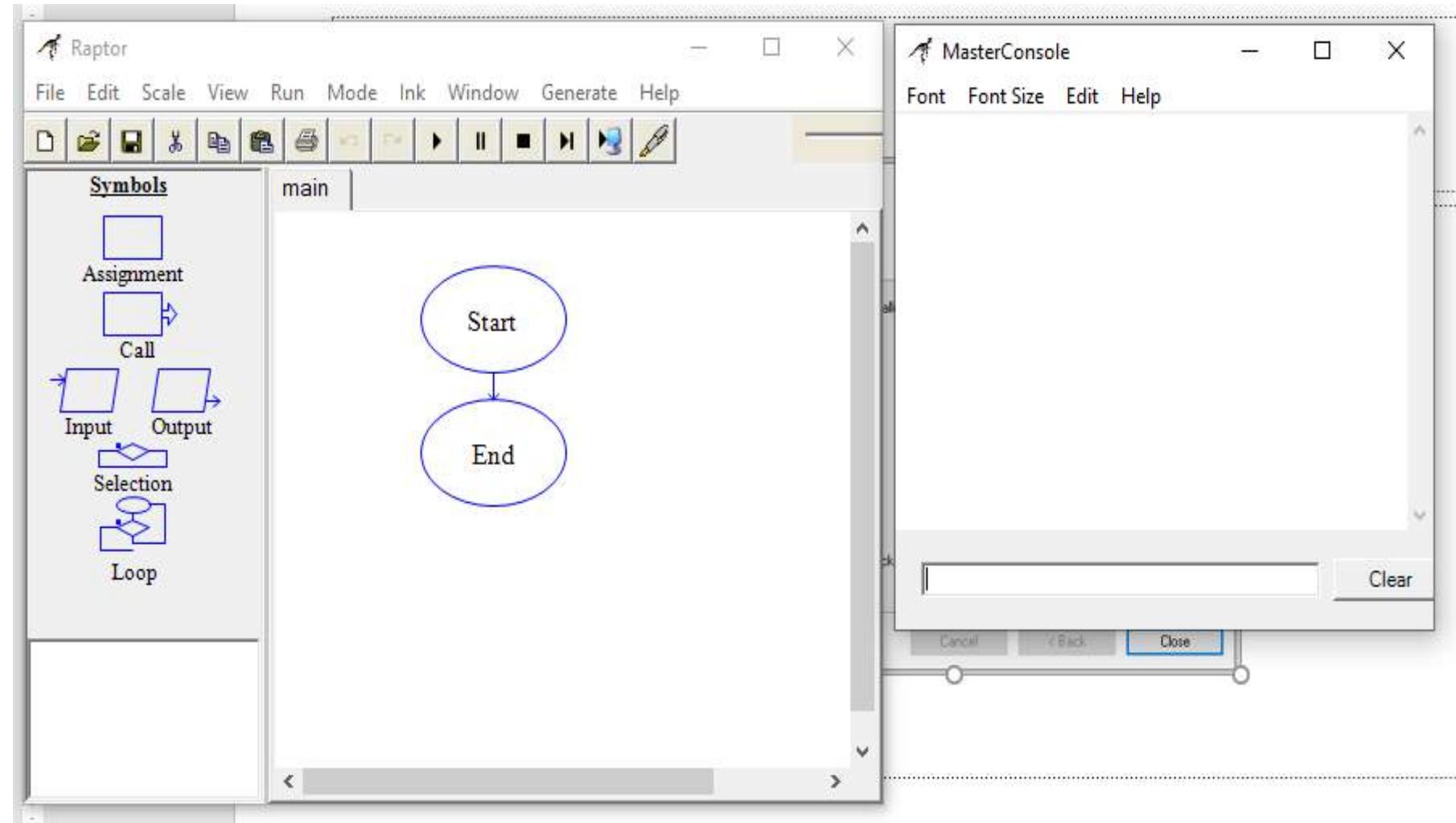
Installation process!!!



Installation complete!!



When u click on Raptor application!!!





Raptor controls!!!



Assignment



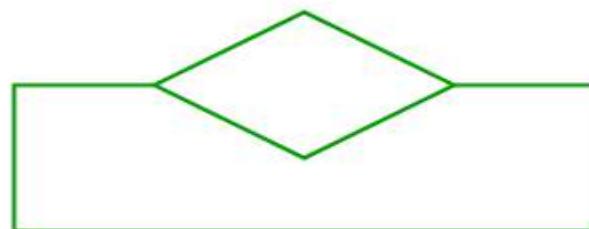
Call



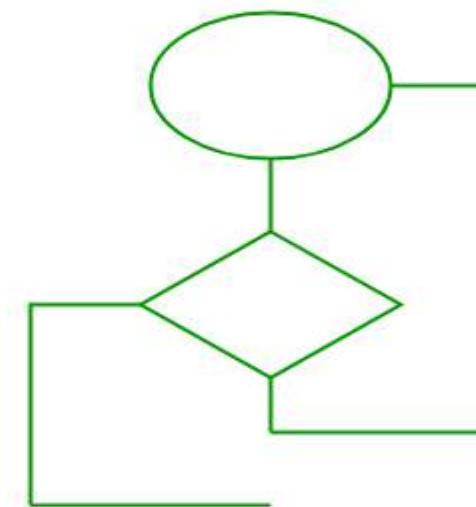
Input



Output



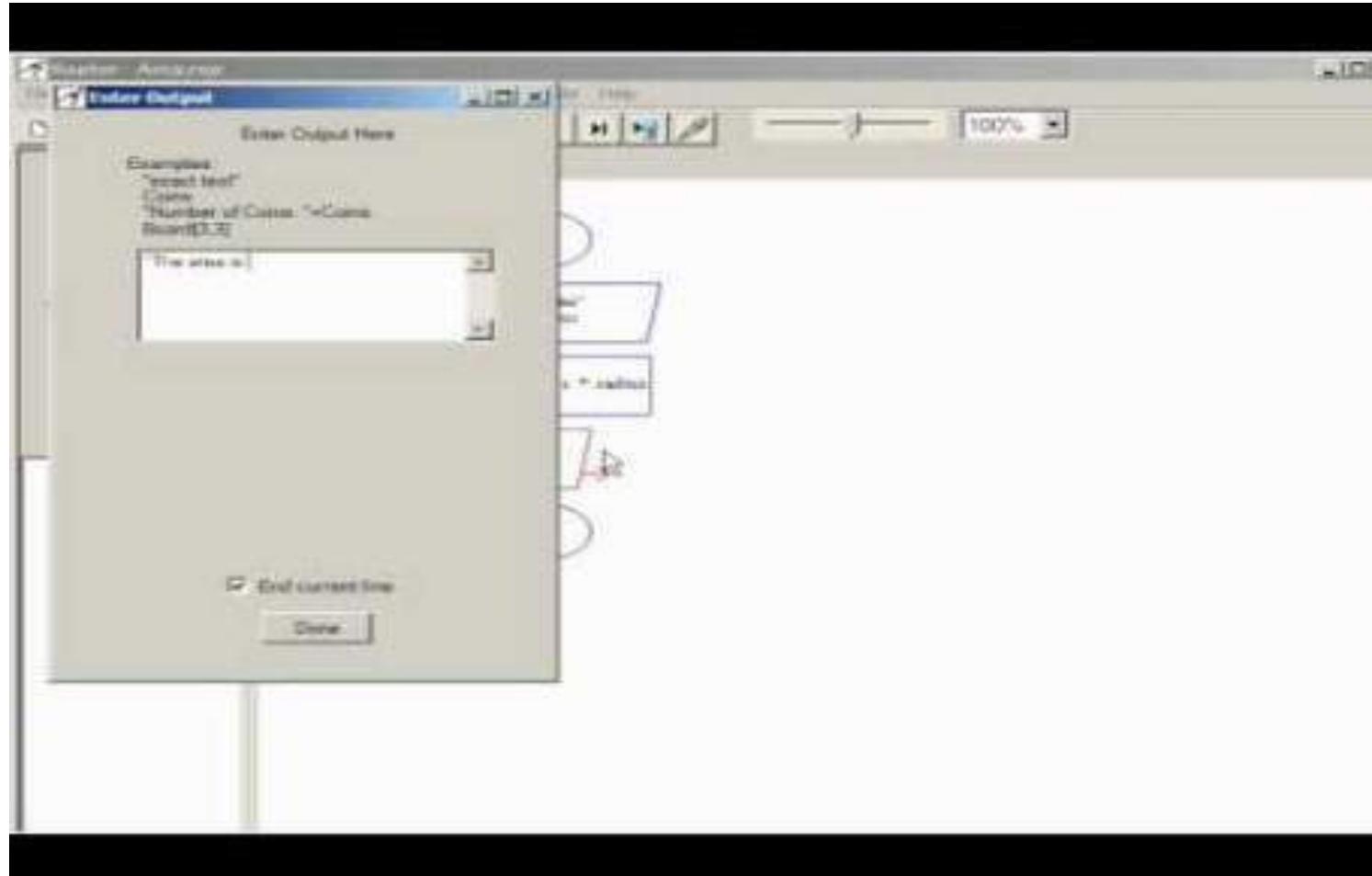
Selection



Loop



Video tutorial for RAPTOR!!!



<https://www.youtube.com/watch?v=ZcAALK3movs>

Session 2 Summary

- ✓ Introduction to algorithms
- ✓ Algorithms for simple problems
- ✓ Introduction to flowcharts
- ✓ Installation of RAPTOR tool

S3_1

Flowchart - Tutorial



Learning objectives!!!

To learn and appreciate the following concepts

- ✓ Draw flowcharts for simple problems
- ✓ Run and check output in RAPTOR tool



Area of the circle – Algorithm to Flowchart

Name of the algorithm: Compute the area of a circle

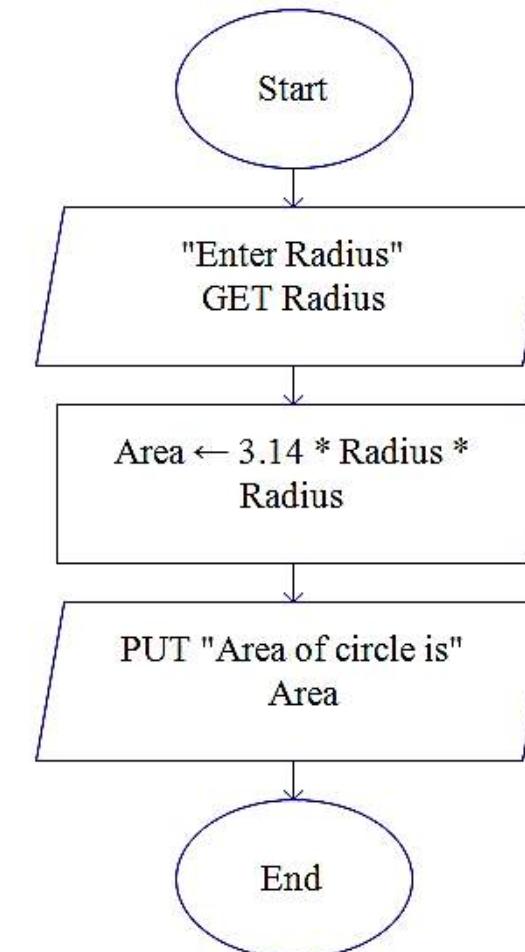
Step1: Input radius

Step 2: [Compute the area]
 $\text{Area} \leftarrow 3.1416 * \text{radius} * \text{radius}$

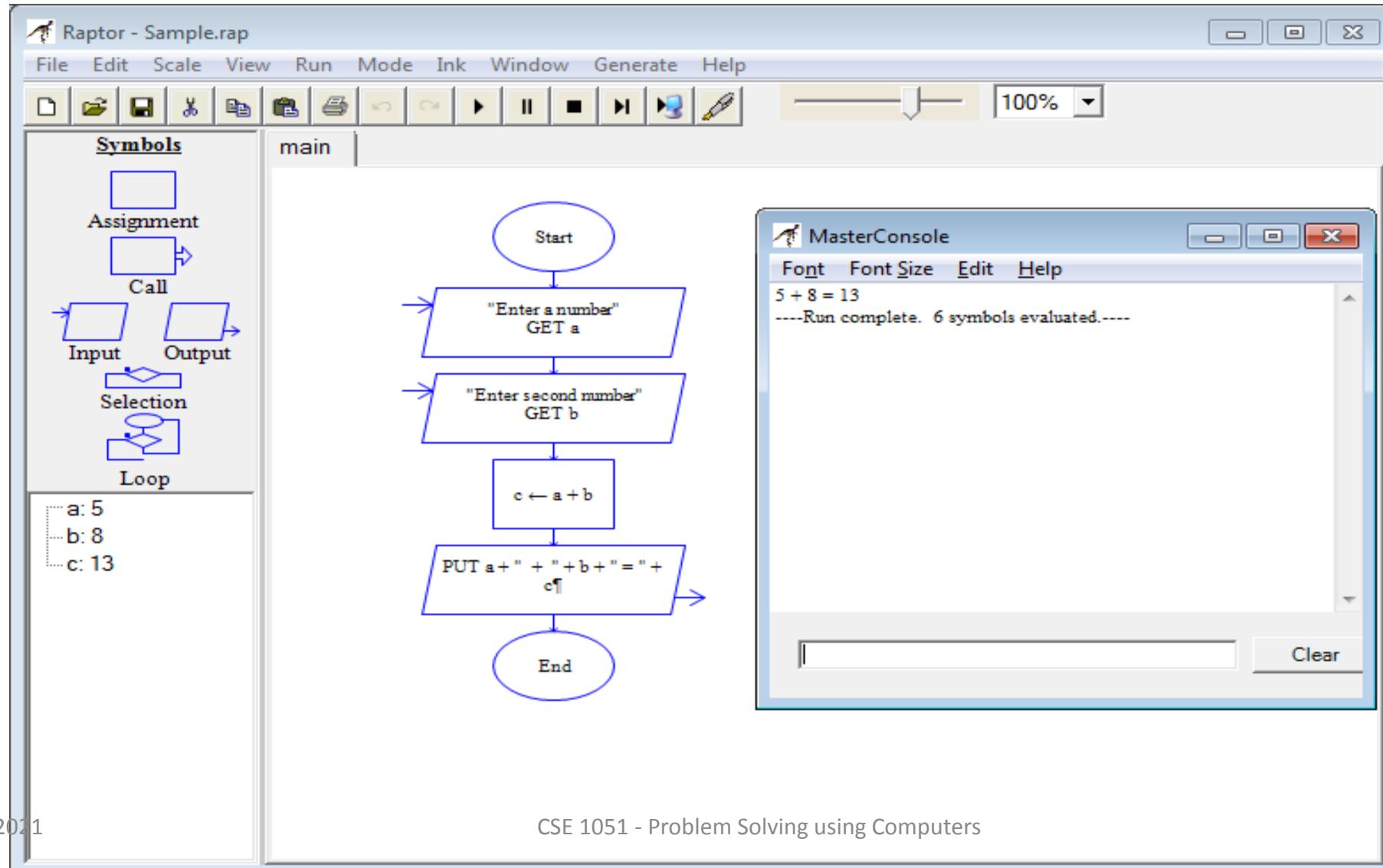
Step 3: [Print the Area]
Print 'Area of a circle =', Area

Step 4: [End of algorithm]
Stop

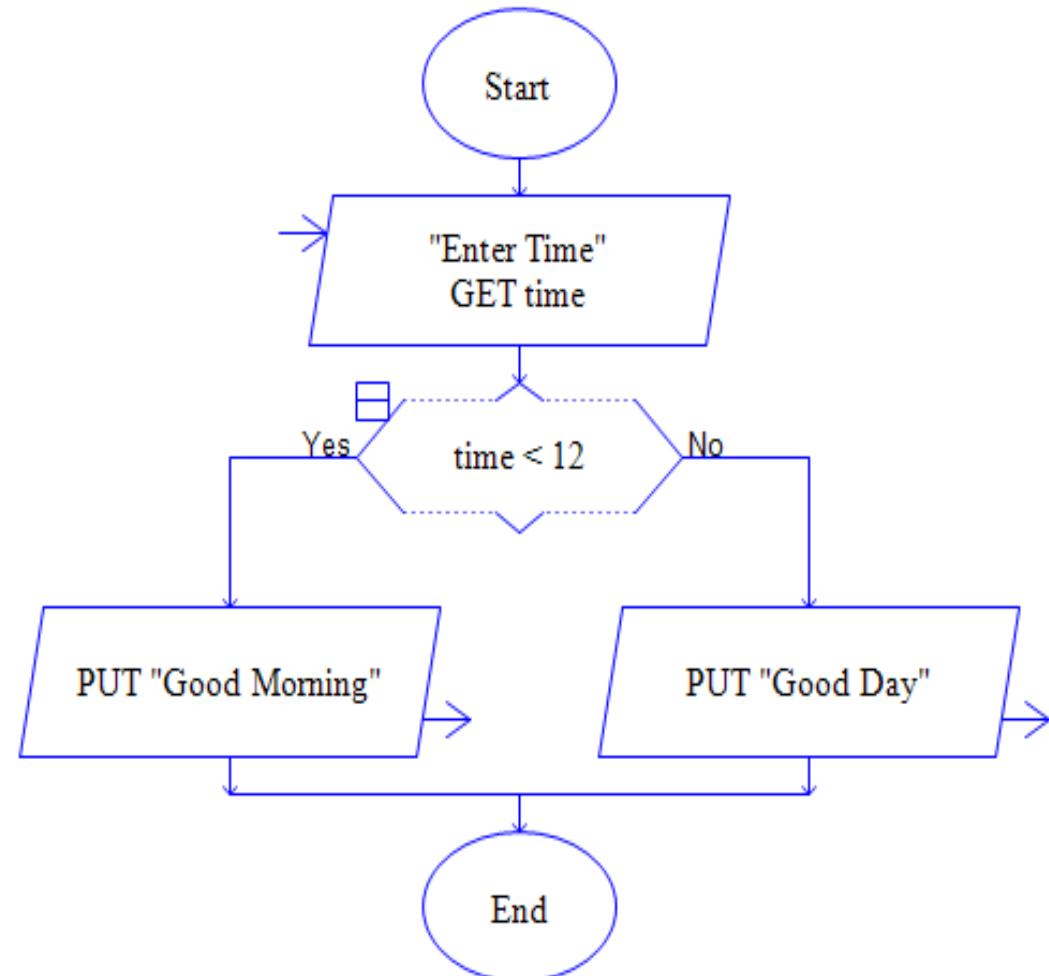
Flowchart



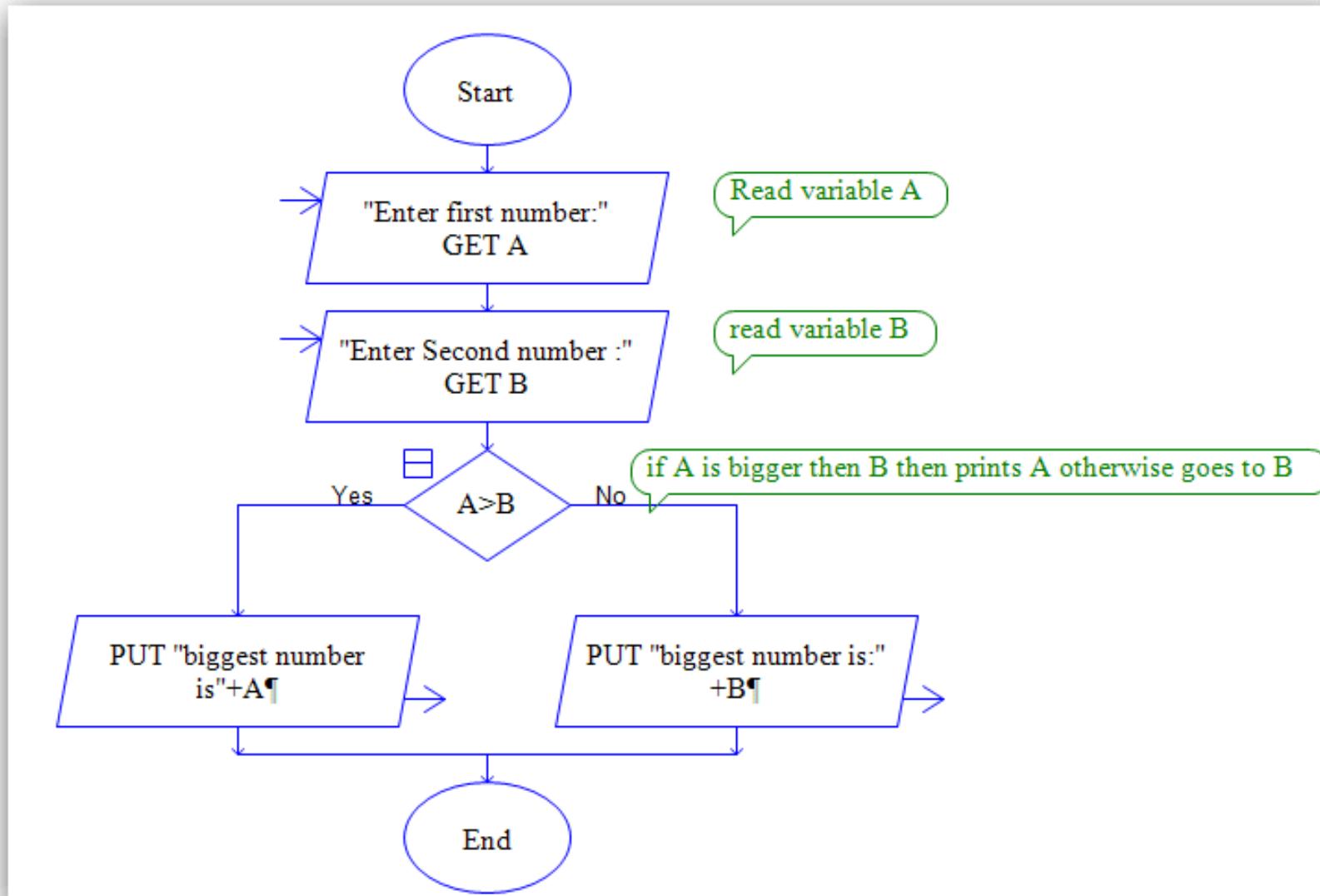
Let's add two numbers!! – Check output in console



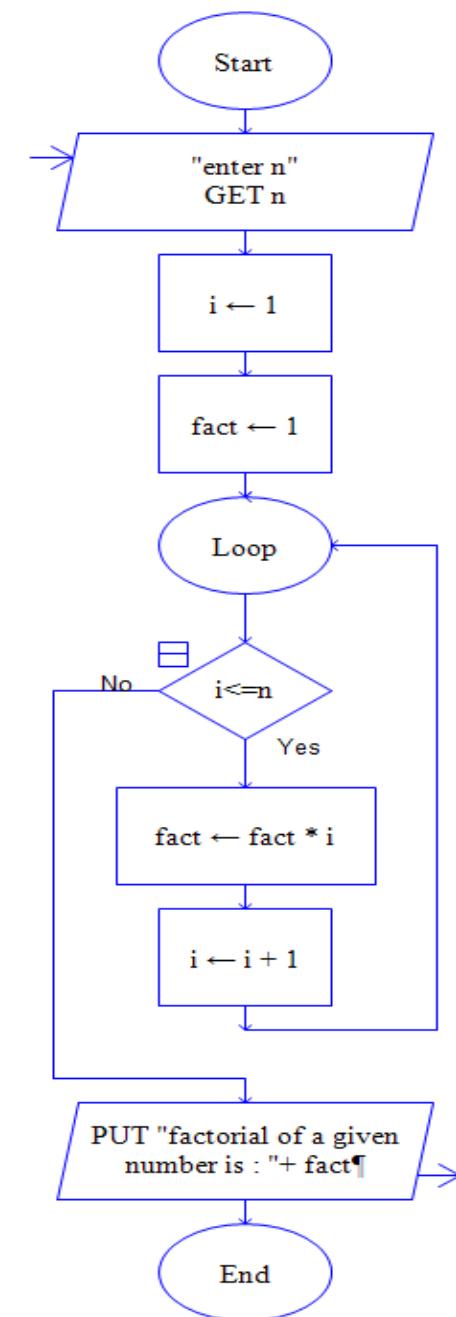
Learn to use selection control!!!



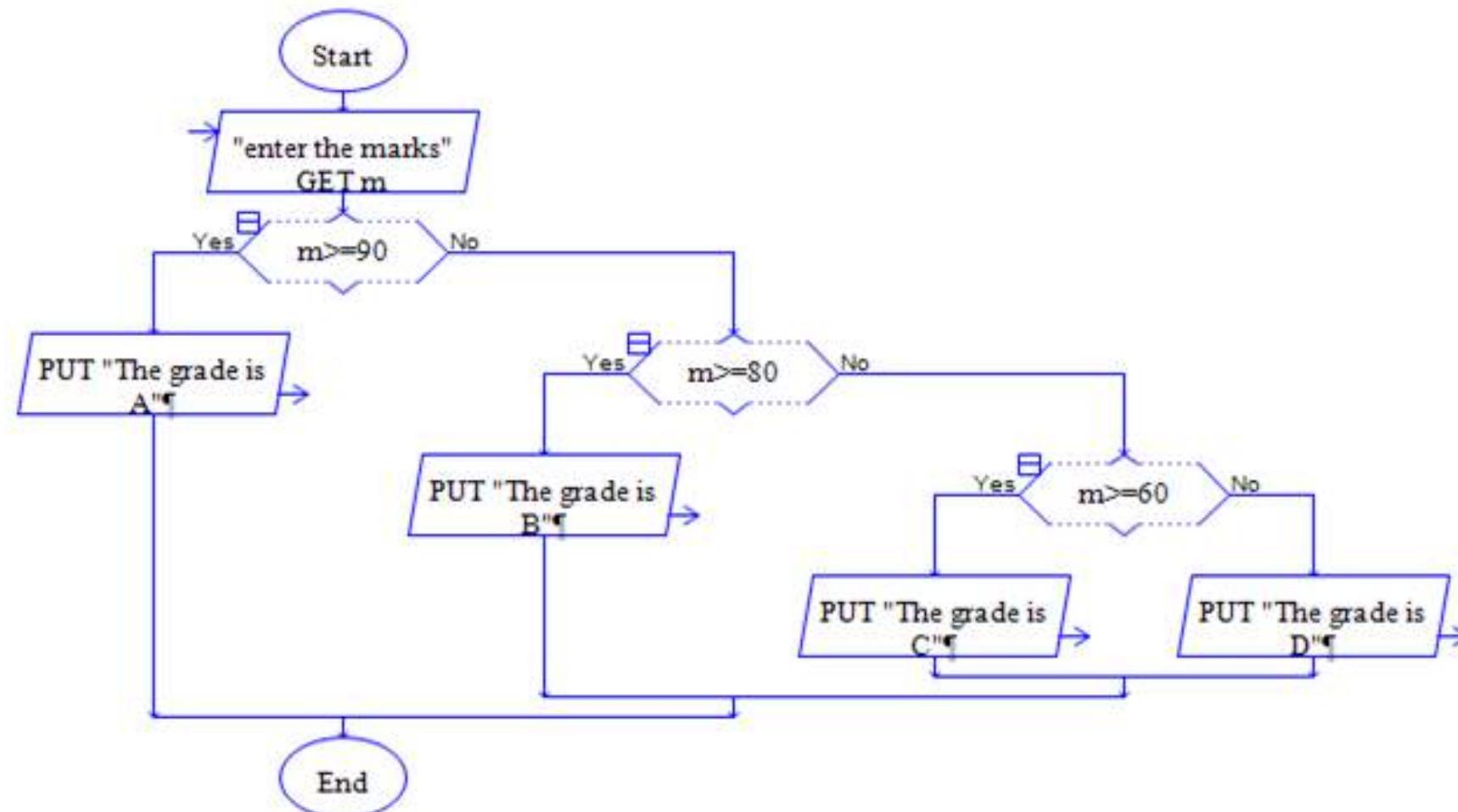
Largest of two numbers!!!



Factorial of a number!!!



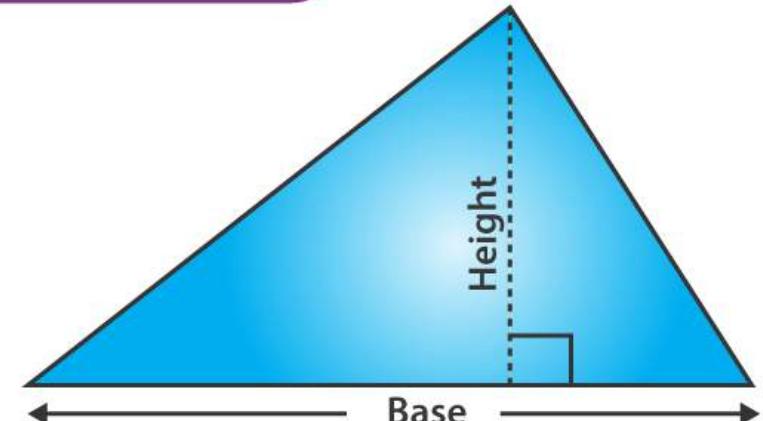
Flowchart for mark and grades!!!



Tutorial

- Realize a flowchart to find the area of triangle when three sides are given using RAPTOR tool
- Realize a flowchart to check whether given integer is positive or negative using RAPTOR tool

AREA OF TRIANGLE



$$\text{Area} = \frac{1}{2} \times \text{base} \times \text{perpendicular height}$$



s3_2

Variables and constants

Learning objectives!!!

To learn and appreciate the following concepts

- ✓ C Tokens
- ✓ C Character set
- ✓ Variables – declaration and initialization

Learning Outcomes

At the end of session the student will be able to

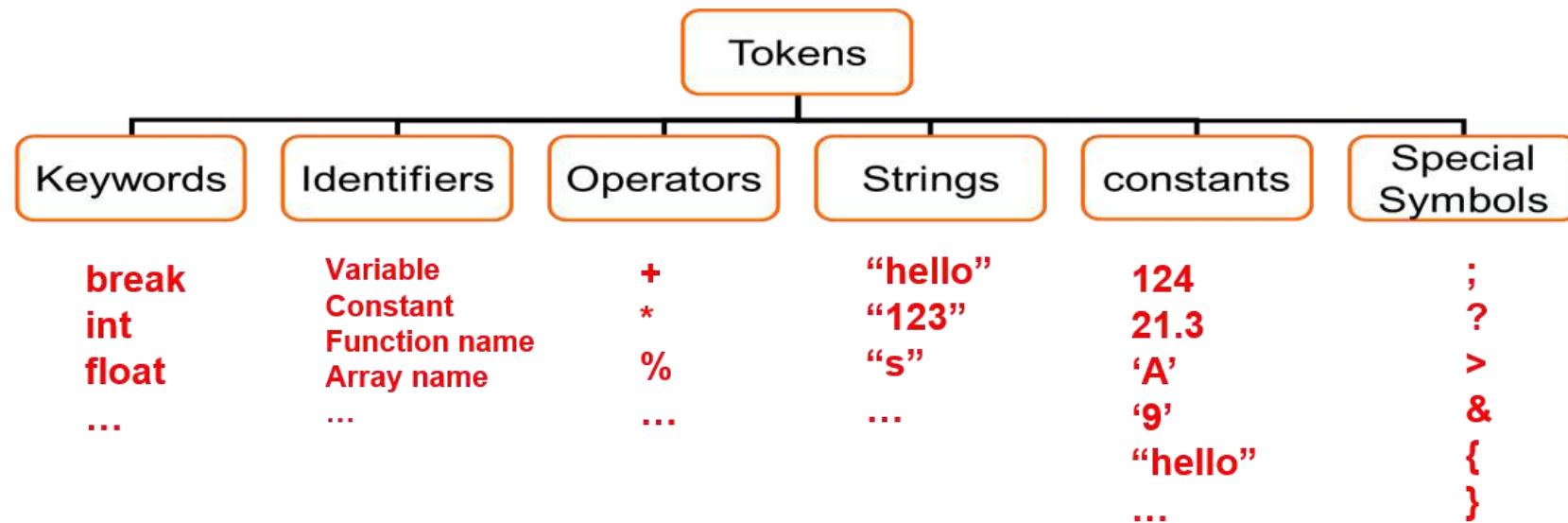
- ✓ Understand the C Tokens
- ✓ C Character set
- ✓ How to declare and initialize the variables

C Character set

- Character set is a set of valid characters that a language can recognize.
 - C character set consists of **letters, digits, special characters, white spaces**.
- (i) Letters → 'a', 'b', 'c',.....'z' Or 'A', 'B', 'C',.....'Z'**
- (ii) Digits → 0, 1, 2,.....9**
- (iii) Special characters → ;, ?, >, <, &, {, }, [,].....**
- (iv) White spaces → New line (\n), Tab(\t), Vertical Tab(\v) etc.**

C Tokens

- ✓ A token is a group of characters that logically belong together.
- ✓ The programmer writes a program by using tokens.
- ✓ C uses the following types of tokens.



Keywords

- These are some reserved words in C which have predefined meaning to compiler called keywords.
- Keywords are not to be used as variable and constant names.
- All keywords have fixed meanings and these meanings cannot be changed.

Compiler specific keywords

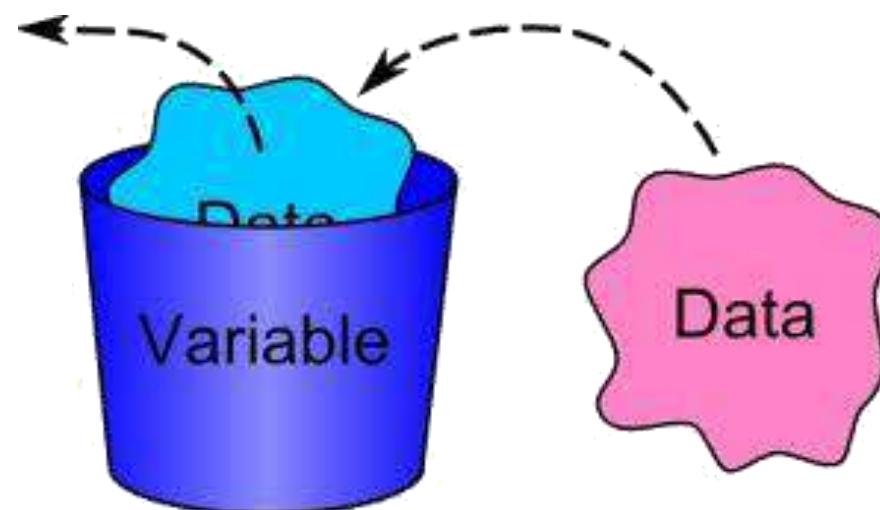
- Some commonly used keywords are given below:

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while



Variables

Variables are data storage locations in the computer's memory.



Variables

- Variables are the symbolic names for storing computational data.
- Variable: a *symbolic name* for a memory location
- In C variables have to be *declared* before they are used
- A variable may take different values at different times during execution.
- *Declarations* reserve storage for the variable.
- Value is assigned to the variable by *initialization* or *assignment*

Variable declarations

Data type

Variable name

**Which data types
are possible in C ?**

**Which variable names
are allowed in C ?**

Ex: int x

Variable Names- Identifiers

- Symbolic names can be used in C for various data items used by a programmer.
- A symbolic name is generally known as an identifier. An identifier is a name for a variable, constant, function, etc.
- The identifier is a sequence of characters taken from C character set.

Variable names

Rules for valid variable names (*identifiers*) :

- Name must begin with a **letter** or **underscore (_)** and can be followed by any combination of letters, underscores, or digits.
- **Key words** cannot be used as a variable name.
- C is **case-sensitive**: sum, Sum, and SUM each refer to a different variable.
- Variable names can be as long as you want, although only the first 63 (or 31) characters might be significant.
- Choice of meaningful variable names can increase the readability of a program



Variable names

- Examples of *valid* variable names:

- 1) **Sum**
- 2) **_difference**
- 3) **a**
- 4) **J5x7**
- 5) **Number_of_moves**

- Examples of *invalid* variable names:

- 1) **sum\$value**
- 2) **3val**
- 3) **int**

Declaring variables

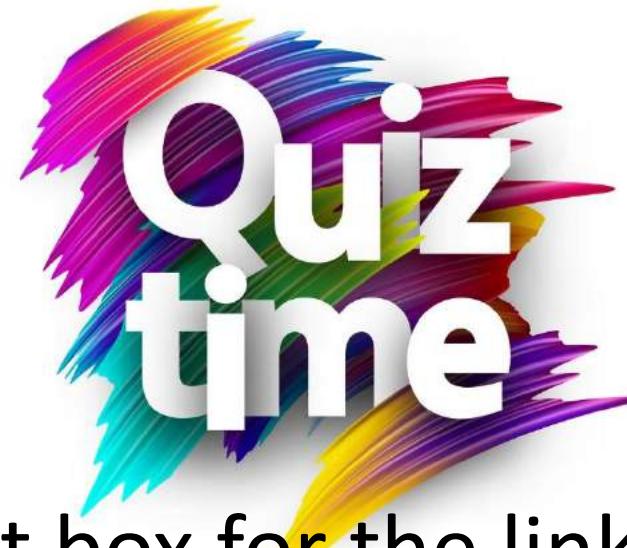
- C imposes to declare variables before their usage.
- Advantages of variable declarations:
 - Putting all the variables in one place makes it easier for a reader to understand the program.
 - Thinking about which variables to declare encourages the programmer to do some planning before writing a program.
 - The obligation to declare all variables helps prevent bugs of misspelled variable names.
 - Compiler knows the amount of memory needed for storing the variable.
 - Compiler can verify that operations done on a variable are allowed by its type.



Variables (Video)

The image is a video thumbnail with a dark background. At the top center is a white icon of a computer monitor displaying a blue letter 'C'. Below this, the title 'PROGRAMMING AND DATA STRUCTURES' is written in large, white, hand-drawn style letters. 'PROGRAMMING AND' is above a yellow rectangular box containing the word 'DATA', and 'DATA' is above 'STRUCTURES'. A small white rocket ship icon is positioned to the left of the word 'INTRODUCTION'. An arrow points from the word 'STRUCTURES' down towards the word 'INTRODUCTION'. Below the title, the words 'INTRODUCTION TO VARIABLES' are written in pink. In the bottom left corner, the text 'NESO ACADEMY' is visible.

<https://youtu.be/fO4FwJOSHdc>



Go to posts/chat box for the link to the question

submit your solution in next 2 minutes

The session will resume in 3 minutes

Variables – summary

- Variables are named memory locations.
- Helps us to store data and retrieve data
- In C language , some rules are to be followed while declaring the variables and while naming them.
- It is better to use the variable names which are meaningful and helps the user to know what data might have been stored in that variable

Session 3 Summary

- ✓ Draw the flowcharts for simple problems
- ✓ Use the RAPTOR tool to write, run and check the output of flowchart
- ✓ C Tokens and Character set
- ✓ C Variables
- ✓ Variable declaration and initialization

Data-types in C

Objectives of this session

- To learn about basic data types in C
- To learn how use declare data types in C

Learning outcomes

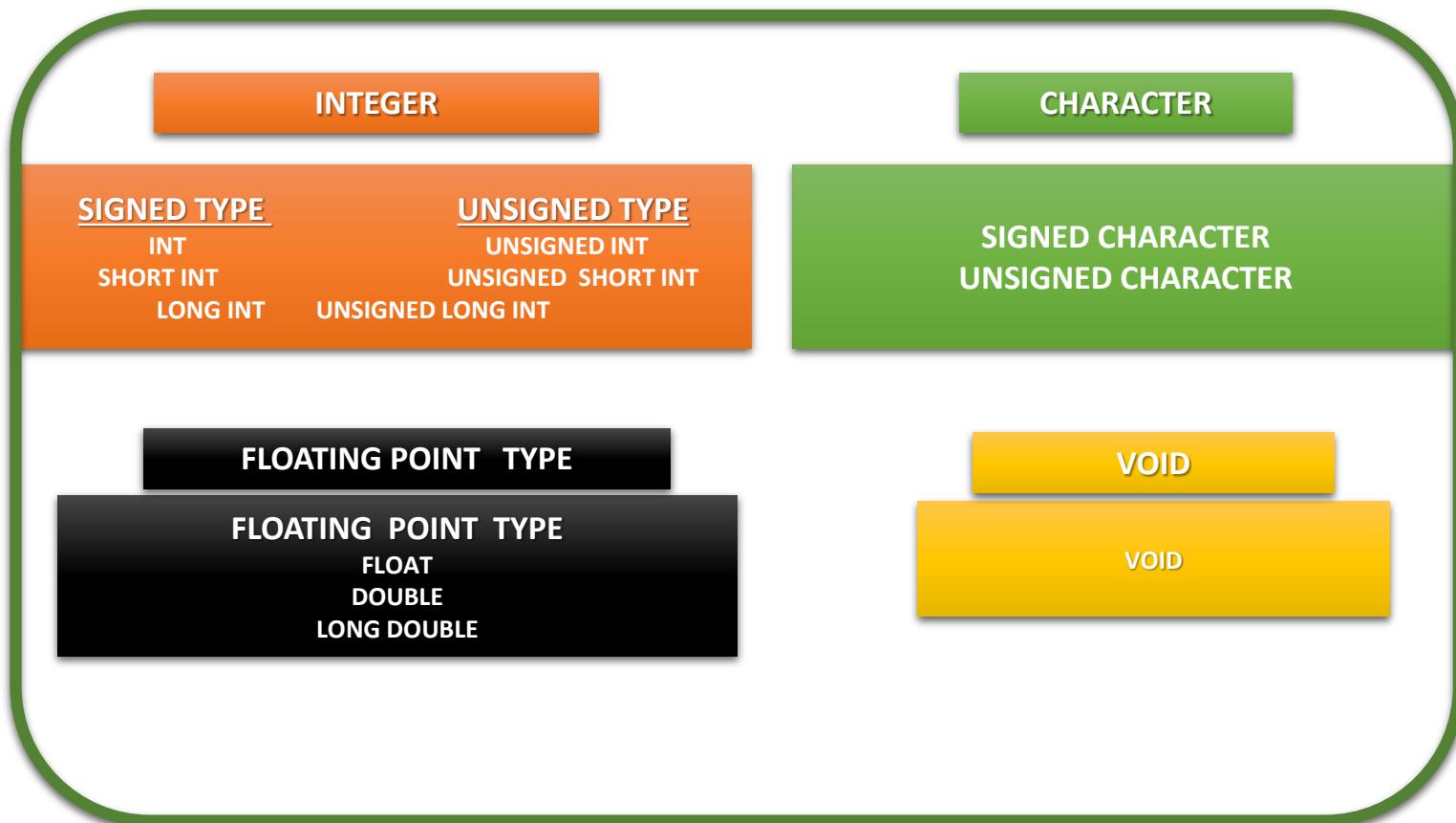
- At the end of this session you will learn
 - About basic data types in C
 - How to declare the basic data types

Big Picture

- Processor works with finite-sized data
- All data implemented as a sequence of *bits*
 - Bit = 0 or 1
 - Represents the level of an electrical charge
- *Byte* = 8 bits
- *Word* = largest data size handled by processor
 - 32 bits on most older computers
 - 64 bits on most new computers



Primary (built-in or Basic) Data types



Data types

Basic data types: int, float, double, char, and void.

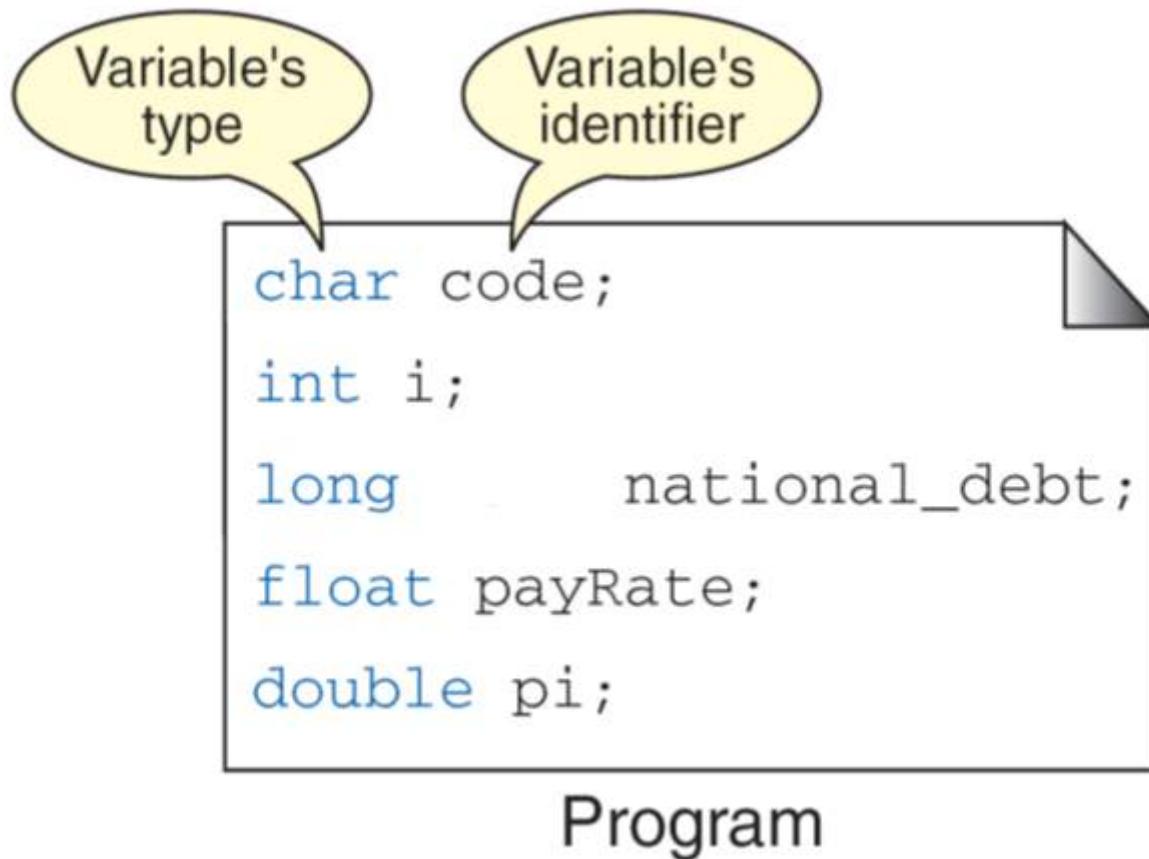
- ✓ **int:** can be used to store integer numbers (values with no decimal places).
- ✓ **float:** can be used for storing floating-point numbers (values containing decimal places).
- ✓ **double:** the same as type float, and roughly twice the size of float.
- ✓ **char:** can be used to store a single character, such as the letter *a*, the digit character *6*, or a semicolon.
- ✓ **void:** is used to denote nothing or empty.

Variables - Examples

```
int a;           // declaring a variable of type int  
  
int sum, a1, a2; // declaring 3 variables  
  
int x = 7;      // declaring and initializing a variable  
  
a = 5;          // assigning to variable a the value 5  
  
a1 = a;         // assigning to variable a1 the value of a  
L-value    R-value
```

```
a1=a1+1; // assigning to variable a1 the value of a1+1  
// (increasing value of a1 with 1)
```

Variables -Example



Integer Types

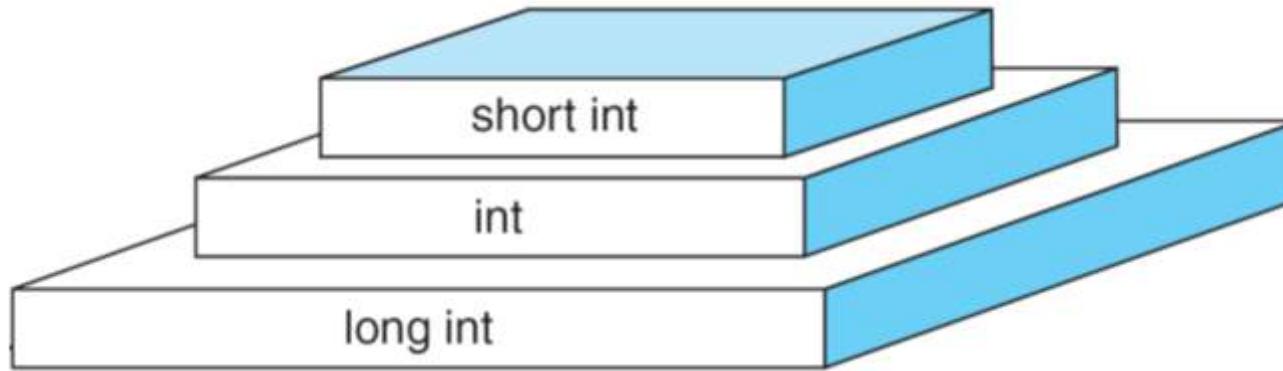
- The basic integer type is **int**
 - The size of an **int** depends on the machine and on PCs it is normally 16 or 32 or 64 bits.
- **modifiers (type specifiers)**
 - **short:** typically uses less bits
 - **long:** typically uses more bits
 - **Signed:** both negative and positive numbers
 - **Unsigned:** only positive numbers

SIZE AND RANGE OF VALUES FOR A 16-BIT MACHINE (INTEGER TYPE)

	Type	Size	Range
short	short int or signed short int	8	-128 to 127
	unsigned int	8	0 to 255
Integer	int or signed int	16	-32,768 to 32,767
	unsigned int	16	0 to 65,535
Long	long int or signed long int	32	-2,147,483,648 to 2,147,483,647
	unsigned long int	32	0 to 4,294,967,295

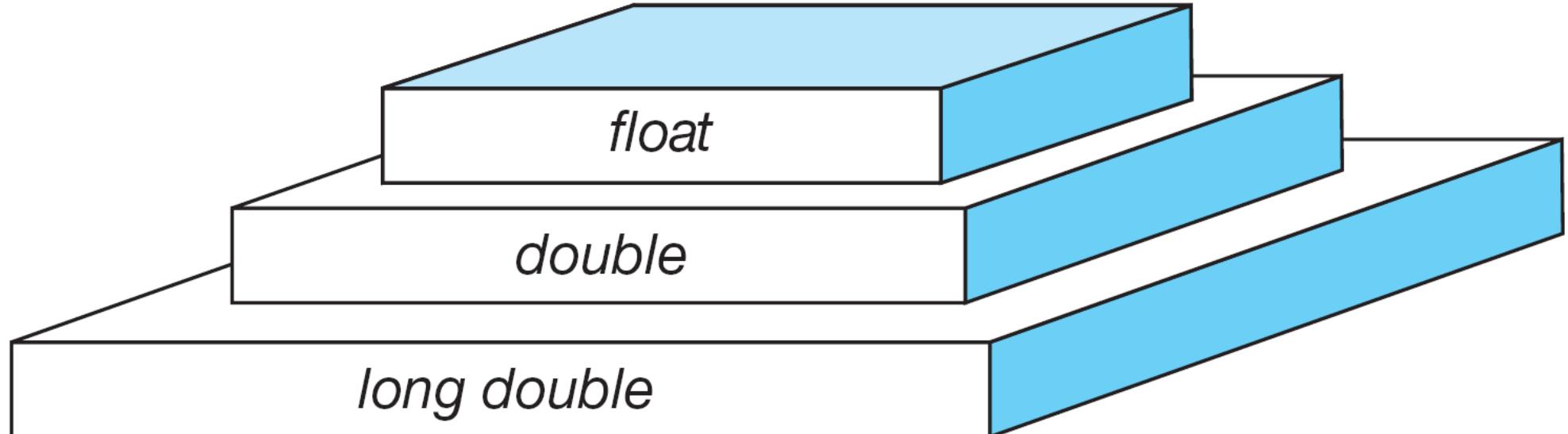
Data- Types in C

Integer Data type



Data- Types in C

Float data type



Summary

- The basic data types in C are
 - int
 - float
 - char
 - Double
- Size of the data types in machine dependent
- If we want a big integer to be stored , we can make use of keyword long[long int]
- Similarly, if we want higher precision to be achieved from a computation involving real numbers , then use long double



Go to posts/chat box for the link to the question

submit your solution in next 2 minutes

The session will resume in 3 minutes

Data Types in C-Part-2

Objectives of this session

- To learn about basic data types in c
- How to declare them in program
- Different operators in C

Learning outcomes

- At the end of this session, you will understand
 - About different types of basic data types available in C
 - How to declare them in a C program
 - Different types of operators available in C

The character type `char`

- A **char variable** can be used to store a single character.
- A **character constant** is formed by enclosing the character within a pair of single quotation marks. Valid examples: '`a`' .
- Character zero ('`0`') is not the same as the number (integer constant) `0`.
- The character constant '`\n`'—the newline character—is a valid character constant. It is called as an **escape character**.
- There are other ***escape sequences*** like, `\t` for tab, `\v` for vertical tab, `\n` for new line etc.

Character Types

- Character type **char** is related to the integer type.
- Modifiers(type specifiers) ***unsigned*** and ***signed*** can be used
 - **char → 1 byte (-128 to 127)**
 - **signed char → 1 byte (-128 to 127)**
 - **unsigned char → 1 byte (0 to 255)**
- ASCII (American Standard Code for Information Interchange) is the dominant encoding scheme for characters.
 - Examples
 - ✓ '' encoded as 32 '+' encoded as 43
 - ✓ 'A' encoded as 65 'Z' encoded as 90
 - ✓ 'a' encoded as 97 'z' encoded as 122
 - ✓ '0' encoded as 48 '9' encoded as 57

Assigning values to char

```
char letter; /* declare variable letter of type char */  
  
letter = 'A'; /* OK */  
letter = A; /* NO! Compiler thinks A is a variable */  
letter = "A"; /* NO! Compiler thinks "A" is a string */  
letter = 65; /* ok because characters are internally stored  
as numeric values (ASCII code) */
```

Floating-Point Types

- Floating-point types represent real numbers
 - Integer part
 - Fractional part
- The number 108.1517 breaks down into the following parts
 - 108 - integer part
 - 1517 - fractional part
- Floating-point constants can also be expressed in *scientific notation*. The value **1.7e4** represents the value **1.7×10^4** .

The value before the letter e is known as the *mantissa*, whereas the value that follows e is called the *exponent*.

- There are three floating-point type specifiers
 - float
 - double
 - long double

SIZE AND RANGE OF VALUES FOR 16-BIT MACHINE (FLOATING POINT TYPE)

	Type	Size
Single Precision	Float	32 bits 4 bytes
Double Precision	double	64 bits 8 bytes
Long Double Precision	long double	80 bits 10 bytes

void

➤ **2 uses of void are**

- **To specify the return type of a function when it is not returning any value.**
- **To indicate an empty argument list to a function.**

Best Practices for Programming

Naming Variables According to Standards

Prefix	Data Type	Example
✓ i	int and unsigned int	iTotalMarks
✓ f	float	fAverageMarks
✓ d	double	dSalary
✓ l	long and unsigned long	lFactorial
✓ c	signed char and unsigned char	cChoice
✓ ai	Array of integers	aiStudentId
✓ af	Array of float	afQuantity
✓ ad	Array of double	adAmount
✓ al	Array of long integers	alSample
✓ ac	Array of characters	acEmpName

Example: Using data types

```
#include <stdio.h>
int main ()
{
    int integerVar = 100;
    float floatingVar = 331.79;
    double doubleVar = 144368.4411;
    char charVar = 'W';
    printf("%d\n", integerVar);
    printf("%f\n",floatingVar);
    printf("%g\n",doubleVar);
    printf("%c\n",charVar);
    return 0;
}
```

Operators

- The different operators are:
 - Arithmetic
 - Relational
 - Logical
 - Increment and Decrement
 - Bitwise
 - Assignment
 - Conditional

Summary

- Character data type (char) takes 1 byte(8-bits) in memory
- ASCII format is used to encode character data
- Floating point numbers (real numbers) can be stored in float, double or long double depending on the precision we want
- There are different types of operators available in c for different purpose

L6-L7 Operators

Learning Objectives

To learn and appreciate the following concepts

- Arithmetic Operators
- Relational and Logical Operators
- Type conversions
- Increment and Decrement Operators
- Bitwise Operators

Session outcome

- At the end of session student will be able to learn and understand
 - Arithmetic Operators
 - Relational and Logical Operators
 - Type conversions
 - Increment and Decrement Operators
 - Bitwise Operators

Operators

- The different operators are:
 - Arithmetic
 - Relational
 - Logical
 - Increment and Decrement
 - Bitwise
 - Assignment
 - Conditional

Arithmetic Operators

- The binary arithmetic operators are +, -, *, / and the modulus operator %.
- The / operator when used with integers truncates any fractional part i.e. E.g. $5/2 = 2$ and not 2.5
- Therefore % operator produces the remainder when 5 is divided by 2 i.e. 1
- The % operator cannot be applied to float or double
- E.g. $x \% y$ wherein % is the operator and x, y are operands

The unary minus operator

```
#include <stdio.h>
int main ()
{
    int a = 25;
    int b = -2;
    printf("%d\n", -a);
    printf("%d\n", -b);
    return 0;
}
```

Working with arithmetic expressions

- Basic arithmetic operators: +, -, *, /, %
- **Precedence:** One operator can have a higher priority, or *precedence*, over another operator. The operators within C are grouped hierarchically according to their *precedence* (i.e., order of evaluation)
 - Operations with a higher precedence are carried out before operations having a lower precedence.

High priority operators * / % ...

Low priority operators + - ...

- Example: * has a higher precedence than +
 $a + b * c \rightarrow a + (b * c)$
- If necessary, you can always use parentheses in an expression to force the terms to be evaluated in any desired order.
- **Associativity:** Expressions containing operators of the same precedence are evaluated either from left to right or from right to left, depending on the operator. This is known as the *associative* property of an operator
 - Example: + has a *left to right* associativity

For both the precedence group described above, *associativity is “left to right”*.

Working with arithmetic expressions

```
#include <stdio.h>
int main ()
{
    int a = 100;
    int b = 2;
    int c = 25;
    int d = 4;
    int result;
    result = a * b + c * d;    //Precedence
    printf ("%d\n", result);
    result = a * (b + c * d); //Associativity
    printf ("%d\n", result);
    return 0;
}
```

Relational operators

Operator	Meaning
<code>==</code>	Is equal to
<code>!=</code>	Is not equal to
<code><</code>	Is less than
<code><=</code>	Is less or equal
<code>></code>	Is greater than
<code>>=</code>	Is greater or equal

The relational operators have lower precedence than all arithmetic operators:
 $a < b + c$ is evaluated as $a < (b + c)$

ATTENTION !

the “is equal to” operator `==` and the “assignment” operator `=`

Relational operators

- An expression such as $a < b$ containing a relational operator is called a *relational expression*.
- The value of a relational expression is one, if the specified relation is true and zero if the relation is false.

E.g.:

$10 < 20$ is TRUE
 $20 < 10$ is FALSE

- A simple relational expression contains only one relational operator and takes the following form.

$ae1 \text{ relational operator } ae2$

$ae1$ & $ae2$ are arithmetic expressions, which may be simple constants, variables or combinations of them.

Relational operators

The arithmetic expressions will be evaluated first & then the results will be compared. That is, arithmetic operators have a higher priority over relational operators. $>$ \geq $<$ \leq all have the same precedence and below them are the next precedence equality operators i.e. $==$ and $!=$

Suppose that i, j and k are integer variables whose values are 1, 2 and 3 respectively.

<u>Expression</u>	<u>Interpretation</u>	<u>Value</u>
$i < j$	true	1
$(i+j) \geq k$	true	1
$(j+k) > (i+5)$	false	0
$k != 3$	false	0
$j == 2$	true	1

Logical operators

Truth Table

op-1	op-2	value of expression	
		op-1 && op-2	op-1 op-2
Non-zero	Non-zero	1	1
Non-zero	0	0	1
0	Non-zero	0	1
0	0	0	0

Operator	Symbol	Example
AND	&&	expression1 && expression2
OR		expression1 expression2
NOT	!	!expression1

The result of logical operators is always either 0 (FALSE) or 1 (TRUE)

Logical operators

Expressions	Evaluates As
(5 == 5) && (6 != 2)	True (1) because both operands are true
(5 > 1) (6 < 1)	True (1) because one operand is true
(2 == 1) && (5 == 5)	False (0) because one operand is false
! (5 == 4)	True (1) because the operand is false
!(FALSE) = TRUE !(TRUE) = FALSE	

Increment and Decrement operators (++ and --)

- The operator ++ adds 1 to the operand.
- The operator -- subtracts 1 from the operand.
- Both are unary operators.
- Ex: ++i or i++ is equivalent to $i = i + 1$
- They behave differently when they are used in expressions on the R.H.S of an assignment statement.

Increment and Decrement operators

Ex:

m=5;

y=++m; Prefix Mode

In this case, the value of y and m would be 6.

m=5;

y=m++; Postfix Mode

Here y continues to be 5. Only m changes to 6.

Prefix operator ++ appears before the variable.

Postfix operator ++ appears after the variable.

Increment and Decrement operators

Don'ts:

Attempting to use the increment or decrement operator on an expression other than a modifiable variable name or reference.

Example:

$++(5)$ is a syntax error

$++(x + 1)$ is a syntax error

Bitwise Operators

- Bitwise Logical Operators
- Bitwise Shift Operators
- Ones Complement operator

Bitwise Logical operators

- **&(AND), |(OR), ^(EXOR)**
- These are *binary operators* and require two integer operands.
- These work on their operands bit by bit starting from LSB (rightmost bit).

op 1	op 2	&	 	^
1	1	1	1	0
1	0	0	1	1
0	1	0	1	1
0	0	0	0	0

Example

Suppose $x = 10$, $y = 15$

$z = x \& y$ sets $z=10$ like this

0000000000001010 ← x

0000000000001111 ← y

0000000000001010 ← $z = x \& y$

Same way $|$, $^{\wedge}$ according to the table are computed.

Bitwise Shift operators

- **<< , >>**
- These are used to move bit patterns either to the left or right.
- They are used in the following form
- **op<<n** or **op>>n** here op is the operand to be shifted and n is number of positions to shift.

Bitwise Shift operator: <<

- << causes all the bits in the operand op to be shifted to the left by n positions.
- The *leftmost* n bits in the original bit pattern will be lost and the *rightmost* n bits that are vacated are filled with 0's

Bitwise Shift operator: >>

- **>>** causes all the bits in operand op to be shifted to the right by n positions.
- The *rightmost* n bits will be lost and the left most vacated bits are filled with 0's if number is unsigned integer

Examples

- Suppose X is an unsigned integer whose bit pattern is 0000 0000 0000 1011

✓ $x \ll 1$ 0000 0000 0001 0110 ← Add ZEROS

✓ $x \gg 1$ Add ZEROS → 0000 0000 0000 0101

Examples

- Suppose X is an unsigned integer whose bit pattern is 0000 0000 0000 1011 whose equivalent value in decimal number system is 11.

✓ $x \ll 3$ 0000 0000 0101 1000 ← Add ZEROS = 88

✓ $x \gg 2$ Add ZEROS → 0000 0000 0000 0010 = 2

Note:

- ✓ $x = y \ll 1$; same as $x = y * 2$ (Multiplication)
- ✓ $x = y \gg 1$; same as $x = y / 2$ (Division)

Bitwise Shift operators

- Op and n can be constants or variables.
- There are 2 restrictions on the value of n
 - ✓ n cannot be –ve
 - ✓ n should not be greater than number of bits used to represent Op.(E.g.: suppose op is *int* and size is 2 bytes then n cannot be greater than 16).

Bitwise complement operator

- The complement operator(\sim) is an *unary operator* and inverts all the bits represented by its operand.
- Suppose $x=1001100010001111$
 $\sim x=0110011101110000$ (complement)
- Also called as 1's complement operator.

Poll Question

Go to chat box/posts for the link to the Poll question

Submit your solution in next 2 minutes

Click the result button to view your score

L6-L7 Operators

Learning Objectives

To learn and appreciate the following concepts

- Type conversions
- Assignment Operators and Conditional Expressions
- Precedence and Order of Evaluation

Session Outcome

- At the end of session student will be able to learn and understand
 - Type conversions
 - Assignment Operators and Conditional Expressions
 - Precedence and Order of Evaluation

Type Conversions in Expressions

- C permits mixing of constants and variables of different types in an expression
- C automatically converts any intermediate values to the proper type so that the expression can be evaluated without losing any signification
- This automatic conversion is known as implicit type conversion

Type Conversions in Expressions

- The final result of an expression is converted to the type of the variable on the left of the assignment sign before assigning the value to it
- However the following changes are introduced during the final assignment
 - Float to int causes truncation of the fractional part
 - Double to float caused rounding of digits
 - Long int to int causes dropping of the excess higher order bits

Type Conversions in Expressions

- **Explicit type conversion**
 - There are instances when we want to force a type conversion in a way that is different from the automatic conversion
 - E.g ratio=57/67
 - Since 57 and 67 are integers in the program , the decimal part of the result of the division would be lost and ratio would represent a wrong figure
 - This problem can be solved by converting locally as one of the variables to the floating point as shown below:
 - The general form of a cast is
 - (type-name) expression
 - Eg: ratio= (float) 57/67

Type Conversions in Expressions

- The operator (**float**) converts the 57 to floating point then using the rule of automatic conversion
- The division is performed in floating point mode, thus retaining the fractional part of result
- The process of such a local conversion is known as explicit conversion or casting a value



The Type Cast Operator

```
int a =150;  
  
float f; f = (float) a / 100; // type cast operator
```

- The type cast operator has the effect of converting the value of the variable ‘a’ to type float for the purpose of evaluation of the expression.
- This operator does NOT permanently affect the value of the variable ‘a’;
- The type cast operator has a higher precedence than all the arithmetic operators except the unary minus and unary plus.
- Examples of the use of type cast operator:

(int) 29.55 + (int) 21.99 results in 29 + 21

(float) 6 / (float) 4 results in 1.5

(float) 6 / 4 results in 1.5

Type Conversions in Expressions

Example	Action
<code>x=(int) 7.5</code>	7.5 is converted to integer by truncation
<code>a=(int) 21.3/(int)4.5</code>	Evaluated as 21/4 and the result would be 5
<code>b=(double)sum/n</code>	Division is done in floating point mode
<code>y=(int)(a+b)</code>	The result of a+b is converted to integer
<code>z=(int)a+b</code>	a is converted to integer and then added to b
<code>p=cos((double)x)</code>	Converts x to double before using it

Integer and Floating-Point Conversions

- Assign an integer value to a floating variable: does not cause any change in the value of the number; the value is simply converted by the system and stored in the floating format.
- Assign a floating-point value to an integer variable: the decimal portion of the number gets truncated.
- Integer arithmetic (division):
 - int divided to int => result is integer division
 - int divided to float or float divided to int => result is real division (floating-point)

Integer and Floating-Point Conversions

```
#include <stdio.h>
int main ()
{
    float f1 = 123.125, f2;
    int i1, i2 = -150;
    i1 = f1; // float to integer conversion
    printf ("float assigned to int produces");
    printf("%d\n",i1);
    f2 = i2; // integer to float conversion
    printf("integer assigned to float produces");
    printf("%d\n",f2);
    printf("integer assigned to float produces");
    printf("%f\n",f2);
    i1 = i2 / 100; // integer divided by integer
    printf("integer divided by 100 produces");
    printf("%d\n",i1);
    f1 = i2 / 100.0; // integer divided by a float
    printf("integer divided by 100.0 produces");
    printf("%f\n",f1);
    return 0;
}
```

123

0

-150.0

-1

-1.500

The assignment operators

- The C language permits you to join the arithmetic operators with the assignment operator using the following general format: op=, where op is an arithmetic operator, including +, -, *, /, and %.

- Example:

count += 10;

- Equivalent to:

count=count+10;

- Example:

a /= b + c

- Equivalent to:

a = a / (b + c)

The conditional operator (? :)

condition ? expression1 : expression2

- *condition* is an expression that is evaluated first.
- If the result of the evaluation of *condition* is TRUE (nonzero), then *expression1* is evaluated and the result of the evaluation becomes the result of the operation.
- If *condition* is FALSE (zero), then *expression2* is evaluated and its result becomes the result of the operation.

maxValue = (a > b) ? a : b;

Equivalent to:

```
if ( a > b )
    maxValue = a;
```

```
else
```

```
    maxValue = b;
```

Comma (,)operator

- The coma operator is used basically to separate expressions.

i = 0, j = 10; // in initialization [i → r]

- The meaning of the comma operator in the general expression e1, e2 is

“evaluate the sub expression e1, then evaluate e2; the value of the expression is the value of e2”.

Operator precedence & Associativity

Operator Category	Operators	Associativity
Unary operators	+ - + + -- ~ !	R→L
Arithmetic operators	* / %	L→R
Arithmetic operators	+ -	L→R
Bitwise shift left	<< >>	L→R
Bitwise shift right		
Relational operators	< <= > >=	L→R
Equality operators	== !=	L→R
Bitwise AND, XOR, OR	& ^	L→R
Logical and	&&	L→R
Logical or		L→R
Assignment operator	= += -= *= /= %=	R→L

Summary of Operators



Precedence	Operator	Description	Associativity
1 highest	::	Scope resolution	None
2	++ -- () [] . ->	Suffix increment Suffix decrement Parentheses (Function call) Brackets (Array subscripting) Element selection by reference Element selection through pointer	Left-to-right
3	++ -- + - ! ~ (type) * & sizeof	Prefix increment Prefix decrement Unary plus Unary minus Logical NOT Bitwise NOT (One's Complement) Type cast Indirection (dereference) Address-of Size-of	Right-to-left
4	* ->*	Pointer to member Pointer to member	Left-to-right
5	*	Multiplication	Left-to-right
6	/	Division	Left-to-right
7	%	Modulo (remainder)	Left-to-right
8	+	Addition	Left-to-right
9	-	Subtraction	Left-to-right
10	<<	Bitwise left shift	Left-to-right
11	>>	Bitwise right shift	Left-to-right
12	<	Less than	Left-to-right
13	<=	Less than or equal to	Left-to-right
14	>	Greater than	Left-to-right
15	>=	Greater than or equal to	Left-to-right
16	==	Equal to	Left-to-right
17	!=	Not equal to	Left-to-right
18	&	Bitwise AND	Left-to-right
19	^	Bitwise XOR (exclusive or)	Left-to-right
20		Bitwise OR (inclusive or)	Left-to-right
21	&&	Logical AND	Left-to-right
22		Logical OR	Left-to-right
23	?:	Ternary conditional	Right-to-left
24	=	Direct assignment	
25	+=	Assignment by sum	
26	-=	Assignment by difference	
27	*=	Assignment by product	
28	/=	Assignment by quotient	
29	%=	Assignment by remainder	
30	<<=	Assignment by bitwise left shift	Right-to-left
31	>>=	Assignment by bitwise right shift	
32	&=	Assignment by bitwise AND	
33	^=	Assignment by bitwise XOR	
34	=	Assignment by bitwise OR	
35 lowest	,	Comma	Left-to-right

Detailed
Precedence
Table

Example:

Show all the steps how the following expression is evaluated. Consider the initial values of $i=8$, $j=5$.

$$2*((i/5)+(4*(j-3))%(i+j-2))$$

Example solution:

$$2*((i/5)+(4*(j-3))%(i+j-2)) \quad i \rightarrow 8, j \rightarrow 5$$

$$2*((8/5)+(4*(5-3))%(8+5-2))$$

$$2*(1+(4*2)%11)$$

$$2*(1+8%11)$$

$$2*(1+8)$$

$$2*9$$

$$\underline{18}$$

Operator precedence & Associativity

Ex: **(x==10 + 15 && y < 10)**

Assume x=20 and y=5

Evaluation:

- + (x==25 && y< 10)
- < (x==25 && true)
- == (False && true)
- && (False)

Tutorial Problems

- Suppose that $a=2$, $b=3$ and $c=6$, What is the answer for the following: $(a==5)$
 $(a * b > =c)$
 $(b+4 > a * c)$
 $((b=2)==a)$
- Evaluate the following:
 1. $((5 == 5) \&\& (3 > 6))$
 2. $((5 == 5) || (3 > 6))$
 3. $7==5 ? 4 : 3$
 4. $7==5+2 ? 4 : 3$
 5. $5>3 ? a : b$
 6. $K = (num > 5 ? (num <= 10 ? 100 : 200) : 500);$ where $num =30$
- In $b=6.6/a+(2*a+(3*c)/a*d)/(2/n)$; which operation will be performed first.
- If a is an integer variable, $a=5/2$; will return a value
- The expression, $a=7/22*(3.14+2)*3/5$; evaluates to
- If a is an Integer, the expression $a = 30 * 1000 + 2768$; evaluates to

Session 5 Summary

- Arithmetic Operators
- Relational and Logical Operators
- Increment and Decrement Operators
- Bitwise Operators
- Type conversions
- Assignment Operators and Conditional Expressions
- Precedence and Order of Evaluation

Poll Question

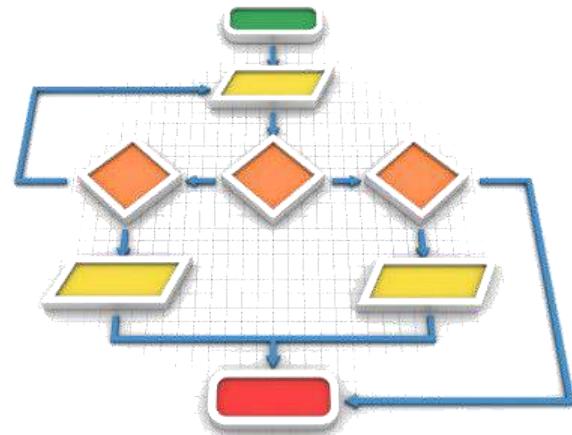
Go to chat box/posts for the link to the Poll question

Submit your solution in next 2 minutes

Click the result button to view your score

Decision Making, Branching & Switch

L8



Learning objectives

To learn and appreciate the following concepts

- The if Statement
- The if-else Statement

Learning Outcomes

- At the end of session student will be able to learn and understand
 - The if Statement
 - The if-else Statement

Control Structures

- A **control structure** refers to the order of executing the program statements.
- The following three approaches can be chosen depending on the problem statement:

✓ **Sequential (Serial)**

- In a **Sequential approach**, all the statements are executed in the same order as it is written.

✓ **Selectional (Decision Making and Branching)**

- In a **Selectional approach**, based on some conditions, different set of statements are executed.

✓ **Iterational (Repetition)**

- In an **Iterational approach** certain statements are executed repeatedly.

DECISION MAKING AND BRANCHING

C decision making and branching statements are:

1. **if statement**
2. **switch statement**

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;
}
next_statement;
```

If expression is true (non-zero), executes statement.
It gives you the choice of executing statement or skipping it.

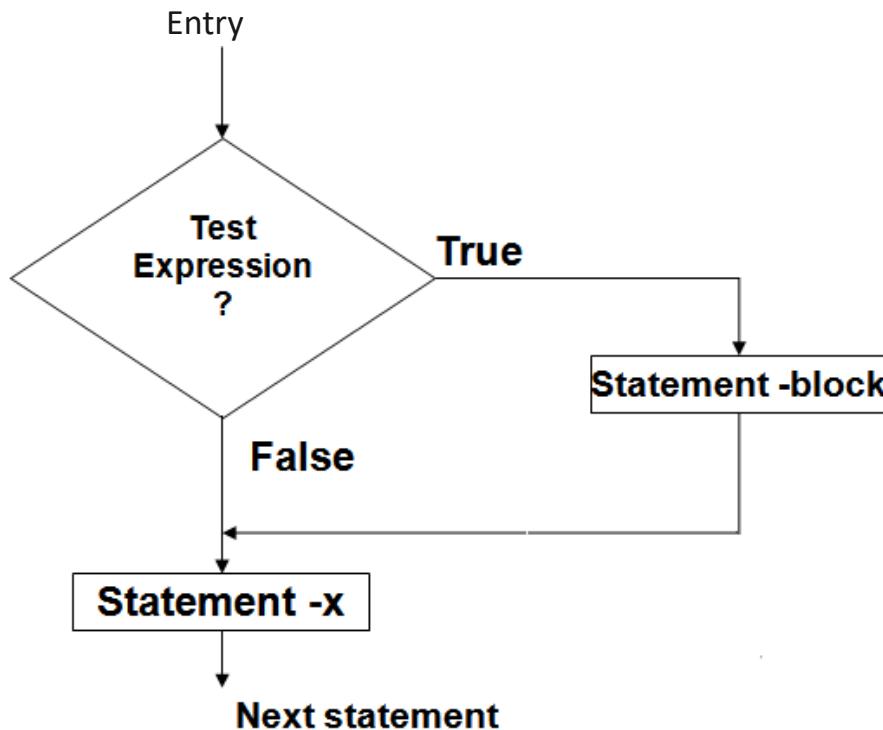
if Statement- explanation

- (*test Expression*) is first evaluated.
- If **TRUE** (non-zero), the ‘if’ statement block is executed.
- If **FALSE** (zero) the next statement following the if statement block is executed.
- So, during the execution, based on some condition, some code will not be executed (skipped).

For example: **bonus = 0;**

```
if (hours > 70)
bonus = 10000;
salary= salary + bonus;
```

Flow chart of simple if



Find out whether a number is even or odd.

```
#include <stdio.h>

int main()
{
    int x;
    printf("input an integer\n");
    scanf("%d",&x);
    if ((x % 2) == 0)
    {
        printf("It is an even number\n");
    }
    if ((x%2) == 1)
    {
        printf("It is an odd number\n");
    }
    return 0;
}
```

Example - if

// Program to calculate the absolute value of an integer

```
int main ()  
{  
    int number;  
    printf("Type in your number: ");  
    scanf("%d",&number);  
    if ( number < 0 )  
        number = -number;  
    printf("The absolute value is");  
    printf("%d",number);  
    return 0;  
}
```

The **if-else** statement

```
if (test expression )  
{  
    statement_block1  
}  
else  
{  
    statement_block2  
}  
Next_statement
```

if-else
statement:
enables you to
choose between
two statements

if-else statement

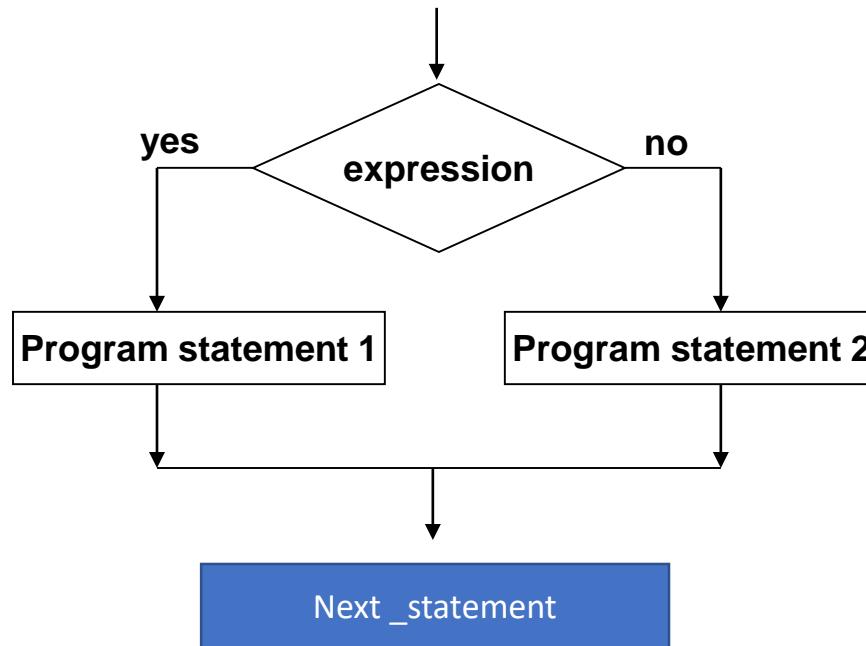
Explanation:

1. First ,the (test expression) is evaluated.
2. If it evaluates to **non-zero (TRUE)**, **statement_1** is executed, otherwise, if it evaluates to **zero (FALSE)**, **statement_2** is executed.
3. They are **mutually exclusive**, meaning, either **statement_1** is executed or **statement_2**, but not both.
4. If the **statements_1** and **statements_2** take the **form of block** , **they** must be put in curly braces.

Example:

```
if(job_code == 1)
    rate = 7.00;
else
    rate = 10.00;
printf("%d",rate);
```

The **if-else** statement



Find out whether a number is even or odd.

```
#include <stdio.h>

int main()
{
    int x;
    printf("Input an integer\n");
    scanf("%d",&x);
    if ((x % 2) == 0)
    {
        printf("It is an even number\n");
    }
    else
    {
        printf("It is an odd number\n");
    }
    return 0;
}
```

WAP to find largest of 2 numbers

```
#include<stdio.h>
int main()
{
    int a, b;
    printf("Enter 2 numbers\n");
    scanf("%d %d",&a,&b);

    if(a > b)
        printf("Large is %d\t",a);
    else
        printf("Large is %d\t",b);

    return 0;
}
```

Attention on if-else syntax !

```
if ( expression )  
    program  
statement 1  
else  
    program  
statement 2
```

In C, the ; is part
(end) of a
statement !

```
if ( remainder == 0 )  
    printf("The number is even.\n");  
else  
    printf("The number is odd.\n");
```

Syntactically OK (void
statement on if) but a
semantic error !

```
if ( x == 0 );  
    printf("The number is zero.\n");
```

Example: determine if a year is a leap year

```
#include<stdio.h>
int main()
{
    int year;
    printf("Enter the year");
    scanf("%d",&year);
    if(year%4 == 0)
    {
        if( year%100 == 0)
        {
            if ( year%400 == 0)
                printf("%d is a leap year",year);
            else
                printf("%d is not a leap year",year);
        } else printf("%d is a leap year",year);
    } else  printf("%d is not a leap year",year);
    return 0;
}
```

A leap year is exactly divisible by 4 except for century years (years ending with 00). The century year is a leap year only if it is perfectly divisible by 400.

Testing for character ranges

```
#include<stdio.h>
int main()
{
    char ch;
    printf("enter a character\n");
    scanf("%c",&ch);
    if (ch >= 'a' && ch <= 'z')
        printf("lowercase char\n");
    if (ch >= 'A' && ch <= 'Z')
        printf("uppercase char\n");
    if (ch >= '0' && ch <= '9')
        printf("digit char\n");
    else
        printf(" special char\n");
    return 0;
}
```

Output:
enter a character:
C
uppercase char
special char

enter a character:
j
lowercase char
special char

enter a character:
5
digit char

Testing for ranges



```
if (x >= 5 && x <= 10)
    printf("in range");
```



```
if (5 <= x <= 10)
    printf("in range");
```

Testing for ranges

YES

```
if (x >= 5 && x <= 10)
    printf("in range");
```

NO!

```
if (5 <= x <= 10)
    printf("in range");
```

Syntactically correct, but semantically an error !!!

Because the order of evaluation for the `<=` operator is left-to-right, the test expression is interpreted as follows:

`(5<= x) <= 10`

The subexpression `5 <= x` either has the value 1 (for true) or 0 (for false). Either value is less than 10, so the whole expression is always true, regardless of `x` !

Poll Question

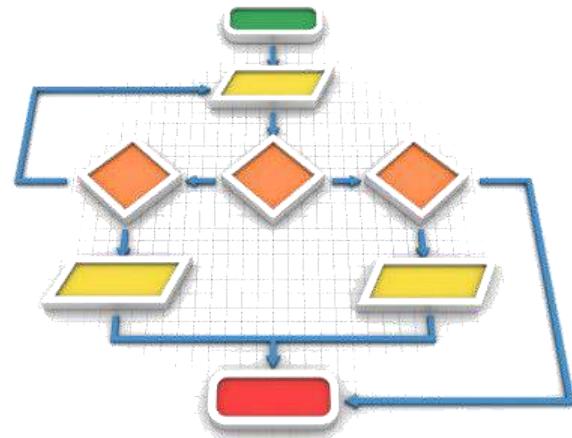
Go to chat box/posts for the link to the Poll question

Submit your solution in next 2 minutes

Click the result button to view your score

Decision Making, Branching & Switch

L8



Learning objectives

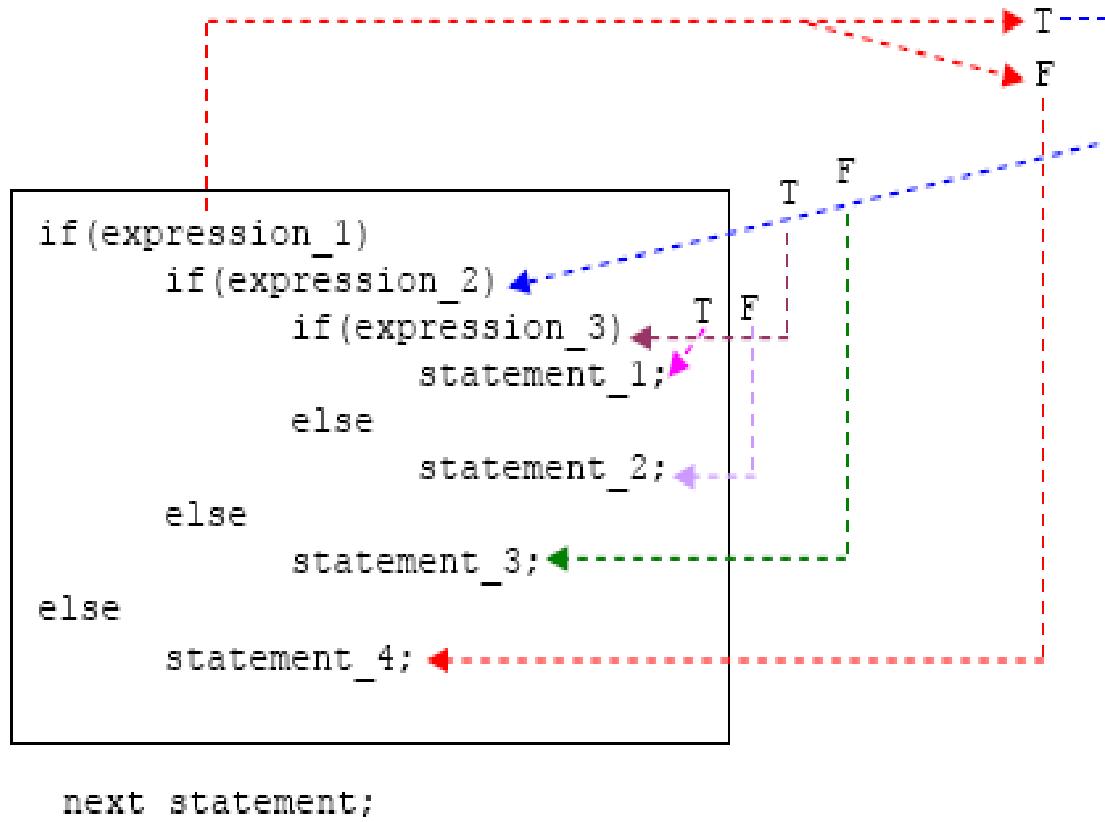
To learn and appreciate the following concepts

- Nested if Statements
- Else-if ladder

Learning Outcomes

- At the end of session student will be able to learn and understand
 - Nested if Statements
 - Else-if ladder

Nested if-else Statement



If-else nesting -Explanation

1. The if-else constructs **can be nested** (placed one within another) to any depth.
2. In this nested form, **expression_1** is evaluated.

- If it is zero (FALSE-F), **statement_4** is executed and the **entire nested if statement is terminated**;
- If not (TRUE-T), control goes to the second if (within the first if) and **expression_2** is evaluated. If it is zero, **statement_3** is executed;
- If not, control goes to the third if (within the second if) and **expression_3** is evaluated.
- If it is zero, **statement_2** is executed;
- If not, **statement_1** is executed. The **statement_1** (inner most) will only be executed if all the if statement is true.

Smallest among three numbers

```
#include <stdio.h>
int main()
{
    int a, b, c, smallest;

    printf("Enter a, b & c\n");
    scanf("%d %d %d", &a,&b,&c);
```

```
        if (a < b)
        {
            if (a < c)
                { smallest = a; }
            else
                { smallest = c; }

        }
        else
        {
            if (b < c)
                { smallest = b; }
            else
                { smallest = c; }

        }
    printf("Smallest is %d",smallest);
    return 0;
}
```

Nested if statements

```
if (number > 5)
    if (number < 10)
        printf("1111\n");
    else printf("2222\n");

if (number > 5) {
    if (number < 10)
        printf("1111\n");
}
else printf("2222\n");
```

Rule: an else goes with the most recent if, unless braces indicate otherwise

The else-if ladder

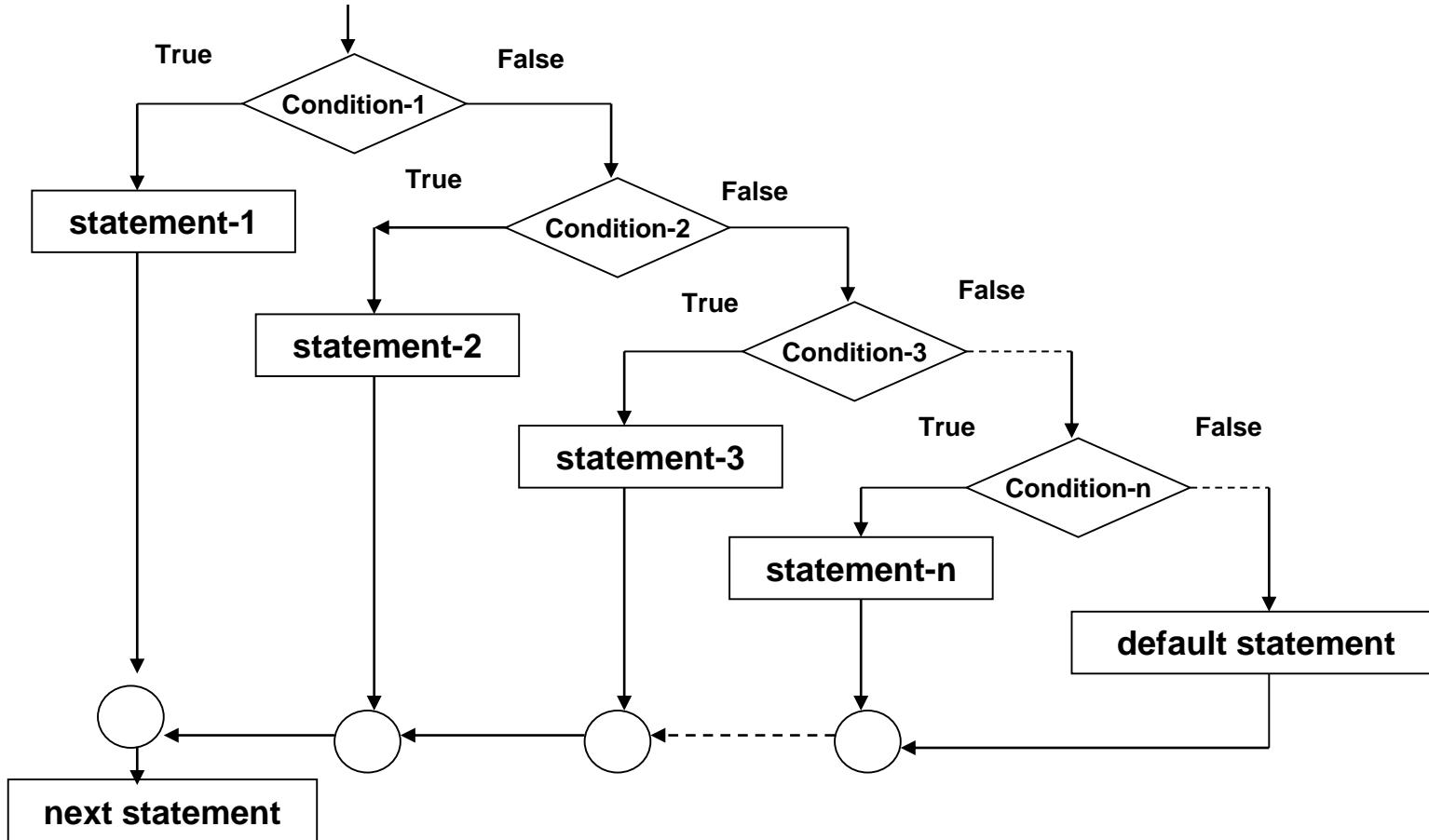
```
if (Expression_1 )  
{  
    statement _block1  
}  
else if (Expression_2)  
{  
    statement _block2  
}  
.....  
else if (Expression_n)  
{  
    statement _blockn  
}  
else  
{  
    last_statement  
}
```

Next_statement

else if ladder -Explanation

- **expression_1** is first evaluated. If it is TRUE, **statement_1** is executed and the whole statement terminated and the **next_statement** is executed.
- On the other hand, if **expression_1** is FALSE, control passes to the **else if** part and **expression_2** is evaluated.
- If it is TRUE, **statement_2** is executed and the whole system is terminated.
- If it is False, **other else if parts** (if any) are tested in a similar way.
- Finally, if **expression_n** is True, **statement_n** is executed; if not, **last_statement** is executed.
- Note that only one of the statements will be executed others will be skipped.
- The **statement_n's** could also be a **block of statement** and must be put in curly braces.

else-if ladder Flow of control



Testing for character ranges

```
#include<stdio.h>
int main()
{
    char ch;
    printf("enter a character\n");
    scanf("%c",&ch);
    if (ch >= 'a' && ch <= 'z')
        printf("lowercase char\n");
    else if (ch >= 'A' && ch <= 'Z')
        printf("uppercase char\n");
    else if (ch >= '0' && ch <= '9')
        printf("digit char\n");
    else
        printf(" special char\n");
return 0;
}
```

WAP using else-if ladder to calculate grade for the marks entered

```
int main() {  
    char cgrade;  
    int imarks;  
    printf("enter marks");  
    scanf("%d",&imarks);  
  
    if(imarks>79)  
        cgrade = 'A';  
    else if (imarks>59)  
        cgrade = 'B';  
    else if (imarks>49)  
        cgrade = 'C';  
    else if (imarks>39)  
        cgrade = 'D';  
    else  
        cgrade = 'F';  
}
```

For inputs
imarks= 46
grade = D
imarks= 64
grade = B

```
printf("Grade :%c\n",cgrade);  
return 0;
```

Example: else-if

// Program to implement the sign function

```
#include <stdio.h>
int main ( )
{
    int number, sign;
    printf("Please type in a number: ");
    scanf("%d",&number);
    if ( number < 0 )
        sign = -1;
    else if ( number == 0 )
        sign = 0;
    else // Must be positive
        sign = 1;
    printf("Sign = %d",sign);
    return 0;
}
```

Example – multiple choices

```
/* Program to evaluate simple expressions of the form number operator number */  
#include <stdio.h>  
int main ( )  
{  
    float value1, value2,result;  
    char operator;  
    printf("Type in your expression.\n");  
    scanf("%f %c %f", &value1,&operator,&value2);  
    if ( operator == '+' )  
        {result=value1+value2;  
         printf("%f",result);}  
    else if ( operator == '-' )  
        {result=value1-value2;  
         printf("%f",result);}  
    else if ( operator == '*' )  
        {result=value1*value2;  
         printf("%f",result);}  
    else if ( operator == '/' )  
        {result=value1/value2;  
         printf("%f",result);}  
    else  
        printf("Unknown operator.\n");  
    return 0;  
}
```

Problem...

- Find the roots of a quadratic equation ax^2+bx+c using if else control statements.
- Roots of a quadratic equation

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- 3 cases
 - Discriminant < 0 ; roots are imaginary $\rightarrow 1 + i 2.45$
 - Discriminant = 0 ; roots are real and equal $\rightarrow - b/2a$
 - Discriminant > 0 ; roots are real and unequal \rightarrow
 $r1 = (-b + \sqrt{\text{disc}})/(2a)$
 $r2 = (-b - \sqrt{\text{disc}})/(2a)$

Find the roots of Quadratic equation using if-else statement

```
#include<stdio.h>
#include <math.h>
int main()
{
float a,b,c,root1,root2,re,im, disc;
scanf("%f %f %f",&a,&b,&c);
disc=b*b-4*a*c;

if (disc<0)
{
    printf("imaginary roots\n");
    re= - b / (2*a);
    im = pow(fabs(disc),0.5)/(2*a);
    printf("root1=% .21f+% .21fi and
root2 =% .21f-% .2fi", re,im,re,im);
}
}
```

```
else if (disc==0)
{
    printf("Real & equal roots");
    re=-b / (2*a);
    printf("Root1 and root2 are
%.21f",re);
}

else /*disc > 0 */
{
    printf("Real & distinct roots");
    printf("Roots are");
    root1=(-b + sqrt(disc))/(2*a);
    root2=(-b - sqrt(disc))/(2*a);
    printf("Root1 = % .21f and root2
=% .21f",root1,root2);
}
return 0;
}
```

Session 6 Summary

At the end of session the student will be able to

- The if Statement
- The if-else Statement
- Nested if Statements
- Else-if ladder

Poll Question

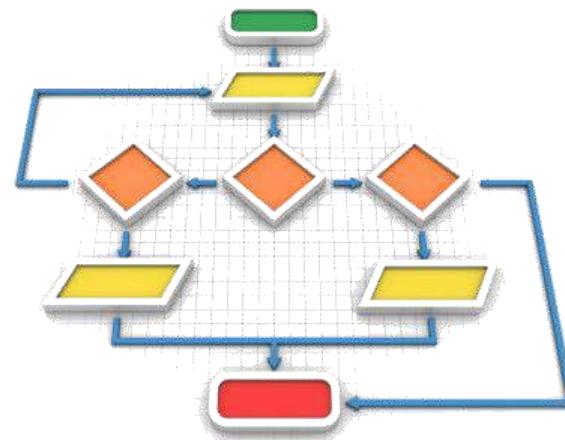
Go to chat box/posts for the link to the Poll question

Submit your solution in next 2 minutes

Click the result button to view your score

Decision Making, Branching & Switch

L8-T3



Learning Objectives

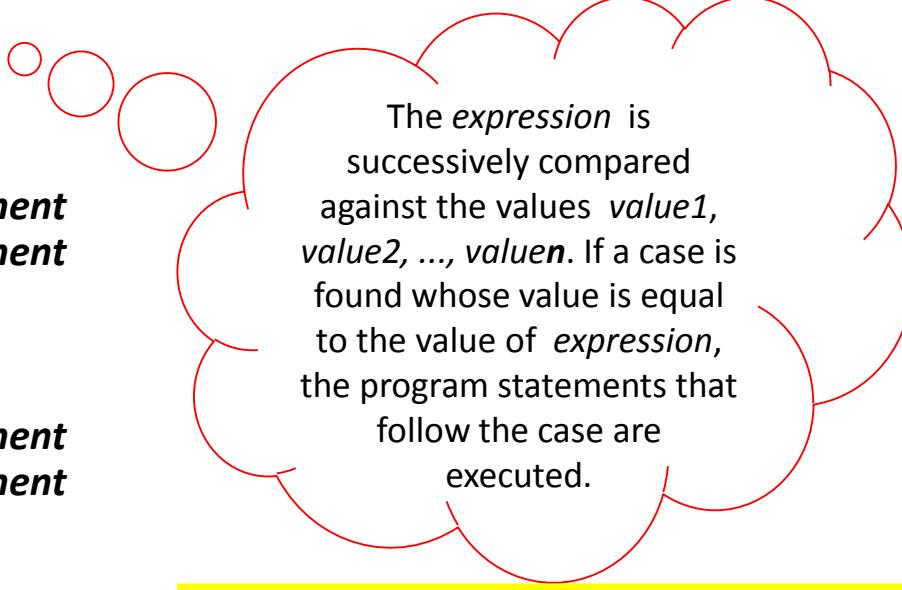
- To learn and appreciate the following concepts
 - The switch Statement
 - Examples

Learning Outcome

- At the end of session student will be able to learn and understand
 - The switch Statement
 - Use Switch statement

The switch statement

```
switch ( expression )  
{  
    case value1:  
        program statement  
        program statement  
        ...  
        break;  
    case value2:  
        program statement  
        program statement  
        ...  
        break;  
    case value n:  
        program statement  
        program statement  
        ...  
        break;  
    default:  
        program statement  
        program statement  
        ...
```

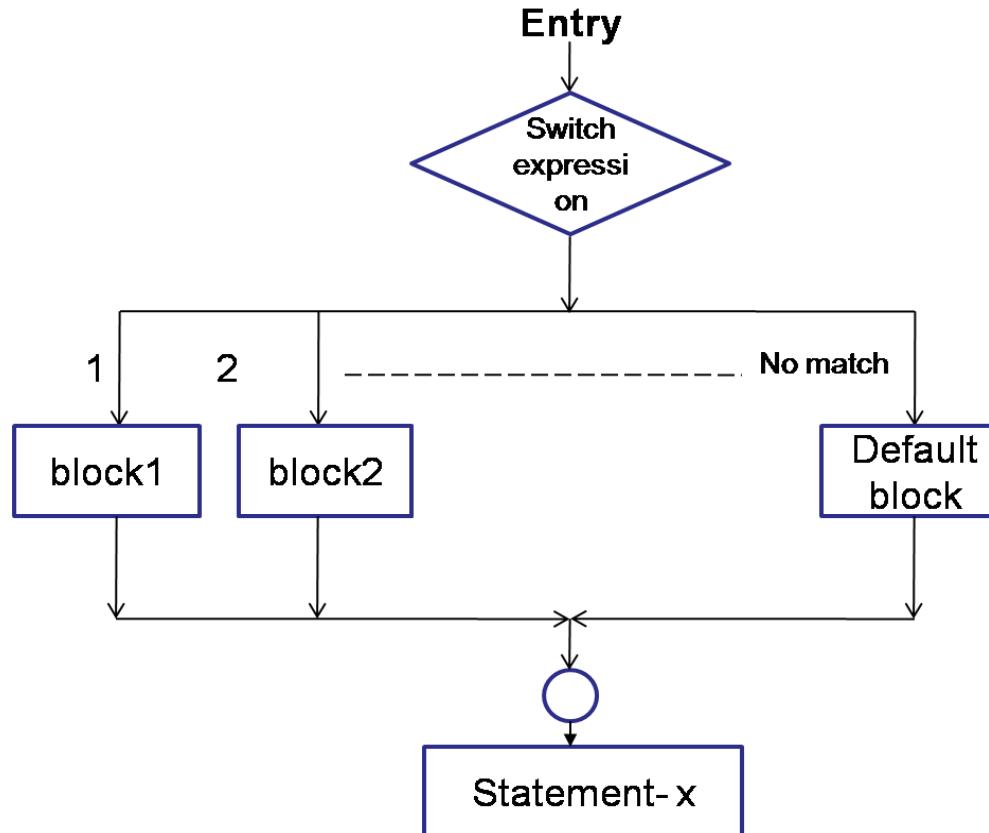


The *expression* is successively compared against the values *value1*, *value2*, ..., *valuen*. If a case is found whose value is equal to the value of *expression*, the program statements that follow the case are executed.

The switch test expression must be one with an integer value (including type char) (No float!).

The case values must be integer-type constants or integer constant expressions (You can't use a variable for a case label !)

switch- control flow





switch- example 1

```
#include<stdio.h>
int main()
{
    int choice;
    printf("Enter your choice: 1-yes, 2-no\n");
    scanf("%d",&choice);
    switch(choice)
    {
        case 1: printf("YESSSSSS.....");
                  break;
        case 2: printf("NOOOOO.....");
                  break;
        default: printf("DEFAULT CASE.....");
    }
    printf("The choice is %d",choice);
}
return 0;
```

switch- example 2

```
scanf("%d",&mark);

switch (mark)
{
    case 100:
    case 90:
    case 80: grade='A';
               break;

    case 70:
    case 60:
        grade='B';
        break;
}
```

```
case 50:
    grade='C'
    break;

case 40:
    grade='D'
    break;

default: grade='F';
          break;
}

printf("%c",grade);
```

An Example – switch case

```
char ch;  
scanf("%c",&ch);  
  
switch(ch)  
{  
    case 'a' : printf("Vowel");  
    break;  
    case 'e' : printf("Vowel");  
    break;  
    case 'i' : printf("Vowel");  
    break;  
    case 'o' : printf("Vowel");  
    break;  
    case 'u' : printf("Vowel");  
    break;  
    default: printf("Not a Vowel");    }
```

An Example – switch case

```
char ch;  
scanf("%c",&ch);  
  
switch(ch)  
{  
    case 'a':  
    case 'e':  
    case 'i' :  
    case 'o' :  
    case 'u' : printf("Vowel");  
    break;  
    default: printf("Not a Vowel");    }  
}
```

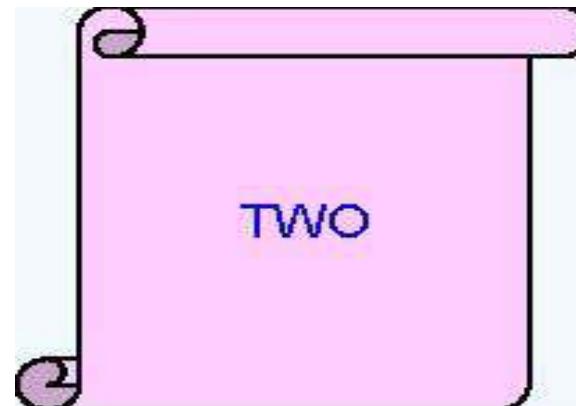
Example - switch

```
/* Program to evaluate simple expressions
of the form value operator value */
#include <stdio.h>
int main (void)
{
    float value1, value2;
    char operator;
    int result;
    printf("Type in your expression.\n");
    scanf("%f %c %f",
    &value1,&operator,&value2);
    switch (operator)
    {case '+':
        result=value1+value2;
        printf("%f",result);
        break;
    case '-':
        result=value1-value2;
        printf("%f",result);
        break;
    }
```

```
case '*':
    result=value1*value2;
    printf("%f",result);
    break;
case '/':
    if ( value2 == 0 )
        printf("Division by
zero.\n");
    else result=value1 / value2;
    printf("%f",result);
    break;
default:
    printf("Unknown Operator");
}
return 0;
```

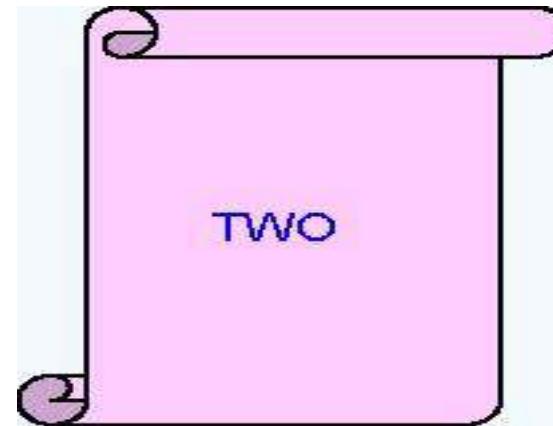
What is the output of the following code snippet?

```
int iNum = 2;  
switch(iNum)  
{  
    case 1:  
        printf("ONE");  
        break;  
    case 2:  
        printf("TWO");  
        break;  
    case 3:  
        printf("THREE");  
        break;  
    default:  
        printf("INVALID");  
        break;  
}
```



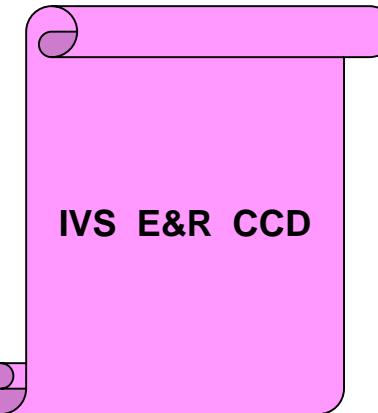
What is the output of the following code snippet?

```
iNum = 2;  
switch(iNum)  
{  
    default:  
        printf("INVALID");  
    case 1:  
        printf("ONE");  
    case 2:  
        printf("TWO");  
        break;  
    case 3:  
        printf("THREE");  
}  
}
```



What is the output of the following code snippet?

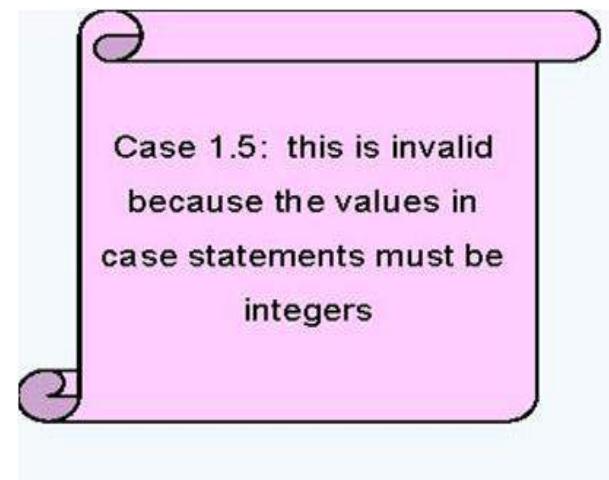
```
switch (iDepartmentCode)
{
    case 110 : printf("HRD ");
    case 115 : printf("IVS ");
    case 125 : printf("E&R ");
    case 135 : printf("CCD ");
}
```



Assume iDepartmentCode is 115
find the output ?

What is the output of the following code snippet?

```
int iNum = 2;  
switch(iNum)  
{  
    case 1.5:  
        printf("ONE AND HALF");  
        break;  
    case 2:  
        printf("TWO");  
    case 'A' :  
        printf("A character");  
}
```



Problem: Find the roots of Quadratic equation using switch statement

```
#include<stdio.h>
int main()
{
    Int d;
    float a,b,c,root1,root2,re,im, disc;
    printf("Enter the values of a, b & c:");
    scanf("%f %f %f",&a,&b,&c);
    disc=b*b-4*a*c;
    printf("\nDiscriminant= %f",disc);

    if(disc<0) d=1;
    if(disc==0) d=2;
    if(disc>0) d=3;
switch(d)
{
    case 1:
        printf("imaginary roots\n");
        re= - b / (2*a);
        im = pow(fabs(disc),0.5)/(2*a);
        printf("root1=% .21f+% .21fi and root2 =% .21f-% .2fi", re,im,re,im);
        break;
```

case 2:

```
printf("Real & equal roots");
re=-b / (2*a);
printf("Root1 and root2 are %.21f",re);
break;
```

case 3:

```
printf("Real & distinct roots");
printf("Roots are");
root1=(-b + sqrt(disc))/(2*a);
root2=(-b - sqrt(disc))/(2*a);
printf("Root1 = %.21f and root2 =%.21f",root1,root2);
break;
} // end of switch
return 0;
} //End of Program
```

Some guidelines for writing switch case statements

- (1) Order the cases alphabetically or numerically – improves readability.**
- (2) Put the normal cases first ; put the exceptional cases later.**
- (3) Order cases by frequency:-put the most frequently executed cases first and the least frequently used cases later.**
- (4) Use default case to detect errors and unexpected cases [user friendly messages].**

Poll Question

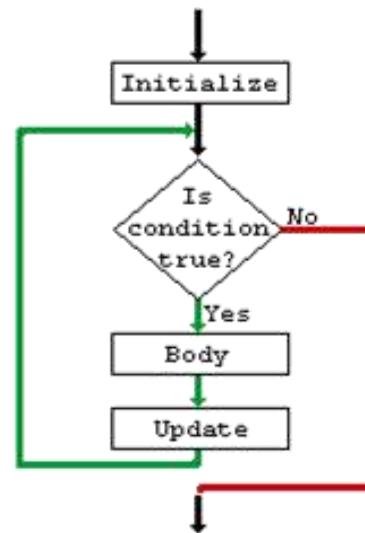
Go to chat box/posts for the link to the Poll question

Submit your solution in next 2 minutes

Click the result button to view your score

Loop Control Structures

L9



Learning Objectives

- To learn and appreciate the following concepts
 - The while Statement
 - Programs

Learning Outcome

- At the end of session student will be able to learn and understand
 - The while Statement
 - Sample programs

Controlling the program flow

- Forms of controlling the program flow:
 - Executing a sequence of statements
 - Using a test to decide between alternative sequences (**branching**)
 - Repeating a sequence of statements (until some condition is met) (**looping**)

Statement1
Statement2
Statement3
Statement4
Statement5
Statement6
Statement7
Statement8

Program Looping

- A set of statements that executes repetitively for a number of times.
- Simple example: displaying a message 100 times:

```
printf(hello !\n");
printf(hello !\n")
printf(hello !\n")
...
printf(hello !\n")
printf(hello !\n")
```

Repeat 100 times
printf(hello !\n")

Program looping: enables you to develop concise programs containing repetitive processes that could otherwise require many lines of code !

The need for program looping

Example problem: computing triangular numbers.

(The n-th triangular number is the sum of the integers from 1 through n)

```
#include <stdio.h>
int main (void)
{
    int triangularNumber;
    triangularNumber = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8;
    printf("The eighth triangular number is
%d",triangularNumber);
    return 0;
}
```

What if we have to compute the 200-th (1000-th, etc) triangular number ?

We have 3 different statements for looping.

Iterative (loop) control structures

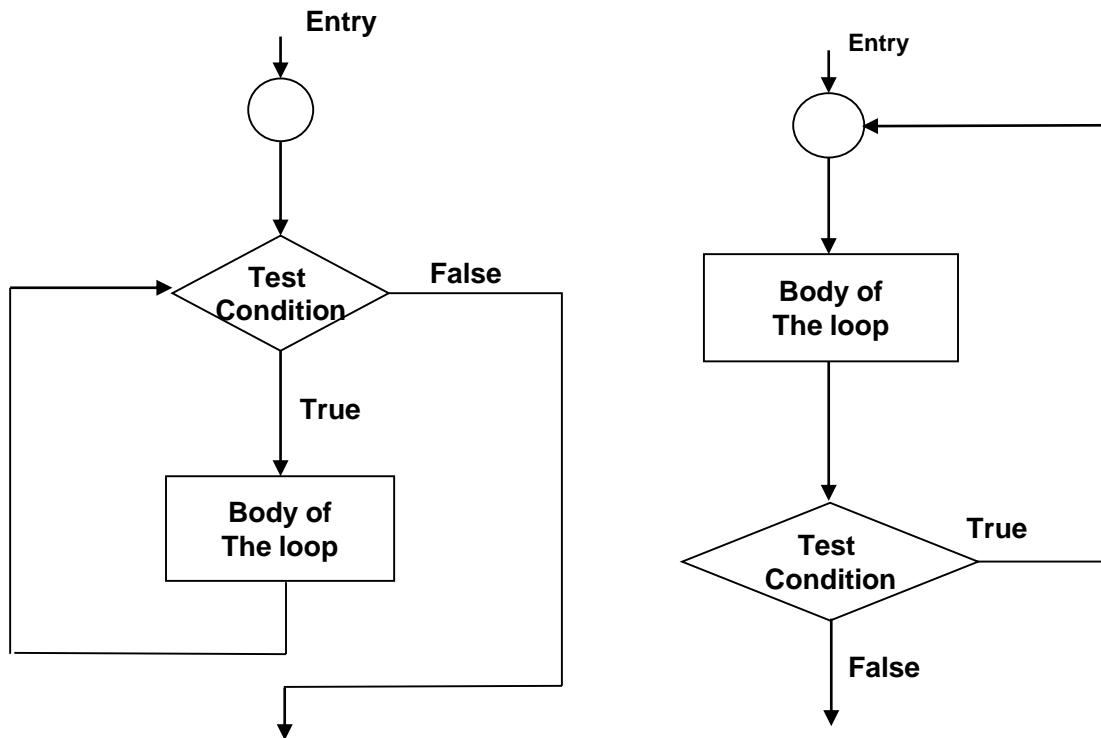
- Three kinds of loop control structures:

- ✓ while
 - ✓ do while
 - ✓ for

Iterative (loop) control structures

- Each loop control structure will have
 - ✓ **Program loop:** body of loop.
 - ✓ **control statement** → tests certain conditions & then directs repeated execution of statements within the body of loop.
- Two types: Based on position of control statement.
 - 1) **Entry controlled loop:** control is tested before the start of the loop. If false, body will not be executed.
 - 2) **Exit controlled loop:** test is performed at the end of the body. i.e. body of loop executed at least once.

Entry Controlled & Exit controlled loops



while-statement

General format:

while (test expression)

{

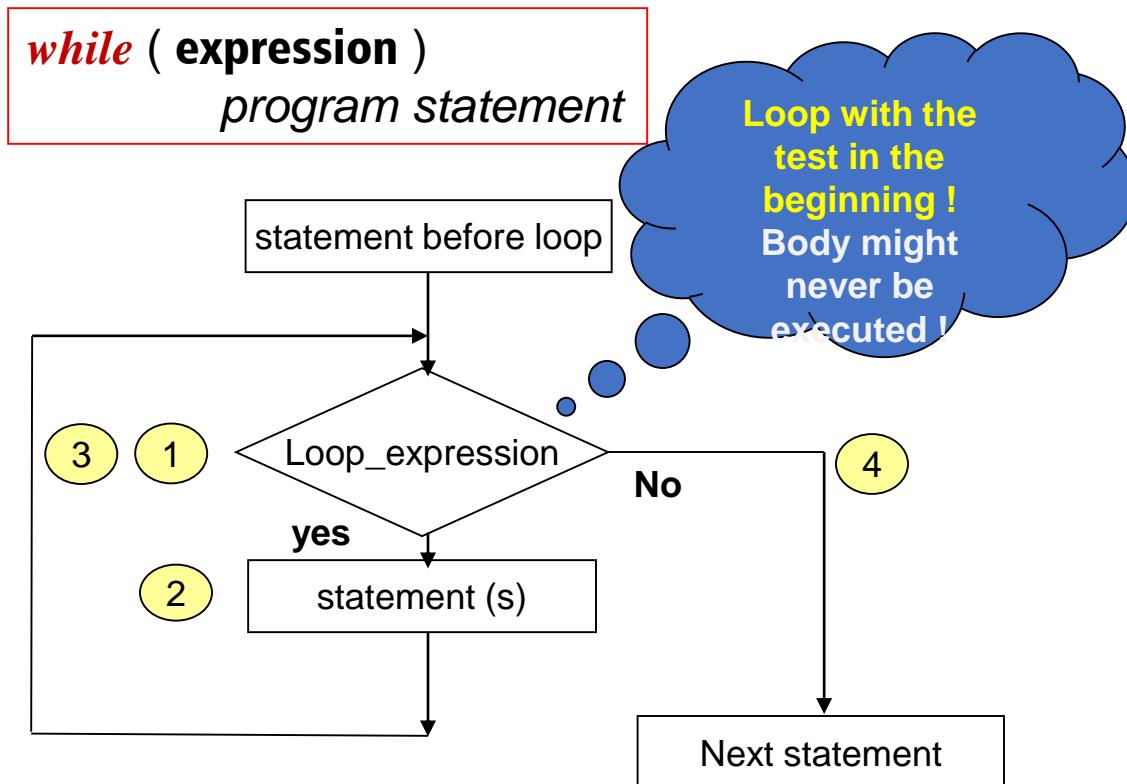
body of the loop

}

***Note: braces optional if
only one statement.***

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

The while statement



Sum and Mean of first N natural numbers

Name of the algorithm: Sum and Mean of natural numbers.

Step 1: Start

Step 2: [Read the maximum value of N]

 Input N

Step 3: [Set sum equal to 0]

 Sum \leftarrow 0

Step 4: [Compute the sum of all first N natural numbers]

 i=1

 While($i \leq N$)

 begin

 Sum \leftarrow Sum + i

 i++;

 end

Sum and Mean of first N natural numbers

Step 5: [Compute mean value of N natural numbers]
Mean \leftarrow Sum / N

Step 6: [Print Sum and Mean]
Print 'Sum of N natural numbers=' , Sum
Print 'Mean of N natural numbers =' , Mean

Step 7: [End of algorithm]
Stop

Finding sum of natural numbers up to 100

```
#include <stdio.h>
int main()
{
    int n;
    int sum;
    sum=0; //initialize
    sum
    n=1;
    while (n < 100)
    {
        sum= sum + n;
        n = n + 1;
    }
    printf("%d",sum);
    return 0;
}
```

Program to reverse the digits of a number

```
#include <stdio.h>
int main()
{
    int number, rev=0, right_digit;

    printf("Enter your number.\n");
    scanf("%d",&number);

    while ( number != 0 )
    {
        right_digit = number % 10;
        rev=rev*10 + right_digit;
        number = number / 10;
    }
    printf("The reversed number is %d", rev);
    return 0;
}
```

Check for palindrome

```
n = num;  
while(num>0)  
{  
    dig = num % 10;  
    rev = rev * 10 + dig;  
    num = num / 10;  
}  
if (n == rev)  
    printf("\n\t GIVEN NO IS A PALINDROME");  
else  
    printf("\n\t GIVEN NO NOT A PALINDROME");
```

Palindrome (number)
e.g.- 121

Session 7 Summary

- **Switch statement**
- **Looping Concepts**
- **While loop**

Poll Question

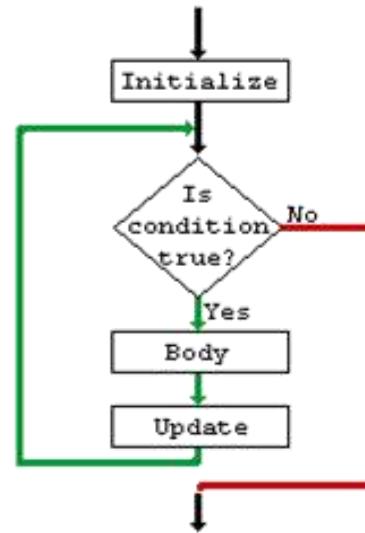
Go to chat box/posts for the link to the Poll question

Submit your solution in next 2 minutes

Click the result button to view your score

Loop Control Structures

L9-T4



Learning Objectives

- To learn and appreciate the following concepts
 - The do-while Statement
 - Nesting of Loops
 - Sample Programs

Learning Outcome

At the end of session the student will be able to

- The do-while Statement
- Nesting of loops
- Write programs

The do – while statement

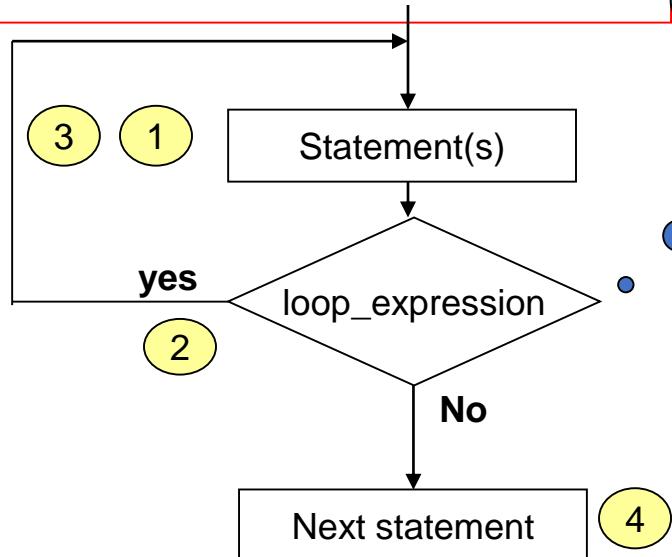
General form:

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

- ✓ **Exit controlled loop.** At the end of the loop, the test condition is evaluated.
- ✓ After do statement, program executes the body of the Loop.
- ✓ Then, the condition is tested, if it is true, body of the loop is executed once again & this process continues as long as the condition is true.
- ✓ **Body of the loop is executed at least once.**
- ✓ **do-while loop can be nested.**

The do statement

do
program
statement
while (loop_expression);



Sum and Mean of first N natural numbers

Name of the algorithm: Sum and Mean of natural numbers.

Step 1: Start

Step 2: [Read the maximum value of N]

 Input N

Step 3: [Set sum equal to 0]

 Sum \leftarrow 0

Step 4: [Compute the sum of all first N natural numbers]

 i=1

 do

 begin

 Sum \leftarrow Sum + i

 i++;

 end

 While(i<=N);

Sum and Mean of first N natural numbers

Step 5: [Compute mean value of N natural numbers]
Mean \leftarrow Sum / N

Step 6: [Print Sum and Mean]
Print 'Sum of N natural numbers=' , Sum
Print 'Mean of N natural numbers =' , Mean

Step 7: [End of algorithm]
Stop

Program to reverse the digits of a number

```
#include <stdio.h>
int main()
{
    int number, rev=0, right_digit;

    printf("Enter your number.\n");
    scanf("%d",&number);

    do
    {
        right_digit = number % 10;
        rev=rev*10 + right_digit;
        number = number / 10;
    }
    while ( number != 0 );

    printf("The reversed number is %d",rev);
    return 0;
}
```

Count the number of digits in a given number

```
scanf("%d",&num);
do
{
    rem=num%10;
    num =num/10;
    ocnt++;
} while(num > 0);

printf("%d digits",ocnt);
```

e.g.- num = 31467
OUTPUT
5 digits

Count the even and odd digits in a given ‘n’ digit number

```
scanf("%d",&num);
do
{
    rem=num%10;
    num =num/10;
    if(rem%2==0)
        ecnt++;
    else
        ocnt++;
} while(num > 0);
```

printf("%d even & %d odd digits",ecnt,ocnt);

e.g.- num = 31467
OUTPUT
2 even & 3 odd digits

Example: Convert binary to decimal

$\text{dec} = \text{bd} * 2^n + \text{bd} * 2^{n-1} + \dots + \text{bd} * 2^1 + \text{bd} * 2^0$

e.g.-given $n=101 \rightarrow 1*2^2 + 0*2^1 + 1*2^0 = 5$

```

int n, p=0, sum=0, k;
printf("Enter a binary number : ");
scanf("%d",&n);

do {
    k=n%10; // binary number in n
    sum= sum + k * pow(2,p);//decimal number in sum
    p++;
    n= n/10;
} while (n!=0);

printf("Decimal Equivalent = %d",sum);

```

Nesting of loop

Do-While Lop

```
i=0;
do
{
    ....
    ....
    j=0;
    do   {
        Statement S;
        j++;
    } while(j<n);
// end of inner 'do' statement

i++; } while(i<m) ;
// end of outer 'do' statement
```

While Loop

```
i=0;
while(i<m)
{
    ....
    ....
    j=0;
    while(j<n)
    {
        Statement S;
        j++;
    } // end of inner
    'while' statement
    i++; } // end of outer 'while'
statement
```

Nesting of loop Examples: Armstrong nos for a given limit ‘n’

```
scanf("%d",&lim);
n=1;
do
{
    sum = 0;
    num = n;
    do
    {
        dig = num%10;
        sum = sum+pow(dig,3);
        num = num/10;
    } while(num>0);
    if(sum == n)
        printf("%d\n\t",n);
    n++;
} while(n<lim);
```

Armstrong Number

e.g. - 371

\sum (cubes of digits) = number

$$3^3 + 7^3 + 1^3 = 371$$

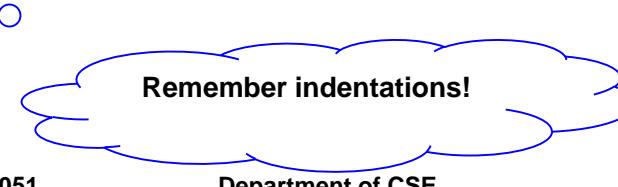
Nested loops

```
#include <stdio.h>
int main()
{
    int n, number, triangularNumber, counter=1;

    while (counter <= 5)
    {
        printf("What triangular number do you want? ");
        scanf("%d",&number);
        triangularNumber=0;
        n=1;

        while(n <= number)
        {
            triangularNumber = triangularNumber + n;
            n++;
        }
        printf("The %d th triangular number is %d:",n-1,triangularNumber);
        counter++;
```

Remember indentations!



Poll Question

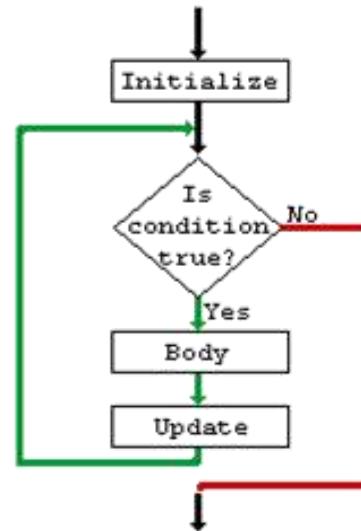
Go to chat box/posts for the link to the Poll question

Submit your solution in next 2 minutes

Click the result button to view your score

Loop Control Structures

L9-T4



Learning Objectives

- To learn and appreciate the following concepts
 - break
 - continue
 - typedef and enum

Learning Outcome

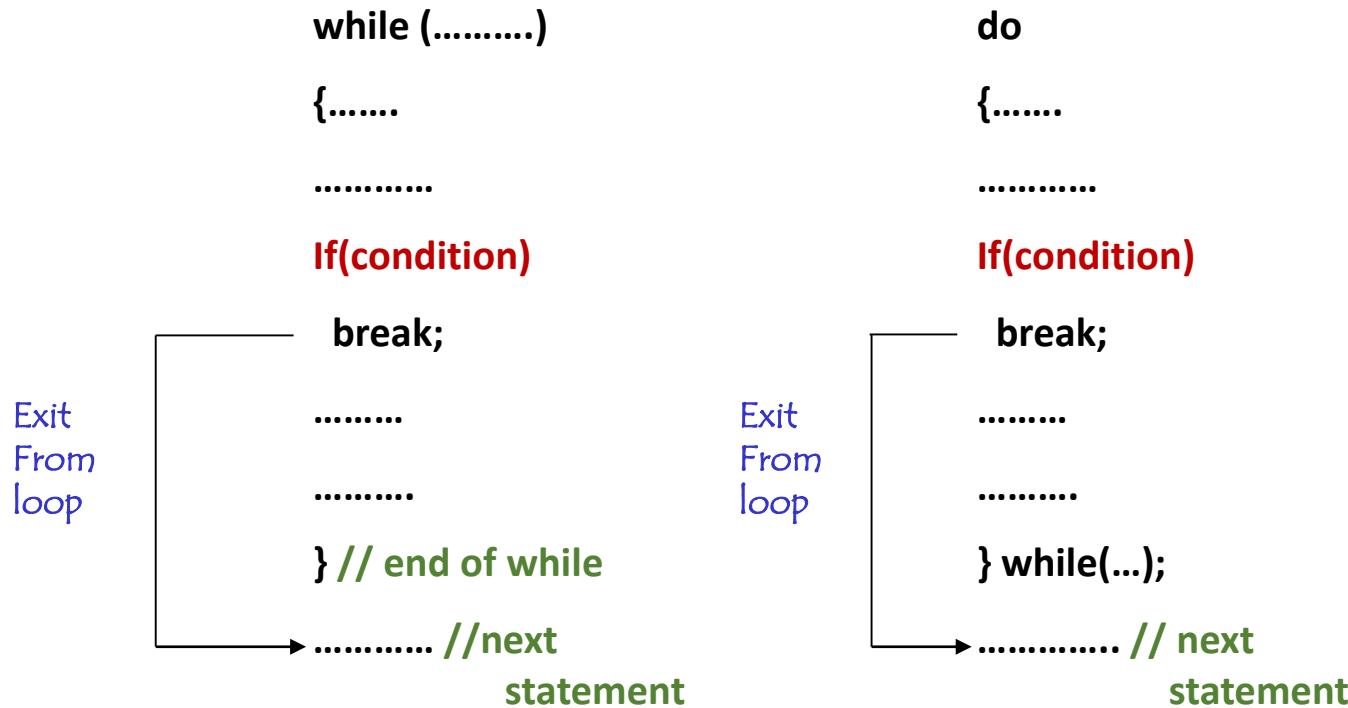
At the end of session the student will be able to

- break
- continue
- typedef and enum

The break Statement

- Used in order to immediately exit from a loop
- After a break, following statements in the loop body are skipped and execution continues with the first statement after the loop
- If a break is executed from within nested loops, only the innermost loop is terminated

Exiting a loop with **break** statement



Break Statement Examples: Check whether given number is prime or not

```
int j=2, prime=1;  
scanf("%d",&N);  
while(j<N)  
{  
    if( (N % j) == 0 )  
    {  
        prime=0;  
        break; /* break out of for loop */  
    }  
    J++;  
}  
if(prime == 1)  
    printf("%d is a prime no",N);  
else  
    printf("%d is a not a prime no",N);
```

Program to generate prime numbers between given 2 limits

```
scanf("%d %d",&m,&n);
i=m;
while(i<n)
{ int prime=1, j=2;
while(j<i)
{
    if( i % j == 0)
    {
        prime=0;
        break; /* break out of inner loop */
    }
    j++;
}
if(prime == 1) printf("%d\t",i);
i++;
}
```

Skipping a part of loop-**continue** statement

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

```
→ while (.....)
{
    Statement-1;
    Statement-2;

    If(condition)
        continue;

    Statement-3;
    Statement-4;
}

Next_statement
```

```
→ do
{
    Statement-1;
    Statement-2;

    If(condition)
        continue;

    Statement-3;
    Statement-4;
} while(...);

Next_statement
```

Continue Statement

```
#include<stdio.h>
int main()
{ int j=0;
    while( j<=8)
    {
        if (j==4)
        { j++ ;
            continue;
        }
        printf("%d ", j);
        j++;
    }
    return 0;
}
```

Output: 0 1 2 3 5 6 7 8

```
#include<stdio.h>
int main()
{ int j=0;
    while( j<=8)
    {
        if (j==4)
        {
            continue;
        }
        printf("%d ", j);
        j++;
    }
    return 0;
}
```

Output: 0 1 2 3

User defined Type declarations

- **typedef**

- **Type definition** - lets you define your own identifiers.

- **enum**

- **Enumerated data type** - a type with restricted set of values.

User defined Type Declaration

- ***typedef type identifier;***

The “type” refers to an existing data type and “identifier” refers to the new name given to the data type.

- After the declaration as follows:

typedef int marks;

typedef float units;

we can use these to declare variables as shown

marks m1,m2 ; //m1 & m2 are declared as integer variables

units u1, u2; //u1 & u2 are declared as floating point variables

The main advantage of **typedef** is that we can create meaningful data type names for increasing the readability of the program.

User defined Type Declaration - enum

```
enum identifier { value1, value2...,valuen };
```

- Here, **identifier** is the name of enumerated data type or tag. And **value1**, **value2**,....,**valueN** are values of type identifier.
- By default, **value1** will be equal to 0, **value2** will be 1 and so on but, the programmer can change the default value.

```
enum card {club, diamonds, hearts, spades};
```

```
enum card {club=0, diamonds=10, hearts=20, spades=3};
```

Algorithm and Program for Fibonacci series

Algorithm : Fibonacci Series

Step 1 : Input Limit

Step 2: First \leftarrow 0,Second \leftarrow 1

Step 3: print First

Step 4: WHILE Second < Limit

begin

 Print Second

 Next \leftarrow First +

 Second

 First \leftarrow Second

 Second \leftarrow Next

end

Step 5:[End of Algorithm]

Stop

```
#include<stdio.h>
int main()
{
    int first=0, second=1;
    int limit, next;

    scanf("%d",&limit);
    printf("%d",first);

    while(second < limit)
    {
        printf("%d",second);
        next = first + second;
        first = second;
        second = next;
    }
    return 0;
}
```

Tutorial Problems

- Write a C program to print all natural numbers from n-0 in reverse using while loop
- Write a C program to find last and first digit of any number
- Write a C program to enter any number and print all its factors
- Write a C program to find LCM of two numbers
- Write a C program to convert Binary to Octal number

Session 8 Summary

- The `do` Loop
- The `break` Statement
- The `continue` Statement
- `Typedef` and `Enum`

Poll Question

Go to chat box/posts for the link to the Poll question

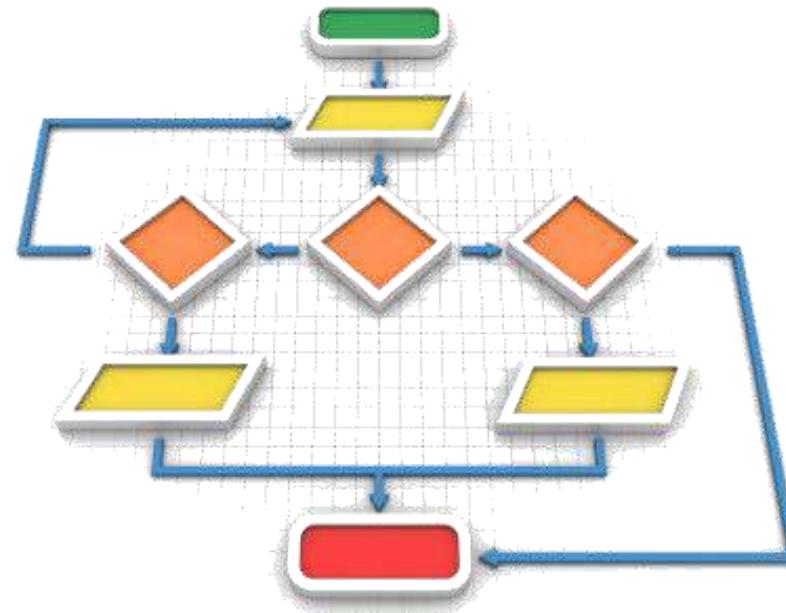
Submit your solution in next 2 minutes

Click the result button to view your score

s9_1

Control Structures

Branching & Looping



Control Structures – Review

- A **control structure** refers to the order of executing the program statements.
- The following three approaches can be chosen depending on the problem statement:
 - ✓ **Sequential (Serial)**
 - In a **Sequential approach**, all the statements are executed in the same order as it is written.
 - ✓ **Selectional (Decision Making and Branching) [if & switch statements]**
 - In a **Selectional approach**, based on some conditions, different set of statements are executed.
 - ✓ **Iterational (Repetition) [while, do-while & for statements]**
 - In an **Iterational approach** certain statements are executed repeatedly.

Session Objectives

- To learn and appreciate the following concepts
 - The `for` Statement
 - Nested `for` Loops
 - `for` Loop Variants

Session Outcomes

At the end of session student will be able to learn and understand

- The `for` Statement
- Nested `for` Loops
- `for` Loop Variants

for statement

General form:

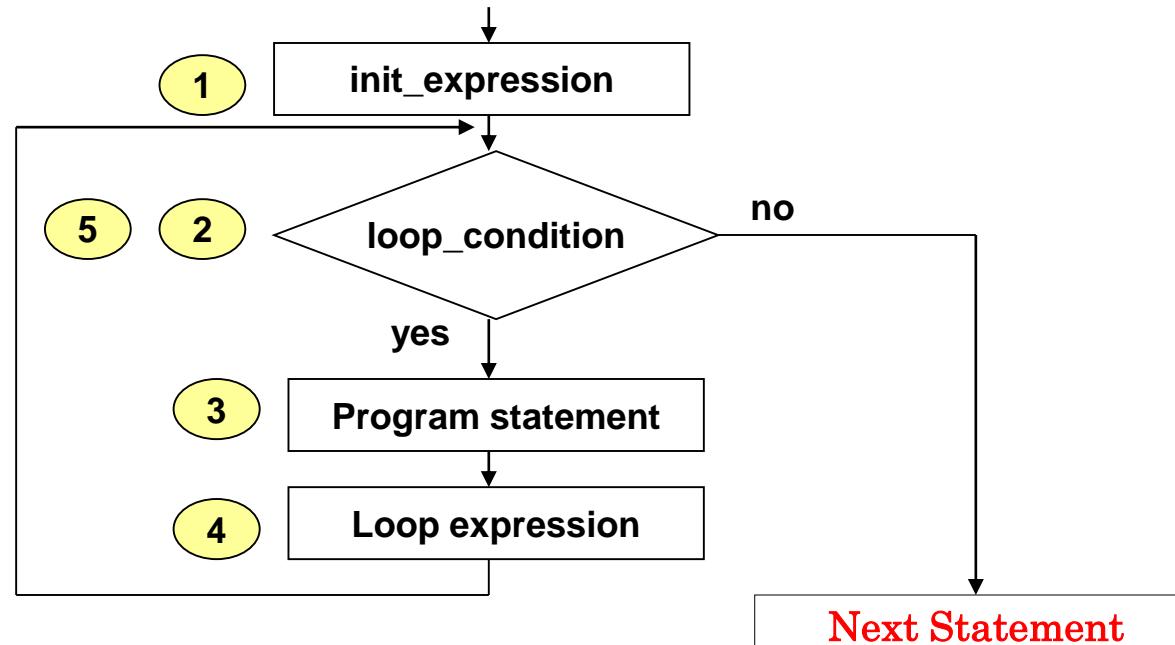
```
for (initialization; loop_condition; loop_expression)
{
    body of the loop
}
```

*Note: braces optional if
only one statement.*

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

The `for` statement

```
for ( init_expression; loop_condition;  
      loop_expression )  
{  
    program statement(s)  
}
```



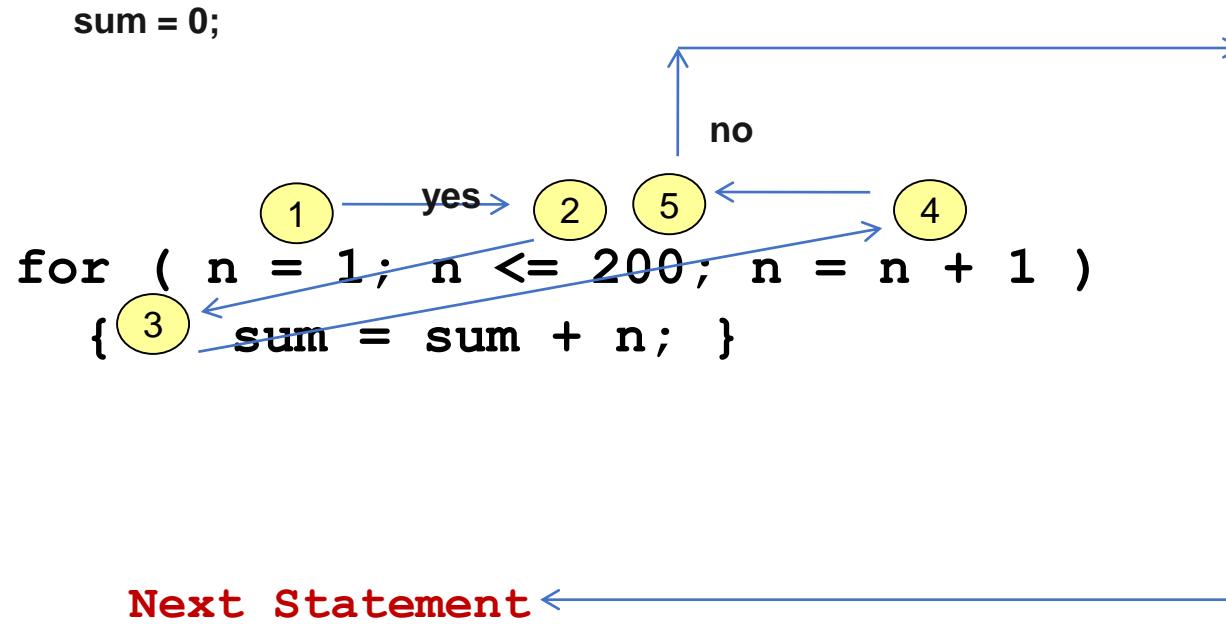
How **for** works

The execution of a for statement proceeds as follows:

1. The **initial expression** is evaluated first. This expression usually sets a variable that will be used inside the loop, generally referred to as an **index variable**, to some initial value.
2. The **looping condition** is evaluated. If the condition is not satisfied (the expression is false – has value 0), the loop is immediately terminated. Execution continues with the program statement that immediately follows the loop.
3. The **program statement** that constitutes the body of the loop is executed.
4. The **looping expression** is evaluated. This expression is generally used to change the value of the index variable
5. Return to step 2.



The `for` statement



```
for ( init_expression; loop_condition; loop_expression )  
{ program statement(s)  
}
```

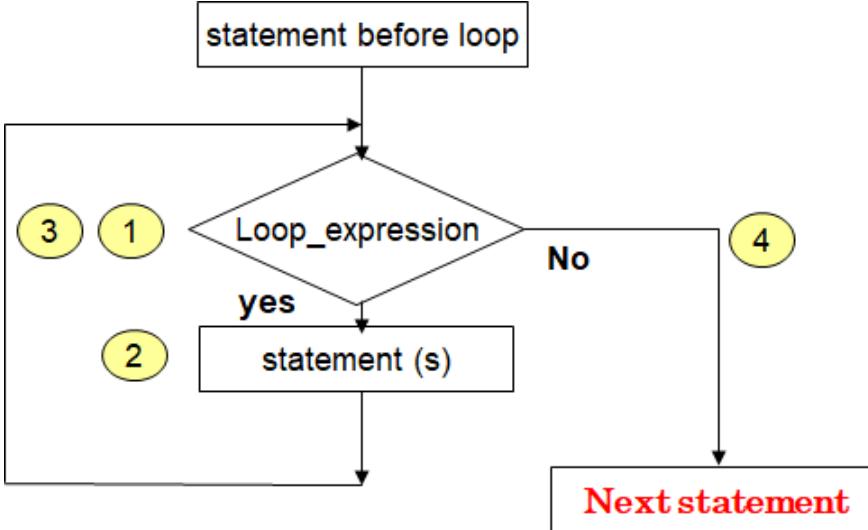
Finding sum of natural numbers up to 100

```
#include <stdio.h>
int main() {
    int n;
    int sum;
    sum=0; //initialize sum
    n=1;
    while (n <= 100)
    {
        sum= sum + n;
        n = n + 1;
    }
    printf("%d", sum);
    return 0;
}
```

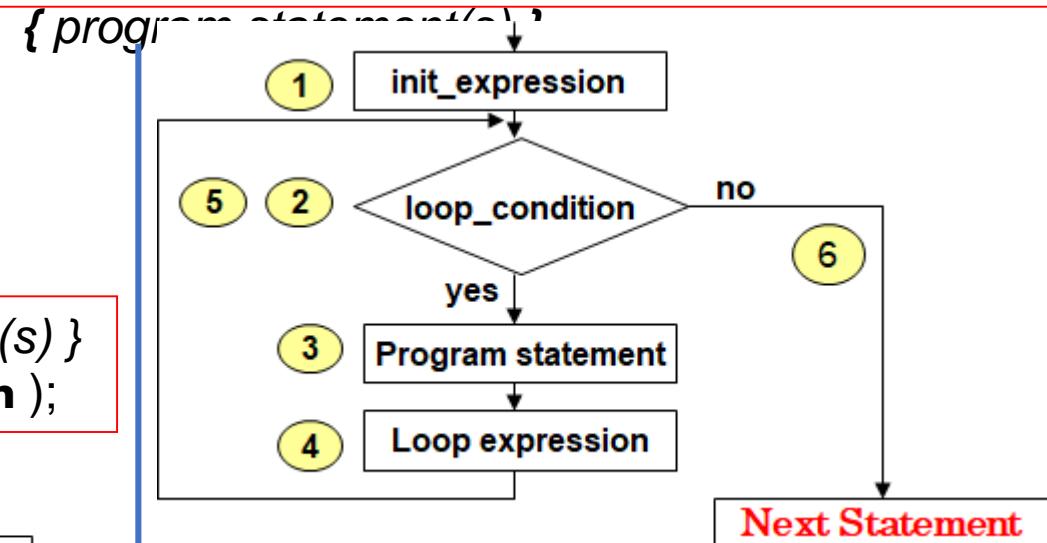
```
#include <stdio.h>
int main(){
    int n;
    int sum;
    sum=0; //initialize sum
    for(n = 1; n <= 100; n=n + 1)
    {
        sum=sum + n;
    }
    printf("%d", sum);
    return 0;
}
```

Review on decision making & looping

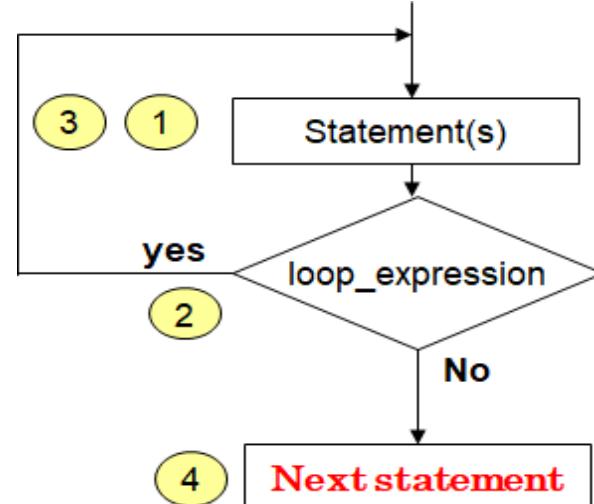
while (expression)
{ program statement(s) }



***for (init_expression; loop_condition;
loop_expression)***



***do { program statement (s) }
while (loop_expression);***

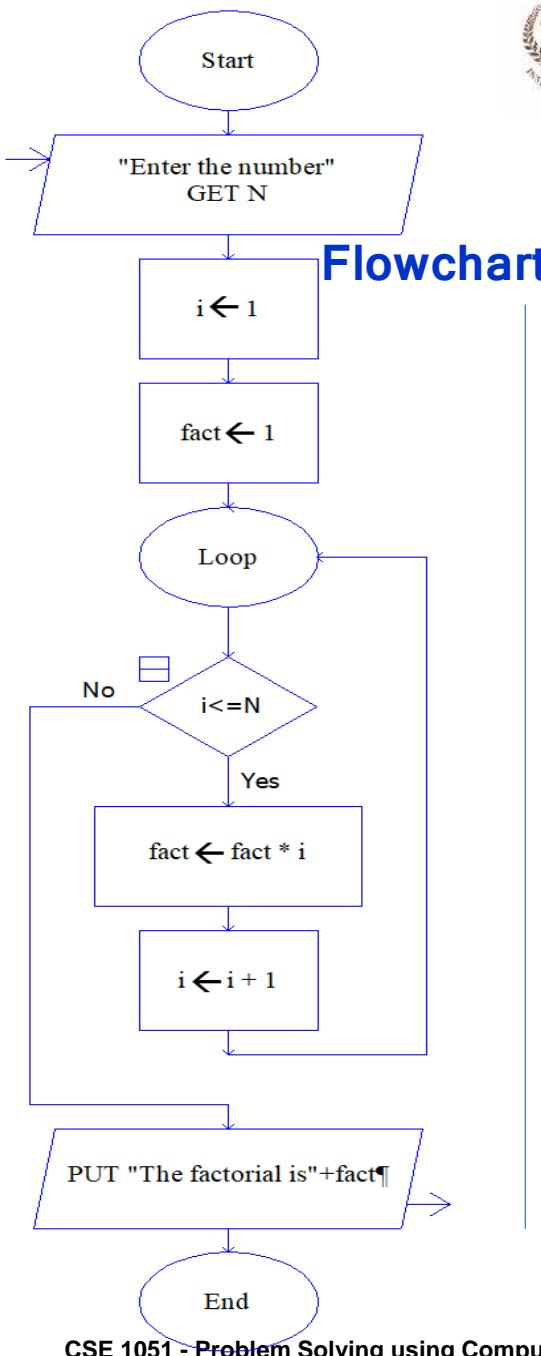


Compute the factorial of a number

Algorithm

Name of the algorithm: Compute the factorial of a number

- Step1: Start
- Step 2: Input N
- Step 3: fact \leftarrow 1
- Step 4: **for i=1 to N in step of 1 do**
begin
fact \leftarrow fact*i
end
- Step 5: Print 'fact of N=' , fact
- Step 6: [End of algorithm]
- Stop



Program

```
#include <stdio.h>
int main()
{
    int N, i, fact=1;
    printf("Enter the number");
    scanf("%d", &N);

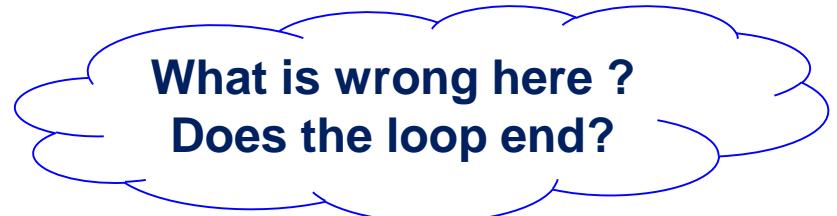
    for(i=1; i<=N; i++)
        fact=fact * i;

    printf("The factorial is %d", fact);
    return 0;
}
```

Infinite loops

It's the task of the programmer to design correctly the algorithms so that loops end at some moment !

```
// Program to count 1+2+3+4+5
#include <stdio.h>
int main() {
    int i, n = 5, sum = 0;
    for ( i = 1; i <= n; n = n + 1 ) {
        sum = sum + i;
        printf("%d",sum);
    }
    return 0;
}
```



What is wrong here ?
Does the loop end?

Nesting of **for** loop

One **for** statement can be nested within another **for** statement.

```
for (i=0; i< m; ++i)
{
    ....
    ....
    for (j=0; j < n;++j)
    {
        Statement S;
    } // end of inner 'for' statement
} // end of outer 'for' statement
```



Multiplication table for 'n' tables up to 'k' terms

```
scanf("%d %d",&n,&k);

for (i=1; i<=k; i++)
{
    for (j=1; j<=n; j++)
    {
        prod = i * j;
        printf("%d * %d = %d\n", j, i, prod);
    }
    printf("\n");
}
```

Enter n & k values: 3 5

The table for 3 X 5 is

1 * 1= 1	2 * 1= 2	3 * 1= 3
1 * 2= 2	2 * 2= 4	3 * 2= 6
1 * 3= 3	2 * 3= 6	3 * 3= 9
1 * 4= 4	2 * 4= 8	3 * 4= 12
1 * 5= 5	2 * 5= 10	3 * 5= 15

for loop variants

- Multiple expressions (*comma between...*)

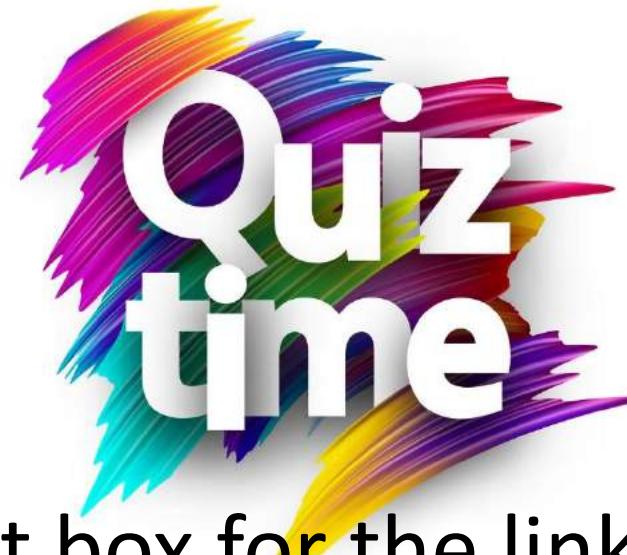
```
for(i=0 , j=10 ; i<j ; i++ , j--)
```

- Omitting fields (*semicolon have to be still...*)

```
i=0;  
for( ; i<10 ; i++ )
```

- Declaring variables

```
for(int i=0 ; i=10 ; i++)
```



Go to posts/chat box for the link to the question

submit your solution in next 2 minutes

The session will resume in 3 minutes

Which loop to choose ?

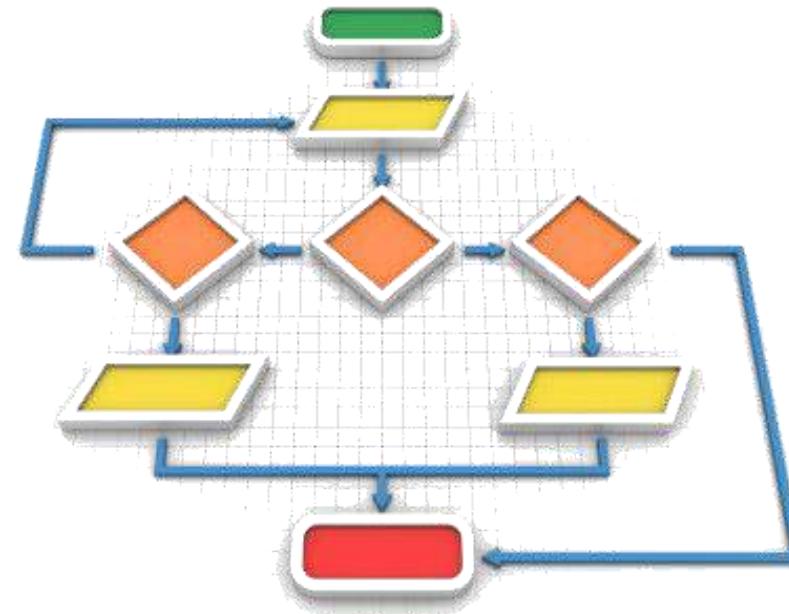
- *Criteria:* category of looping
 - Entry-controlled loop → **for, while**
 - Exit-controlled loop → **do**
- *Criteria:* Number of repetitions:
 - Indefinite loops → **while**
 - Counting loops → **for**
- You can actually rewrite any **while** using a **for** and vice versa !

Summary

- The `for` Statement
- Nested `for` Loops
- `for` Loop Variants

S9_2

Control Structures- Branching & Looping



Session Objectives

- To learn and appreciate the following concepts
 - The break with for statement
 - The continue with for statement
 - Problems on Control Structures

Session Outcomes

At the end of session student will be able to learn and understand

- The break with for statement
- The continue with for statement
- Problems on Control Structures

Exiting a loop with **break** statement in **for** statement

```

for (.....)
{ .....
.....
if(condition)
break;
.....
.....
}
.....next Stmt;

for (.....)
{ .....
for(.....)
{ .....
if(condition)
break;
... stmts of inner loop;
} // inner for loop ends
...stmts of outer loop;
} // outer for loop ends
..... next Stmt;

```

Exit From loop

Exit From inner loop

Check whether given number is prime or not - example

```
int j, prime=1;
scanf("%d", &N);
for( j=2; j<N; j++ )
{
    if( (N % j) == 0)
    {
        prime=0;
        break; /* break out of for loop */
    }
}
if (prime == 1)
printf("%d is a prime no",N);
else
printf("%d is a not a prime no",N);
```

Skipping a part of loop – **continue** in **for** statement

```

for( .........)
{
.....
.....
if(condition)
    continue;
.....
.....
}
.....next Stmts;

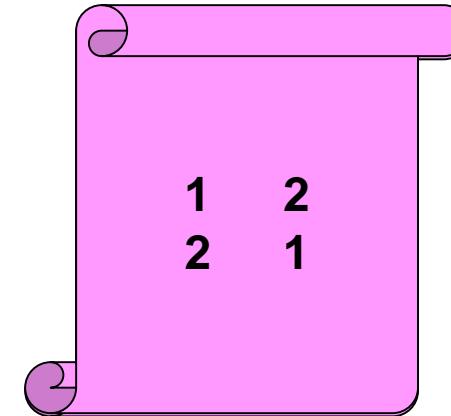
for(.....)
{
.....
for(.....)
{
    if(condition)
        break;
    ... stmts of inner loop;
}
// inner for loop ends
....stmts of outer loop;
}
// outer for loop ends
..... next Stmts;

```

Exit From loop Exit From inner loop

Skipping a part of loop – **continue** in **for** statement

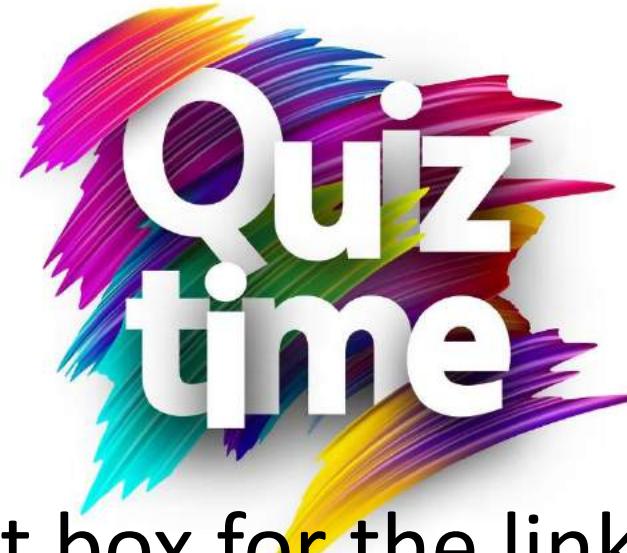
```
for ( i = 1 ; i <= 2 ; i++ )  
{  
    for ( j = 1 ; j <= 2 ; j++ )  
    {  
        if ( i == j )  
            continue ;  
        printf("\n %d\t %d\n",i, j);  
    }  
}
```



Generate prime numbers between given 2 limits

```
scanf ("%d %d", &m, &n);

for( i=m; i<=n; i++) {
    int prime=1;
    for( j=2; j<i; j++ ) {
        if( i % j == 0)
            {
                prime=0;
                break; /* break out of inner loop */
            }
    }
    if (prime == 1) printf ("%d\t", i);
}
```



Go to posts/chat box for the link to the question

submit your solution in next 2 minutes

The session will resume in 3 minutes



Problems on Control Structures



Factors of a Positive Integer

```
#include <stdio.h>
int main() {
    int num, i;
    printf("Enter a positive integer: ");
    scanf("%d", &num);
    printf("Factors of %d are: ", num);
    for (i = 1; i <= num; ++i) {
        if (num % i == 0) {
            printf("%d ", i);
        }
    }
    return 0;
}
```

For num = 20

OUTPUT

Factors of 20 are: 1 2 4 5 10 20

```
Enter a positive integer: 20
Factors of 20 are: 1 2 4 5 10 20
```



Armstrong no's for a given limit 'n'

```
scanf ("%d", &lim);
printf ("The armstrong numbers are:");
for(n=1;n<lim;n++) {
    sum = 0; //initialized for each number
    num = n; //store it for comparison
    while(num>0) {
        dig = num%10; //extracting digits
        sum = sum+pow(dig,3); //sum of cube of digits
        num = num/10; //remaining digits for next iteration
    }
    if(sum == n)
        printf ("%d\n\t",n);
}
```

Armstrong Number
 Σ (cubes of digits)= num
 $3^3 + 7^3 + 1^3 = 371$

```
Enter the limit: 400
The armstrong numbers are:
1
153
370
371
```

Sine series for a given 'n' terms & angle 'x'

```
# define PI 3.141592
```

```
scanf ("%d %f", &n , &x) ;
no=x;
x=x*PI/180.0; // degrees to radians
term=x; // first term value
sum=x; //term stored in sum
for (i=1;i<=n;i++)
{
    term= term* (((-1)*x*x)/(2*i*(2*i+1)));
    sum+=term;
}
printf("Library value of Sin(%.2f) = %.2f ", no, sin(x));
printf("\nSin (%.2f) = %.2f", no, sum);
```

Sine series

$$\sin(x) = x - x^3/3! + x^5/5! - \dots x^n/n!$$

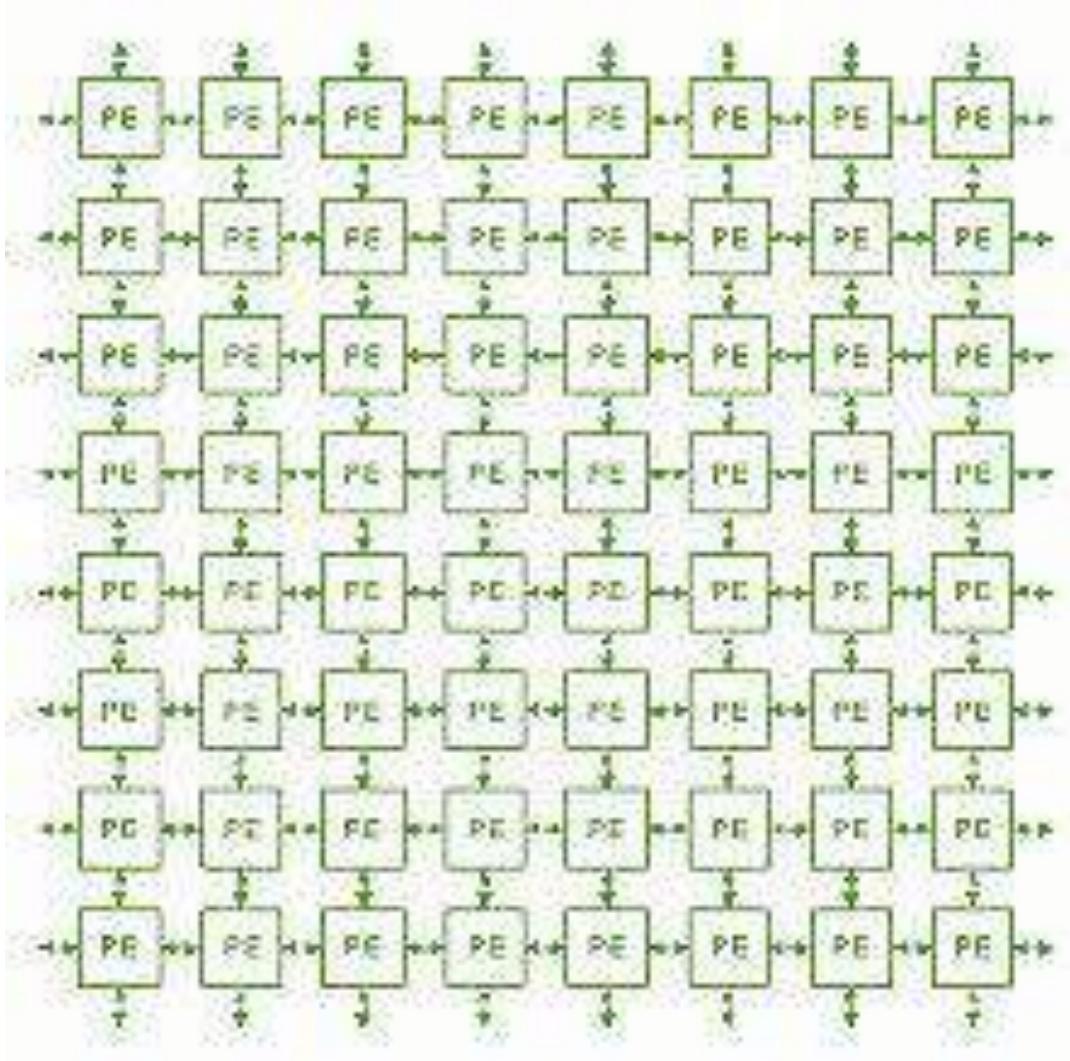
```
Enter the values for number of terms(n) and angle (x):
5
90
Library value of Sin(90.00) = 1.00
Computed Sin (90.00) = 1.00
```

Tutorial Problems

1. Write a C program to count number of digits in any number
2. Write a C program to find last and first digit of any number
3. Write a C program to enter any number and print all its factors
4. Write a C program to find LCM of two numbers
5. Write a C program to convert Binary to Octal number

• Session 9 Summary

- The `for` Statement
- Nested `for` Loops
- `for` Loop Variants
- The `break` with `for` statement
- The `continue` with `for` statement
- Problems on Control Structures



S10_1
1 D - A r r a y s

Objectives

To learn and appreciate the following concepts:

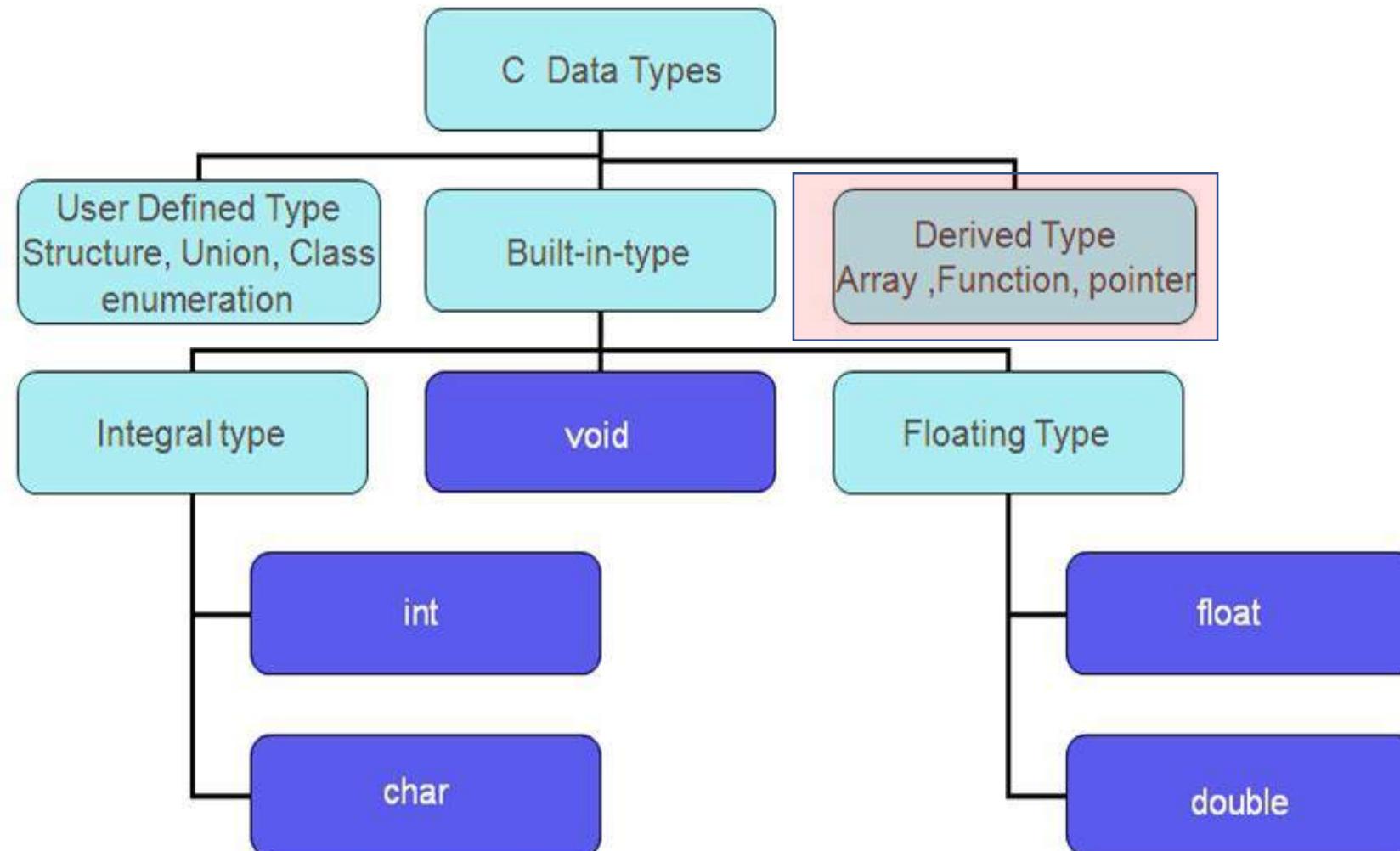
- Declare, initialize and access 1D array.
- Write programs using common data structures namely arrays and strings and solve problems.

Session outcome

- At the end of session student will be able to
 - Declare, initialize and access 1D array
 - Write programs using 1D array



Data types - Revisit





Arrays

- **An array is a group of related data items that share a common name.**
- The array elements are placed in 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..

Arrays

Array Declaration:

```
data-type name [size];
```

where data-type is a valid data type (like int, float, char...)

- ✓ 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.
 - For example an array salary containing 5 elements is declared as follows

```
int salary [5];
```

Arrays - One Dimensional



Name	roll[0]	roll[1]	roll[2]	roll[3]	roll[4]	roll[5]	roll[6]	roll[7]
Values	12	45	32	23	17	49	5	11
Address	1000	1002	1004	1006	1008	1010	1012	1014
<u>1-D Array memory arrangement</u>								

- 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. **roll[0], roll[1].... or 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**.



Arrays - 1D

Total size:

The Total memory that can be allocated to 1Darray is computed as

Total size =size *(**sizeof(data_type)**);

where size → number of elements in 1-D array

data_type → basic data type

sizeof() → is an unary operator which returns the size of data type in bytes.

Arrays - 1D

How to read & display the values of an array and store it !

```
int main() {
    int arr[50],n; // declaration of 'arr'
    printf(" enter value of n\n"); // no of elements
    scanf("%d", &n);           // reading the limit into n
    for(int i=0;i<n;i++)
    {
        scanf("%d", &arr[i]); // reading n elements
    }
    for(int j=0; j<n;j++) //displaying n elements
    {
        printf("%d",arr[j]);
        printf("\t");
    }
    return 0;
}
```



Initializing one-dimensional array

```
int number[3] ={0,0,0}; or {0} ;
```

→ declares the variable number as an array of size 3 and will assign 0 to each element.

```
int age[ ] ={16,25,32,48,52,65};
```

→ declares the age array to contain 6 elements with initial values 16, 25, 32, 48, 52, 65 respectively



Initializing one-dimensional array

Initialize all the elements of an integer array ‘values’ to zero

`int values[20];`

Begin for loop

Initialize counter

Set limit for counter

`for (int i=0; i<20; i++)`

Initialize element in array ‘values’

`values[i]=0;`

Increment counter



Printing one-dimensional array

For example

```
int x[3] = {9,11,13};
```

```
printf("%d\n",x[0]);
```

Output:

9

```
printf("%d\n",x[1]);
```

11

```
printf("%d\n",x[2]);
```

13

or

```
int x[3] = {9,11,13};
```

```
for (int i = 0; i<3; i++)
```

```
printf("%d\n",x[i]);
```

Program to read n elements into an array and print it

```
int a[10], i, n;  
  
printf("enter no of numbers");  
  
scanf("%d",&n);  
  
printf("enter n numbers \n");  
  
for(i=0;i<n;i++)  
  
scanf ("%d\n", &x[i]);  
  
printf("\nNumbers entered are:\n");  
  
for(i=0;i<n;i++)  
  
printf("%d\n", a[i]);
```

Output:

enter no of numbers

3

enter n numbers

9

11

13

Numbers entered are:

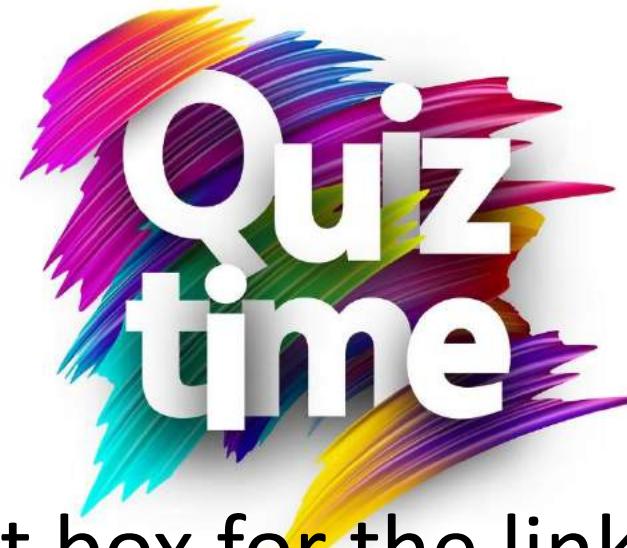
9

11

13

Program to add two array elements and store the corresponding sum elements in another array

```
int a[10], b[10], c[10], n, m, i;  
printf("enter no. of numbers in first array\n");  
scanf("%d", &n);  
//first array reading  
for (i=0; i<n; i++)  
    scanf ("%d", &a[i]);  
printf("enter no of numbers in second array\n");  
scanf("%d", &m);  
//second array reading  
for (i=0; i<m; i++)  
    scanf ("%d", &b[i]);  
  
if(m==n)  
{    //addition  
    for (i=0; i<m; i++)  
        c[i]=a[i]+b[i];  
  
printf("Sum of given array elements\n");  
  
    for(i=0;i<n;i++)  
        printf("%.d\n",c[i]);  
}  
else  
printf("cannot add");  
}
```



Go to posts/chat box for the link to the question

submit your solution in next 2 minutes

The session will resume in 3 minutes



Write a program to reverse an array

```
int a[20], i, j, n, temp;  
  
printf("enter n \n");  
  
scanf("%d", &n);  
  
printf("\n Enter values for an array");  
  
for(i=0;i<n;i++)  
  
scanf("%d", &a[i]);
```

Example : a[]={1, 2, 3, 4, 5}

Enter values

n=5

1 2 3 4 5

Reversed array

5 4 3 2 1

Array	Reversed
a[0]=1	a[0]=5
a[1]=2	a[1]=4
a[2]=3	a[2]=3
a[3]=4	a[3]=2
a[4]=5	a[4]=1

Contd...



Reversing an array

```
for(i=0, j=n-1; i<n/2; i++, j--)  
{  
    temp=a[i] ;  
    a[i]=a[j] ;  
    a[j]=temp ;  
}  
printf("\n Reversed array: \n");  
for(i=0;i<n;i++)  
printf("%d\t", a[i]);  
}
```

Example :

a[]={1, 2, 3, 4, 5}

Output:

Enter values for an array
n=5

1 2 3 4 5

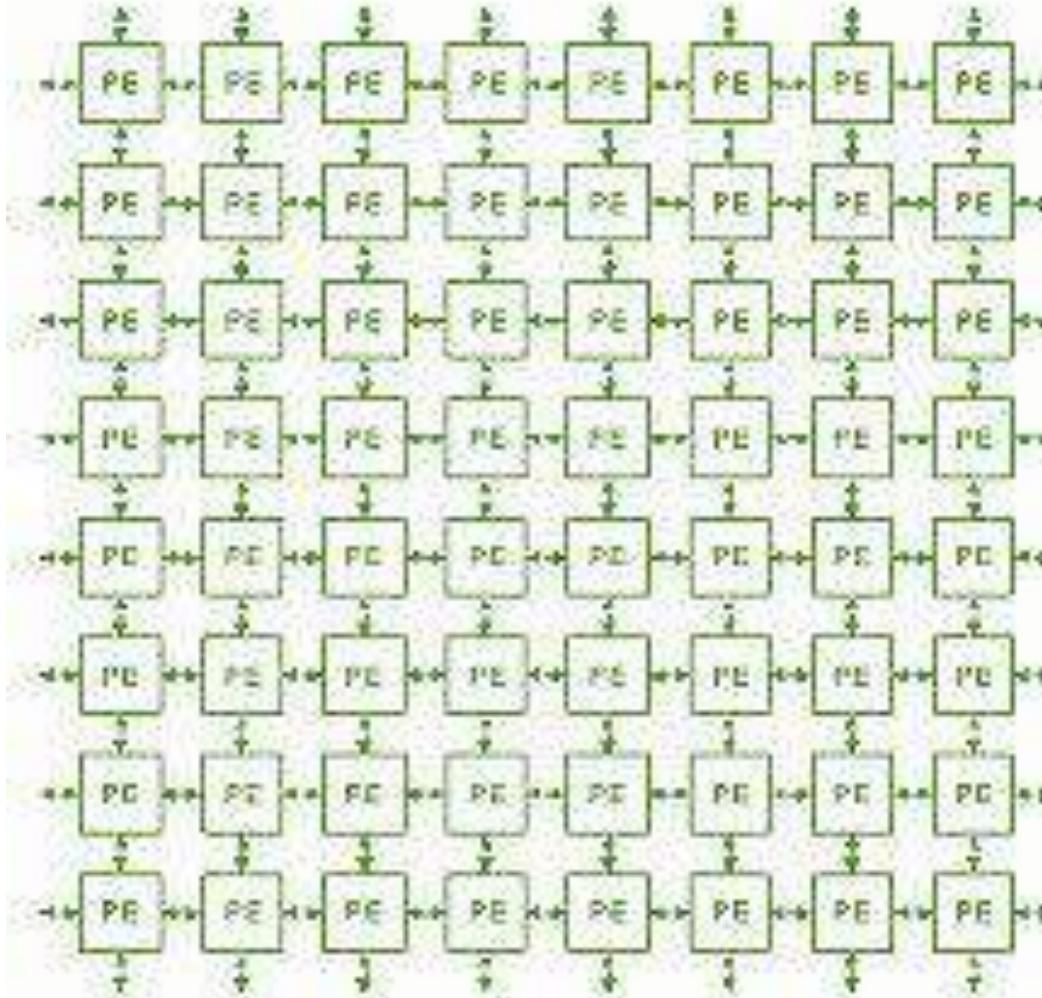
Reversed array

5 4 3 2 1



Summary

- Arrays
- 1 Dimensional arrays (lists)
- Problems on 1D arrays



S10_2
1 D - A r r a y s

Objectives

To learn and appreciate the following concepts:

- Declare, initialize and access 1D array.
- Write programs using common data structures namely arrays and strings and solve problems.

Session outcome

- At the end of session student will be able to
 - Declare, initialize and access 1D array
 - Write programs using 1D array

Arrays – recap

1D Array:

- Syntax: **type array_name[size];**
- Memory Requirement:
Total size =size *(sizeof(data_type));
- Initialization:
type array-name [size]={list of values}
- Write and Read:
for(i=0;i<n;i++) **for(i=0;i<n;i++)**
scanf("%d",&a[i]); **printf("%d\n",a[i]);**

WAP to insert an element to an array at a given position

```
int a[100], n,i, pos, ele;  
scanf("%d",&n); // number of elements  
printf("\nEnter the elements of array:");  
for(i=0;i<n;i++)  
    scanf ("%d" , &a[i] ) ;  
  
printf("\nEnter the element and position of insertion:");  
scanf ("%d %d" , &ele , &pos) ;  
  
for(i=n; i>=pos; i--) //shift the elements to right  
    a[i]=a[i-1];  
  
a[pos-1] = ele; //ele is inserted at the specified pos.  
  
n = n + 1; // increment the count of no of elements  
  
printf("\nThe array after insertion is:");  
for(i=0;i<n; i++) printf("%d\n",a[i]);
```

Example : insert 9 at 2nd position
a[]={1, 2, 3, 4, 5}

New array after inserting 9 :
a[]={1, 9, 2, 3, 4, 5}

WAP to delete an element from an array

```
printf("enter no of numbers");
scanf ("%d", &n) ;
printf("enter n numbers \n");
for(i=0;i<n;i++)
    scanf ("%d", &a[i]) ;
printf("enter the position at which the element to be deleted");
scanf ("%d", &pos) ;
for(i=pos-1; i<n-1; i++)
    a[i] =a[i+1]; //shift the elements to left
n = n-1;      //decrement the count of no of elements
for(i=0;i<n;i++)
    printf ("%d", a[i]);
```

Example : delete ele at 2nd position
a[]={1, 2, 3, 4, 5}

New array after deleting 2:
a[]={1, 3, 4, 5}

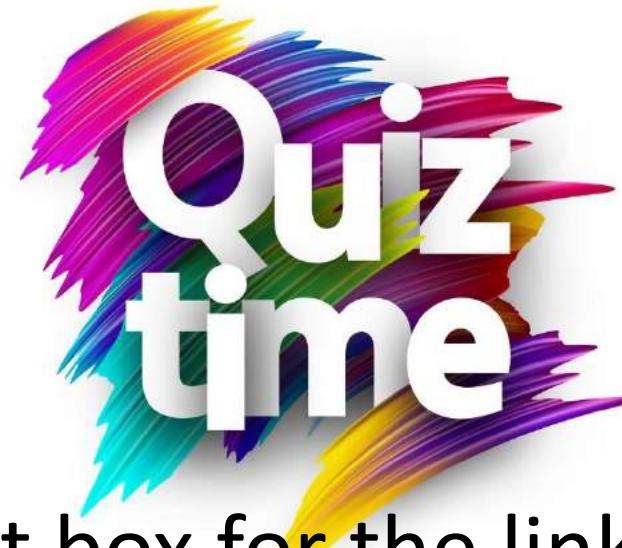
Insert an element into a sorted array

Read array elements (in sorted order) & element 'ele' to be inserted

```
//finding position  
for(i=0;i<n;i++)  
  
    if (ele<a[i])  
        break;  
  
pos = i+1; //position of insertion  
  
for(i=n; i>=pos; i--) //shift the elements to right  
  
    a[i]=a[i-1];  
  
a[pos-1] = ele; //ele is inserted at the specified pos.  
  
n = n + 1; // increment the count of no of elements
```

Example: insert 3 into the array
a[] = {1, 2, 4, 5, 6}

New array after inserting 3 :
a[] = {1, 2, 3, 4, 5, 6}



Go to posts/chat box for the link to the question
submit your solution in next 2 minutes
The session will resume in 3 minutes



Tutorials on Array

- Write a C program to find average of an 1-D array.
- Write a C program to find second largest element in an array.
- Write a C program to find union and intersection of two arrays.

Largest and second largest element in an array

```
/* assume first element of the array as the largest & second largest */  
largest1 = array[0];  
largest2 = array[1];  
  
for (i = 1; i < MAX; i++)  
{  
    if (array[i] >= largest1)  
    {  
        largest2 = largest1;  
        largest1 = array[i];  
    }  
    else if (array[i] > largest2)  
    {  
        largest2 = array[i];  
    }  
}
```

Example: array[] = {22,44, 34, 9, 21}

44 is largest
34 is second largest



Summary

- Problems on 1D arrays



0	1	2	3	...	M-2	M-1	M+1	M+2	M+3
0	*	*	*	...	*	*	*	*	0
1	0			...					0
2	*			...					*
3	*	*	*	...	*	*	*	*	*
...				...					
M-2				...					
M-1	0			...					0
M+1	0	0	+	+	0	+	+	0	0
M+2	0	0	+	+	0	+	+	0	0



S11_1
CHARACTER ARRAYS
STRINGS

Objectives

To learn and appreciate the following concepts

- Strings definition, declaration, initialization
- Reading Strings
- Programs using strings

Session outcome

At the end of session student will be able to

- Declare and initialize strings
- Write programs using strings

Strings

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.

Strings

Declaration and initialization

```
char string_name[size];
```

The size determines the number of characters in the `string_name`.

For example, consider the following array:

```
char name [20];
```

is an array that can store up to 20 elements of type `char`.

It can be represented as:



Strings

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

name

H e l l o \0

name

Merry Christmas \0



Initialization of null-terminated character sequences

- **arrays of characters or strings** are ordinary arrays that follow the same rules of arrays.

For example

To initialize an array of characters with some predetermined sequence of characters one can initialize like any other array:

```
char myWord[ ] = { 'H', 'e', 'T', 'I', 'o', '\0' };
```

Initialization of null-terminated character sequences

- Arrays of char elements have an additional methods to initialize their values: **using string literals**
- “**Manipal** ” is a constant string literal.

For example,

char result[14] =“Manipal”;

- **Double quoted ("")** strings are literal constants whose type is in fact a null-terminated array of characters.

So string literals enclosed between double quotes always have a null character ('\0') automatically appended at the end.

Initialization

- **Initialization:**

char myWord [] = { 'H', 'e', 'l', 'l', 'o', '\0' };

char myWord [] = "Hello";

- In both cases the array of characters **myword** is declared with a size of 6 elements of type **char**:
 - ✓ The 5 characters that compose the word **"Hello"** plus a final null character (**'\0'**) which specifies the end of the sequence and that,
 - ✓ In the second case, when using double quotes ("") null character (**'\0'**) is appended automatically.

Example

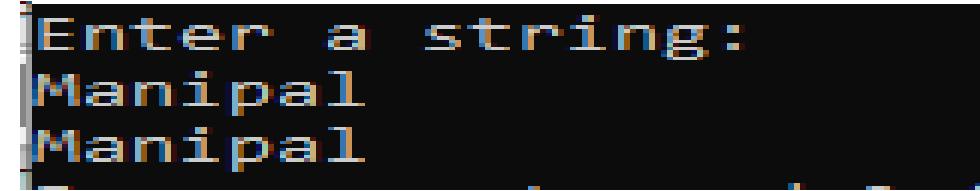
```
#include<stdio.h>

int main()      {
    char greeting[]="Hello ";
    char yourname[80];
    printf("enter your name:");
    scanf("%s",yourname);
    printf("%s,%s",greeting, yourname);
    return 0;    }
```

```
enter your name:David
Hello ,David
Process returned 0 (0x0)  execution time : 24.636 s
Press any key to continue.
```

Example

```
#include <stdio.h>
int main()
{
    const int MAX = 80;          //max characters in string
    char str[MAX];              //string variable str
    printf("Enter a string: \n");
scanf("%s",str);           //put string in str
printf("%s",str);           //display string from str
    return 0;
}
```



Enter a string:
Manipal
Manipal

Reading Embedded Blanks

To read everything that you enter from the keyboard until the ENTER key is pressed (including space).

Syntax:

`gets(string) ;`

To write/display that you entered.

Syntax:

`puts(stringname) ;`

Example

```
#include <stdio.h>
int main()
{
    const int MAX = 80; //max characters in string
    char str[MAX]; //string variable str
    printf("\nEnter a string: ");
gets(str);
    printf(" the string is \n");
    puts(str);
    return 0;
}
```

```
Enter a string: Technology
the string is
Technology
```

The function will continue to accept characters until enter key is pressed.



Reading multiple lines: Example

```
#include <stdio.h>

int main() {

    const int MAX = 2000; //max characters in string

    char str[MAX]; //string variable str

printf("\nEnter a string:\n");

scanf("%[^#]",str); //read characters to str until a # character is encountered

printf("You entered:\n");

printf("%s",str);

return 0;

}
```

The function will continue to accept characters until termination key (#) is pressed.



Go to posts/chat box for the link to the question

submit your solution in next 2 minutes

The session will resume in 3 minutes



Count the number of characters in a string

```
#include <stdio.h>
int main()
{
    const int Max = 100;
    char sent[Max];
    int i=0, count=0;
    printf("enter sentence \n");
    gets(sent);
    puts(sent);
```

```
while(sent[i]!='\0')
{
    count++;
    i++;
}
printf("\n no of characters = %d", count);
return 0;
```

```
enter sentence
Manipal Institute of Technology
```

```
enter sentence
Manipal
Manipal

no of characters = 7
```



Count the number of words in a sentence

```
#include <stdio.h>
int main()
{
    const int MAX = 100;
    char sent[MAX];
    int i=0,count=1;
    printf("enter sentence \n");
    gets(sent);
    printf("\n");
```

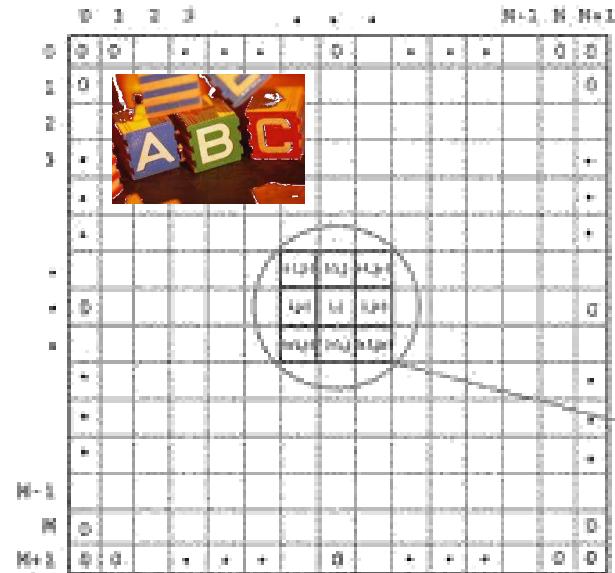
```
while(sent[i]!='\0')
{
    if ( ( sent[i] ==' ') && ( sent[i+1]!=' ' ) )
        count++;
    i++;
}
printf("\n no. of words =%d", count);
return 0;
```

```
enter sentence
Manipal Institute of Technology

no. of words =4
```

Summary

- Strings definition, declaration, initialization
- Reading Strings
- Programs using strings



S11_2
**STRINGS and STRING
HANDLING FUNCTIONS**

Objectives

To learn and appreciate the following concepts

- String
- String Handling Functions
- Programs using strings

Session outcome

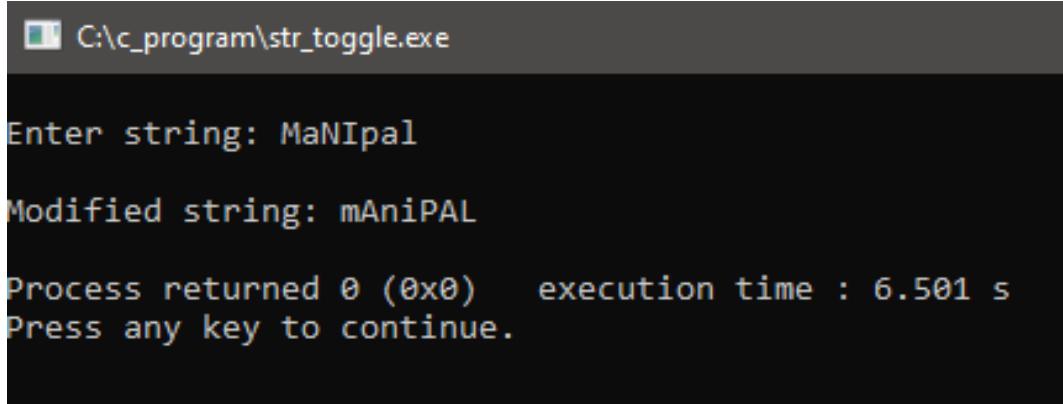
At the end of session student will be able to

- Understand string and String Handling Functions
- Write programs using strings

Input a string and toggle the case of every character in the input string.

```
#include<stdio.h>
int main()
{char string[100];
int i;
printf("\nEnter string: ");
gets(string);
for(i=0;string[i]!='\0';i++)
{
if(string[i]>='A'&&string[i]<='Z')
string[i]+=32;
else if(string[i]>='a'&&string[i]<='z')
string[i]-=32;
}
```

```
printf("\nModified string: ");
puts(string);
return 0;
}
```



```
C:\c_program\str_toggle.exe

Enter string: MaNIpal

Modified string: mAniPAL

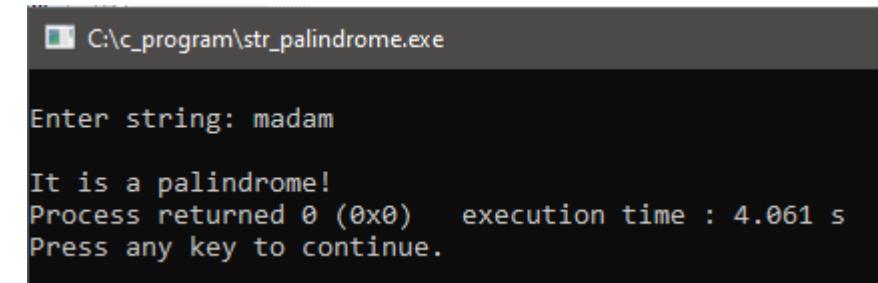
Process returned 0 (0x0) execution time : 6.501 s
Press any key to continue.
```



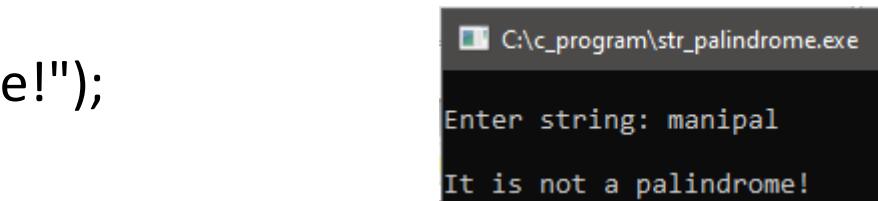
Check whether the given string is a palindrome or not.

```
#include<stdio.h>
int main()
{
    char string[100];
    int i,n=0,flag=0;
    printf("\nEnter string: ");
    gets(string);
    for(i=0;string[i]!='\0';i++)
        n++;
}
```

```
for(i=0;i<n/2;i++)
{
    if(string[i]!=string[n-1-i])
    {
        flag=1;
        break;
    }
    if(flag==0)
        printf("\nIt is a palindrome!");
    else
        printf("\nIt is not a palindrome!");
    return 0;
}
```



```
C:\c_program\str_palindrome.exe
Enter string: madam
It is a palindrome!
Process returned 0 (0x0)  execution time : 4.061 s
Press any key to continue.
```



```
C:\c_program\str_palindrome.exe
Enter string: manipal
It is not a palindrome!
Press any key to continue.
```

Library functions: String Handling functions (built-in)

- 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()`

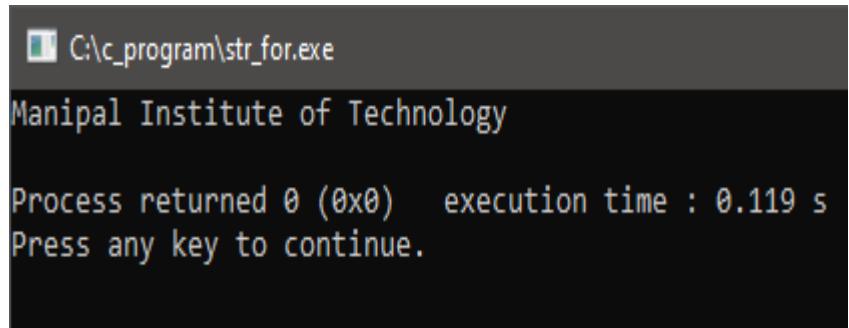
- String length can be obtained by using the following function

`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.
Eg: `printf("%d",strlen("Manipal"));` prints out 7.

Copies a string using a for loop

```
#include <stdio.h>
#include<string.h>
int main()
{
    char str1[ ] = "Manipal Institute of Technology";
    const int MAX = 80;                      //size of str2 buffer
    char str2[MAX]; //empty string
    int j;
    for(j=0 ; j<strlen(str1); j++)          //copy strlen characters
        str2[j] = str1[j];                  // from str1 to str2
    str2[j] = '\0';                         //insert NULL at end
    printf("%s\n",str2);                   //display str2
    return 0;
}
```



```
C:\c_program\str_for.exe
Manipal Institute of Technology

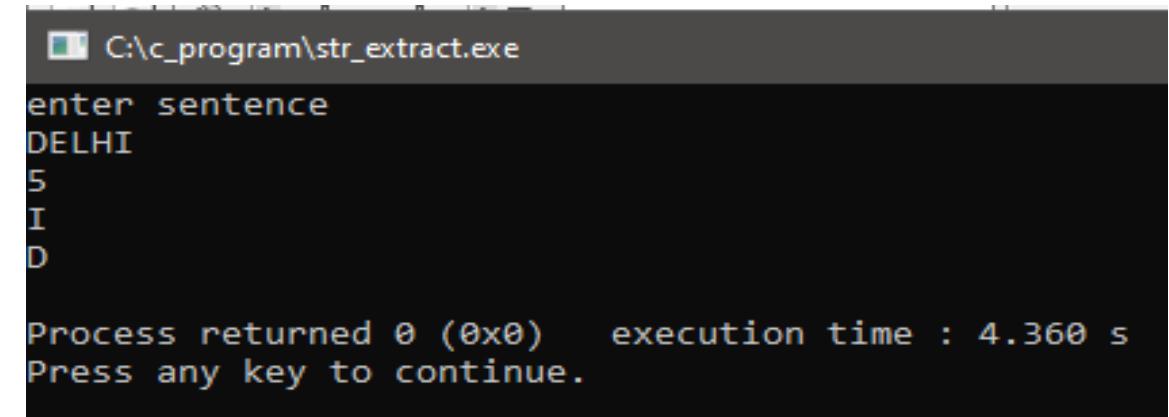
Process returned 0 (0x0)  execution time : 0.119 s
Press any key to continue.
```

Extracting a character from a string

```
#include <stdio.h>
#include<string.h>
int main()
{
const int MAX = 100;
char sent[MAX];
int len;
printf("enter sentence \n");
gets(sent);
len=strlen(sent);
printf("%d\n",len);
printf("%c\n",sent[len-1]);
printf("%c\n",sent[0]); }
```



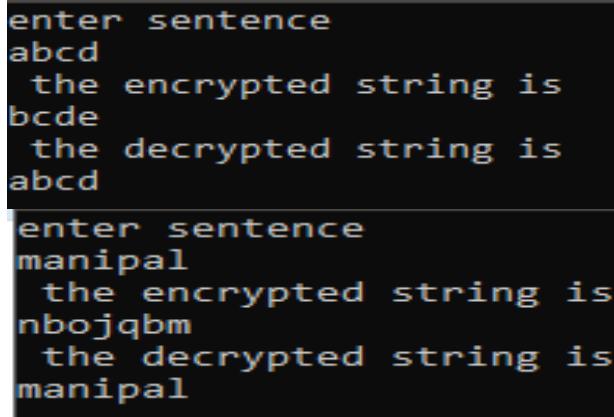
sent[0] sent[1] sent[2] sent[3] sent[4]



```
C:\c_program\str_extract.exe
enter sentence
DELHI
5
I
D
Process returned 0 (0x0)  execution time : 4.360 s
Press any key to continue.
```

To encrypt and decrypt a string

```
#include <stdio.h>
#include<string.h>
int main()
{
const int MAX = 100;
char sent[MAX];
int len,i;
printf("enter sentence \n");
gets(sent);
for(i=0;sent[i]!='\0';i++)
sent[i]=sent[i]+1;
printf(" the encrypted string is \n");
puts(sent);
for(i=0;sent[i]!='\0';i++)
sent[i]=sent[i]-1;
printf(" the decrypted string is \n");
puts(sent);}
```



The terminal window displays two examples of string manipulation. In the first example, the user enters "abcd", which is then encrypted by incrementing each character by 1, resulting in "bcde". This encrypted string is then decrypted by decrementing each character by 1, returning to the original "abcd". In the second example, the user enters "manipal", which is encrypted to "nbojqbm" by shifting each character forward in the ASCII sequence. This encrypted string is then decrypted back to "manipal".

Library function: `strcpy()`

Copying a String the EASY WAY using

`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.

The size of the array city1 should be large enough to receive the contents of city2.

strcpy(): Example

```
#include <stdio.h>

#include<string.h>

int main() {

char str1[ ] = "Tiger, tiger, burning bright\n"
    "In the forests of the night";

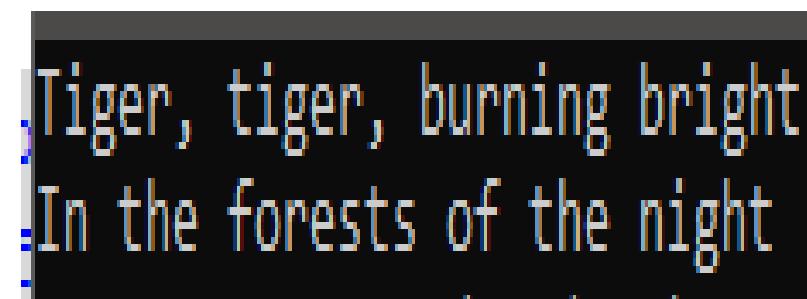
const int MAX = 80; //size of str2 buffer

char str2[MAX]; //empty string

strcpy(str2, str1); //copy str1 to str2

printf("%s",str2);//display str2

}
```



The terminal window displays the output of the C program. It shows two lines of text: "Tiger, tiger, burning bright" and "In the forests of the night". The text is colored in blue and orange, likely due to syntax highlighting in the terminal or code editor.

Summary

- Strings and String Handling Functions
- Programs using strings

S12_1

String handling Functions

Objectives

To learn the following concepts

- String handling function

Session outcome

At the end of session student will be able to understand

- The String handling functions.
- Strcmp
- strcat

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.

If the value is negative, string1 is alphabetically above string2.

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 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.

Concatenation of 2 strings

```
#include <stdio.h>
#include <string.h>
int main()
{   char s1[40], s2[50];
    printf("\nEnter the first string: ");
    gets(s1);
    printf("\nEnter the second string: ");
    gets(s2);
    strcat(s1, s2);
    printf("\nConcatenated string is: ");
    printf("%s",s1);
    return 0; }
```

```
Enter the first string: Manipal
Enter the second string: Institute
Concatenated string is: ManipalInstitute
```

Reversing a string

```
#include<stdio.h>
int main()
{
char str[70];
char temp;
int i, n=0;
printf("\nEnter the string:");
gets(str);
for(i=0;str[i]!='\0';i++)
    n++;
}
```

```
for(i=0;i<n/2;i++)
{
    temp=str[i];
    str[i]=str[n-i-1];
    str[n-i-1]=temp;
}
printf("\nReversed string is:");
puts(str);
return 0;
}
```

Enter the string: Manipal

Reversed string is:lapinaM

Print an alphabet in decimal [ASCII] & character form

```
#include<stdio.h>
int main()
{
char c;
printf("\n");
for(c=65;c<=122;c++)
{
if(c>90 && c<97)
    continue;
    printf("%c", c);
    printf("-");
    printf("%d\t",(int)c);
}
printf("\n");
return 0;
}}}
```

A-65	B-66	C-67	D-68	E-69	F-70	G-71	H-72	I-73	J-74	K-75	L-76	M-77	N-78	O-79
P-80	Q-81	R-82	S-83	T-84	U-85	V-86	W-87	X-88	Y-89	Z-90	a-97	b-98	c-99	
d-100	e-101	f-102	g-103	h-104	i-105	j-106	k-107	l-108	m-109	n-110	o-111	p-112	q-113	
r-114	s-115	t-116	u-117	v-118	w-119	x-120	y-121	z-122						

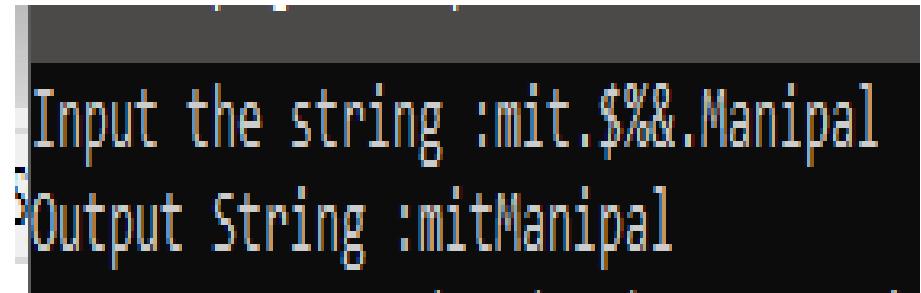
Write a C Program to input a String & store their Ascii Values in an Integer Array & print the Array.

```
#include<stdio.h>
void main()
{ char string[20]; int asc[20];
int n, count = 0;
printf("Enter the no of characters present in an array \n ");
scanf("%d", &n);
printf(" Enter the string of %d characters \n ", n);
scanf("%s", string);
while (count < n)
{ asc[count]=string[count];
printf(" %c = %d\n", string[count], asc[count] );
++ count ;}}
```

```
Enter the no of characters present in an array
5
Enter the string of 5 characters
APpLE
A = 65
P = 80
p = 112
L = 76
E = 69
```

Write a C program to remove special characters and digits leaving the alphabets un altered in a given string.

```
#include <stdio.h>
int main(){
char str[150];
int i,j;
printf("Input the string :");
scanf("%s",str);
for(i=0; str[i]!='\0'; ++i){
while (!((str[i]>='a'&&str[i]<='z') || (str[i]>='A'&&str[i]<='Z' || str[i]=='\0'))){
for(j=i;str[j]!='\0';++j){
str[j]=str[j+1];}
str[j]='\0'; }
printf("Output String :%s", str);
return 0;}
```



Write a C program to read a sentence and replace all the alphabets in the input sentence with '#' whose ASCII value is even and with '%', whose ASCII value is odd. Display the resultant sentence.

```
#include<stdio.h>
#include<string.h>
int main()
{
const int Max = 100;
char sent[Max];
int i=0,count=0;
printf("Enter sentence \n");
gets(sent);
puts(sent);
```

```
while(sent[i]!='\0') {
    if( (sent[i]>='a'&& sent[i]<='z') ||
        sent[i]>='A' && sent[i]<='Z')) {
        if(sent[i]%2==0)
            sent[i]='#';
        else
            sent[i]='%'; }
    i++; }
printf("\n Modified sentence is %s\n",sent);
return 0;}
```

```
Enter sentence
APpLE
APpLE
Modified sentence is %##%
```

Arrange 'n' names in alphabetical order

(hint: use string handling function-*strcpy*)

```
#include<stdio.h>
#include<string.h>
int main()
{
char a[10][10],temp[10];
int n,i,j;
printf("\nEnter how many names: ");
scanf("%d",&n);
printf("\nEnter the names: \n");
fflush(stdin);
for(i=0;i<n;i++)
gets(a[i]);
```

```
for(i=0;i<n-1;i++)
for(j=i+1;j<n;j++){
if(strcmp(a[i],a[j])>0){
strcpy(temp,a[i]);
strcpy(a[i],a[j]);
strcpy(a[j],temp);
}}
printf("\nThe sorted array is:\n ");
for(i=0;i<n;i++){
puts(a[i]);
}}
```

```
Enter how many names: 4
Enter the names:
abc
bca
aaa
dcs
The sorted array is:
aaa
abc
bca
dcs
```

Tutorials on Simple Operations on String

- Write a simple C program to retrieve first word from a sentence.
- Write a C program to remove blank space from the string
- Write a C program to count the number of vowels and consonants in a given string.



Go to posts/chat box for the link to the question **PQn. S12.1**
submit your solution in next 2 minutes
The session will resume in 3 minutes

Summary

The String handling functions.

- strcmp
- strcat

S12_2

Searching Techniques

Objectives

To learn and appreciate the following concepts

Searching Technique

- **Linear Search**

- **Binary Search**

Session outcome

- At the end of session student will be able to understand
 - Searching Techniques

Arrays – A recap

1D Array:

Syntax: **type array_name[size];**

- **Initialization:**

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

- **Read:**

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

scanf(“%d”,&a[i]);

- **Write:**

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

printf(“%d”,a[i]);

Searching

- Finding whether a data item is present in a set of items
 - **linear** search / sequential search

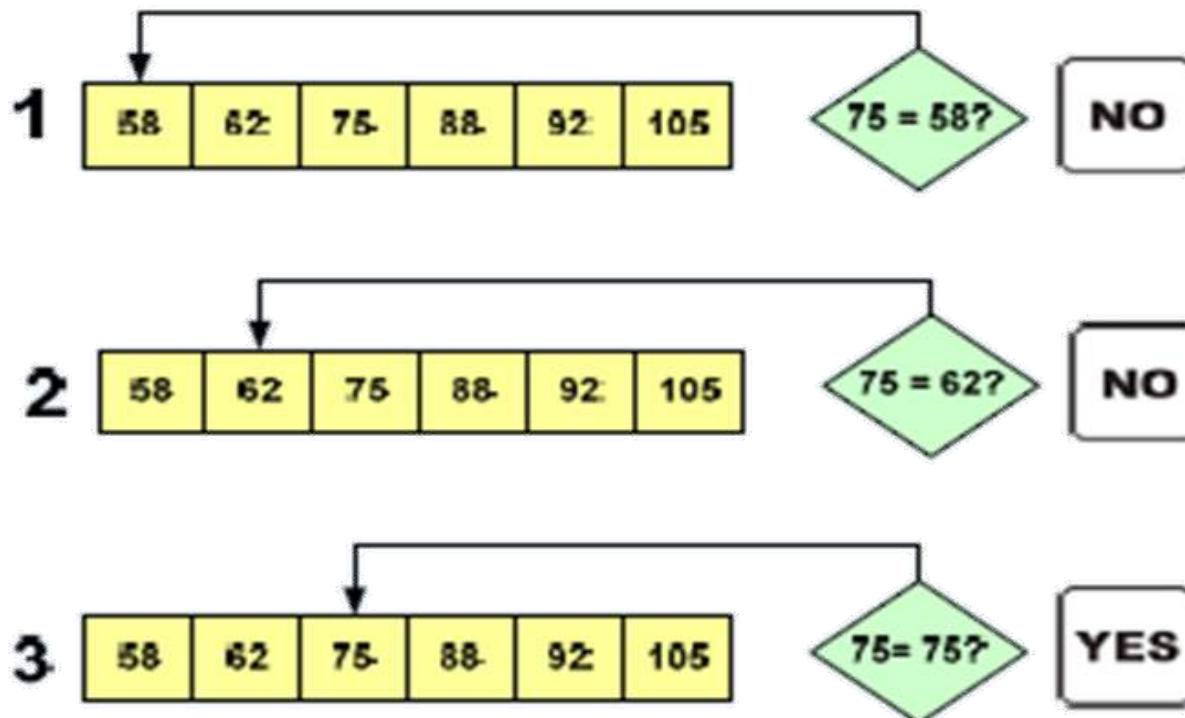
Linear search- illustration 1

List of Data:

58, 62, 75, 88, 92, 105

Data to be searched is

75



The “item is found” and stop the searching process

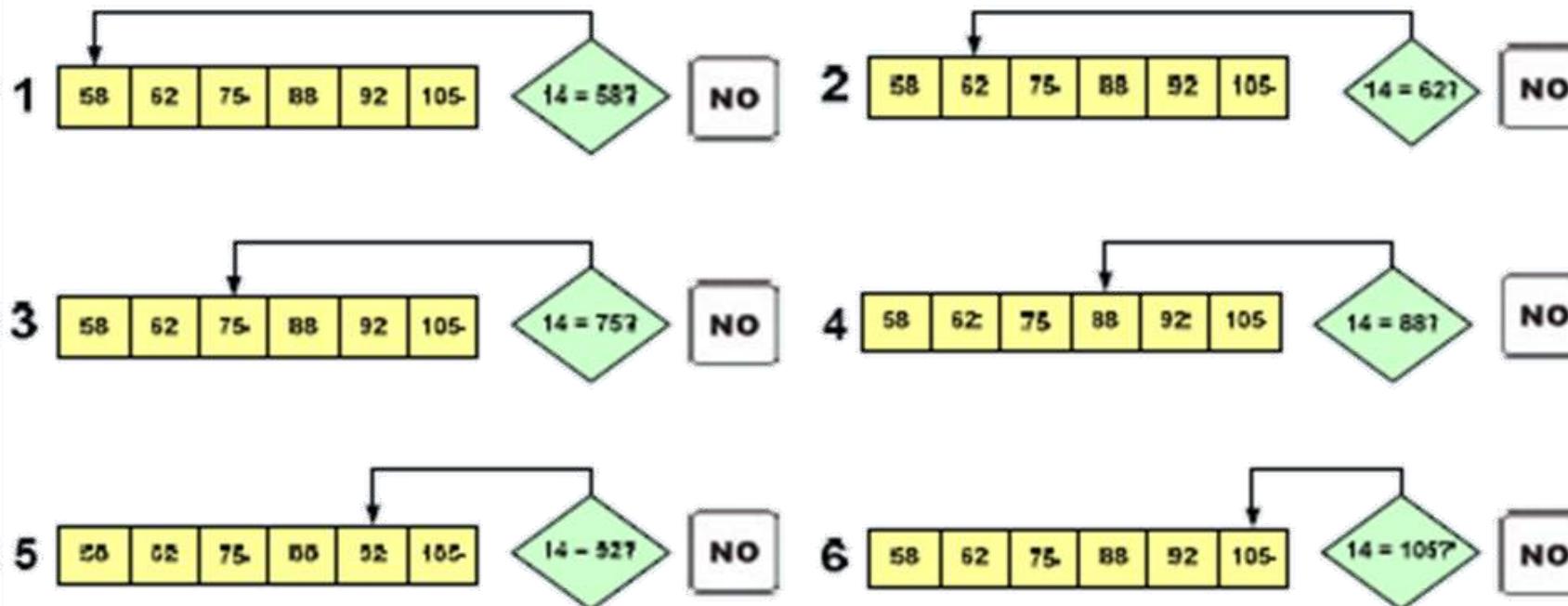
Linear search- illustration 2

List of Data:

58, 62, 75, 88, 92, 105

Data to be searched is

14



Now the end of the list is reached. There are no more elements in the list.
So the item 14 is “not found” in the list.

Pseudo code for linear search

```
int found=0; //setting flag  
  
Print "enter no of numbers";  
  
Input n;  
  
for(i=0;i<n;i++){  
  
Print "enter number\n";  
  
Input a[i]; // entered data items  
}  
  
Print "enter the element to be  
searched";  
  
Input key; // data to be searched
```

```
/*search procedure*/  
  
for(i=0; i<n; i++) {  
  
if(a[ i ]==key) // comparison  
{  
    found=1;  
    pos=i+1;  
    break;  
}  
}  
  
if(found==1)  
    Print“data_found_in”,pos,  
    "position";  
otherwise  
    Print “data is not found”;
```

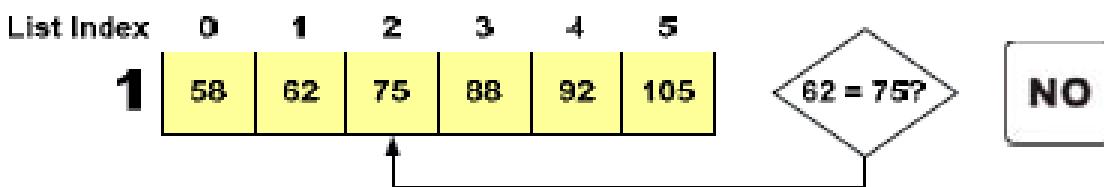
Binary Search

- A binary search is a searching technique that can be applied only to a **sorted list** of items
- This searching technique is similar to dictionary search.
- **Algorithm:**
 - **Step 1:** Set First = 0 and Last = Number of Items – 1
 - **Step 2:** Find the middle of the list as mid = (First + Last) /2. Take only the integer part, if the result is a real number.
 - **Step 3:** Compare the middle item with the searching item. If they are equal then “**Item is found**” and go to step 8.
 - **Step 4:** If the searching item is less than the middle item then the searching item comes before this middle element. So, set Last = mid -1 and there is no change in the value of First. Go to step 6.
 - **Step 5:** Since the above conditions are false the searching element should be greater than the middle element. So, set First = mid +1 and there is no change in the value of Last. Go to the next step.
 - **Step 6:** If First <= Last then go to step 2.
 - **Step 7:** Since end of the list is reached, the searching item is “**not found**” in the list
 - **Step 8:** End of the algorithm.

Binary Search – example-1

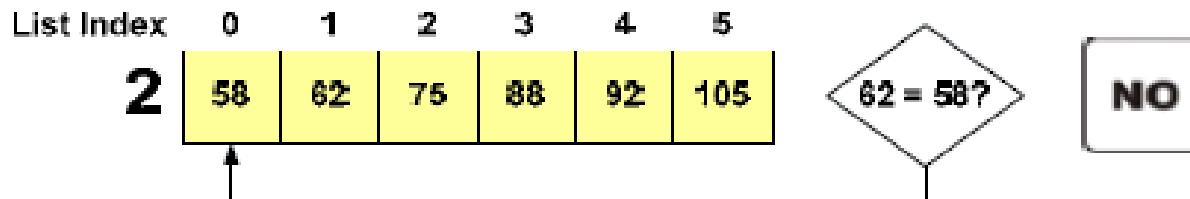
List of Data: **58, 62, 75, 88, 92, 105**

Data to be searched is **62**

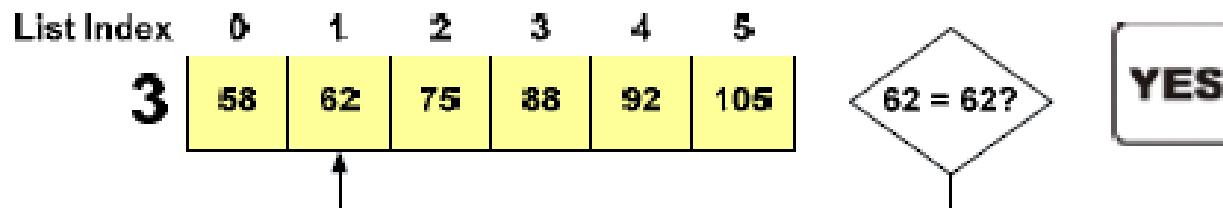
- Step 1: First = 0 and Last = 5
- Step 2: Step 2: Mid = $(0 + 5) / 2$. That is Mid = 2 (Only the integer part is taken)
- Step 3: 
- Step 4: The searching item 62 is less than 75. So it should appear before 75. Now First = 0 and Last = mid-1 that is Last = 2-1. So Last = 1
- Step 5: Compute Mid = $(0 + 1) / 2$ that is Mid = 0 (Integer part)

Binary Search – example-1

- Step 6: Compare 0th item with 62. That is compare 58 and 62. Since they are not equal proceed to the next step



- Step 7: Since the searching item 62 is greater than 58, the searching item comes after 58. First = mid+1 that is First = 0+1. So, First = 1 and Last=1. Now, mid=(1+1)/2=1
- Step 8: Compare 62 with the item in position 1. That is also 62. So, the “item is found” and stop the searching process

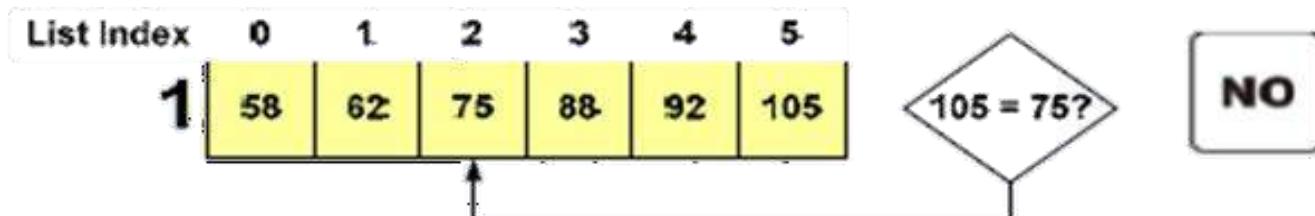


Binary Search – example-2

List of Data: 58, 62, 75, 88, 92, 105

Data to be searched is 105

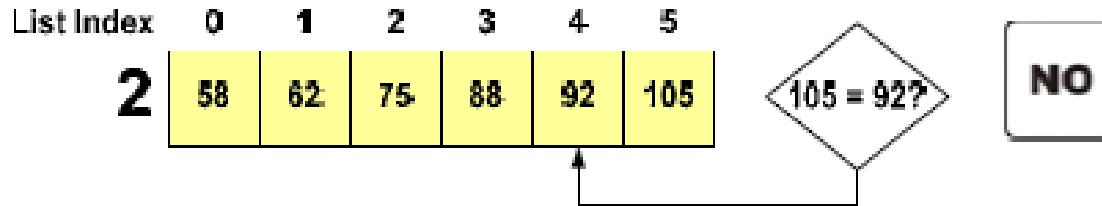
- Step 1: First = 0 and Last = 5
- Step 2: Step 2: Mid = $(0 + 5) / 2$. That is Mid = 2 (Only the integer part is taken)



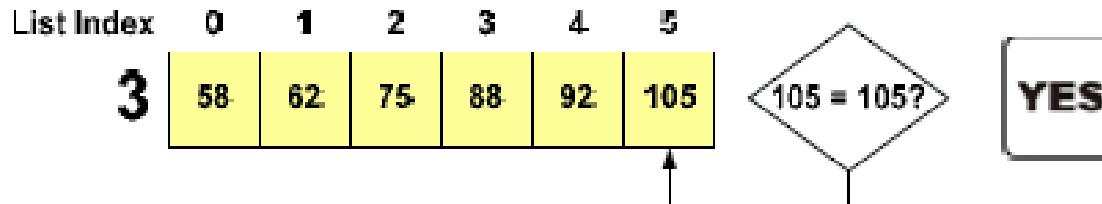
- Step 3:
- Step 4: The searching item 105 is greater than 75. So it comes after 75. First = 2 + 1 = 3. That is, First = 3 and Last = 5

Binary Search – example-2

- Step 5: Compute Mid = $(3+5) / 2 = 4$



- Step 6: The searching item 105 is greater than 92. So the searching item 105 comes after 92. First= $(4+1) = 5$ and Last =5. So Mid= $(5+5)/2 = 5$
- Step 7: Compare Searching element 105 with the 5th element. Since they are equal "**Item is found**" and stop the searching process

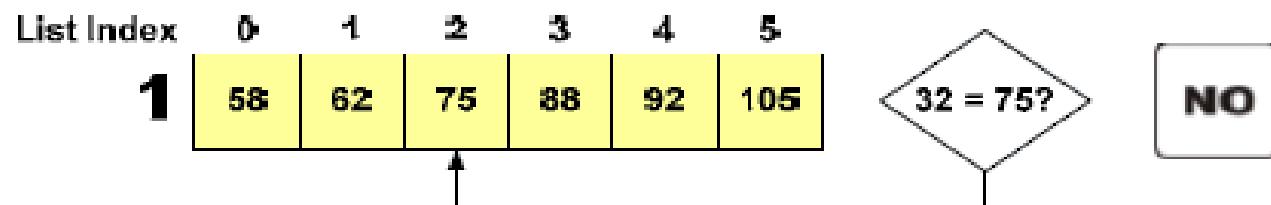


Binary Search – example-3

List of Data: **58, 62, 75, 88, 92, 105**

Data to be searched is **32**

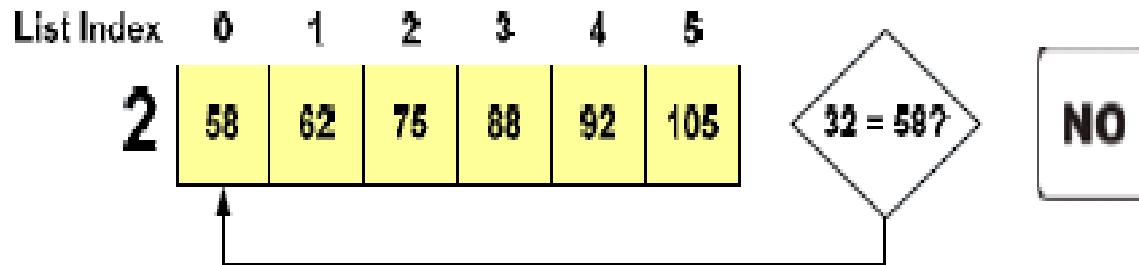
- Step 1: First = 0 and Last = 5
- Step 2: Mid = $(0 + 5) / 2$. That is Mid = 2 (Only the integer part is taken)
- Step 3: Compare the searching item 32 and 75. Since they are not equal proceed with the next step



- Step 4: The searching item 32 is less than 75. So First=0 and Last=1. $\text{Mid}=(0+1)/2=0$

Binary Search – example-3

- Step 5: Compare 32 and 58. Since they are not equal proceed with the next step



- Step 6: The searching item 32 is less than 58. So First = Mid-1 that is Last=0-1= -1 and First = 0. Since First > Last, “**Item is not found**” and stop the searching process

Binary Search – procedure

```
/* Binary search on sorted array */
```

```
low=0;  
high=N-1;  
do  
{  
    mid= (low + high) / 2;  
    if ( key < array[mid] )  
        high = mid - 1;  
    else if ( key > array[mid] )  
        low = mid + 1;  
} while( key!=array[mid] && low <= high);  
  
    if( key == array[mid] )  
    {  
        printf("SUCCESSFUL SEARCH\n");  
    }  
    else  
    {  
        printf("Search is FAILED\n");  
    }
```

Linear *versus* Binary Search

Linear Search	Binary Search
Can be applied on sorted and unsorted list of items	Can be applied only on sorted list of items
Searching time is more	Searching time is less



Go to posts/chat box for the link to the question
submit your solution in next 2 minutes
The session will resume in 3 minutes

Summary

- ❖ Linear Search
- ❖ Binary Search



A circular word cloud centered around the word "problem". Other words include "solution", "complex", "process", "work", "goal", "skill", "cognitive", "steps", "fail", "difficult", "using", and "computers".

problem solving using computers

CSE 1051



Bubble Sorting

S13_1

Objectives

To learn and appreciate the following concepts

Sorting Technique

- Bubble Sort
- Bubble Sort with strings

Session Outcome

- At the end of the session the student will be able to understand:
- Importance of bubble sorting on integers and strings

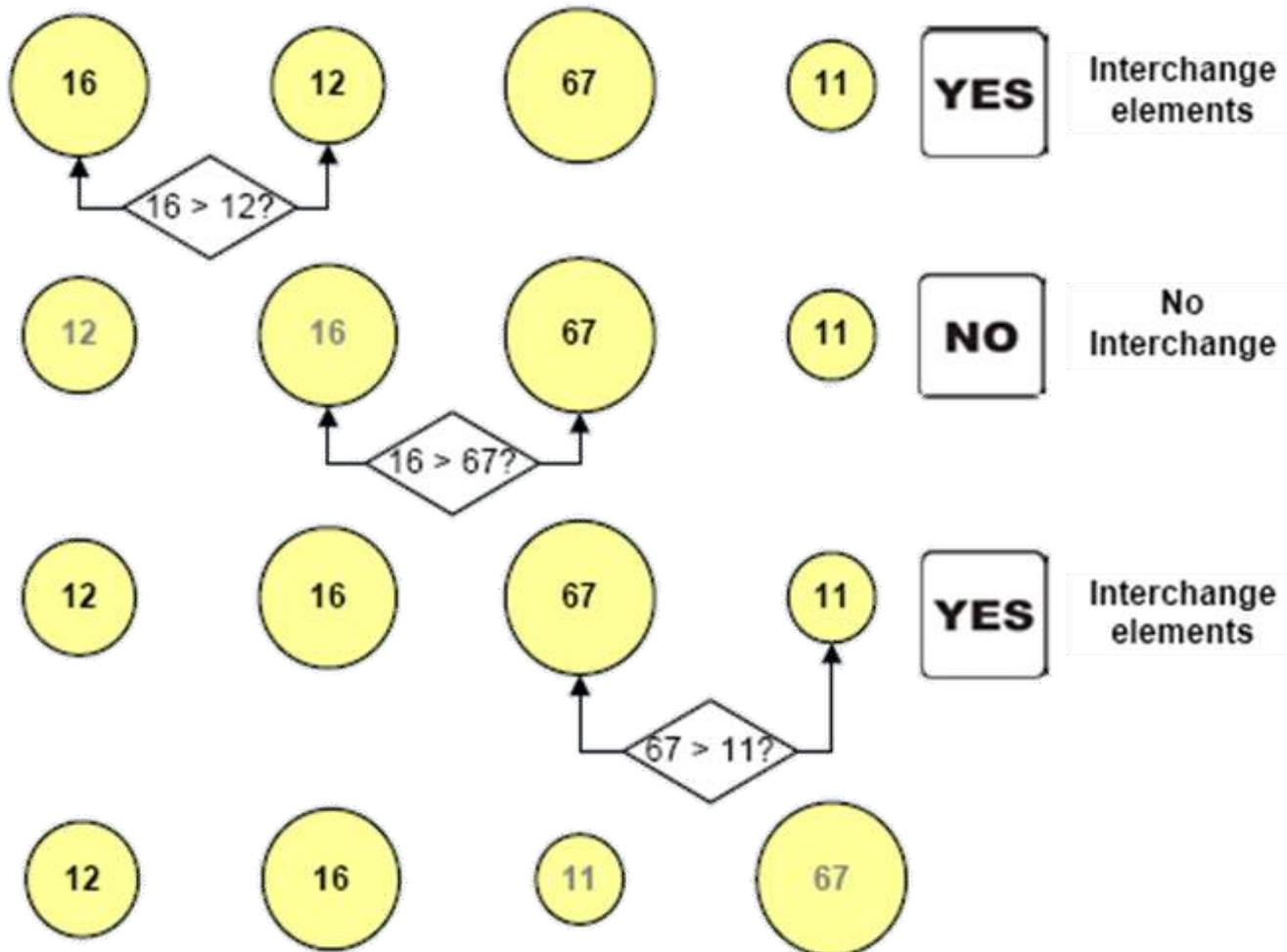
Sorting

Arrangement of data elements in a particular order

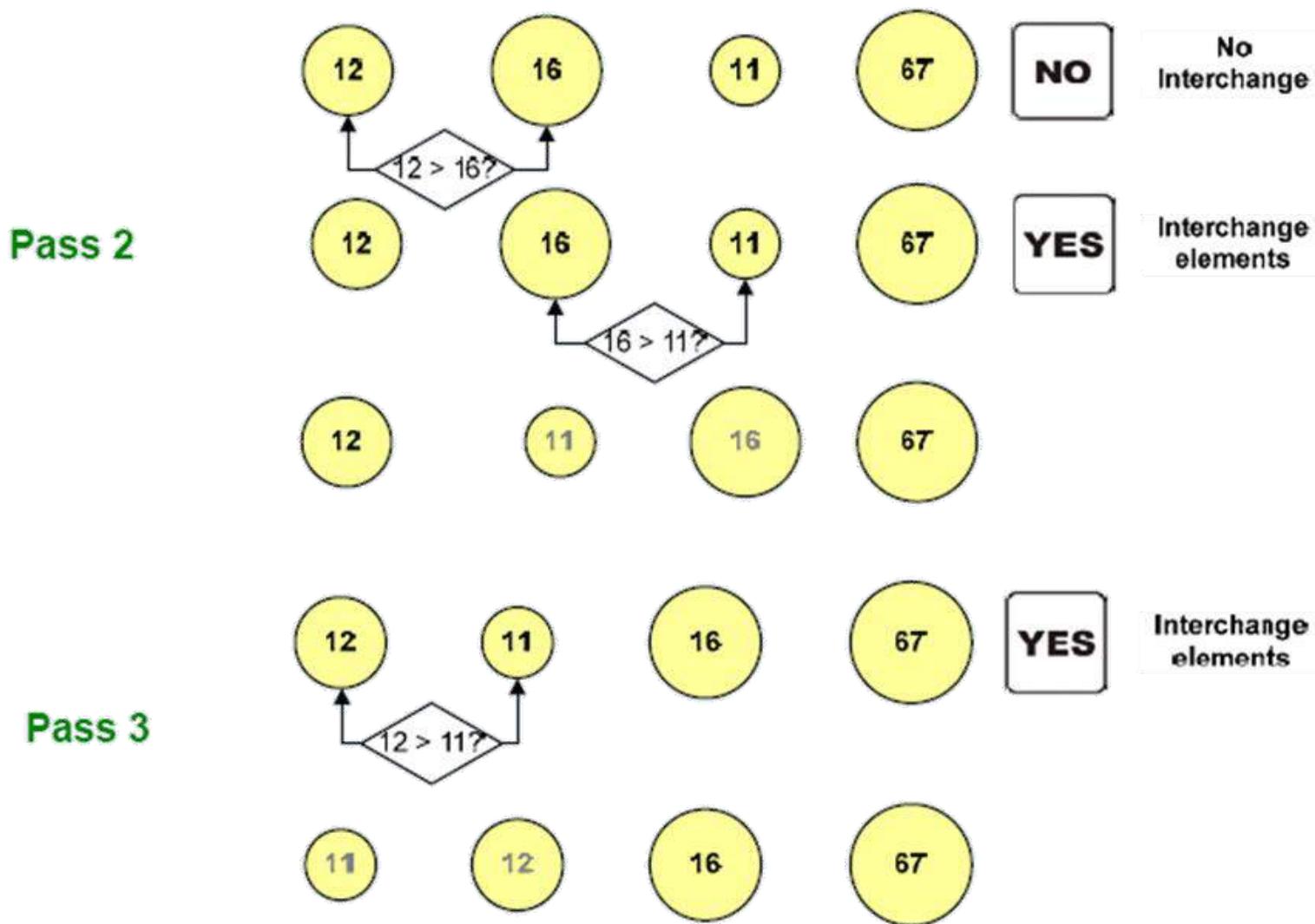
→ Bubble sorting

Bubble Sort- Illustration

Pass 1



Bubble Sort- Illustration



Pseudo code for Bubble Sort procedure

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

```
Input a[i]; // entered elements
```

```
for(i=0;i<n-1;i++) //pass
```

```
{     for(j=0;j<n-i-1;j++)
```

```
{         if(a[j]>a[j+1]) // comparison
```

```
{ // interchange
```

```
temp=a[j];
```

```
a[j]=a[j+1];
```

```
a[j+1]=temp;
```

```
} } }
```

Example :

a[]={ 16, 12, 11, 67 }

Array after sorting (ascending):

a[]={ 11, 12, 16, 67 }



Go to posts/chat box for the link to the question **PQn. S13.1**

submit your solution in next 2 minutes
The session will resume in 3 minutes

Strings Bubble Sort

```
int main()
{
char string[30][30],temp[30];
int no, i, j;
printf("\nEnter the no of strings:");
scanf("%d",&no);
printf("\nEnter the strings:");
for(i=0;i<no; i++)
    gets(string[i]);
```

```
for(i=0;i<no-1;i++)
{
    for(j=i+1;j<no;j++)
    {
        if(strcmp(string[i],string[j])>0)
        {
            strcpy(temp,string[i]);
            strcpy(string[i],string[j]);
            strcpy(string[j],temp);
        }
    }
    printf("\n The sorted array is:");
    for(i=0;i<no;i++)
        puts(string[i]);
return 0;
}
```

String Bubble Sort input/output

D	E	L	H	I	\0		
A	G	R	A	\0			
B	A	R	E	L	I	\0	

A	G	R	A	\0			
B	A	R	E	L	I	\0	
D	E	L	H	I	\0		

Summary

- Bubble Sort
- Bubble sort with strings



problem
solving
using
computers

CSE 1051

Selection Sorting

S13_2

Objectives

To learn and appreciate the following concepts

Sorting Technique

- Selection Sort
- Selection Sort with Strings

Session Outcome

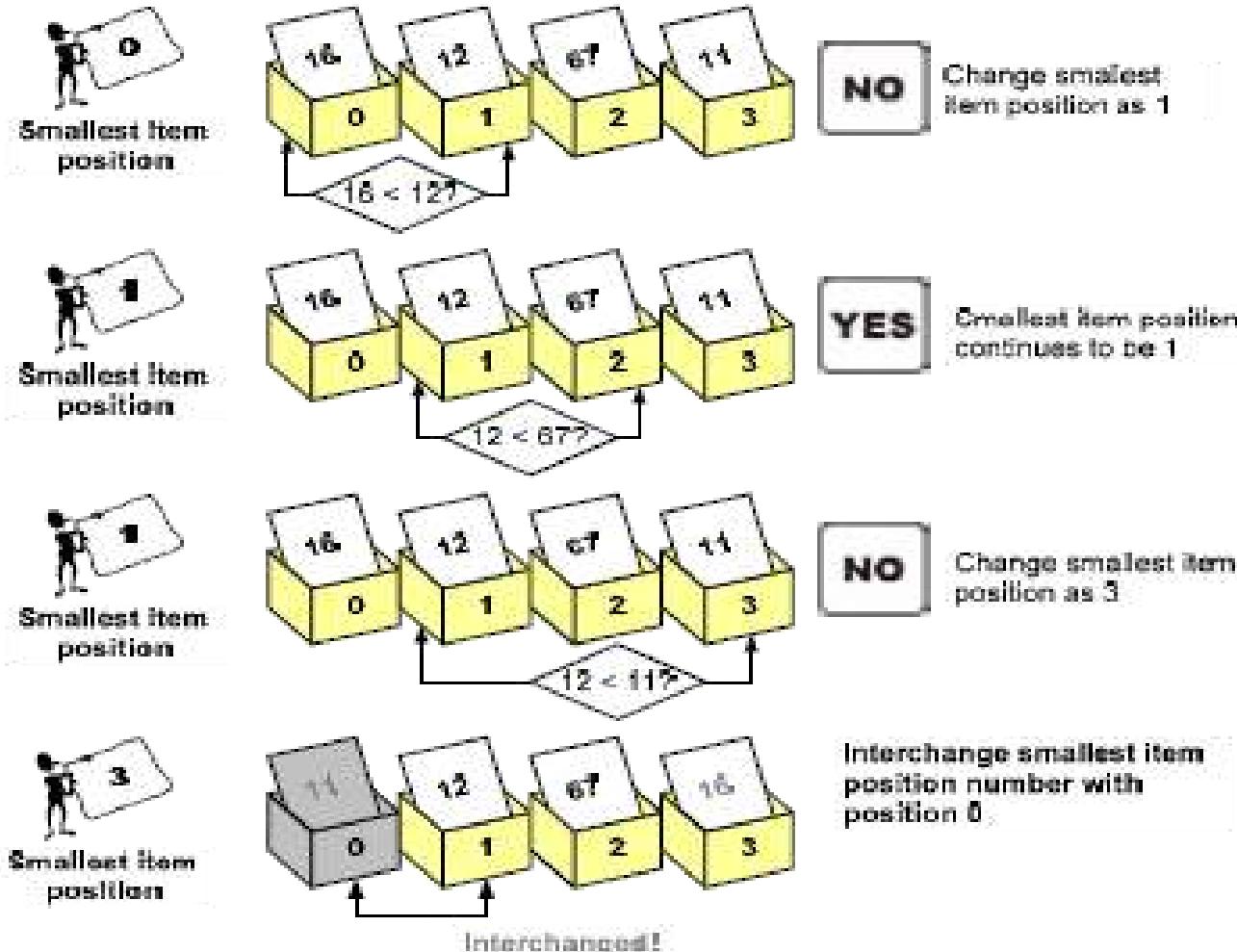
- At the end of the session the student will be able to understand:
- Importance of selection sorting on integers and strings

Selection Sort

- Each pass selects the smallest data item from the unsorted set and move it to its position
- **Procedure:**
 - Here 'N' indicates the number of data items to be sorted
 - Step 1: From the data items in positions 0 to N-1, select the smallest data item and interchange with the 0th data item. Now the first data item is sorted
 - Step 2: From the data items in positions 1 to N-1, select the smallest data item and interchange with the 1st data item. Now the second data item is sorted
 - Step 3: The steps are repeated N-1 times. At the end of N-1 th time the entire data set is sorted

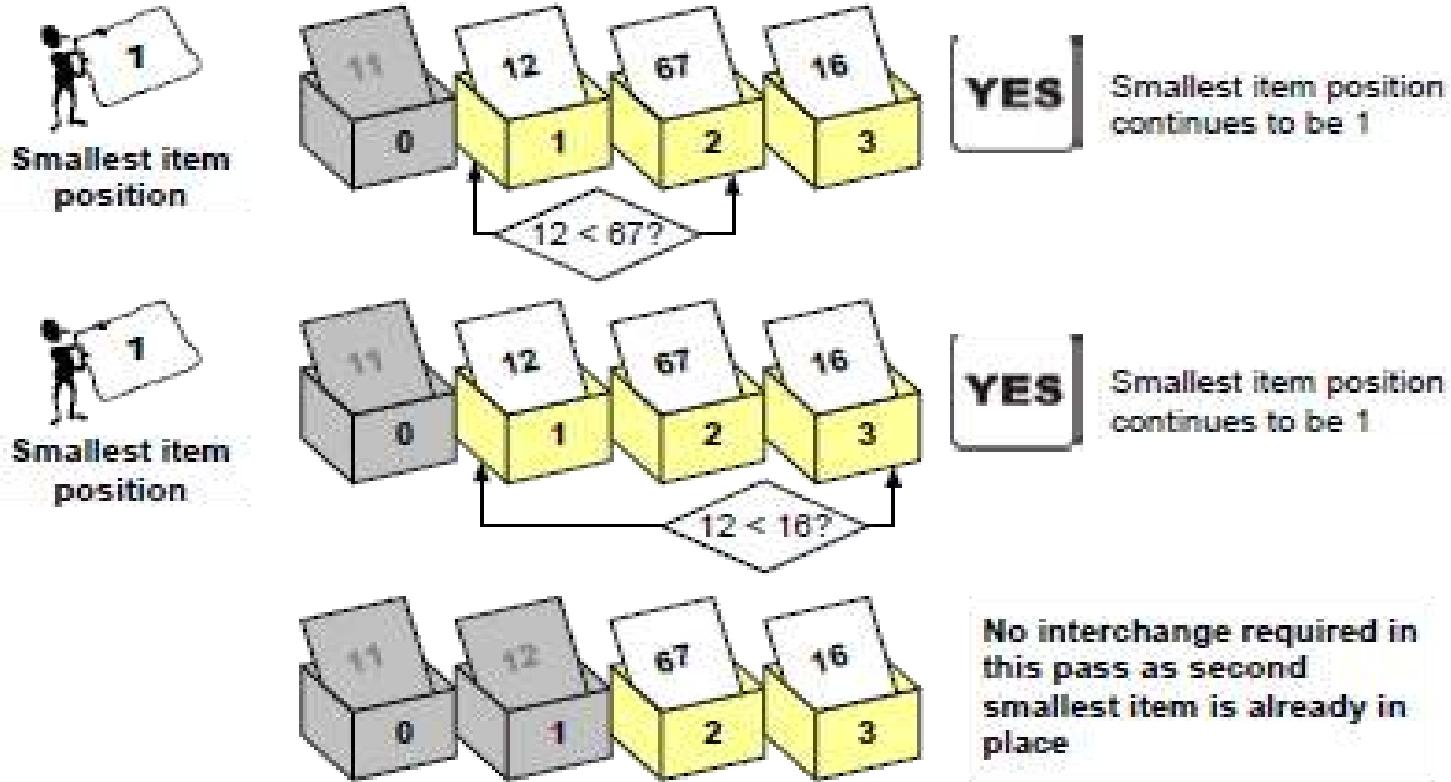
Selection Sort – example

Pass 1



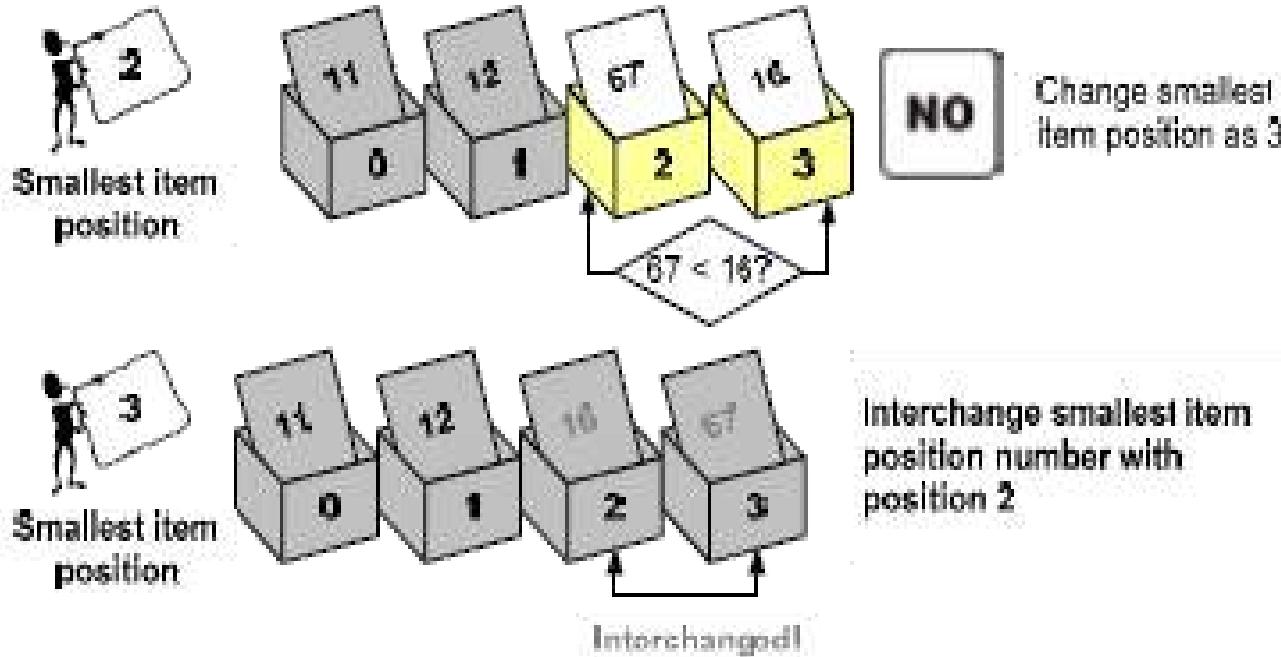
Selection Sort – example

Pass 2



Selection Sort – example

Pass 3



Selection Sort – procedure

```
for(i = 0; i < n-1; i++) // loop for number of pass
{
    pos = i; small = a[i];
    for(j=i+1; j<n; j++) //loop for searching the smallest
    {
        if(small > a[j]) // finding the smallest
        {
            pos = j; // pos for interchanging
            small = a[j]; // assigning current small value
        }
    }
    a[pos] = a[i]; //interchanging values
    a[i] = small; }
```



Go to posts/chat box for the link to the question **PQn. S13.2**

submit your solution in next 2 minutes
The session will resume in 3 minutes

Selection Sort for Strings

```
int main() {  
    char arr[100][100], temp[100], minStr[100];  
    int n,i, j, min_idx=i;  
    printf("enter the number of strings");  
    scanf("%d", &n);  
    printf("Enter the array\n");  
    fflush(stdin); // clear the buffer  
    for (i = 0; i < n; i++)  
        gets(arr[i]);  
    // Moving boundary of unsorted subarray  
    for (i = 0; i < n-1; i++) {  
        // Find the minimum element in unsorted array  
        strcpy(minStr, arr[i]);  
        for (j = i+1; j < n; j++)  
            if (strcmp(minStr, arr[j]) > 0)  
                strcpy(minStr, arr[j]);  
        if (strcmp(minStr, arr[i]) > 0)  
            strcpy(arr[i], minStr);  
    }  
}
```

Selection Sort for Strings



```
for (j = i+1; j < n; j++)      {  
    // If minStr is greater than arr[j]  
    if (strcmp(minStr, arr[j]) > 0)      {  
        // Make arr[j] as minStr and update min_idx  
        strcpy(minStr, arr[j]);  
        min_idx = j;      }      }  
  
// Swap the found minimum element with the first element  
if (min_idx != i)      {  
    strcpy(temp, arr[i]);  
    strcpy(arr[i], arr[min_idx]); //swap item[pos] and item[i]  
    strcpy(arr[min_idx], temp); } }  
  
printf("\nSorted array is\n");  
for (i = 0; i < n; i++)  
    puts( arr[i]); } }
```

String Selection Sort input/output

```
Select E:\PSUC_ONLINE\sel.exe
enter the number of strings3
Enter the array
Mango
Banana
Apple

Sorted array is
0: Apple
1: Banana
2: Mango

Process returned 0 (0x0)  execution time : 6.820 s
Press any key to continue.
```

Comparison between Bubble Sort and Selection Sort

Basis for Comparison	Bubble Sort	Selection Sort
Basic	Adjacent element is compared and then swapped	Largest element is selected and swapped with the last element(in case of ascending order)
Efficiency	Inefficient	Improved efficiency when compared to Bubble sort
Method	Exchanging	selection
Speed	Slow	Fast when compared to Bubble sort

Summary

- Selection Sort
- Selection Sort with strings

Summary of S13

- Bubble Sort
- Bubble sort with Strings
- Selection Sort
- Selection Sort with strings
- Comparison of bubble and selection sort



problem solving using computers

CSE 1051

2 D A R R A Y S

S14_1

Objectives

To learn and appreciate the following concepts

- 2D Array declaration, initialization
- Simple Programs using 2D arrays

Session outcome

At the end of session student will be able to

- Declare, initialize and access 2D array
- Write simple programs using 2D array

2 Dimensional Array

- 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 rows and columns.
- It is also called as a two-subscripted variable.

2 Dimensional Arrays

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.

2 dimensional Arrays

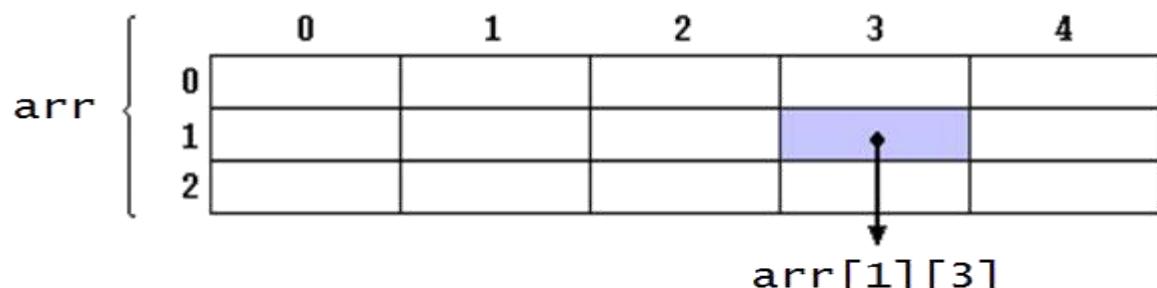
Declaration

```
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.



2 Dimensional Arrays

Initialization of two dimensional arrays

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

```
int table [2][3]={0,0,0,1,1,1};
```

→ initializes the elements of the first row to zero and the second row to 1.

Initialization is always done row by row.

The above statement can be equivalently written as

```
int table [2][3]={ {0,0,0},{1,1,1} };
```

OR in matrix form it can be written as

```
int table [2][3]= { {0,0,0},  
                          {1,1,1} };
```

2 Dimensional Arrays

When array is completely initialized with all values, need not necessarily specify the first dimension.

```
int table [][] = { { 0,0,0 },  
                   { 1,1,1 } };  
};
```

If the values are missing in an initializer, they are set to zero

```
int table [2][3] = { { 1,1 },  
                     { 2 } };
```

will initialize the first two elements of the first row to 1, the first element of the second row to two, and all other elements to zero.

To set all elements to zero

```
int table [3][3] = { { 0 }, { 0 }, { 0 } };
```



Go to posts/chat box for the link to the question **PQn. S14.1**

submit your solution in next 2 minutes
The session will resume in 3 minutes

Read a matrix and display it

```
int main()
{
    int i, j, m, n, a[100][100];
    printf("enter dimension for a:");
    scanf("%d %d",&m,&n);
    printf("\n enter elements\n");
    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
            scanf("%d", &a[i][j]);
    }
    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
            printf("%d\t",a[i][j]);
        printf("\n");
    }
    return 0;
}
```



```
#include<stdlib.h>
#include<stdio.h>
int main()
{
    int i, j, m, n, p, q, a[10][10],
        b[10][10], c[10][10];
    printf("enter dimension for a \n");
    scanf("%d %d",&m,&n);
    printf("enter dimension for b\n");
    scanf("%d %d",&p,&q);
    if (m!=p||n!=q)
    {
        printf("cannot add \n");
        exit(0);
    }
    //Reading the elements
    printf("enter elements for a \n");
    for (i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
            scanf("%d",&a[i][j]);
    }
    printf("enter elements for b \n");
    for (i=0;i<p;i++)
    {
        for(j=0;j<q;j++)
            scanf("%d",&b[i][j]);
    }
    for (i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
            c[i][j]=a[i][j]+b[i][j];
    }
    printf("Resultant matrix is \n");
    for (i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
            printf("%d ",c[i][j]);
        printf("\n");
    }
}
```

Matrix Addition

```
printf("\n enter elements for b\n");
for(i=0;i<p;i++)
    for(j=0;j<q;j++)
        scanf("%d", &b[i][j]);
//Addition
for(i=0;i<m;i++)
    for(j=0;j<n;j++)
        c[i][j]=a[i][j]+b[i][j];
//Display
printf("\n final matrix is \n");
for(i=0;i<m;i++)
{
    for(j=0;j<n;j++)
        printf("%d",c[i][j]);
    printf("\n");
}
```

Syntax Recap

Declaration:

```
data-type array_name[row_size][column_size];
```

Initialization of two dimensional arrays:

```
type array-name [row size] [col size ] ={list of values};
```

Reading a Matrix:

```
int a[100][100];
for(i=0;i<m;i++)
{
    for(j=0;j<n;j++)
        scanf("%d",&a[i][j]);
}
```

Display a Matrix:

```
int a[100][100];
for(i=0;i<m;i++)
{
    for(j=0;j<n;j++)
        printf("%d",a[i][j]);
    printf(" ");
    printf("\n");
}
```

Summary

- Declare, initialize and access 2D array
- Write simple programs using 2D array



CSE 1051

Department of CSE

2 D A R R A Y S

S14_2

Objectives

To learn and appreciate the following concepts

- Programs using 2D arrays

Session outcome

At the end of session student will be able to

- Write programs using 2D array

Syntax Recap

Declaration:

```
data-type array_name[row_size][column_size];
```

Initialization of two dimensional arrays:

```
type array-name [row size] [col size ] ={list of values};
```

Reading a Matrix:

```
int a[100][100];
for(i=0;i<m;i++)
{
    for(j=0;j<n;j++)
        scanf("%d",&a[i][j]);
}
```

Display a Matrix:

```
int a[100][100];
for(i=0;i<m;i++)
{
    for(j=0;j<n;j++)
        printf("%d",a[i][j]);
    printf(" ");
    printf("\n");
}
```

Row Sum & Column Sum of a matrix

```
int a[10][10];                                //Row sum

int rowsum[10], colsum[10];                    for(i=0;i<m;i++)

printf("enter dimension for a \n");            {

scanf("%d %d",&m, &n);                      rowsum[i]=0;

//Reading                                     for(j=0;j<n;j++)

printf("enter elements for a \n");           rowsum[i]=rowsum[i]+a[i][j];

for (i=0;i<m;i++){                           }

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

scanf("%d", &a[i][j]);                      printf("\n");

}
```

Row Sum & Column Sum of a matrix

```
//Display

for(i=0;i<m;i++) {
    for(j=0;j<n;j++) {
        printf("\t %d",a[i][j]);
        printf("->");
        printf("%d\n",rowsum[i]);
    }
    printf("\n");
    for(i=0;i<n;i++)
        printf("\t %d",colsum[i]);
```



Row Sum & Column Sum of a matrix

```
C:\Users\Admin\Desktop\Programs\2.exe
enter dimension for a
3 3
enter elements for a
1 2 3 4 5 6 7 8 9

      1          2          3->6
      4          5          6->15
      7          8          9->24

     12          15          18
```



Go to posts/chat box for the link to the question **PQn. S14.2**

submit your solution in next 2 minutes
The session will resume in 3 minutes



Multiplication of two Matrices

```
#include <stdlib.h>

int main(){ int i, j, m, n, p, q;
int a[10][10], b[10][10],c[10][10];
printf("enter dimension for a \n");
scanf("%d %d",&m,&n);
printf("\n enter dimension for b\n");
scanf("%d %d", &p,&q);
if(n!=p){
    printf("not multiplicable \n");
    exit(0); // Terminate the execution
}
```

```
printf("enter elements for a \n");
for (i=0;i<m;i++)
{
    for(j=0;j<n;j++)
        scanf("%d",&a[i][j]);
}
printf("\n enter elements for b\n");
for(i=0;i<p;i++)
{
    for(j=0;j<q;j++)
        scanf("%d",&b[i][j]);
}
```

$$\begin{aligned} A \times B &= \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \\ A \times B &= \begin{bmatrix} 7 & 10 \\ 15 & 22 \end{bmatrix} \text{ 2x2 matrices} \end{aligned}$$



Multiplication of two Matrices

```
for(i=0;i<m;i++) {  
    for(j=0;j<q;j++) {  
        c[i][j]=0;  
        for(k=0;k<n;k++)  
            c[i][j]=c[i][j]+a[i][k]*b[k][j];  
    }  
}
```

```
printf("\n The product matrix is \n");  
for(i=0;i<m;i++){  
    for(j=0;j<q;j++)  
        printf("%d\t",c[i][j]);  
    printf("\n");  
}
```

$$\boxed{A \times B = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}}$$
$$A \times B = \begin{bmatrix} 7 & 10 \\ 15 & 22 \end{bmatrix} \text{ 2x2 matrices}$$

Summary

- Write programs using 2D array



**problem
solving
using
computers**

CSE 1051

A central word cloud is composed of various terms related to problem-solving, including: solve, emotional, goals, tasks, logic, skills, research, tracing, defined, attempts, novel, system, process, work, solution, best, goal, skill, try, relation, simulation, main, existing, applied, researchers, step, taken, control, found, life, simple, resolution, psychology, cognitive, steps, fail, difficult, testing, capacity, initiated, theory, analysis, study, technique, and skills.

2 D A R R A Y S

S15-1

Objectives

To learn and appreciate the following concepts

- Programs using 2D arrays

Session outcome

At the end of session student will be able to

- Write programs using 2D array

Syntax Recap

Declaration:

```
data-type array_name[row_size][column_size];
```

Initialization of two dimensional arrays:

```
type array-name [row size] [col size ] ={list of values};
```

Reading a Matrix:

```
int a[100][100];
for(i=0;i<m;i++)
{
    for(j=0;j<n;j++)
        scanf("%d",&a[i][j]);
}
```

Display a Matrix:

```
int a[100][100];
for(i=0;i<m;i++)
{
    for(j=0;j<n;j++)
        printf("%d",a[i][j]);
    printf(" ");
    printf("\n");
}
```

Trace and Norm of a Matrix

Trace is sum of principal diagonal elements of a square matrix.

Norm is Square Root of sum of squares of elements of a matrix.

```
int trace=0, sum=0,i,j,norm;
int m=3,n=3;
printf("enter elements for a
\n");
for (i=0;i<m;i++){
    for(j=0;j<n;j++)
        scanf("%d",&a[i][j]);
}
for(i=0;i<m;i++)
    trace=trace + a[i][i];
for(i=0;i<m;i++)
    for(j=0;j<n;j++)
        sum=sum+a[ i ][ j ]*a[ i ][ j ];
norm=sqrt(sum);
printf(" trace is %d", trace );
printf(" norm is %d", norm );
```

Check whether a given Matrix is Symmetric or not

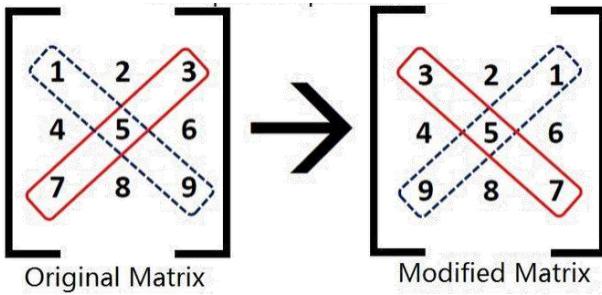
```
printf("enter dimension \n");
scanf("%d %d",&m,&n);
if(m!=n)
printf("it is not a square \n");
else
{ printf("enter elements \n");
for(i=0;i<m;i++)
    for(j=0;j<n;j++)
        scanf("%d",&a[i][j]);
    for(i=0;i<m;i++){
        for(j=0;j<n;j++){
            if (a[ i ][ j ]!=a[ j ][ i ]){
                printf("\n matrix is not
symmetric \n");
                exit(0);    }
            }
        }
    printf("\n matrix is symmetric");
}
```



Go to posts/chat box for the link to the question **PQn. S15.1**

submit your solution in next 2 minutes
The session will resume in 3 minutes

Exchange the elements of principal diagonal with secondary diagonal in an N dimensional Square matrix



```
int main(){
int i, j, temp, arr[4][4],n;
printf("\nEnter dimension:");
scanf("%d",&n);
printf("\nEnter elements:\n");
for(i=0;i<n;i++)
for(j=0;j<n;j++)
printf("%d",arr[i][j]);
```

```
for(i=0;i<n;i++)
for(j=0;j<n;j++)
if(i==j){
    temp=arr[i][j];
    arr[i][j]=arr[i][n-i-1];
    arr[i][n-i-1]=temp;
}
printf("\nModified Matrix:\n");
for(i=0;i<n;i++){
    for(j=0;j<n;j++)
        printf(" ");
    printf("%d",arr[i][j]);
    printf("\n");
}
return 0;}
```

Exchange the Rows and Columns of a ‘mxn’ matrix

```
printf("\nEnter the cols to exchange: ");
scanf("%d %d",&c1,&c2);
/*Column exchange : c1 ⇔ c2 */
for(i=0;i<m;i++) {
    temp=arr[i][c1-1];
    arr[i][c1-1]=arr[i][c2-1];
    arr[i][c2-1]=temp; }

/*read 'mxn' matrix */
printf("\nEnter the rows to exchange:
");
scanf("%d %d",&r1,&r2);
/*Row exchange r1 ⇔ r2 */
for(j=0;j<n;j++) {
    temp=arr[r1-1][j];
    arr[r1-1][j]=arr[r2-1][j];
    arr[r2-1][j]=temp; }
```

Tutorials

- Write a program to check whether the given matrix is sparse matrix or not.
- Write a program to find the sum of the elements above and below diagonal elements in a matrix.
- Write program to check the given matrix is a magic square or not

(A magic square of order n is an arrangement of n^2 numbers, usually distinct integers, in a square, such that the n numbers in all rows, all columns, and both diagonals sum to the same constant. A normal magic square contains the integers from 1 to n^2 .)

2	7	6	→15
9	5	1	→15
4	3	8	→15
15	15	15	15

11

Summary

- Declare, initialize and access 2D array
- Write programs using 2D array

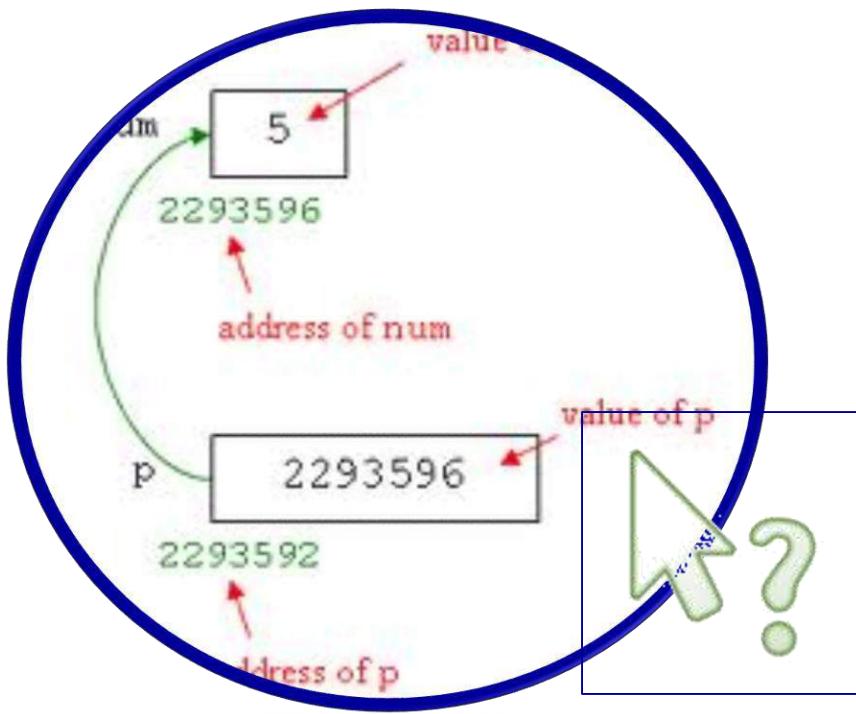
Summary of 2D arrays

- Declare, initialize and access 2D array
- Write simple programs using 2D array
- Advance programming in 2D arrays

problem solving using computers

CSE 1051





Pointers S15 - 2

Objectives

- To learn and appreciate the following concepts:
 - Concept of Basic Pointers – declaration and initialization

Session outcome

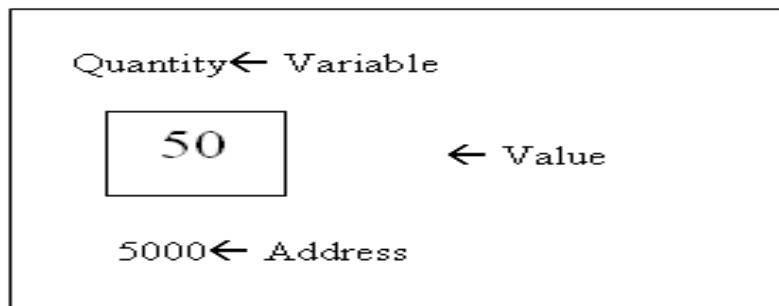
- At the end of session one will be able to:
 - Understand the concept of Basic Pointers

Pointers - Concept

- Consider the following statement

```
int Quantity = 50;
```

- Compiler will allocate a memory location for Quantity and places the value in that location. Suppose the address of that location is 5000, then



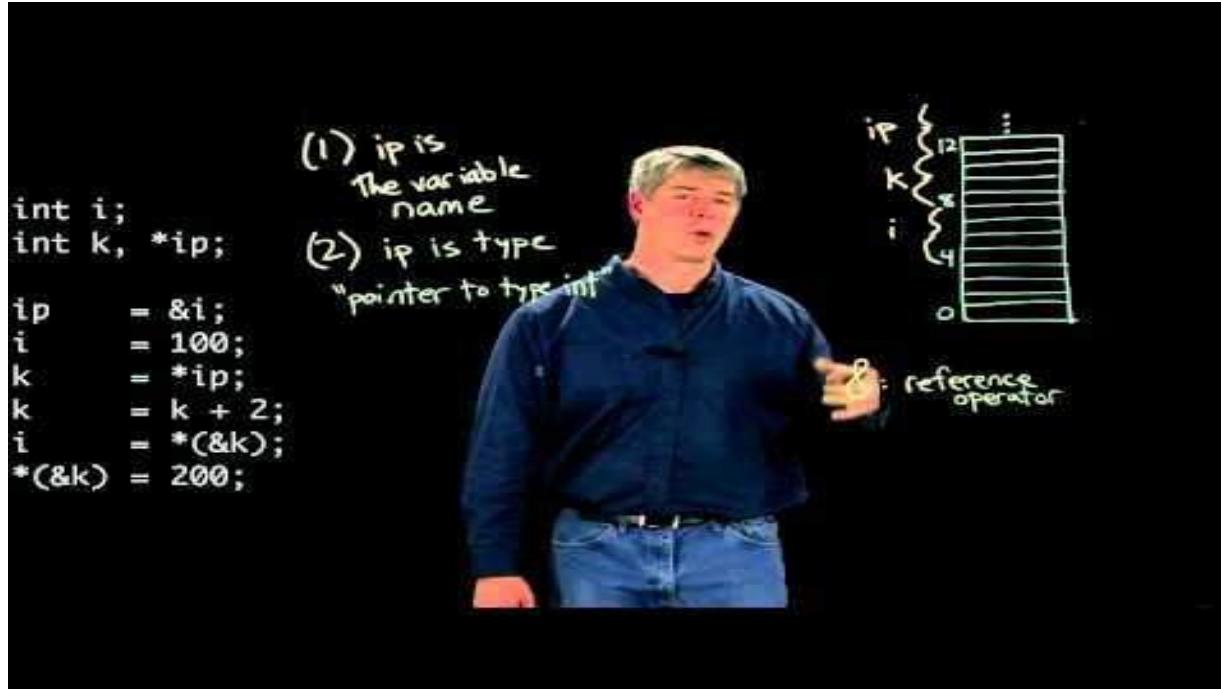
Pointers - Concept

- During Execution of the program, the system always associates the name quantity with the address 5000.
- We may have access to the value 50 by using either the name of the variable quantity or the address 5000.
- Since memory addresses are simply numbers, they can be assigned to some variables which can be stored in memory, like any other variable.

Pointer

- A memory location or a variable which stores the address of another variable in memory
- Commonly used in C than in many other languages (such as BASIC, Pascal, and certainly Java, which has no pointers).

Basics of Pointer



<https://www.youtube.com/watch?v=47IS8VtAM9E>



Go to posts/chat box for the link to the question **PQn. S15.2**

submit your solution in next 2 minutes
The session will resume in 3 minutes

Declaring and initializing pointers

- Syntax:

```
data_type * pt_name;
```

- This tells the compiler 3 things about the pt_name:
 - The asterisk(*) tells the variable pt_name is a pointer variable.
 - pt_name needs a memory location.
 - pt_name points to a variable of type data_type

Summary

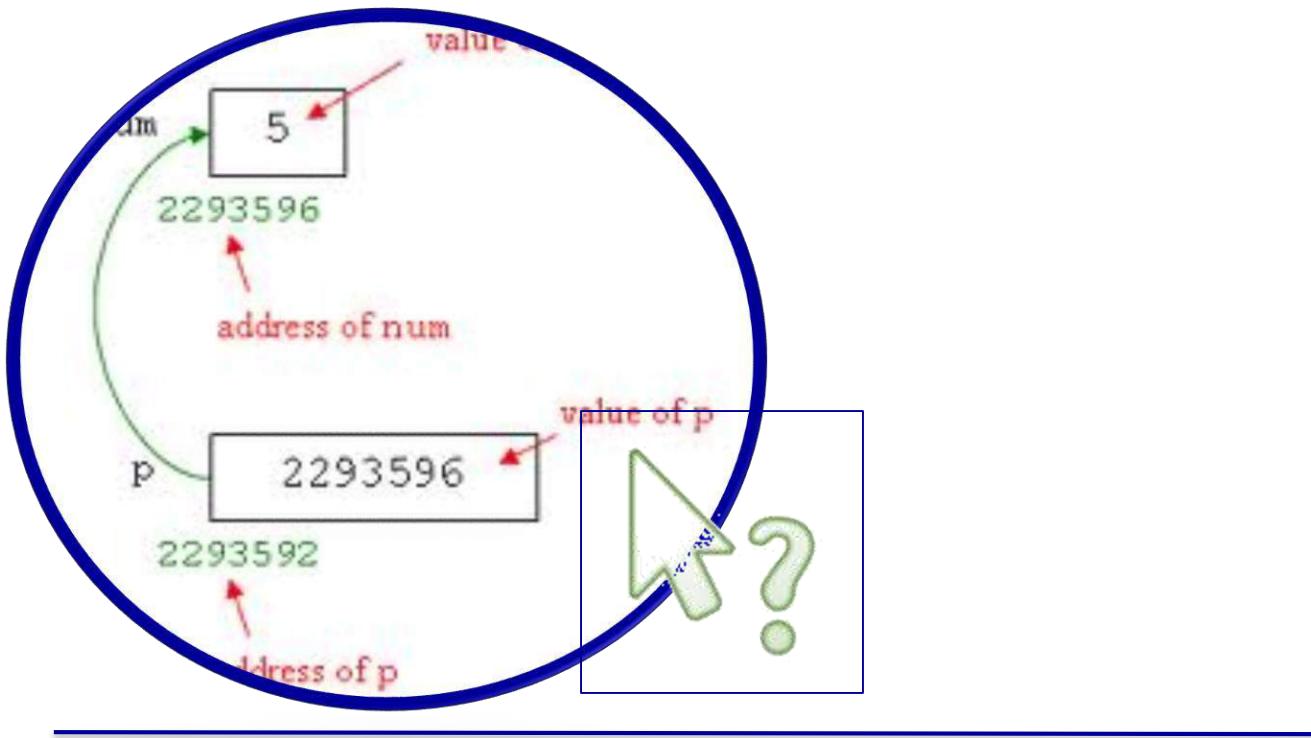
- Pointers – declaration and initialization



**problem
solving
using
computers**

CSE 1051

A central word cloud is composed of various terms related to problem-solving, including: solve, emotional, goals, tasks, logic, directly, research, tracing, defined, novel, emphasis, attempts, approaches, past, used, system, causes, focused, process, large, work, solution, best, try, goal, skill, relation, simulation, field, psychologists, book, realbeverages, existing, applied, researchers, step, taken, control, found, life, simple, resolution, check, psychology, target, include, decision, target, point, simple, share, prove, private, assume, assuming, steps, fail, smaller, difficult, testing, capacity, approach, solved, difficult, difficulty, attention, relatively, analysis, study, technique, developing, initiated.



Pointers S16-1

Objectives

- To learn and appreciate the following concepts:
 - Accessing the variable using address-of operator

Session outcome

- At the end of session one will be able to:
 - Access the variable using address-of operator

Pointers Concept

- **The Address-of Operator &**
 - To find the address occupied by a variable

Accessing the address of a variable

```
int Quantity=50 ;
```

To assign the address 5000 (the location of quantity) to a variable p, we can write:

```
int *p = &Quantity ;
```

Such variables that hold memory addresses are called Pointer Variables.

Variable	Value	Address
Quantity	50	5000
p	5000	5048

Program to illustrate the address of operator

```
#include <stdio.h>

int main()
{
    int var1 = 11;                                0x29feec
    int var2 = 22;                                0x29fee8
    int var3 = 33;                                0x29fee4

    //print the addresses of these variables
    printf("%x",&var1);
    printf("%x",&var2);
    printf("%x",&var3);

    return 0;
}
```

Output:

Declaring and initializing pointers

Example:

```
int *p; //declares a variable p as a pointer variable  
         that points to an integer data type.
```

```
float *x; //declares x as a pointer to floating point  
           variable.
```

Once a pointer variable has been declared, it can be made to point to a variable using an assignment statement :

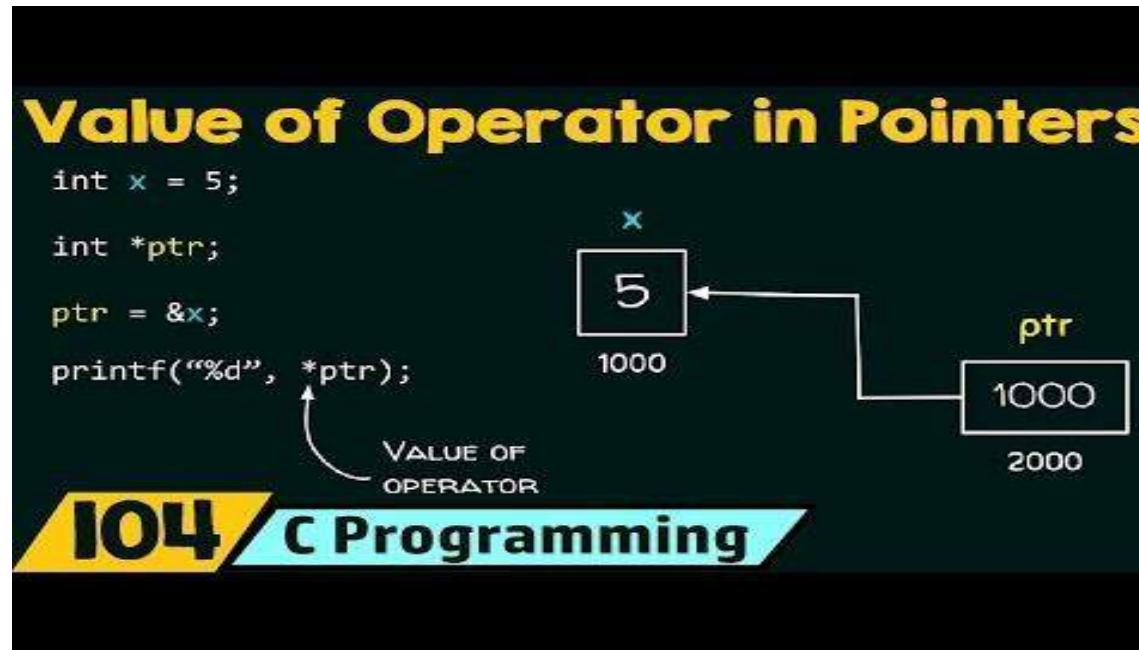
```
int quantity = 10;  
p = &quantity; // p now contains the address of quantity. This is known as pointer initialization.
```



Go to posts/chat box for the link to the question **PQn. S16.1**

**submit your solution in next 2 minutes
The session will resume in 3 minutes**

Understanding pointers better



https://www.youtube.com/watch?v=xlt_bEqfnxg

Summary till now ...

```
int v; //defines variable v of type int
```

```
int* p; //defines p as a pointer to int
```

```
p = &v; //assigns address of variable v to pointer p
```

Now...

```
v = 3; //assigns 3 to v
```

To be taken care ...

Before a pointer is initialized, it should not be used.

We must ensure that the pointer variables always point to the corresponding type of data.

Assigning an absolute address to a pointer variable is prohibited. i.e `p=5000`

A pointer variable can be initialized in its declaration itself.

Example:

```
int x, *p=&x; //declares x as an integer  
variable and then initializes  
p to the address of x.
```

To be taken care ...

The statement

int *p = & x, x; not valid.

i.e target variable ‘x’ must be declared first.

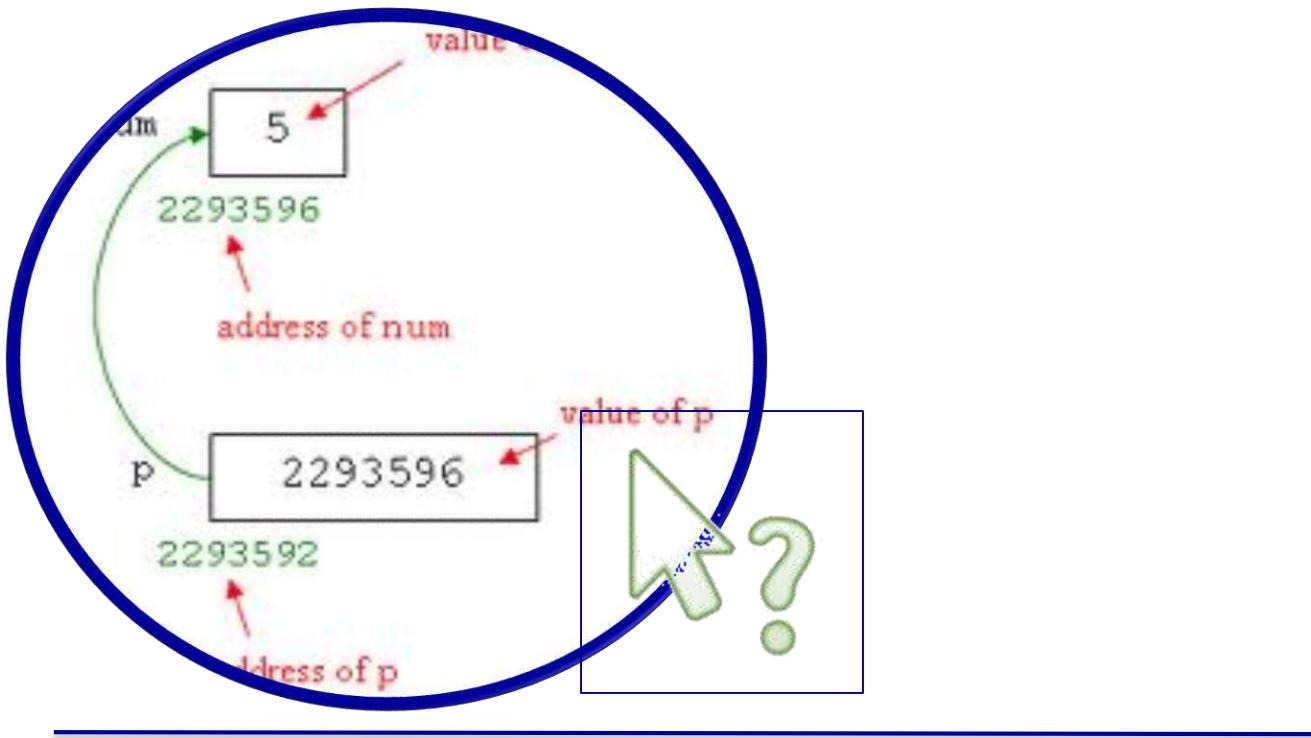
Summary

- Accessing the address of a variable using & operator



CSE 1051

Department of CSE



Pointers S16-2

Objectives

- To learn and appreciate the following concepts:
 - Simple Programs using pointers

Session outcome

- At the end of session one will be able to:
 - Write simple Programs using pointers

Accessing variable through a pointer

- A variable's value can be accessed by its pointer using unary operator *(asterisk) known as indirection operator.

Consider the following statements:

```
int quantity, *p, n;    // 2 int variables & 1 integer pointer  
quantity =50;           // assigns value 50 to quantity  
p=&quantity;           // assigns the address of quantity to p  
n=*p;                  // contains the indirection operator *
```

* Operator - value at address operator

Example – Accessing variable through a pointer

```
#include <stdio.h>

int main()
{
    int var1 = 11;          //two integer variables
    int var2 = 22;
    int *ptr;              //pointer to integer
    ptr = &var1;            //pointer points to var1
    printf("%d",*ptr);     //print contents of pointer (11)
    ptr = &var2;            //pointer points to var2
    printf("%d",*ptr);     //print contents of pointer (22)
    return 0;
}
```

Output :
11
22

Example - Accessing via pointers.

```
#include <stdio.h>
int main()
{
    int var1, var2;          //two integer variables
    int *ptr;                //pointer to integers
    ptr = &var1;              //set pointer to address of var1
    *ptr = 37;                //same as var1=37 ( Dereferencing)
    var2 = *ptr;              //same as var2=var1
    printf("%d", var2); //verify var2 is 37
    return 0;
}
```

Reference and dereference operators

- `&` is the ‘reference’ operator and can be read as “address of”
- `*` is the ‘dereference’ operator and can be read as “value at address” or “value pointed by”



Go to posts/chat box for the link to the question **PQn. S16.2**

submit your solution in next 2 minutes

The session will resume in 3 minutes

Example- understanding pointers

```
#include <stdio.h>

int main()
{
    int firstvalue = 5, secondvalue = 15;
    int * p1, * p2;
    p1 = &firstvalue;      // p1 = address of firstvalue
    p2 = &secondvalue; // p2 = address of secondvalue
    *p1 = 10;          // value pointed by p1 = 10
    *p2 = *p1;          // value pointed by p2 = value pointed by p1
    p1 = p2;            // p1 = p2 (value of pointer is copied)
    *p1 = 20;          // value pointed by p1 = 20
    printf("firstvalue is %d ", firstvalue );
    printf( "secondvalue is %d" ,secondvalue);
    return 0;
}
```

Output :
firstvalue is 10
secondvalue is 20

Summary of pointers

- Pointer concept
- Reference operator &
- Dereference operator *



S-17_1 MODULAR PROGRAMMING

Objectives:

To learn and appreciate the following concepts

- Modularization and importance of modularization
- Understand how to define and invoke a function
- Understand the flow of control in a program involving function call

Session outcome:

At the end of session one will be able to

- Understand modularization and function
- Write simple programs using functions

Programming Scenario . . .

In a large complex software development,

- Several Functionalities needs to be implemented
- Development needs to be done in a Team
- Lengthier code

Programming Scenario . . .

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
- Readability
- Debugging is easier
- Build Library
- Manageability
- Develop in a Team
- Quality

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)

General form of function definition

```
return_type function_name(parameter_definition)
{
    variable declaration;

    statement1;
    statement2;

    .
    .
    .

return(value_computed);
}
```

Defining a Function

✓ Name (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.

✓ Parameter list (parameter_definition)

- 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)

Understanding of main function

The diagram illustrates the structure of the `main` function. It shows the declaration `int main (void)` at the top, followed by a brace indicating its body. The declaration is annotated with three labels: "Return type" pointing to `int`, "Function name" pointing to `main`, and "Parameter List" pointing to `(void)`. Below the declaration, the function body begins with a brace, followed by the statements `printf("hello world\n");` and `return 0;`, which are enclosed within curly braces labeled "Body".

```
int main (void)
{
    printf("hello world\n");
    return 0;
}
```

Return type Function name Parameter List

{ Body

Function Definition and Call

```
// FUNCTION DEFINITION
Return type    Function name    Parameter List
void DisplayMessage(void)
{
    printf("Hello from function DisplayMessage\n");
}
int main()
{
    printf("Hello from main \n");
    DisplayMessage(); // FUNCTION CALL
    printf("Back in function main again.\n");
    return 0;
}
```

Multiple Functions- An example

```
void First (void){ // FUNCTION DEFINITION
    printf("I am now inside function First\n");
}

void Second (void){ // FUNCTION DEFINITION
    printf( "I am now inside function Second\n");
    First(); // FUNCTION CALL
    printf("Back to Second\n");
}

int main (){
    printf( "I am starting in function main\n");
    First (); // FUNCTION CALL
    printf( "Back to main function \n");
    Second (); // FUNCTION CALL
    printf( "Back to main function \n");
    return 0;
}
```

Arguments and parameters

- Both arguments and 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**.

Functions

Formal parameters

```
void dispChar(int n, char c) {  
    printf(" You have entered %d & %c“,n,c);  
}
```

```
int main(){ //calling program  
    int no; char ch;  
    printf("Enter a number & a character: \n");  
    scanf("%d %c",&no,&ch);  
    dispChar( no, ch);  
    return 0;
```

Actual parameters



Go to posts/chat box for the link to the question **PQn. S17.1**
submit your solution in next 2 minutes
The session will resume in 3 minutes

Summary

- Modularization and importance of modularization
- Defining and invoking a function
- Flow of control of a program involving function call



problem solving using computers

CSE 1051

S-17_2 CATEGORIES OF FUNCTIONS

Objectives:

To learn and appreciate the following concepts

- Function prototypes
- To understand scope of variables
- Understand the different categories of functions
- Write programs using functions

Session outcome:

At the end of session one will be able to

- Describe function prototypes
- Differentiate local and global variables
- Understand modularization and function
- Write simple programs using functions

Function Prototypes

- Must be included for each function that will be defined, (required by Standards for C++ but optional for C) if not directly defined before main().
- In most cases it is recommended to include a function prototype in your program to avoid ambiguity.
- Identical to the function header, with semicolon (;) added at the end.
- Function prototype (declaration) includes
 - Function name
 - Parameters – what the function takes in and their type
 - Return type – data type function returns (default **int**)
- Parameter names are Optional.

Function Prototypes

- Function prototype provides the compiler the name and arguments of the functions and must appear before the function is used or defined.
- It is a model for a function that will appear later, somewhere in the program.
- General form of the function prototype:

fn_return_type fn_name(type par1, type par2, ..., type parN);

- Example:

int maximum(int, int, int);

- Takes in 3 **ints**
- Returns an **int**

Scope of Variables

- A scope is a region of the program where a defined variable can have its existence and beyond that variable it cannot be accessed.
- There are two types of variables in C
 - 1) **local** variables
 - 2) **global** variables

Local Variables

- Variables that are declared inside a function are called local variables.
- They can be used only by statements that are inside that function.
- In the following example all the variables a, b, and c are local to main() function.

```
#include <stdio.h>
int main () {
    /* local variable declaration */
    int a, b, c;
    a = 10; b = 20; c = a + b;
    printf ("value of a = %d, b = %d and c = %d\n", a, b, c);
    return 0;
}
```

Global Variables

- Global variables are defined outside a function, usually on top of the program.
- Global variables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.

```
#include <stdio.h>
int g; /* global variable declaration */
int main () {
    int a, b; /* local variable declaration */
    a = 10; b = 20; g = a + b;
    printf ("value of a = %d, b = %d and g = %d\n", a, b, g);
    return 0;
}
```

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 ch){  
    printf(" You have entered %d &%c", num,ch);  
}
```

4. Use of parameter names in the declaration(prototype) is optional but parameter type is a must.

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

Functions- **points to note**

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

Functions- Categories

Categorization based on the arguments and return values

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
(will see later with parameter passing techniques).

Function with No Arguments/parameters & No return values

```
void dispPattern(void); // prototype
```

```
int main(){
    printf("fn to display a line of stars\n");
    dispPattern();
    return 0;
}
```

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

Function with No Arguments but A return value

```
int readNum(void); // prototype

int main(){
    int c;
    printf("Enter a number \n");
    c=readNum();
    printf("The number read is %d",c);
    return 0;
}

int readNum(){
    int z;
    scanf("%d",&z);
    return(z);
}
```

Fn with Arguments/parameters & No return values

```
void dispPattern(char ch); // prototype

int main(){
    printf("fn to display a line of patterns\n");
    dispPattern('#');
    dispPattern('*');
    dispPattern('@');
    return 0;
}

void dispPattern(char ch ){
    int i;
    for (i=1;i<=20 ; i++)
        printf("%c",ch);
}
```

Function with Arguments/parameters & One return value

```
int main(){
    int a,b,c;
    printf("\nEnter numbers to be added\n");
    scanf("%d %d",&a,&b);
    c=fnAdd(a,b);
    printf("Sum is %d ", c);
    return 0;
}
int fnAdd(int x, int y ){
    int z;
    z=x+y
    return(z);
}
```



Go to posts/chat box for the link to the question **PQn. S17.2**
submit your solution in next 2 minutes
The session will resume in 3 minutes

Summary

- Function Prototypes
- Local variables and global Variables
- Different categories of functions
- Simple programs using functions



problem solving using computers

CSE 1051

S-18_1 PARAMETER PASSING TECHNIQUES

Objectives:

To learn and appreciate the following concepts

- Write C functions
- Invoking functions
- Write programs using functions
- Parameter passing techniques
- Pass by value

Session outcome

At the end of session one will be able to understand:

- The overall ideology of parameter passing techniques

Problems:

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.

Factorial of a given number ‘n’

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

int main() {
    int n;
    long int f;

    printf("Enter a number :");
    scanf("%d",&n);
    f =factFn(n);
    printf("Fact= %ld",f);

    return 0;
}
```

```
long int factFn(int num) {
    //function definition
    int i;
    long int fact=1;

    //factorial computation
    for (i=1; i<=num; i++)
        fact=fact * i;

    // return the result
    return (fact);
}
```

Reversing a given number ‘n’

```
int Reverse(int); //prototype

int main()
{
    int n,r;
    printf("Enter a number : \n");
    scanf("%d", &n);

    r= Reverse(n);
    printf(" reversed no=%d",r)

    return 0;
}
```

```
int Reverse(int num)
{
    int rev=0;
    int digit;

    while(num!=0)
    {
        digit = num % 10;
        rev = (10 * rev) + digit;
        num = num/10;
    }
    return (rev);
}
```

Check whether given number is prime or not

```
int IsPrime(int); //prototype
```

```
int main() {
    int n;

    printf("Enter a number : ");
    scanf("%d",&n);
    if (IsPrime(n))
        Printf("%d is a prime no",n);
    else
        printf("%d is not a prime no",n);
    return 0;
}
```

```
int IsPrime(int num) //prime check
{
    int p=1;
    for(int j=2;j<=num/2;j++)
    {
        if(num%j==0)
        {
            p=0;
            break;
        }
    }
    return p;
}
```

First n Fibonacci number generation

```
void fibFn(int); //prototype

int main() {
    int n;
    printf("Enter the limit ");
    scanf("%d",&n);
    fibFn(n); //function call
    return 0;
}
```

```
void fibFn(int lim) { //fib generation
    int i, first, sec, next;
    if (lim<=0)
        printf("limit should be +ve.\n");
    else
    {
        printf("\nFibonacci nos\n");
        first = 0, sec = 1;
        for (i=1; i<=lim; i++) {
            printf("%d", first)
            next = first + sec;
            first = sec;
            sec = next;
        }
    }
}
```

Functions-Overview:

Parameters/Arguments

```
→ void dispNum( int n ) // function definition
```

```
{
```

```
    printf(" The entered num=%d", n);
```

```
}
```

Formal parameters

```
int main(){ //calling program
```

```
    int no;
```

```
    printf("Enter a number \n");
```

```
    scanf("%d",&no);
```

```
    dispNum( no ); //Function reference
```

```
    return 0;
```

```
}
```

Actual parameters

Functions- Parameter Passing

- Pass by value (call by value)
- Pass by reference (call by reference)

Pass by value:

```
void swap(int x, int y )  
{  
    int t=x;  
    x=y;  
    y=t;  
    printf("In fn: x= %d and y=%d ",x,y);  
}  
  
int main()  
{  
    int a=5,b=7;  
    swap(a, b);  
    printf("After swap: a= %d and b= %d",a,b);  
    return 0;  
}
```

Output:

In fn: x = 7 & y = 5

After swap: a = 5 & b = 7



Go to posts/chat box for the link to the question **PQn. S18.1**
submit your solution in next 2 minutes
The session will resume in 3 minutes

Summary

- Write C Functions and Invoke them
- Simple programs using functions
- Parameter Passing
- Pass by Value



problem solving using computers

CSE 1051

S-18_2 PARAMETER PASSING TECHNIQUES

Objectives

To learn and appreciate the following concept:

- Pass by reference
- Returning multiple values from functions
- Nested Functions

Session outcome

At the end of session one will be able to understand:

- The overall ideology of parameter passing techniques

Functions-Overview:

Parameters/Arguments

```
→ void dispNum( int n ) // function definition
```

```
{
```

```
    printf(" The entered num=%d", n);
```

```
}
```

Formal parameters

```
int main(){ //calling program
```

```
    int no;
```

```
    printf("Enter a number \n");
```

```
    scanf("%d",&no);
```

```
    dispNum( no ); //Function reference
```

```
    return 0;
```

```
}
```

Actual parameters

Functions- Parameter Passing

- Pass by value (call by value)
- Pass by reference (call by reference)

Pass by value:

```
void swap(int x, int y )  
{  
    int t=x;  
    x=y;  
    y=t;  
    printf("In fn: x= %d and y=%d ",x,y);  
}  
  
int main()  
{  
    int a=5,b=7;  
    swap(a, b);  
    printf("After swap: a= %d and b= %d",a,b);  
    return 0;  
}
```

Output:

In fn: x = 7 & y = 5

After swap: a = 5 & b = 7

Pass by Reference – Using Pointers:

```
void swap(int *x, int *y )
```

```
{
```

```
    int t=*x;  
    *x=*y;  
    *y=t;
```

```
}
```

Change is directly on the variable using the reference to the address.

When function is called:

address of a → x
address of b → y

```
int main()
```

```
{
```

```
    int a=5,b=7;
```

```
    swap(&a, &b);
```

```
    printf("After swap: a=%d and b= %d",a,b);
```

```
    return 0; }
```

Output:

After swap: a = 7 and b = 5

Pointers as functions arguments:

When we pass addresses to a function, the parameters receiving the addresses should be pointers.

```
#include <stdio.h>

int change (int *p)
{
    *p = *p + 10 ;
    return 0;
}

int main()
{
    int x = 20;
    change(&x);
    printf("x after
change==%d",x);
    return 0;
}
```

Output :
X after change=30

Pointers as function arguments

- When the function `change()` is called, the address of the variable `x`, not its value, is passed into the function `change()`.
- Inside `change()`, the variable `p` is declared as a pointer and therefore `p` is the address of the variable `x`. The statement
 - `*p=*p +10` adds 10 to the value stored at the address `p`. Since `p` represents the address of `x`, the value of `x` is changed from 20 to 30. therefore it prints 30.

Function that return multiple values-Using pointers

Using pass by reference we can write a function that return multiple values.

```
void fnOpr(int a, int b, int *sum, int *diff) {
```

```
    *sum = a + b;
```

```
    *diff = a - b; }
```

```
int main() {
```

```
    int x, y, s, d;
```

```
    printf("Enter two numbers: \n");
```

```
    scanf("%d %d", &x, &y);
```

```
    fnOpr(x, y, &s, &d);
```

```
    printf("Sum = %d & Diff = %d ", s, d);
```

```
    return 0; }
```

Output:

x= 5 & y= 3

Sum = 8 & Diff = 2

Nesting of functions:

- C language allows nesting of functions by calling one function inside another function.
- Nesting of function does not mean that we can define an entire function inside another function. The following examples shows both valid and invalid function nesting in C language

// Wrong way of function nesting

```
void fun()
{
    printf("I am having Fun....");

    void sleep()
    {
        printf("I am having sleep");
    }
}
```

// Right way of function nesting

```
void sleep()
{
    printf("I am having sleep");
}

void fun()
{
    printf("I am having Fun....");
    sleep();
}
```

Nesting of Functions:

```
void First (void){    // FUNCTION DEFINITION
    printf("I am now inside function First\n");
}

void Second (void){ // FUNCTION DEFINITION
    printf( "I am now inside function Second\n");
    First();        // FUNCTION CALL
    printf("Back to Second\n");
}

int main (){
    printf( "I am starting in function main\n");
    First ();       // FUNCTION CALL
    printf( "Back to main function \n");
    Second ();     // FUNCTION CALL
    printf( "Back to main function \n");
    return 0;
}
```

Nesting of Functions:

```
void fnOpr(int a, int b, int *sum, int *diff)
{
    *sum = a + b;
    if (fnDiff(a,b))
        *diff = a -b;
    else
        *diff = b - a;
}
int main() {
    int x, y, s, d;
    printf("Enter the values: \n");
    scanf("%d %d", &x, &y);
    fnOpr(x, y, &s, &d);
    printf("The results are, Sum =%d and Diff = %d", s, d);
    return 0; }
```

```
int fnDiff(int p, int q) {
    if (p>q)
        return(1);
    else
        return (0);}
```

Output:

$x= 3 \& y= 5$
 $s =8 \& d = 2$



Go to posts/chat box for the link to the question **PQn. S18.2**
submit your solution in next 2 minutes
The session will resume in 3 minutes

Summary:

- Parameter passing techniques
 - pass by value
 - pass by reference
- Pointers as functions arguments
- Function returning Multiple values
- Nesting of functions



problem solving using computers

CSE 1051

19.1 PASSING 1-D AND 2-D ARRAY TO FUNCTIONS

Passing 1D-Array to Function

Rules to pass an array to a function

- The function must be called by passing only the name of the array.
- In the function definition, the formal parameter must be an array type; the size of the array does not need to be specified.
- The function prototype must show that argument is an array.

Passing 1D-Array to Function:

```
int fnSum( int a[ ], int n ) {  
    int sum=0,i;  
    for(i=0;i<n;i++)  
        sum+=a[i];  
    return (sum); }
```

```
int main() {  
    int n, a[20], x, y,i;  
    printf("Enter the limit \n");  
    scanf("%d",&n);  
    printf("Enter the values: \n");  
    for (i=0; i<n; i++)  
        scanf("%d",&a[i]);  
    printf("The sum of array elements is =%d ",fnSum(a, n));//fn call  
    return 0; }
```

Output: n=5

1, 2, 3, 4, 5

Sum of elements = 15

Array name is passed along
with number of elements



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.

Passing 2D-Array to Function:

```
int fn2d(int x[ ][10], int m, int n)
{
    int i, j, sum=0;
    for(i=0; i<m; i++)
        for(j=0; j<n; j++)
            sum+=x[i][j];
    return(sum);
}

int main() {
    int i, j, a[10][10], m, n;
    printf("Enter dimensions of matrix");
    scanf("%d%d", &m, &n);
    printf("Enter the elements");
    for(i=0;i<m;i++)
        for(j=0;j<n;j++)
            scanf("%d",&a[i][j]);
    printf ("Sum of elements of 2D array is=%d",fn2d(a, m, n));
    return 0;
}
```

Output: m=2 n=3

1 2

3 4

5 6

Sum of elements = 21

Extra problems:

- Write a c program to add all the even elements of an array using a function Add().
- Write a C program to replace all odd numbers of an array with the largest number in the array using a function Replace().
- Write a C program to replace all the zeros in the matrix by the trace of the matrix using a function Trace().
- Write a C program using pass-by-pointer method to compute the compound interest using a function Compound().

Write a c program to add all the even elements of an array using a function Add()

```
#include <stdio.h>
int Add( int a[ ], int n )
{
    int sum=0,i;
    for(i=0;i<n;i++)
    {
        if((a[i]%2) == 0)
            sum+=a[i];
    }
    return (sum);
}
```

```
int main()
{
    int n, a[20], x, y,i;
    printf("Enter the limit \n");
    scanf("%d",&n);
    printf("Enter the values: \n");
    for (i=0; i<n; i++)
        scanf("%d",&a[i]);
    printf("The sum of odd array
elements is =%d
",Add(a,n));
    Return 0;
}
```

Write a C program to replace all odd numbers of an array with the largest number in the array using a function Replace()

```
#include <stdio.h>
void Replace( int arr[ ],  
int n)  
{  
    // To find the largest  
    int i, max = arr[0];  
    for (i = 1; i < n; i++)  
        if (arr[i] > max)  
            max = arr[i];  
  
    // To replace  
    for(i=0;i<n;i++)  
    {  
        if(arr[i]%2 != 0)  
            arr[i]=max;  
    }  
}  
  
int main()  
{  
    int n, a[20], x, y,i;  
    printf("Enter the limit \n");  
    scanf("%d",&n);  
    printf("Enter the values: \n");  
    for (i=0; i<n; i++)  
        scanf("%d",&a[i]);  
    Replace(a,n);  
    printf("The array after replacement is\n");  
    for (i=0; i<n; i++)  
        printf("%d \n",a[i]);  
  
    return 0;  
}
```

Write a C program to replace all the zeros in the matrix by the trace of a square matrix using a function Trace()

```
#include <stdio.h>
```

```
void Trace(float a[ ][10], int n)
{
    int i, j, tr=0;

    // Finding Trace
    for(i=0;i<n;i++)
    {
        tr=tr+a[ i ][ i ];
    }
    //Replacing zeros
    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
            if(a[i][j]==0)
                a[i][j]=tr;
}
```

```
int main()
{
    int i, j, n;
    float a[10][10];
    printf("Enter the rows or columns of a
square matrix");
    scanf("%d", &n);
    printf("Enter the elements");
    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
            scanf("%f",&a[i][j]);
    Trace(a, n);
    printf("Matrix after replacement \n");
    for(i=0;i<n;i++)
    {
        for(j=0;j<n;j++)
        {
            printf("%f",a[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

Write a C program using pass-by-pointer method to compute the simple interest and compound interest using a function SI_CI()

```
#include <stdio.h>
#include <math.h>
int main() {
    float p,q,r,SI,CI;
    int n;
    printf("Enter the value of Principal p = ");
    scanf("%f",&p);
    printf("Enter the value of Rate r = ");
    scanf("%f",&r);
    printf("Enter the value of Period in year n = ");
    scanf("%d",&n);

    SI_CI(&p,&r,&n,&SI,&CI);

    printf("Simple Interest SI=%f \n",SI);
    printf("Compound Interest CI=%f \n",CI);

    return 0;
}
```

Write a C program using pass-by-pointer method to compute the simple interest and compound interest using a function SI_CI()

```
void SI_CI(float *pr, float *ra, int *yr, float *smp, float  
*cmp)  
{  
    int amount;  
  
    // Simple interest  
    *smp = ((*pr)*(*ra)*(*yr)/100);  
  
    // Compound interest  
    amount= (*pr)*pow((1 +(*ra)/100),(*yr));  
    *cmp= amount-(*pr);  
}
```



Go to posts/chat box for the link to the question **PQn. S19.1**
submit your solution in next 2 minutes
The session will resume in 3 minutes

Summary:

- Parameter passing techniques
 - pass by value
void swap(int x, int y)
 - pass by reference
void swap(int *x, int *y)
- Passing 1 D array
 int fnParr(int a[], int n)
- Passing 2 D array
 int fn2d(int x[][10], int m, int n)



problem solving using computers

CSE 1051

19.2 INTRODUCTION TO RECURSION

Objectives:

To learn and understand the following concepts:

- ✓ To design a recursive algorithm
- ✓ To solve problems using recursion
- ✓ To understand the relationship and difference between recursion and iteration

Session outcome:

At the end of session one will be able to :

- Understand recursion
- Write simple programs using recursive functions

What is Recursion ?

- Sometimes, the best way to solve a problem is by solving a smaller version of the exact same problem first.
- Recursion is a technique that solves a problem by solving a smaller problem of the same type.
- A *recursive function* is a function that invokes / calls itself directly or indirectly.
- In general, code written recursively is shorter and a bit more elegant, once you know how to read it.
- It enables you to develop a natural, straightforward, simple solution to a problem that would otherwise be difficult to solve.

What is Recursion ?

- Mathematical problems that are recursive in nature like factorial, fibonacci, exponentiation, GCD, Tower of Hanoi, etc. can be easily implemented using recursion.
- Every time when we make a call to a function, a stack frame is allocated for that function call where all the local variables in the functions are stored. As recursion makes a function to call itself many times till the base condition is met, many stack frames are allocated and it consumes too much main memory and also makes the program run slower.
- We must use recursion only when it is easy and necessary to use, because it takes more space and time compared to iterative approach.

Let us consider the code ...

```
int main() {  
    int i, n, sum=0;  
    printf("Enter the limit");  
    scanf("%d",n);  
    printf("The sum is %d",fnSum(n));  
    return 0;  
}
```

```
int fnSum(int n)  
{  
    int sum=0;  
    for(i=1;i<=n;i++)  
        sum=sum+i;  
    return (sum);  
}
```

Recursive Thinking

Recursion Process

A child couldn't sleep, so her mother told a story about a little frog,
who couldn't sleep, so the frog's mother told a story about a little
bear,

who couldn't sleep, so bear's mother told a story about a little
weasel

...who fell asleep.

...and the little bear fell asleep;

...and the little frog fell asleep;

...and the child fell asleep.

Steps to Design a Recursive Algorithm

- Base case:
 - It prevents the recursive algorithm from running forever.
- Recursive steps:
 - Identify the base case for the algorithm.
 - Call the same function recursively with the parameter having slightly modified value during each call.
 - This makes the algorithm move towards the base case and finally stop the recursion.

Let us consider same code again ...

```
int main() {  
    int i, n, sum=0;  
    printf("Enter the limit");  
    scanf("%d", &n);  
    printf("The sum is %d",fnSum(n));  
    return 0;  
}
```

```
int fnSum(int n){  
    int sum=0;  
    for(i=1;i<=n;i++)  
        sum=sum+i;  
    return (sum);  
}
```

```
int fnSum(int x)  
{  
    if (x == 1) //base case  
        return 1;  
    else  
        return x + fnSum(x-1);  
        //recursive case  
}
```

Factorial of a natural number–

A classical recursive example

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

So $\text{factorial}(5)$
= $5 * \text{factorial}(4)$
= $4 * \text{factorial}(3)$
= $3 * \text{factorial}(2)$
= $2 * \text{factorial}(1)$
= $1 * \text{factorial}(0)$
= 1

Factorial- recursive procedure

```
#include <stdio.h>
```

```
long factorial (long a) {
    if (a ==0) //base case
        return (1);
    return (a * factorial (a-1));
}
```

```
int main () {

    long number;
    printf("Please type a number: ");
    scanf("%ld", &number);
    printf("%ld != %d", number,factorial (number));
    return 0;
}
```

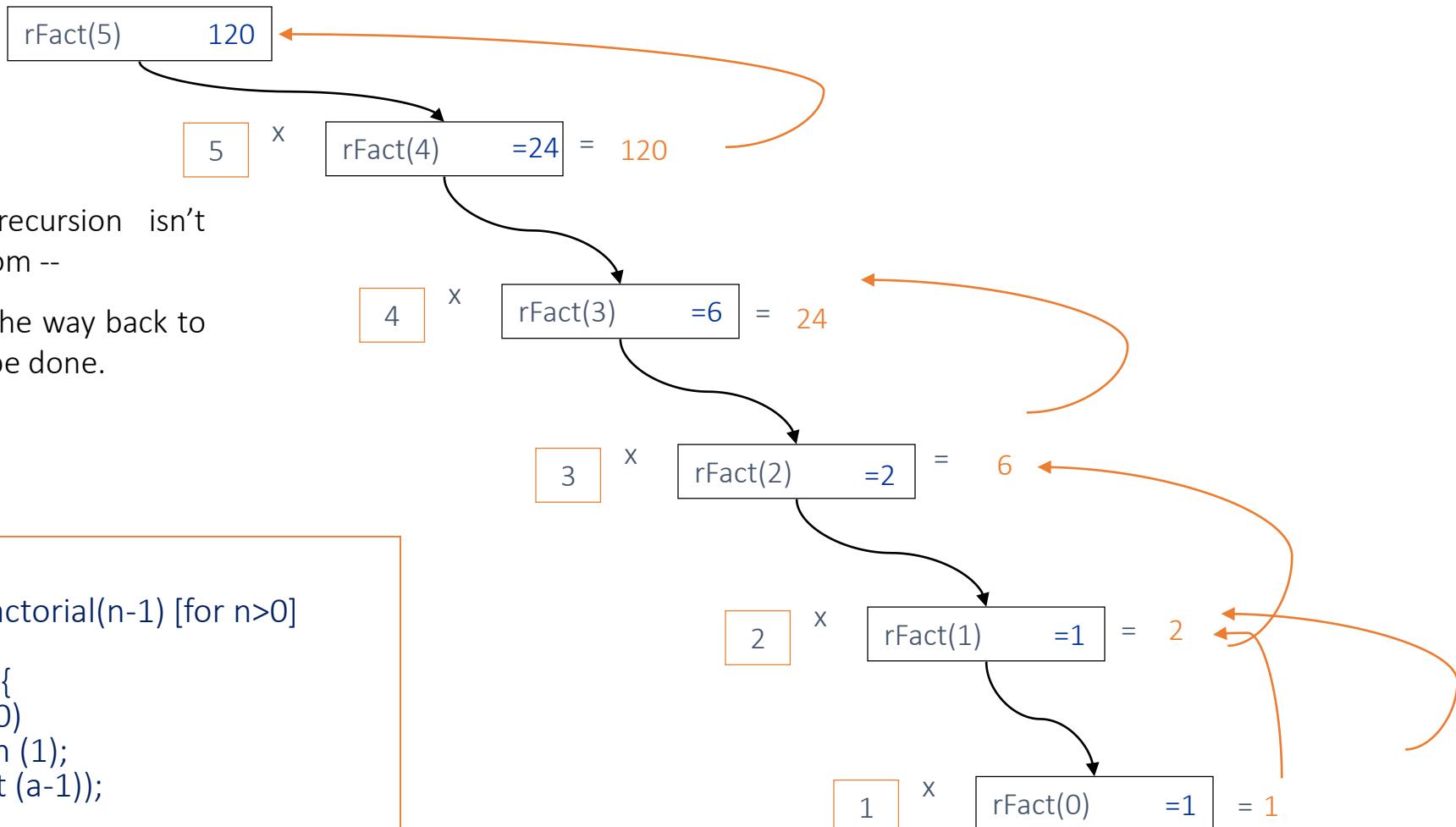
Recursion - How is it doing!

Notice that the recursion isn't finished at the bottom --

It must unwind all the way back to the top in order to be done.

```
factorial(0) = 1  
factorial(n) = n * factorial(n-1) [for n>0]
```

```
long rFact (long a) {  
    if (a ==0)  
        return (1);  
    return (a * rFact (a-1));  
}
```





Go to posts/chat box for the link to the question **PQn. S19.2**
submit your solution in next 2 minutes
The session will resume in 3 minutes

Summary

- Recursion definition
- Example for recursive function



problem solving using computers

CSE 1051

20.1 RECURSION EXAMPLES

Objectives:

To learn and understand the following concepts:

- ✓ To design a recursive algorithm
- ✓ To solve problems using recursion
- ✓ To understand the relationship and difference between recursion and iteration

Session outcome:

At the end of session one will be able to :

- Understand recursion
- Write simple programs using recursive functions

Fibonacci Numbers: Recursion

$$\text{fib}(0) = 0$$

$$\text{fib}(1) = 1$$

$$\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2) \text{ [for } n>1]$$

$$\text{So } \text{fib}(4)$$

$$= \text{fib}(3) + \text{fib}(2)$$

$$= (\text{fib}(2) + \text{fib}(1)) + (\text{fib}(1) + \text{fib}(0))$$

$$= ((\text{fib}(1) + \text{fib}(0)) + 1) + (1 + 0)$$

$$= (1 + 0) + 1 + (1 + 0)$$

$$= 3$$

Fibonacci Numbers: Recursion

Fibonacci series is 0,1, 1, 2, 3, 5, 8 ...

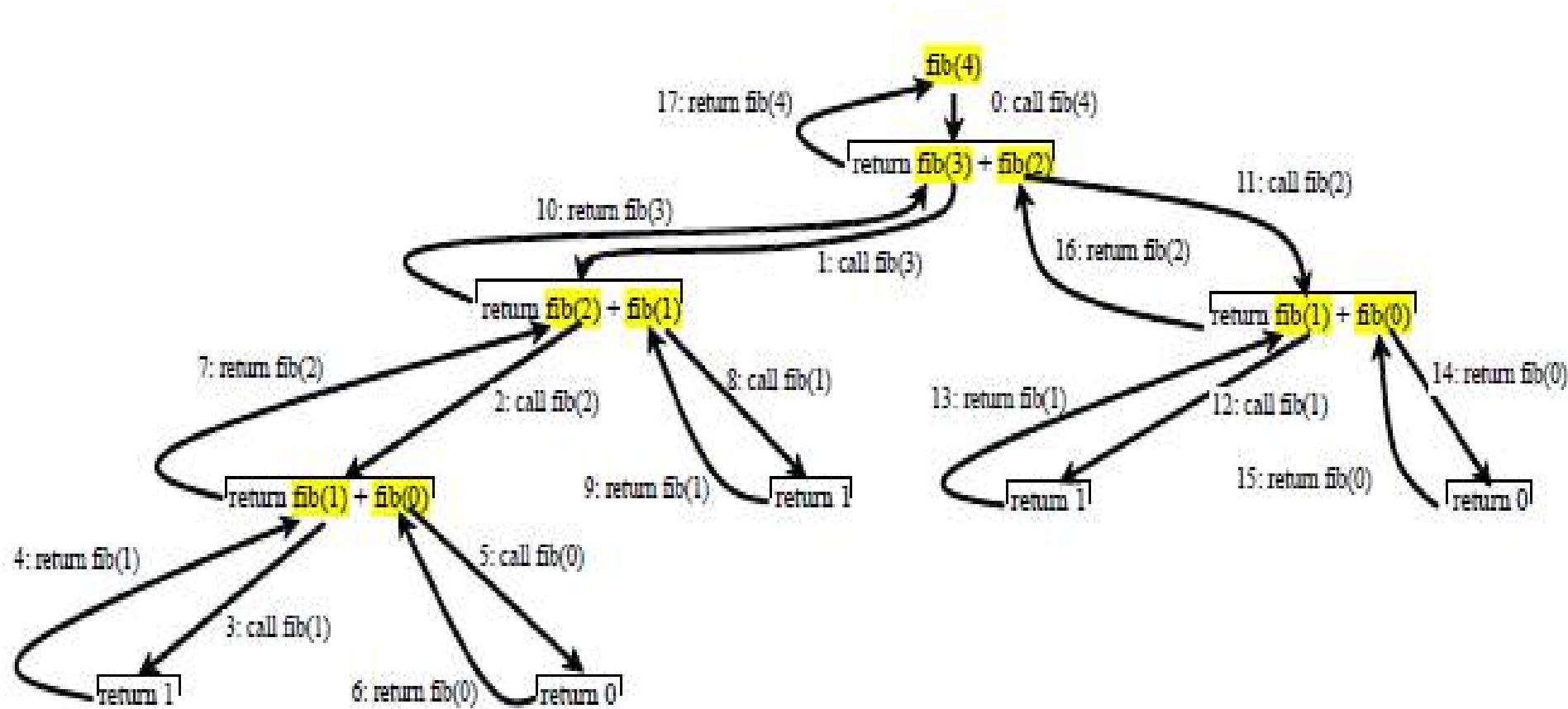
```
int fib(int n)
{
    if (n <= 1)
        return n;
    else
        return (fib(n-1) + fib(n-2));
}
```

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

Output:

$n = 4$
 $\text{fib} = 3$

Recursive Calls initiated by Fib(4)



Fibonacci Series using Recursion

```
int rfibo(int);
```

```
int main(void){  
    int n,i, a[20], fibo;  
    printf("enter any num to n\n");  
    scanf("%d", n);  
    printf("Fibonacci series ");  
    for (i=1; i<=n; i++)  
    {  
        fibo = fib(i);  
        printf("%d ", fibo);  
    }  
    return 0;  
}
```

Static Variable:

The value of static variable persists until the end of the program.

Static variables can be declared as

`static int x;`

A static variable can be either an internal or external type depending on the place of declaration.

```
void fnStat();  
int main() {  
    int i;  
    for( i= 1; i<=3; i++)  
        fnStat();  
    return 0;  
}
```

`static int x = 0;`

```
void fnStat(){  
    int x = 0;  
    x = x + 1;  
    printf("x=%d", x);  
}
```

Output:
`x = 1`
`x = 2`
`x = 3`

GCD: Recursion

$$\text{gcd}(x, y) = \begin{cases} x & \text{if } y = 0 \\ \text{gcd}(y, \text{remainder}(x, y)) & \text{if } x \geq y \text{ and } y > 0 \end{cases}$$

```
int gcd(int x, int y)
{
    if(x == 0)
        return (y);
    if(y==0)
        return (x);
    return gcd(y, x % y);
}
```

gcd(24,9) ← Control In gcd fn on call

gcd(9,24%9)

gcd(6,9%6)

gcd(3,6%3)

return values

gcd(9, 6)

gcd(6, 3)

gcd(3, 0)

return 3

return 3

return 3

Output:

x= 24 , y = 9

gcd = 3

Extra Problem- Finding product of two numbers

```
#include <stdio.h>
int product(int, int);
int main()
{
    int a, b, result;
    printf("Enter two numbers to find their product: ");
    scanf("%d%d", &a, &b);
    result = product(a, b);
    printf("%d * %d = %d\n", a, b, result);
    return 0;
}
```

Output:

```
Enter two numbers to find their product: 10 20
10*20=200
```

```
int product(int a, int b)
{
    if (a < b)
    {
        return product(b, a);
    }
    else if (b != 0)
    {
        return (a + product(a, b - 1));
    }
    else
    {
        return 0;
    }
}
```

Extra Problem- Dividing two numbers

```
#include <stdio.h>
int divide(int a, int b);

int main()
{
    int a,b;
    printf("Enter two numbers for division");
    scanf("%d%d", &a,&b);
    printf("%d/%d=%d",a,b,divide(a,b));

    return 0;
}
```

```
int divide(int a, int b)
{
    if(a - b <= 0)
    {
        return 1;
    }
    else
    {
        return divide(a - b, b) + 1;
    }
}
```

Output:

```
Enter two numbers for division: 20 10
20/10=2
```

Extra Problem- Calculating power of a number

```
#include <stdio.h>
int power(int n1, int n2);

int main()
{
    int base, powerRaised, result;

    printf("Enter base number: ");
    scanf("%d",&base);

    printf("Enter power number");
    scanf("%d",&powerRaised);

    result = power(base, powerRaised);

    printf("%d^%d = %d", base, powerRaised, result);
    return 0;
}
```

```
int power(int base, int powerRaised)
{
    if (powerRaised != 0)
        return (base*power(base, powerRaised-1));
    else
        return 1;
}
```

Output:

```
Enter base number:3
Enter power number: 4
3 ^ 4=81
```

Recursion - Should I or Shouldn't I?

- Pros
 - Recursion is a natural fit for recursive problems
- Cons
 - Recursive programs typically use a large amount of computer memory and the greater the recursion, the more memory used
 - Recursive programs can be confusing to develop and extremely complicated to debug

Recursion *vs* Iteration

RECURSION	ITERATION
Uses more storage space requirement	Less storage space requirement
Overhead during runtime	Less Overhead during runtime
Runs slower	Runs faster
A better choice, a more elegant solution for recursive problems	Less elegant solution for recursive problems

Recursion – Do's

- You must include a **termination** condition or **Base Condition** in recursive function; Otherwise your recursive function will run “forever” or **infinite**.
- Each successive call to the recursive function must be nearer to the base condition.



Go to posts/chat box for the link to the question **PQn. S20.1**
submit your solution in next 2 minutes
The session will resume in 3 minutes

Summary

- Definition
- Recursive functions
- Problems Solving Using Recursion



problem solving using computers

CSE 1051

20.2 FURTHER PROGRAMS ON RECURSION

Objectives:

To learn and understand the following concepts:

- ✓ To understand recursive algorithm
- ✓ To solve few problems including sorting using recursion

Session outcome:

At the end of session one will be able to :

- Understand recursion
- Write programs using recursive functions

Extra Problem- Sum of natural numbers

```
#include <stdio.h>
int sum(int n);

int main()
{
    int number, result;

    printf("Enter a positive integer: ");
    scanf("%d", &number);

    result = sum(number);

    printf("sum=%d", result);
}
```

```
int sum(int num)
{
    if (num!=0)
        return num + sum(num-1);
    else
        return num;
}
```

Output:

```
Enter a positive integer: 10
Sum= 55
```

Extra Problem- To count number of digits

```
#include <stdio.h>
int countDigits(int);
int main()
{
    int number;
    int count=0;

    printf("Enter a positive integer number: ");
    scanf("%d",&number);

    count=countDigits(number);

    printf("Number of digits is: %d\n",count);
    return 0;
}

int countDigits(int num)
{
    static int count=0;

    if(num>0)
    {
        count++;
        countDigits(num/10);
    }
    else
    {
        return count;
    }
}
```

Output:

```
Enter a positive integer number: 123
Number of digits is: 3
```

Extra Problem- To find sum of all digits

```
#include <stdio.h>
```

```
int sumDigits(int num);
```

```
int main()
```

```
{
```

```
    int number,sum;
```

```
    printf("Enter a positive integer number: ");
```

```
    scanf("%d",&number);
```

```
    sum=sumDigits(number);
```

```
    printf("Sum of all digits are: %d\n",sum);
```

```
    return 0;
```

```
}
```

```
int sumDigits(int num)
```

```
{
```

```
    static int sum=0;
```

```
    if(num>0)
```

```
{
```

```
        sum+=(num%10);
```

```
        sumDigits(num/10);
```

```
}
```

```
else
```

```
{
```

```
    return sum;
```

```
}
```

Output:

```
Enter a positive integer number: 123
```

```
Sum of digits is: 6
```

Extra Problem-Reversing a Number

```
#include <stdio.h>
int rev(int);
int main() {
    int num;
    printf("enter number");
    scanf("%d",num);
    printf("%d", rev(num));
    return 0;
}
int rev(int num){
    static int n = 0;
    if(num > 0)
        n = (n* 10) + (num%10) ;
    else
        return n;
    return rev(num/10);
}
```

Output:

num = 234
rev = 432

Extra Problem- To find length of a string

```
#include <stdio.h>
int length(char [], int);

int main()
{
    char str[20];
    int count;

    printf("Enter any string :: ");
    scanf("%s", str);
    count = length(str, 0);
    printf("The length of string=%d.\n",count);
    return 0;
}
```

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

Output:

```
Enter any string :: Manipal
The length of string= 7
```

Extra Problem-Binary Search

```
#include<stdio.h>
int binarySearch(int x[],int element,int start,int end);
int main(){
    int x[20],n,i,index,start=0,end;element;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    end = n;
    printf("Enter array elements: ");
    for(i=0;i<n;i++){
        scanf("%d",&x[i]);
    }
    printf("Enter the element to search: ");
    scanf("%d",&element);
    index = binarySearch(x[element,start,end-1];
    if(index == -1)
        printf("Element Not Found.\n");
    else
        printf("Element found at index : %d\n",index);
    return 0;
}
```

Extra Problem-Binary Search

```
int binarySearch(int x[],int element,int start,int end){  
    int mid,noOfElements,i;  
    mid = (int)(start+end)/2;  
    if(start > end)  
        return -1;  
    if(x[mid] == element)  
        return mid;  
    else if(x[mid] < element){  
        start = mid+1;  
        return binarySearch(x,element,start,end);  
    }  
    else{  
        end = mid-1;  
        return binarySearch(x,element,start,end);  
    }  
}
```

Output:

```
Enter number of elements: 5  
Enter array elements: 1 2 3 4 5  
Enter the element to search: 3  
Element found at index : 2
```

Extra Problem- Recursive Sorting

Base Case:

if length of the list (n) = 1

No sorting, return

Recursive Call:

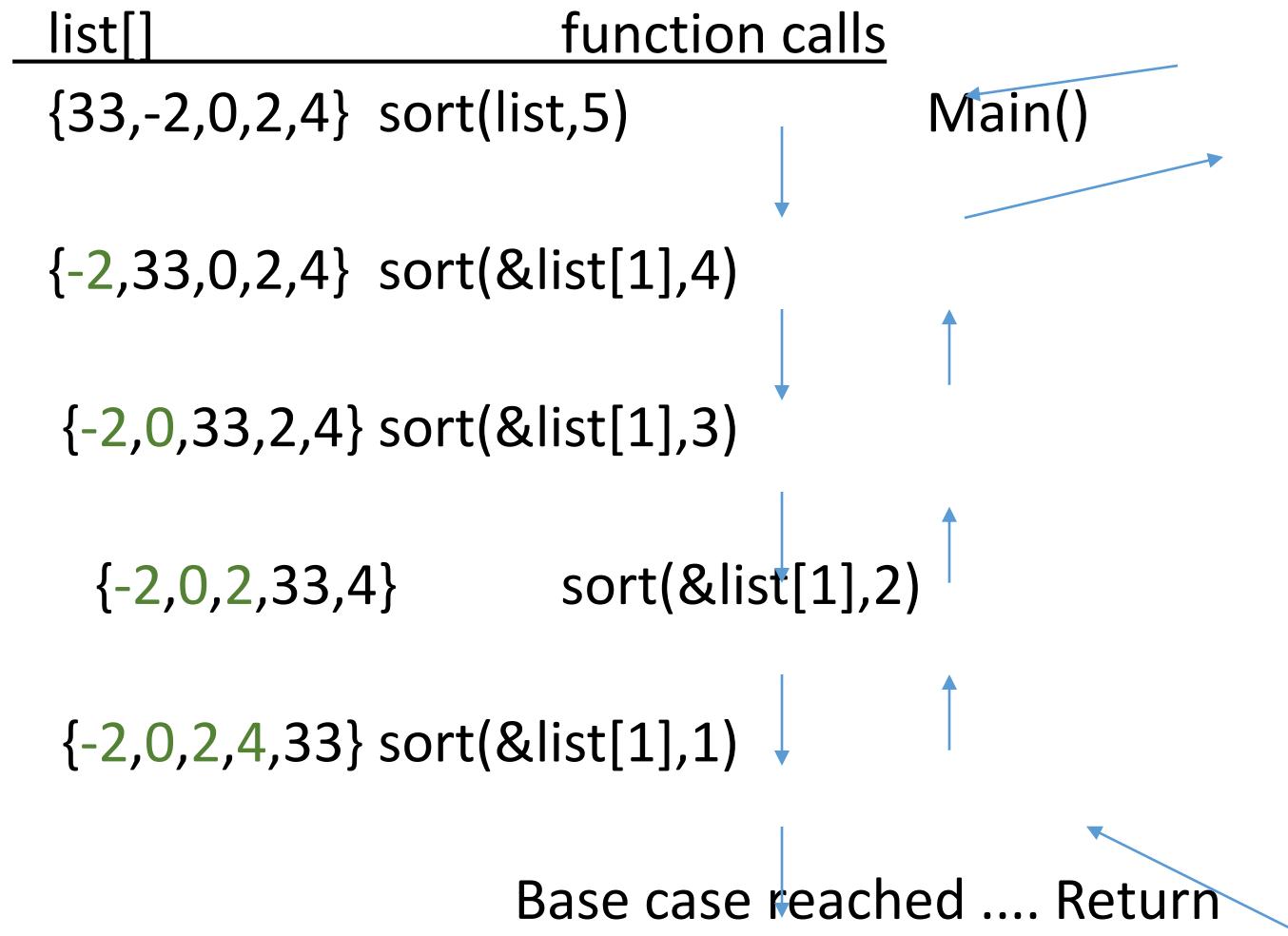
1. Find the smallest element in the list and place it in the 0th position

2. Sort the unsorted array from 1.. $n-1$

`sortR(&list[1], n-1)`

For eg: list [] = {33,-2,0,2,4} n=5

Extra problem-Sorting



Extra problem - Sorting

```
sortR(list, n); // call of fn & display of sorted array in main()
```

```
int sortR(int list[], int ln){  
    int i, tmp, min;  
    if(ln == 1)  
        return 0;  
    /* find index of smallest no */  
    min = 0;  
    for(i = 1; i < ln; i++)  
        if(list[i] < list[min])  
            min = i;  
    /* move smallest element to 0-th  
       element */  
    tmp = list[0];  
    list[0] = list[min];  
    list[min] = tmp;  
    /* recursive call */  
    return sortR(&list[1], ln-1);  
}
```

Output:

Origin. array-: 33 -2 0 2 4
Sorted array -: -2 0 2 4 33



Go to posts/chat box for the link to the question **PQn. S20.2**
submit your solution in next 2 minutes
The session will resume in 3 minutes

Summary

- Definition
- Recursive functions
- Problems Solving Using Recursion



S 2 1 _ 1 S t r u c t u r e s



Objectives

To learn and appreciate the following concepts

- Basic operations and programs using structures
- Advantages of structures over array



Session outcome

At the end of session one will be able to

1. Understand the overall ideology of structures
2. Write programs using structures



Introduction

- We've seen variables of simple data types, such as **float**, **char**, and **int**.
- We saw **derived data** type, **arrays** to store group of related data.
- Variables of such types represent one item of information: a **height**, an **amount**, a **count**, or group of item with same data type: **list[10]** and so on.
- But, these basic types does not support the storage of compound data.
Eg. **Student** {name, address, age, sem, branch}

Introduction

- C provides facility to define one's own type (user-defined) that may be a **composite** of basic types (**int**, **char**, **double**, etc) and other user-defined types.

✓ **Structures**



Introduction

- Definition:
 - collection of *one or more variables, possibly of different types*, grouped together under a *single name* for convenient handling
- A structure type in C is called **struct**.



Structures

- Structures hold data that belong together.
- Examples:
 - ❑ Student record: student id, name, branch, gender, start year, ...
 - ❑ Bank account: account number, name, address, balance, ...
 - ❑ Address book: name, address, telephone number, ...
- In database applications, structures are called records.

Structure versus Array

- A **struct** is **heterogeneous**, that means it can be composed of data of different types.
- In contrast, **array** is **homogeneous** since it can contain only data of the same type.

Structure Definition - Syntax

The general format of a structure **definition**

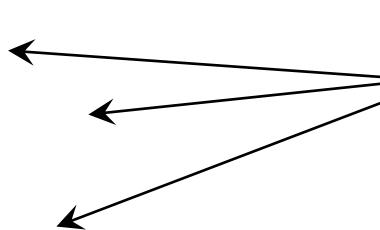
```
struct structure_name
{
    data_type member1;
    data_type member2;
    ...
};
```



Structure Definition - Examples

- Example:

```
struct Date
{
    int day;
    int month;
    int year;
};
```



Members of the
structure Date



struct examples

- Examples:

i) **struct StudentInfo{**
 int Id;
 int age;
 char Gender;
 double CGA;
};

}

ii) **struct Employee{**
 char Name[15];
 char Address[30];
 int Age;
 float Basic;
 float DA;
 float NetSalary;
};

}

The “StudentInfo” structure has 4 members of different types.

The “Employee” structure has 6 members



- **Definition** of Structure reserves **no space**.
- It is nothing but the “ **Template / Map / Shape** ” of the structure .
- Memory is created, very first time when a **variable of structure type is created / Instance** is created.



Declaring Structure Variables

- Declaration of a variable of **struct** type using **struct tag name**, after structure definition:

```
<struct-type> <identifier_list>;
```

- Example:

```
StudentInfo Student1, Student2;
```

Student1

Name	
Id	Gender
Dept	

Student2

Name	
Id	Gender
Dept	

Student1 and Student2 are variables of StudentInfo type.

Declaring Structure Variables

Declare them at the time of structure definition:

```
struct student
{
    int rollno;
    int age;
    char name[10];
    float height;
}s1, s2, s3; /* Defines 3 variables of type student */
```



Members of a structure themselves are not variables. i.e. **rollno** alone does not have any value or meaning.

Member or dot operator

- The link between member and a structure variable is established using the **member operator ‘.’** which is also known as ‘**dot operator**’

<struct-variable>.<member_name>

e.g.: student s1; // s1 is a variable of type
//student structure.

s1. rollno;

s1. age;

s1. name;

Example of Same Member Names in Different structures

```
struct fruit {  
    char name[15];  
    int calories;  
};
```

```
struct vegetable {  
    char name[15];  
    int calories;  
}
```

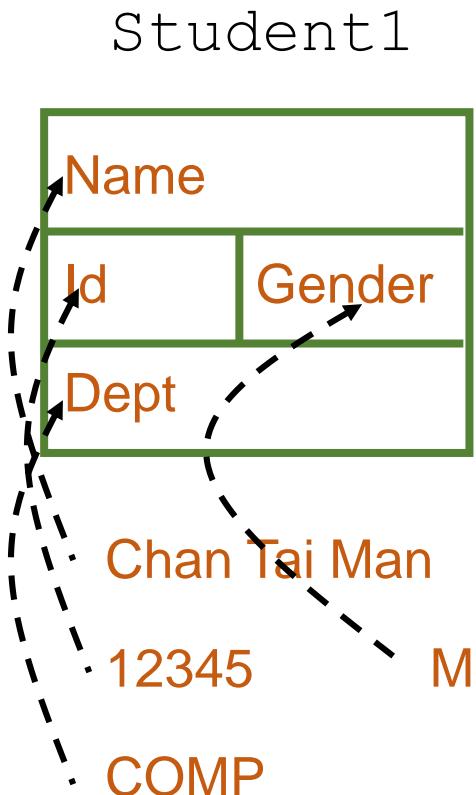
```
struct fruit    a;  
struct vegetable b;
```

We can access **a.calories** and **b.calories** without ambiguity.

Ex: Member accessing using dot operator

- Example:

```
StudentRecord Student1; //Student1 is a variable of type
//StudentRecord
strcpy(Student1.Name, "Chan Tai Man");
Student1.Id = 12345;
strcpy(Student1.Dept, "COMP");
Student1.gender = 'M';
printf("The student is ");
switch (Student1.gender){
    case 'F': cout << "Ms. "; break;
    case 'M': cout << "Mr. "; break;
}
printf("%s \n", Student1.Name);
```



Assigning values to members

Different ways to assign values to the members of a structure:

Assigning string:

```
strcpy(s1.name, "Rama");
```

Assignment statement:

```
s1.rollno = 1335;
```

```
s1.age = 18;
```

```
s1.height = 5.8;
```

```
struct student
{
    int rollno;
    int age;
    char name[20];
    float height;
}s1;
```

Reading the values into members:

```
scanf("%s %d %f %f", s1.name, &s1.age, &s1.rollno,
&s1.height);
```

Omission of the Tag Name

```
struct { /* Since no tag name is */  
    char *last_name; /* used, no variables can */  
    int student_id; /* be declared later in */  
    char grade; /* the program. */  
} s1, s2, s3;
```

```
struct student { /* Variables can now be */  
    char *last_name; /* declared later in */  
    int student_id; /* the program as shown */  
    char grade; /* below. */  
};
```

```
struct student temp, class[100];
```



Summary

- Structure Basics
- Member accessing using dot operator
- Simple problems using structures

Poll Question

Go to chat box/posts for the link to the Poll question

Submit your solution in next 2 minutes

Click the result button to view your score



S 2 1 _ 2 S t r u c t u r e s



Array of Structures

Objectives:

- To learn and appreciate the following concept
 - Array of structures
 - Pointers and Structures



Session outcome

- At the end of session one will be able to
 - Understand Array of Structures
 - Understand the overall ideology of array of structures
 - Write programs using array of structures

Structure Initialization Methods

```
int main ( )
{
    struct Student
    {
        int rollno;
        int age;
    };
    struct Student
    s1={20, 21};
    struct Student
    s2={21, 21};
}
```

```
struct Student
{
    int rollno;
    int age;
} s1={20, 21};

main ( )
{
    struct Student
    s2={21, 21};
    ...
    ...
}
```



Structure: Example

```
struct Book {           // definition
    char title[20];
    char author[15];
    int pages;
    float price;
};

int main( ){
    struct Book b1;
    printf("Input values");
    scanf("%s %s %d %f", b1.title, b1.author, &b1.pages,
          &b1.price);
    //output
    printf("%s %s %d %f", b1.title, b1.author, b1.pages,
          b1.price);
    return 0;
}
```

Structure: Example

```
struct Book {          // definition
    char title[20];
    char author[15];
    int pages;
    float price;
};

int main( ){
    struct Book b1;
    printf("Input values");
    gets(b1.title); gets(b1.author);
    scanf("%d %f", &b1.pages, &b1.price);
    //output
    printf("%s %s %d %f", b1.title, b1.author, b1.pages,
           b1.price);
    return 0;
}
```



Structures: Overview

- Definition & structure variable declaration

```
struct student
{ int rollno;
  int age;
  char name[20];
}s1, s2, s3;
```

- Initialization

```
int main( )
{
struct
{ int rollno;
  int age;
}stud={20, 21};
...
}
return 0;
```

- Giving values to members

Using **dot operator** ‘.’

```
s1. rollno = 25;
scanf("%s",s1.name);
```

‘.’ operator acts as Link between member and a Structure variable.

- Assign & compare members

s1 = s2 ; assignment (allowed)

s1 == s2 comparison (not allowed)

s1!=s2 comparison (not allowed)

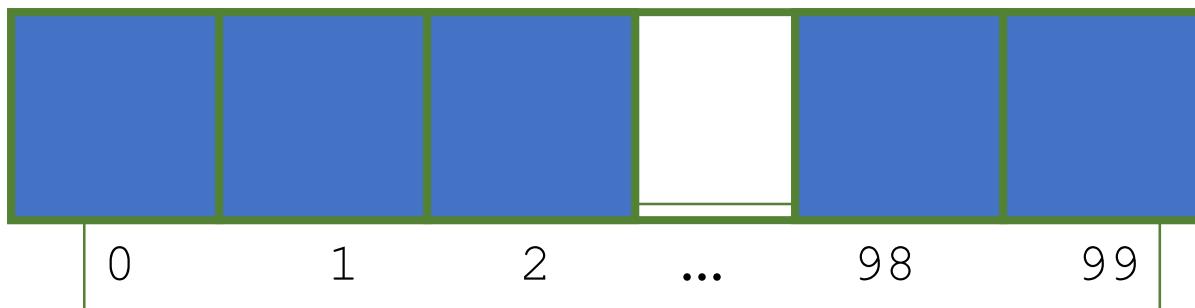
s1.rollno == s2.rollno; (allowed)

s1.rollno!=s2.rollno; (allowed)

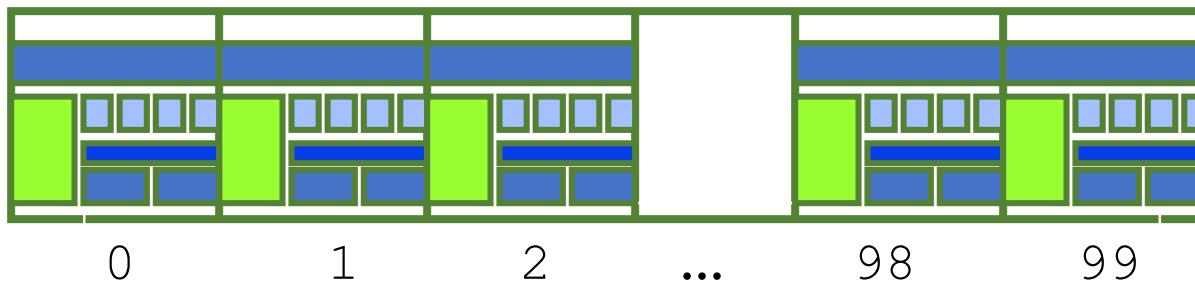


Arrays of structures

- An ordinary array: One type of data



- An array of structs: Multiple types of data in each array element.





Array of structures

We can define single or multidimensional arrays as structure variables.

```
struct marks
{
    int subject1;
    int subject2;
    int subject3;
} ;
struct marks student[80];
```

- Defines an array called student, that consists of 80 elements.
- Each element is defined to be the type marks.

Array of structures – Initialization

```

struct marks {
    int subject1;
    int subject2;
    int subject3;
} ;

main(){
    struct marks student[] = {
        {45,47,49},
                    {43,44,45},
                    {46,42,43}
    };
}

```

Memory
student[0].subject1
student[0].subject2
student[0].subject3
student[1].subject1
student[1].subject2
student[1].subject3
student[2].subject1
student[2].subject2
student[2].subject3

Array of Structure: Example

```
struct Book {           //Structure Definition
    char title[20];
    char author[15];
    int pages;
    float price;
};

int main( ){
    struct Book b[10];
    printf("Input values");
    for (int i=0;i<3;i++)
        scanf("%s %s %d %f", b[i].title, b[i].author, &b[i].pages,
&b[i].price);
    for (int j=0;j<3;j++)
        printf("%s\t %s\t %d\t %f\n", b[j].title, b[j].author, b[j].pages,
b[j].price);
    return 0;
}
```

Arrays within Structures

We can define single or multidimensional arrays inside a structure.

struct marks

```
{   int rollno;  
    float subject[3];  
} student[2] ;
```

The member **subject** contains 3 elements; **subject[0]**, **subject[1]** & **subject[2]**.

student[1].subject[2];

- Refers to the marks obtained in the third subject by the second student.

Arrays within structures : example

```
#include<stdio.h>
```

```
int main(){
    struct marks student[3] ={{0,45,47,49},
                               {0,43,44,45},
                               {0,46,42,43}};
```

```
int i, j ;
```

```
//students total
```

```
for(i=0;i<=2;i++) {
    for(j=0;j<=2;j++)
        student[i].total+=student[i].sub[j]; }
```

```
printf("Grand Total of each student:");
```

```
for(i=0;i<=2;i++)
```

```
    printf("\nTotal of student[%d]= %d", i, student[i].total);
```

```
return 0;
```

```
}
```

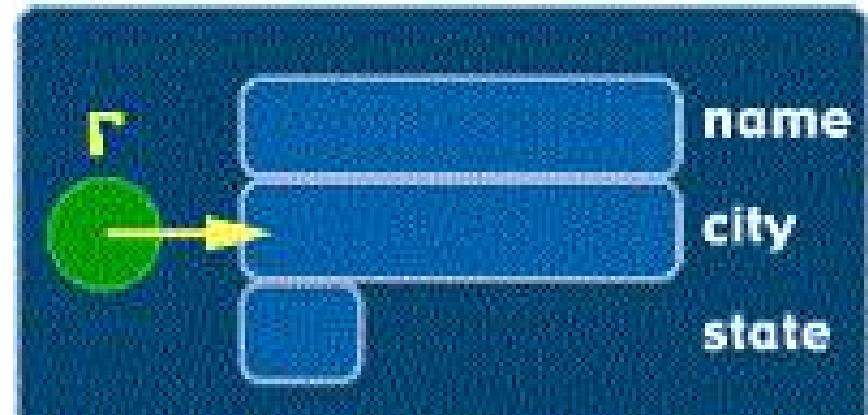
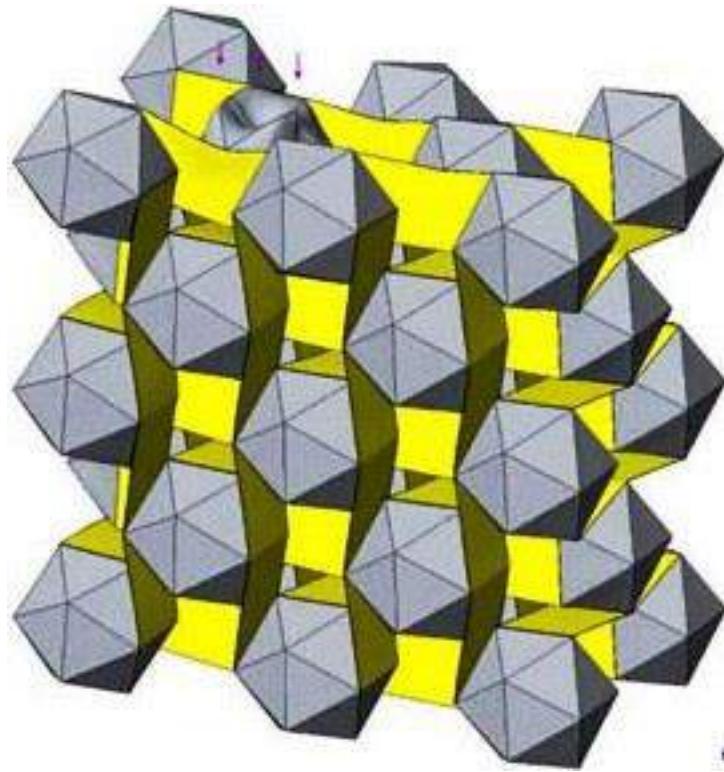
//Structure Definition

```
struct marks{
    int total;
    int sub[3];
};
```



Summary

- Simple problems using structures
- Array of Structures
- Arrays within Structures
- Structures within Structures



Array of Structures & Pointers to Structures

Objectives

- To learn and appreciate the following concept
 - Array of structures

Session outcome

- At the end of session one will be able to
 - Understand the overall ideology of array of structures
 - Write programs using array of structures

Structures within Structures

Structure within structure means nesting of structures.

for instance see the following structure defined to store information about students

```
struct student{  
    int rollno;  
    char name[15];  
    struct { // marks for 3 subjects under structure marks  
        int sub1;  
        int sub2;  
        int sub3;  
    }marks;  
}fs[3]; //3 students
```



Structures within Structures

```
//Structure Definition
struct student{
    int rollno;
    char name[15];
    struct m marks;
}fs[3];
```

```
//Structure Definition
struct m{
    int sub1;
    int sub2;
    int sub3;
}
```

Tag name is used to define inner structure **marks**

The members contained in the inner structure namely **sub1**, **sub2** and **sub3** can be referred to as:

fs[i].marks.sub1;
fs[i].marks.sub2;
fs[i].marks.sub3;

Structures and functions

```

void read(struct book x[]); // prototype
int main() {
int i;
struct book b1[2];
printf("\n Enter IBN, Author name & Price \n");
read(b1); // function call

printf("\nThe book details entered are:\n");
for(i=0;i<2;i++){
    printf("\n Book %d", i+1);
    printf("\nIBN: \t\t%d", b1[i].ibn);
    printf("\nAuthor: \t%s", b1[i].author);
    printf("\nPrice: \t\t%f", b1[i].price);
}
return 0;
}

```

```

//Structure
Definition
struct book
{
    int ibn;
    char author[15];
    float price;
};

```

```

//function definition
void read(struct book a[])
{
int i;
for(i=0;i<2;i++){
    printf("\nBook %d\n", i+1);
    scanf("%d", &a[i].ibn);
    scanf("%s", a[i].author);
    scanf("%f", &a[i].price);
}
}

```

Structures -Problems

Write programs to

1. Create a student record with name, rollno, marks of 3 subjects (m1, m2, m3). Compute the average of marks for 3 students and display the names of the students in ascending order of their average marks.
2. Create an employee record with emp-no, name, age, date-of-joining (year), and salary. If there is 20% hike on salary per annum, compute the retirement year of each employee and the salary at that time. [standard age of retirement is 55]



Structures – Solution for Q1

```
int main()
{
    struct student temp, fs[3] =
        {{1,"manish",45,47,49},
         {2,"ankur",43,44,45},
         {3,"swati",46,42,43}};
    int i, n=3, total[3]={0}, avg[3]={0},tot=0;

    for(i=0; i< n; i++)  {
        total[i]=fs[i].marks.sub1+fs[i].marks.sub2+
        fs[i].marks.sub3; //students total

        avg[i] = total[i]/3;
    }
    //display
    printf("Total & Average of each student.\n");
    for(i=0;i<n;i++){
        printf("\nTotal of %s = %d & avg = %d", fs[i].name, total[i], avg[i]);
    }
}
```

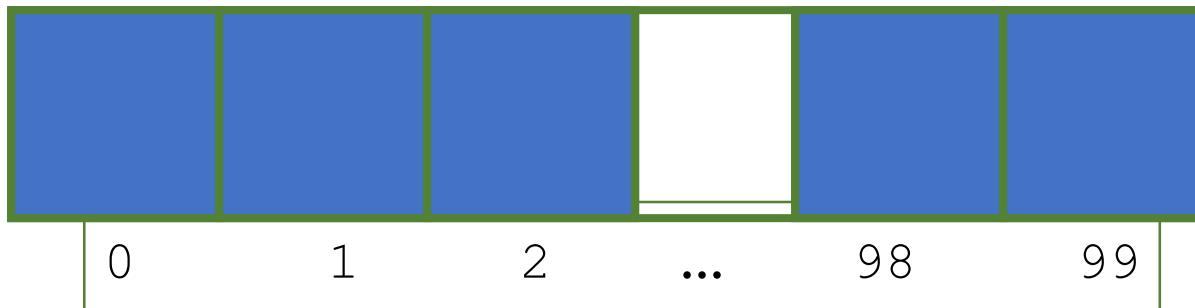
```
struct student{
    int rollno;
    char name[15];
    struct {
        int sub1;
        int sub2;
        int sub3;
    }marks;
};
```

Structures – Solution for Q1

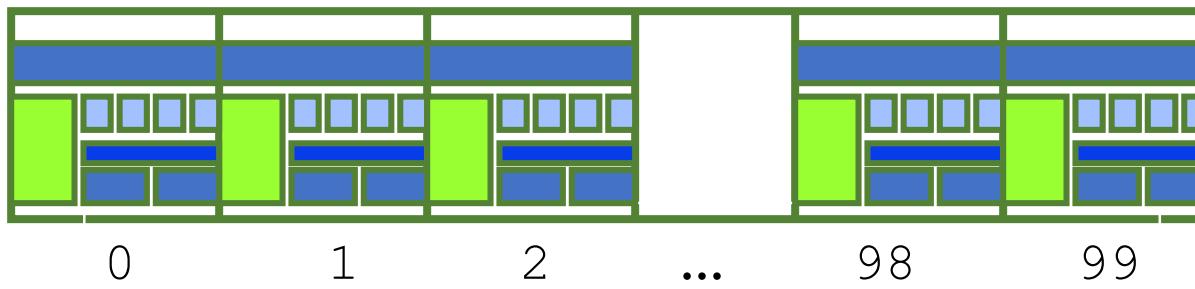
```
// sorting
for(i=0;i<n;i++)
    for(int j=i+1;j<n;j++)
        if(avg[i] > avg[j])
        {
            temp=fs[i]; //Swapping
            fs[i]=fs[j];
            fs[j]=temp;
        }
for(i=0;i<n;i++) //Sorted list w.r.to average marks
    printf("\n%s\n",fs[i].name);
return 0;
} //end of main
```

Arrays of structures

- An ordinary array: One type of data



- An array of structs: Multiple types of data in each array element.



Array of structures

We can define single or multidimensional arrays as structure variables.

```
struct marks
{
    int subject1;
    int subject2;
    int subject3;
} ;
marks student[80];
```

- Defines an array called student, that consists of 80 elements.
- Each element is defined to be the type marks.

Structures and functions

```

void read(struct book x[]); // prototype
int main() {
int i;
struct book b1[2];
printf("\n Enter IBN, Author name & Price \n");
read(b1); // function call

printf("\nThe book details entered are:\n");
for(i=0;i<2;i++){
    printf("\n Book %d", i+1);
    printf("\nIBN: \t\t%d", b1[i].ibn);
    printf("\nAuthor: \t%s", b1[i].author);
    printf("\nPrice: \t\t%f", b1[i].price);
}
return 0;
}

```

```

//Structure
Definition
struct book
{
    int ibn;
    char author[15];
    float price;
};

```

```

//function definition
void read(struct book a[])
{
int i;
for(i=0;i<2;i++){
    printf("\nBook %d\n", i+1);
    scanf("%d", &a[i].ibn);
    scanf("%s", a[i].author);
    scanf("%f", &a[i].price);
}
}

```

Structures as Function Arguments

You can pass a structure as a function argument in very similar way as you pass any other variable or pointer.

```
#include <stdio.h>#include<string.h>
struct Books { char title[50];
    char author[50];
    char subject[100];
    int book_id; }; /* function declaration */
void printBook( struct Books book );
int main( )
{ struct Books Book1; /* Declare Book1 of type Book */
struct Books Book2;
/* Declare Book2 of type Book */
/* book 1 specification */
strcpy( Book1.title, "C Programming");
strcpy( Book1.author, "Nuha Ali");
strcpy( Book1.subject, "C Programming Tutorial");
Book1.book_id = 6495407;
```

Structures as Function Arguments

```
/* book 2 specification */

strcpy( Book2.title, "Telecom Billing");
strcpy( Book2.author, "Zara Ali");
strcpy( Book2.subject, "Telecom Billing Tutorial");
Book2.book_id = 6495700; /*
print Book1 info */ printBook( Book1 );
/* Print Book2 info */
printBook( Book2 ); return 0; }
void printBook( struct Books book )
{
printf( "Book title : %s\n", book.title);
printf( "Book author : %s\n", book.author);
printf( "Book subject : %s\n", book.subject);
printf( "Book book_id : %d\n", book.book_id);
}
```



Structures -Problem

Write a menu driven program for a “***BOOK MART***” with the following menu options

BOOKMART MENU

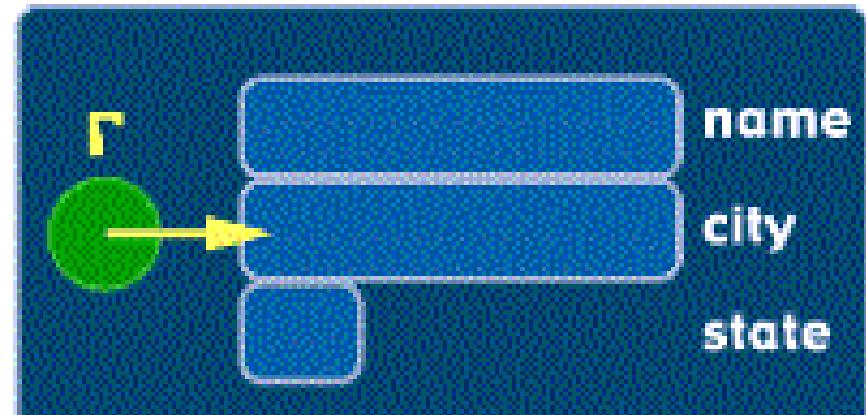
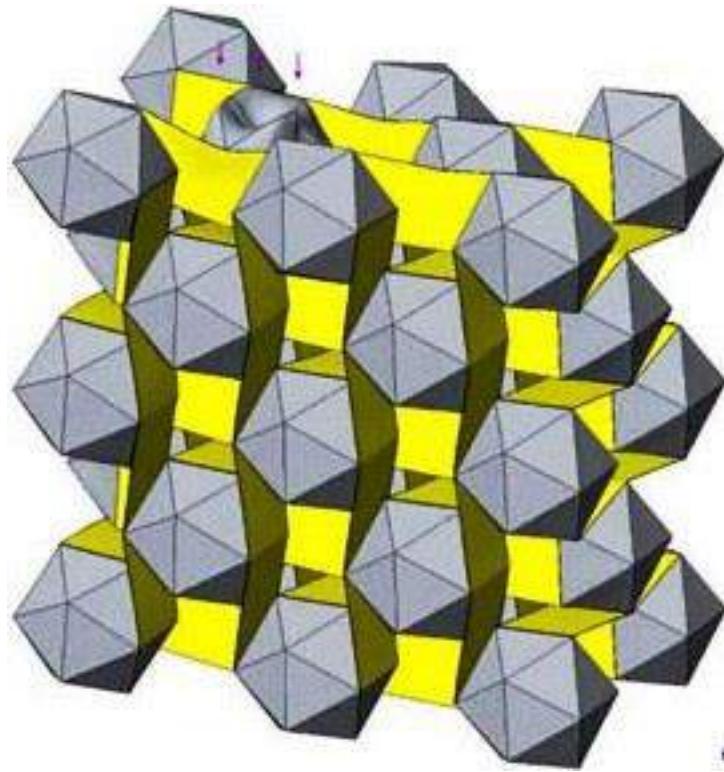
1. *Availability*
2. *Purchase*
3. *Exit*

The details of the books are stored in a structure “***books***” with the member variables ***book_number***, ***book_name***, ***book_price***, ***book_author***, ***number_of_copies***.

Declare the different member variables (use meaningful abbreviations for the variables; e.g. **bno** for *book number*, **noc** for *number of copies* etc.) with appropriate data types. Use an array of structure ***book[]*** to insert details for at least 5 books. Your program shall run continuously for all the operations until you press ***Exit*** option in the menu. Purchase menu should be used to purchase a particular book using the book number as user input. [Hint: usage of SWITCH within WHILE statement (repeating loop)]

Summary

- Array of Structures
- Arrays within Structures
- Structures within Structures
- Structures and Functions



Array of Structures & Pointers to Structures

Objectives

- To learn and appreciate the following concept
 - Structures and Functions
 - Pointers and Structures

Session outcome

- At the end of session one will be able to
 - Understand the concept of pointers to structures
 - Write programs on pointers to structures.

EXAMPLE PROGRAM – PASSING STRUCTURE TO FUNCTION BY VALUE:

- In this program, the whole structure is passed to another function by value.
- It means the whole structure is passed to another function with all members and their values.
- So, this structure can be accessed from called function.
- This concept is very useful while writing very big programs in C.

Structures and Functions

```
#include <stdio.h>
#include <string.h>

struct student
{
    int id;
    char name[20];
    float percentage;
};

void func(struct student record);
```

Structures and Functions

```
int main()
{
    struct student record;

    record.id=1;
    strcpy(record.name, "Raju");
    record.percentage = 86.5;
    func(record);
    return 0;
}
```

```
void func(struct student record)
{
    printf(" Id is: %d \n", record.id);
    printf(" Name is: %s \n", record.name);
    printf(" Percentage is: %f \n", record.percentage);
}
```

Output:

Id is: 1

Name is: Raju

Percentage is: 86.500000

Pointers and structures

Consider the following structure

```
struct inventory {  
    char name[30];  
    int number;  
    float price;  
} product[2],*ptr;
```

This statement declares **product** as an array of 2 elements, each of the type **struct inventory**.

ptr=product; assigns the address of the **zeroth** element of **product** to **ptr**

or **ptr** points to **product[0]**;

Pointers and Structures

Its members are accessed using the following notation

`ptr → name`

`ptr → number`

`ptr → price`

The symbol `→` is called **arrow operator** (also known as **member selection operator**)

When **ptr is incremented by one**, it points to the next record.
i.e. **product[1]**

The member price can also be accessed using

`(*ptr).price`

Parentheses is required because `:` has higher precedence than the operator `*`



Pointers and Structures- example

```
struct invent
```

```
{
```

```
    char name[30];
```

```
    int number;
```

```
    float price;
```

```
};
```

```
Enter 3 (0, 1 and 2 )sets of Name , Number and Price
c_Book
100
250
C++Book
200
350
Java
150
400
c_Book 100 250
C++Book 200 350
Java 150 400
```

```
Process returned 0 (0x0) execution time : 33.424 s
Press any key to continue.
```

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
    struct invent prod[3], *ptr;
```

```
    printf("Enter 3 (0, 1 and 2 )sets of Name,
           Number and Price");
```

```
    for(ptr = prod; ptr < prod+3; ptr++)
```

```
        scanf("%s %d %f",ptr ->name, &ptr ->number,
              &ptr ->price);
```

```
    ptr=prod;
```

```
    while(ptr < prod+3)
```

```
{
```

```
        printf("%s %d %f\n", ptr ->name,
               ptr ->number, ptr ->price);    ptr++;
```

```
}
```

```
    return 0;
```

```
}
```



Pointers and Structures- example

```
main( )
{
    struct s1
    {
        char *z ;
        int i ;
        struct s1 *p ;
    };
    static struct s1 a[ ] = {
        { "Nagpur", 1, a + 1 },
        { "Raipur", 2, a + 2 },
        { "Kanpur", 3, a }
    };
    struct s1 *ptr = a ;
    printf ( "\n%s %s %s", a[0].z, ptr->z, a[2].p->z ) ;
}
```

Output

Nagpur Nagpur Nagpur



Pointers and Structures- example

```
main( )
{
    struct a
    {
        char ch[7];
        char *str;
    };
}
```

```
struct b
{
    char *c;
    struct a ss1;
};

struct b s2 = { "Raipur", "Kanpur", "Jaipur" };

printf( "\n%s %s", s2.c, s2.ss1.str );
printf( "\n%s %s", ++s2.c, ++s2.ss1.str );
}
```

Output

Raipur Jaipur
aipur aipur

Benefits(use) of pointers

- Pointers provide direct access to memory.
- Pointers provide a way to return more than one value to the functions.
- Reduces the storage space and complexity of the program.
- Reduces the execution time of the program.
- Provides an alternate way to access array elements
- Pointers can be used to pass information back and forth between the calling function and called function.

Drawbacks of pointers

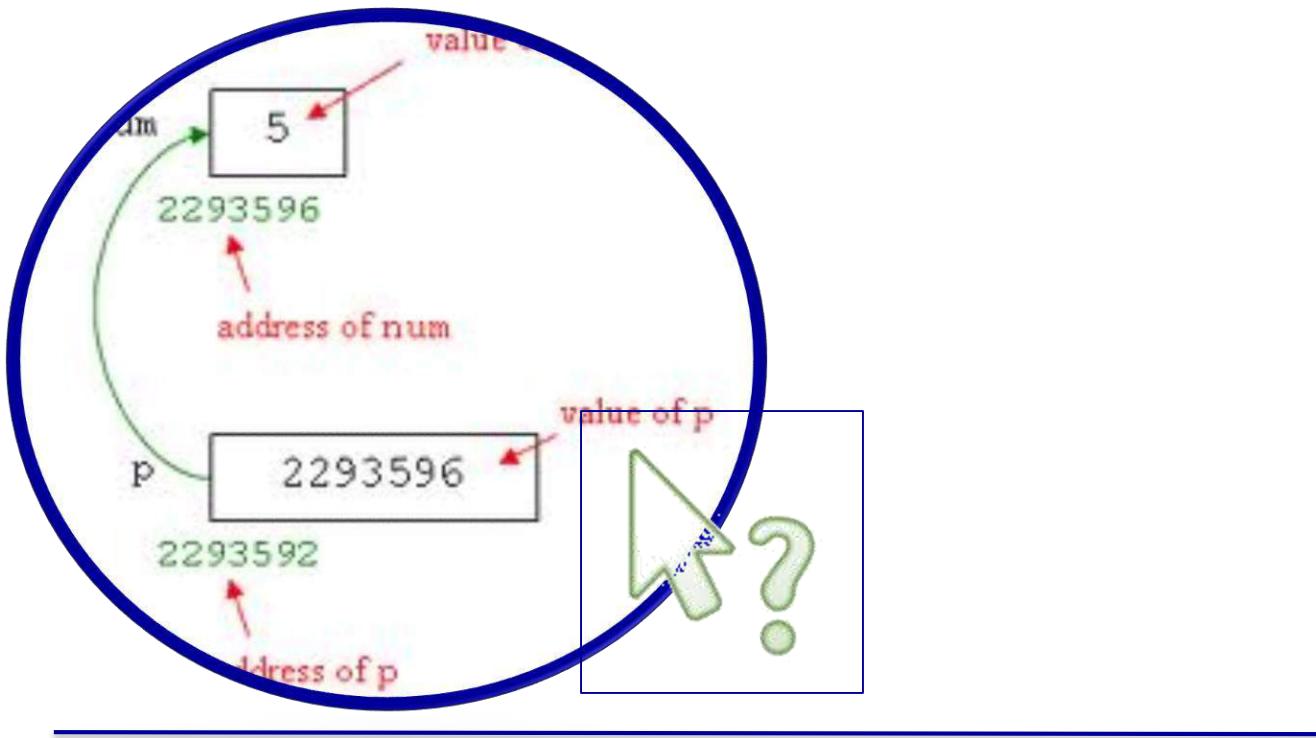
- Uninitialized pointers might cause segmentation fault.
- Dynamically allocated block needs to be freed explicitly. Otherwise, it would lead to memory leak.
- Pointers are slower than normal variables.
- If pointers are updated with incorrect values, it might lead to memory corruption.

Summary

- Structures and Functions
- Pointers and Structures

S23_1 Pointers





P o i n t e r s

Objectives

To learn and appreciate the following concepts

- Basic operations on pointers
- Pointers and Arrays
- Pointers and Character Strings
- Pointers and 2D
- Array of Pointers

Session outcome

At the end of session one will be able to understand

- Basic operations on pointers
- Pointers and Arrays

Pointers- recap

`int Quantity; //defines variable Quantity of type int`

`int* p; //defines p as a pointer to int`

`p = &Quantity; //assigns address of variable Quantity to pointer p`

Variable	Value	Address
<code>Quantity</code>	<code>50</code>	<code>5000</code>
<code>p</code>	<code>5000</code>	<code>5048</code>

Now...

`Quantity = 50; //assigns 50 to Quantity`

`*p = 50; //assigns 50 to Quantity`

Pointer expressions

- Pointers can be used in most valid C expressions. However, some special rules apply.
- You may need to surround some parts of a pointer expression with parentheses in order to ensure that the outcome is what you desire.
- As with any variable, a pointer may be used on the right side of an assignment operator to assign its value to another pointer.

Pointer Expressions - Example

- Eg: int a=10, b=20,c,d=10;
int *p1 = &a, *p2 = &b;

Expression	a	b	c
c= *p1**p2; OR *p1 * *p2 OR (*p1) * (*p2)	10	20	200
c= c + *p1;	10	20	210
c=5 * - *p2 / *p1; OR (5 * (- (*p2)))/(*p1) //space between / and * is required	10	20	-10
*p2 = *p2 +10;	10	30	

Operations on Pointer Variables

- **Assignment** – the value of one pointer variable can be assigned to another pointer variable of the same type
- **Relational operations** - two pointer variables of the same type can be compared for equality, and so on
- **Some limited arithmetic operations**
 - integer values can be added to and subtracted from a pointer variable
 - value of one pointer variable can be subtracted from another pointer variable
 - Shorthand Increment and Decrement Operators

Allowed Pointer Operations - Example

- `int a = 10, b = 20, *p1, *p2, *p3, *p4;`
- `p1 = &a; //assume address of a = 2004`
- `p2 = &b; //assume address of b = 1008`

Assume an integer occupies 4 bytes

Pointer Operations	Example expression	Result
Addition of integers from pointers	<code>p3 = p1 + 2</code>	value of p3 = $2004 + 4*2 = 2012$
Subtraction of integers from pointers	<code>p4 = p2 - 2</code>	value of p4 = $1008 - 4*2 = 1000$
Subtraction of one pointer from another	<code>c = p3 - p1</code>	Value of c = $2012 - 2004 = 2$
Pointer Increment	<code>p1++</code>	Value of p1 = 2008
Pointer Decrement	<code>--p1</code>	Value of p1 = 2004

Allowed Pointer Operations - Example

if (p1<p2)

printf("p1 points to lower memory than p2");

if (p1==p2)

printf(" p1 and p2 points to same location");

if (p1!=p2)

printf(" p1 and p2 NOT pointing to same location");

Invalid Operations:

- Pointers are not used in division and multiplication.

p1/*p2;

p1*p2;

p1/3; are not allowed.

- Two pointers can not be added.

p1 + p2 is illegal.

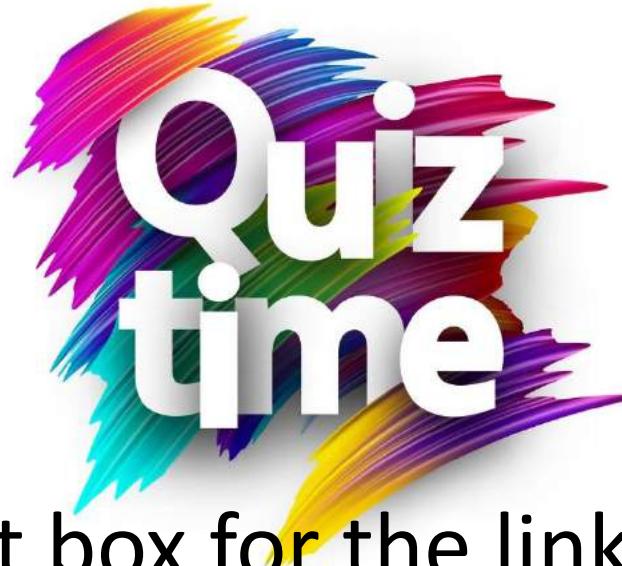
Program to exchange two values

```
#include<stdio.h>
int main()
{
    int x, y, t, *a, *b;
    a=&x; b=&y;
    printf("Enter the values of a and b: \n");
    scanf("%d %d", a, b); // equivalent to scanf("%d %d", &x, &y);
    t=*a;
    *a=*b;
    *b=t;

    printf("x = %d \n", x);
    printf("y = %d", y);

    return 0;
}
```

Enter the values of a and b:
10 5
x= 5
y = 10



Go to posts/chat box for the link to the question

submit your solution in next 2 minutes

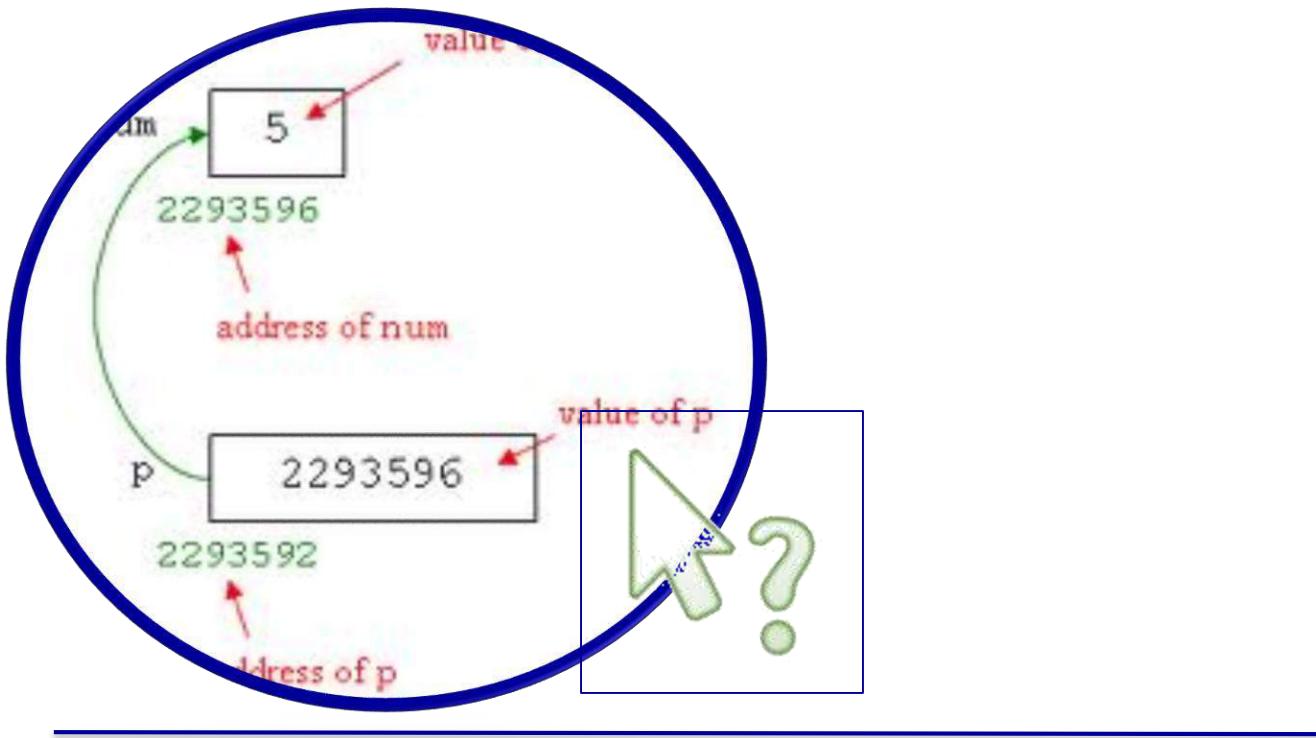
The session will resume in 3 minutes

Summary of pointers

- Basic operations on pointers

S23_1 Pointers





P o i n t e r s

Objectives

To learn and appreciate the following concepts

- Pointers and Arrays

Session outcome

At the end of session one will be able to understand

- Basic operations on pointers
- Pointers and Arrays

Pointers and arrays

- When an array is declared, the compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory locations.
- The base address is the location of the first element (index 0) of the array.
- The compiler also defines the array name as a **constant pointer** to the first element.

Pointers and arrays

- An array **x** is declared as follows and assume the base address of **x** is **1000**.

int x[5] ={ 1,2,3,4,5};

- Array name **x**, is a constant pointer, pointing to the first element **x[0]** .
- Value of **x** is **1000 (Base Address)**, the location of **x[0]**. i.e. **x = &x[0] = 1000 (in the example below)**

Elements	x[0]	x[1]	x[2]	x[3]	x[4]
Value	1	2	3	4	5
Address	1000  Base Address	1004	1008	1012	1016

Array accessing using Pointers

- An **integer pointer variable** p, can be made to point to an array as follows:

```
int x[5] ={ 1,2,3,4,5};  
int *p;  
p = x;    OR    p = &x[0];
```

- **Following statement is Invalid:**

```
p = &x ; //Invalid
```

- Successive array elements can be accessed by writing:

```
printf("%d", *p); p++;  
or  
printf("%d", *(p+i)); i++;
```

Pointers and arrays

- The relationship between **p** and **x** is shown below:

`p= &x[0];` (=1000) BASE ADDRESS

`p+1=>&x[1]` (=1004)

`p+2=>&x[2]` (=1008)

`p+3=>&x[3]` (=1012)

`p+4=>&x[4]` (=1016)

- Address of an element of **x** is given by:**

Address of **x[i]** = **base address + i * scale factor of (int)**

Address of **x[3]**= **1000 +(3*4) = 1012**

Array accessing using array name as pointer - Example

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
    int arr[5] = { 31, 54, 77, 52, 93 };
```

```
    for(int j=0; j<5; j++) //for each element,
```

```
        printf("%d ", *(arr+j)); //print value
```

```
    return 0;
```

```
}
```

```
31 54 77 52 93
Process returned 0 (0x0) execution time : 0.047 s
Press any key to continue.
```

Array accessing using Pointers - Example

```
// array accessed with pointer
```

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
    int arr[5] = { 31, 54, 77, 52, 93 };
```

```
    int* ptr; //pointer to arr
```

```
    ptr = arr; //points to arr
```

```
    for(int j=0; j<5; j++) //for each element,
```

```
        printf("%d ", *ptr++);
```

```
    return 0;
```

```
}
```

```
31 54 77 52 93
```

```
Process returned 0 (0x0) execution time : 0.047 s
```

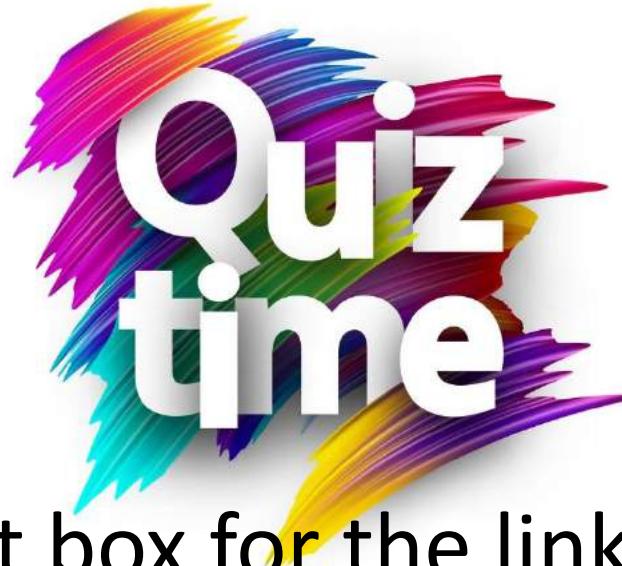
```
Press any key to continue.
```

“ptr” is a pointer which can be used to access the elements.

Sum of all elements stored in an array

```
#include <stdio.h>
int main()
{
    int *p, sum=0, i=0;
    int x[5] ={5, 9, 6, 3, 7};
    p=x;
    while(i<5)
    {
        sum+=*p;
        i++;
        p++;
    }
    printf("sum of elements = %d", sum);
    return 0;
```

```
sum of elements =30
Process returned 0 (0x0) execution time : 0.016 s
Press any key to continue.
```



Go to posts/chat box for the link to the question

submit your solution in next 2 minutes

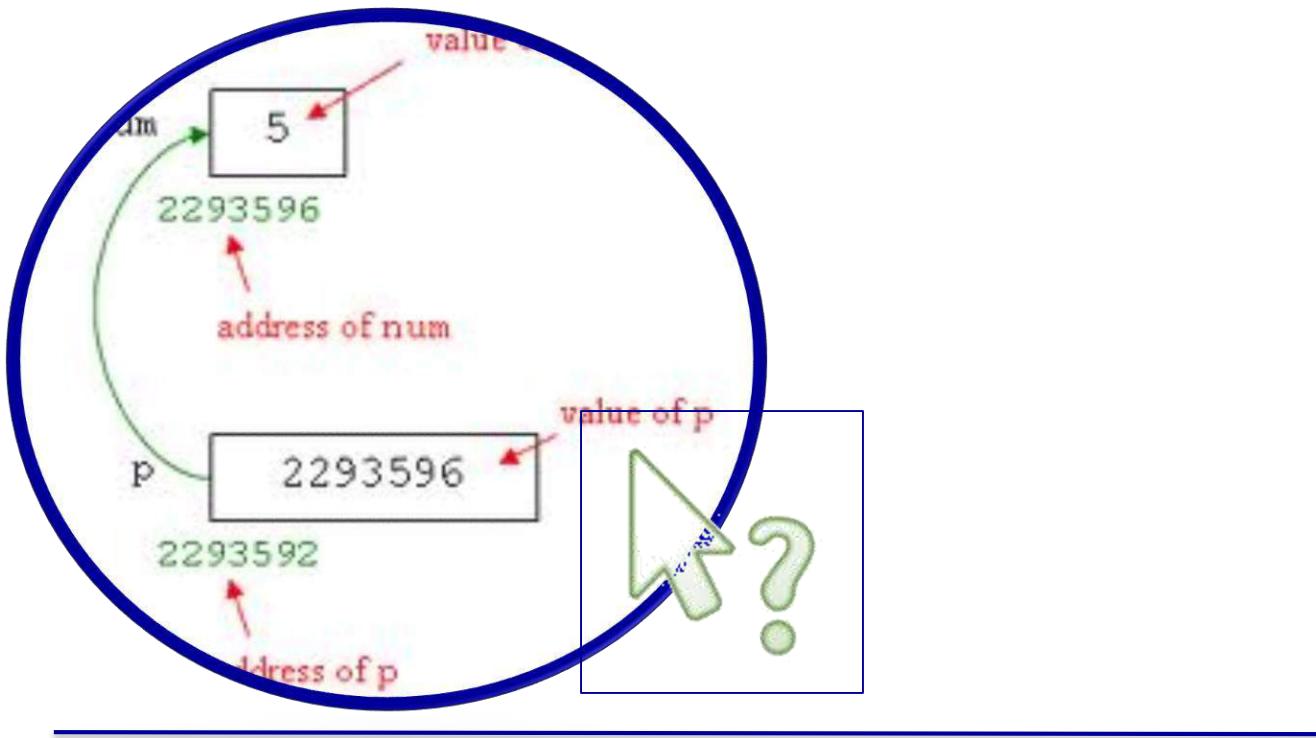
The session will resume in 3 minutes

Summary of pointers

- Pointers and Arrays

S24_1 Pointers





P o i n t e r s

Objectives

To learn and appreciate the following concepts

- Pointers and Character Strings
- Pointers and 2D
- Array of Pointers

Session outcome

At the end of session one will be able to understand

- Pointers and Character Strings
- Pointers and 2D
- Array of Pointers

Pointers & Character strings

//length of the string

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
    char name[15];
```

```
    char *cptr=name;
```

```
    printf("Enter some word to find its length: \n");
```

```
    scanf("%s", name);
```

```
    while(*cptr!= '\0')
```

```
        cptr++;
```

```
    printf("length= %d",cptr-name);
```

```
    return 0;
```

```
}
```

```
Enter some word to find its length
Computer
length=8
Process returned 0 (0x0) execution time : 16.345 s
Press any key to continue.
```

Pointers & Character strings

- The statements

`char name[10];`

`char *cptr = name;`

declares `cptr` as a pointer to a character array and assigns address of the first character of `name` as the initial value.

- The statement `while(*cptr != '\0')`

is true until the end of the string is reached.

- When the while loop is terminated, the pointer `cptr` holds the address of the null character `'\0'`.
- The statement `length = cptr - name;` gives the length of the string `name`.

Pointers & Character strings

- A constant character string always represents a pointer to that string.
- The following statements are valid.

```
char *name;
```

```
name =“Delhi”;
```

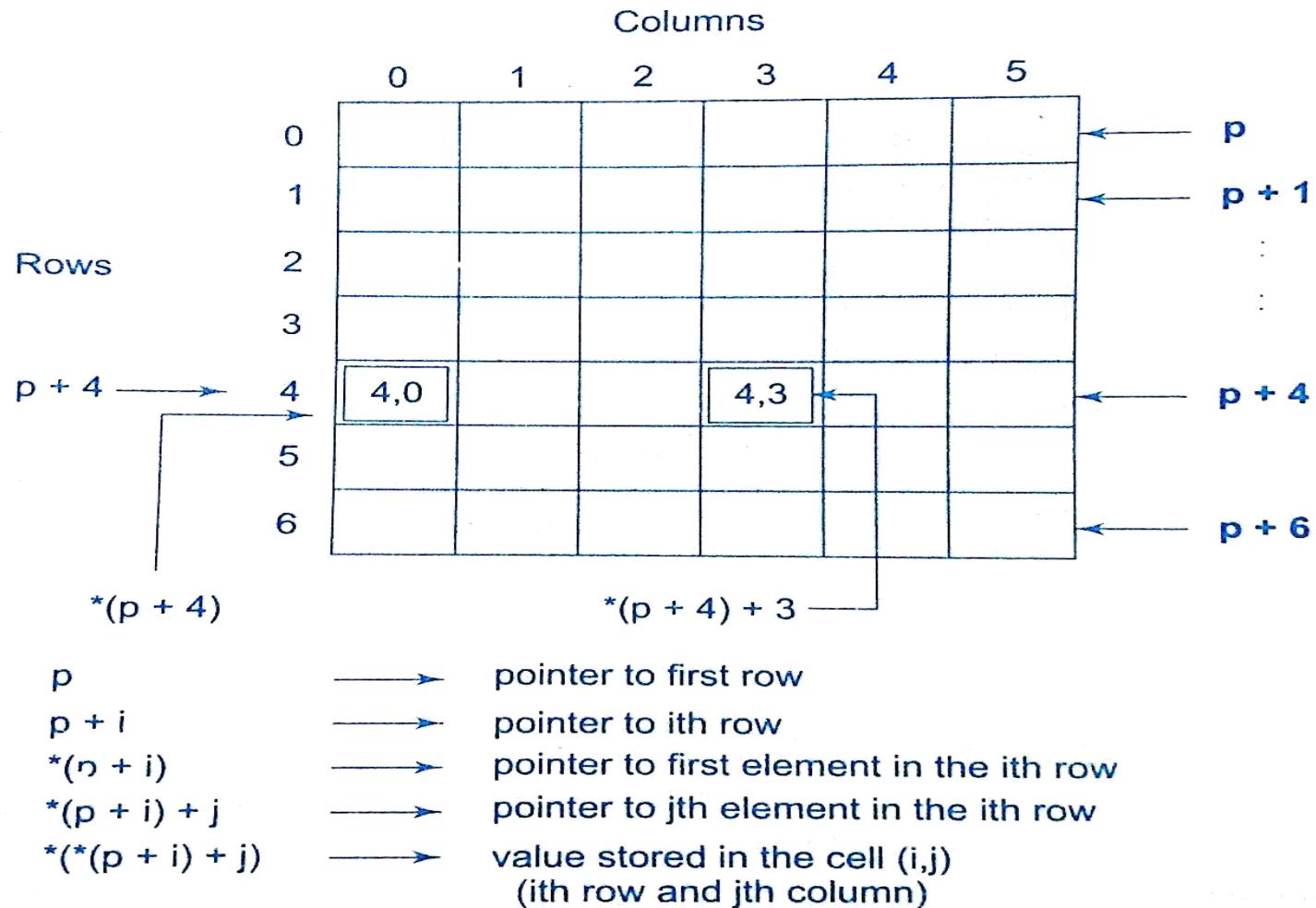
These statements will declare name as a pointer to character array and assign to name the constant character string “Delhi”.

Pointers and 2D arrays

```
int a[][2]={ {12, 22},  
            {33, 44} };  
  
int (*p)[2];  
p=a; // initialization
```

Element in 2d represented as

$*(*(\text{a}+\text{i})+\text{j})$
or
 $*(*(\text{p}+\text{i})+\text{j})$



Pointers and 2D arrays

// 2D array accessed with pointer

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
    int i, j, (*p)[2], a[][2] = {{12, 22}, {33, 44}};
```

```
    p=a;
```

```
    for(i=0;i<2;i++)
```

```
{
```

```
        for(j=0;j<2;j++)
```

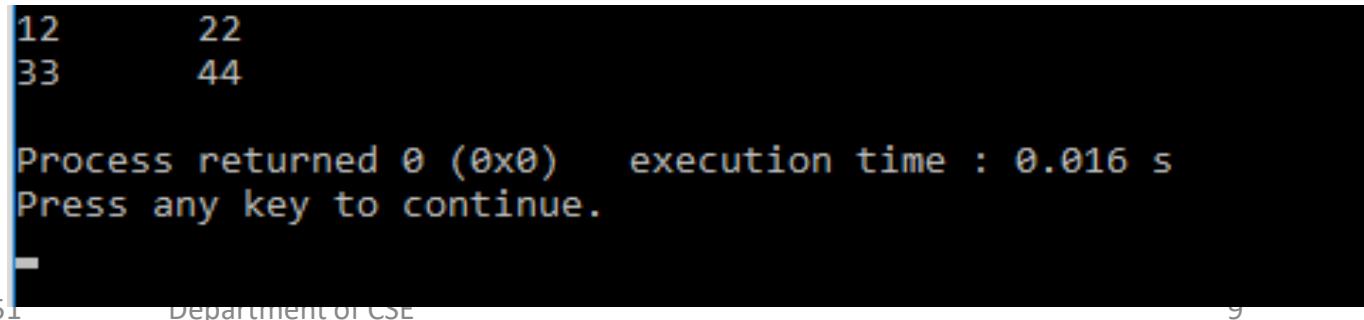
```
            printf("%d \t", *(*(p+i)+j));
```

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

```
}
```

```
    return 0;
```

```
}
```



```
12      22
33      44
Process returned 0 (0x0)  execution time : 0.016 s
Press any key to continue.
```

Array of pointers

- We can use pointers to handle a table of strings.

char name[3][25];

name is a table containing 3 names, each with a maximum length of 25 characters (including '\0')

- Total **storage** requirement for **name** is **75 bytes**.

But rarely all the individual strings will be equal in lengths.

- We can use a pointer to a string of varying length as

char *name[3] = { “New Zealand”, “Australia”, “India” };

Array of pointers

```
So, char *name[3] = {    “New Zealand ”,  
                        “Australia”,  
                        “India”};
```

Declares **name** to be an **array of 3 pointers** to characters, each pointer pointing to a particular name.

name[0] → New Zealand

name[1] → Australia

name[2] → India

This declaration allocates **28 bytes**.

Array of pointers

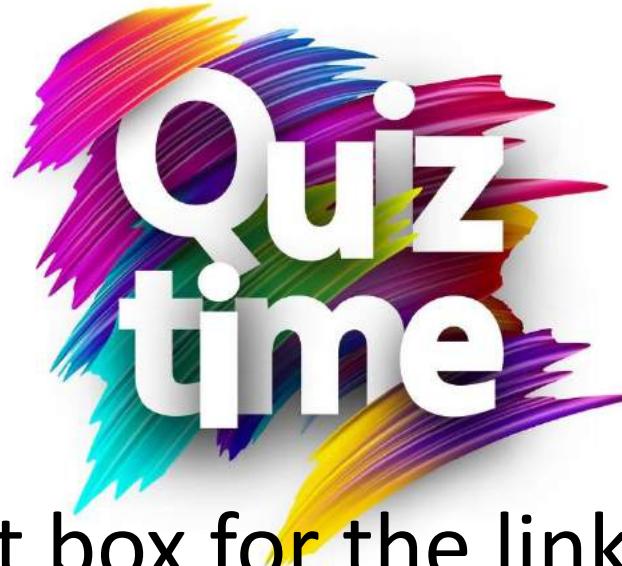
The following statement would print out all the 3 names.

```
for(i=0; i<=2;i++)  
    printf("%s",name[i]);  
or printf("%s", *(name + i));
```

To access the j^{th} character in the i^{th} name, we may write as

$*(name[i] + j)$

The character array with rows of varying lengths are called **ragged arrays** and are better handled by pointers.



Go to posts/chat box for the link to the question

submit your solution in next 2 minutes

The session will resume in 3 minutes

Summary of pointers

- Pointers and Strings
- Array of Pointers



MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
(A constituent unit of MAHE, Manipal)



S24_2 CYBER SECURITY



Objectives

To understand

- Definition of Cyber Crime
- Classification of Cyber crimes
- Computer Intrusions and Hacking
- Computer Security



Cyber Crime

- **Cybercrime** also known as **Computer crime**, refers to any crime that involves a computer/mobile and a network.
- The computer may have been used in the commission of a crime, or it may be the target.
- **Netcrime** is criminal exploitation of the internet.
- Experts defined Cybercrime as "Offences that are committed against individuals or groups of individuals with a criminal motive to intentionally harm the reputation of the victim or cause physical or mental harm to the victim directly or indirectly, using modern telecommunication networks such as Internet (Chat rooms, emails, notice boards and groups) and mobile phones (SMS/MMS)".
- Such crimes may threaten a nation's security and financial health.



Cyber Crime

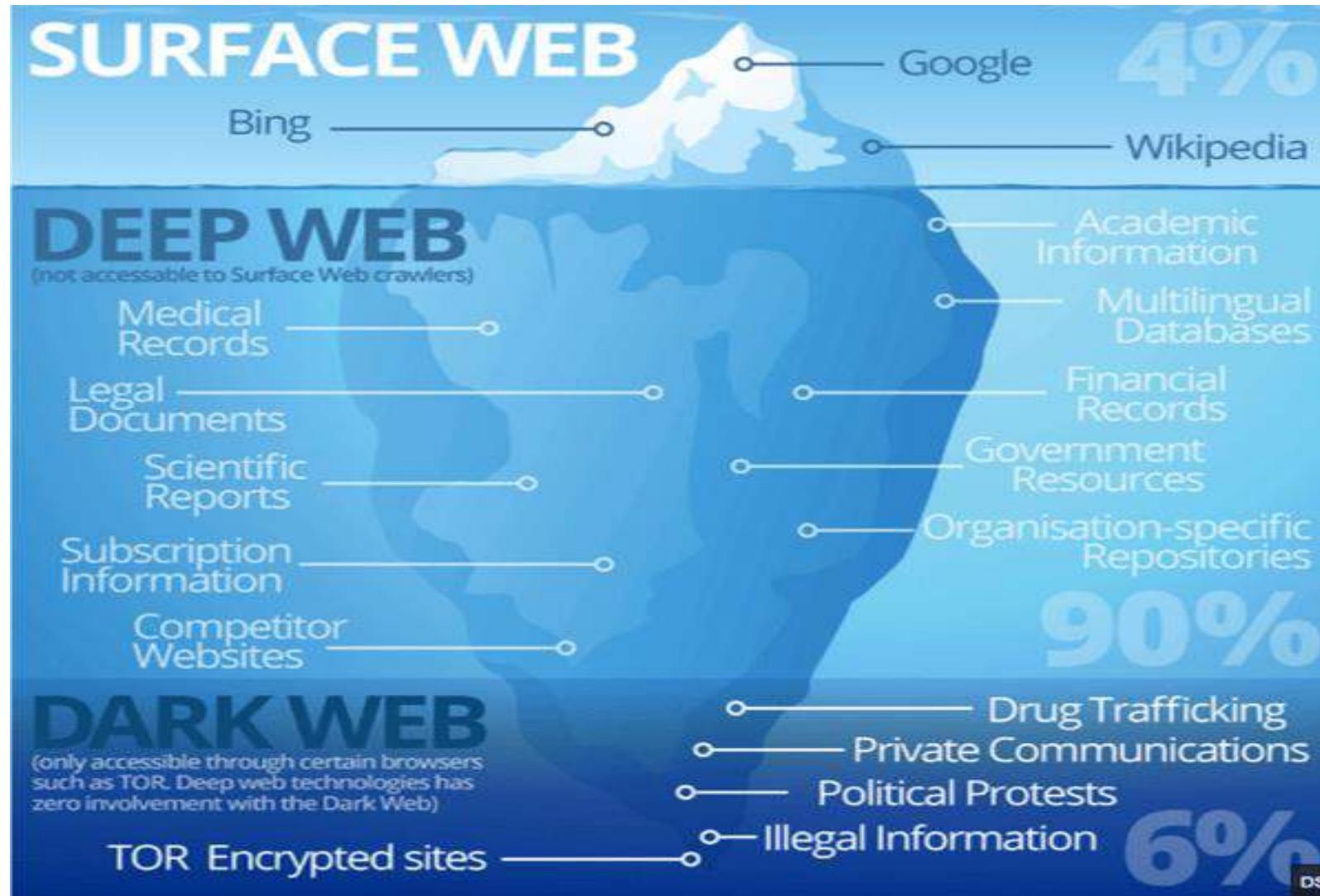
A simple yet sturdy definition of cyber crime would be

“unlawful acts wherein the computer is either a tool or a target or both”.





Surface Web vs Deep Web vs Dark Web





Surface Web vs Deep Web vs Dark Web

Dark Web vs Deep Web vs Surface Web	
Darknet/Dark Web	Restricted to special browsers Not indexed for Search Engines Large scale illegal activity Unmeasurable due to nature
Deep Web	Accessible by password, encryption, or through gateway software Not indexed for Search Engines Little illegal activity outside of Dark Web Huge in size and growing exponentially
Surface Web	Accessible Indexed for Search Engines Little illegal activity Relatively small in size



Cost of Information in Dark Web

- **Bank credential:** \$1,000 plus (6% of the total dollar amount in the account)
- **U.S. credit card with track data (account number, expiration date, name and more):** \$12
- **EU, Asia credit card with track data:** \$28
- **Hacking into a website:** \$100 to \$300
- **Counterfeit social security cards:** \$250 and \$400
- **Counterfeit driver's license:** \$100 to \$150



Classification of Cyber Crimes

Where computers are used to commit crime

- This category includes traditional offenses such as fraud committed through the use of a computer.
- Some examples are:
 1. Financial Crime
 2. Online Gambling
 3. Intellectual Property Crimes
 4. Email spoofing
 5. Cyber defamation
 6. Cyber stalking



1. Financial crimes

- This would include cheating, credit card frauds, money laundering etc.



2. Online gambling

- There are millions of websites; all hosted on servers abroad, that offer online gambling.
- In fact, it is believed that many of these websites are actually fronts for money laundering.





3. Intellectual Property crimes

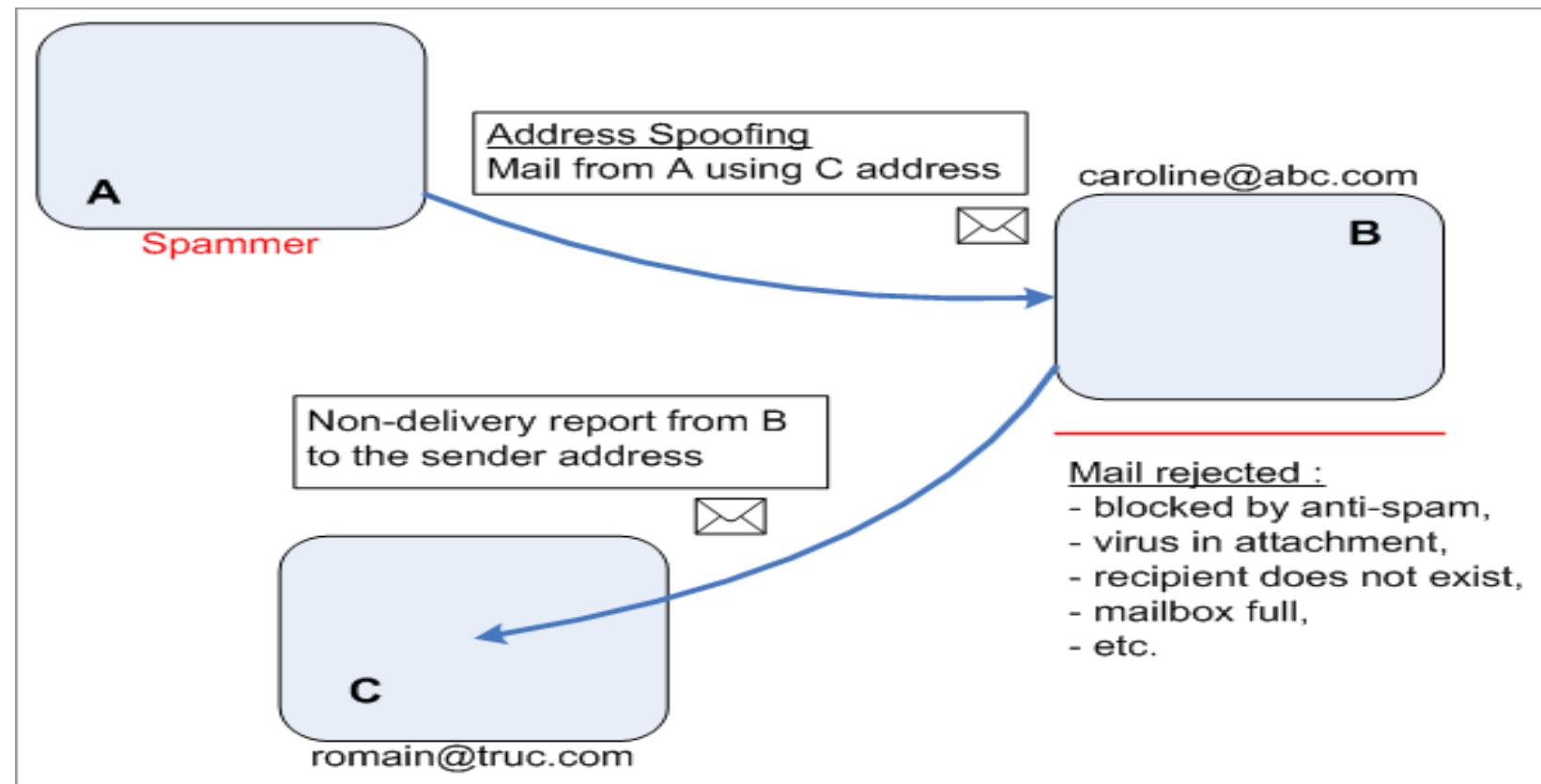
- These include software piracy, copyright infringement, trademarks violations, theft of computer source code etc.





4. Email spoofing

- A spoofed email is one that appears to originate from one source but actually has been sent from another source.





5. Cyber Defamation

- This occurs when defamation takes place with the help of computers and / or the Internet.
- Example: Someone publishes defamatory matter about someone on a website or sends e-mails containing defamatory information to all of that person's contacts.

Defamation





6. Cyber stalking

- Cyber stalking involves following a person's movements across the Internet by posting messages (sometimes threatening) on the bulletin boards frequented by the victim, entering the chat-rooms frequented by the victim, constantly bombarding the victim with emails etc.





Classification of Cyber Crimes

Where computers become target of crime

- This category includes computer oriented cyber crimes.
- Some types are:
 - A. Unauthorized Access(Hacking)
 - B. Malicious Software(Viruses, Trojans- corrupts server)
 - C. Worm (Self-replicating programs)
 - D. Spyware – parasitic software, invades privacy,
 - E. Divulging details through tracking cookies.
 - F. Cyber terrorism



A. Unauthorized Access

- Also known as Hacking.
- Involves gaining access illegally to a computer system or network and in some cases making unauthorized use of this access.
- Hacking is also the act by which other forms of cyber-crime (e.g., fraud, terrorism) are committed.



A. Theft of information

- Theft of any information contained in electronic form such as that stored in computer hard disks, removal storage media, etc.
- Can extend to identity theft.



A. Email Bombing

- This refers to sending large number of emails to the victim resulting in the victim's email account (in case of an individual) or mail servers (in case of a company or an email service provider) crashing.

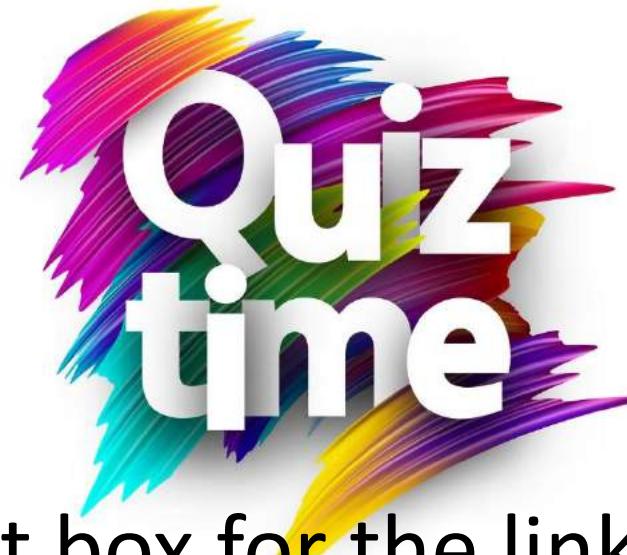


xcitefun.net



A. Salami Attacks

- These attacks are often used in committing financial crime and are based on the idea that an alteration, so insignificant, would go completely unnoticed in a single case.
- E.g. a bank employee inserts a program, into the bank's servers, that deducts a small amount of money (say 5 cents a month) from the account of every customer. This unauthorized debt is likely to go unnoticed by an account holder.



Go to posts/chat box for the link to the question

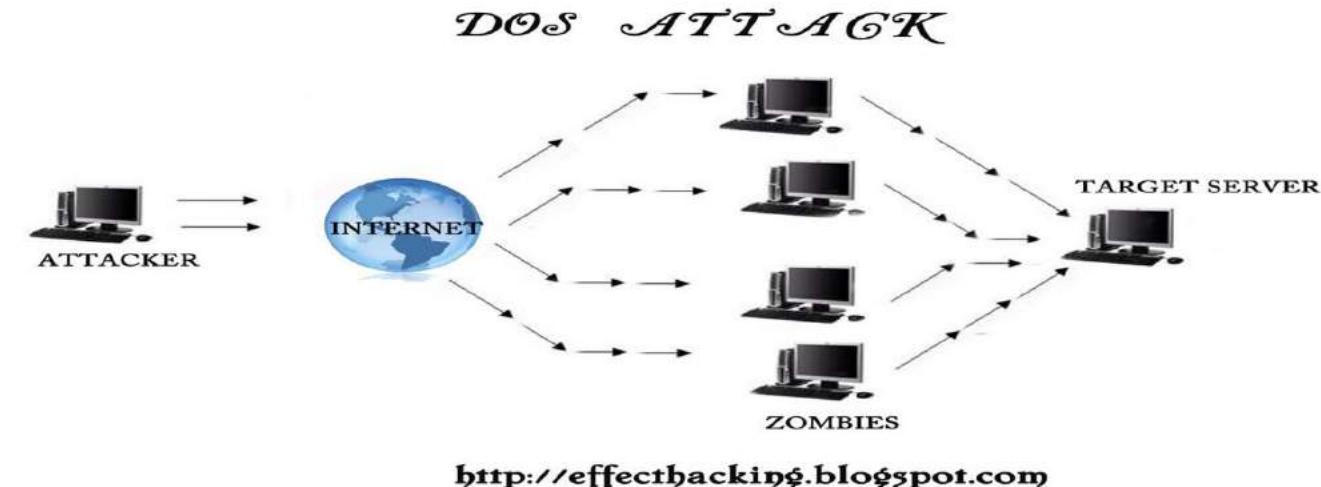
submit your solution in next 2 minutes

The session will resume in 3 minutes



A. Denial of Service (DoS) Attack

- This involves flooding a computer resource with more requests than it can handle, causing the resource (e.g. a web server) to crash thereby denying authorized users the service offered by the resource.





B. Virus

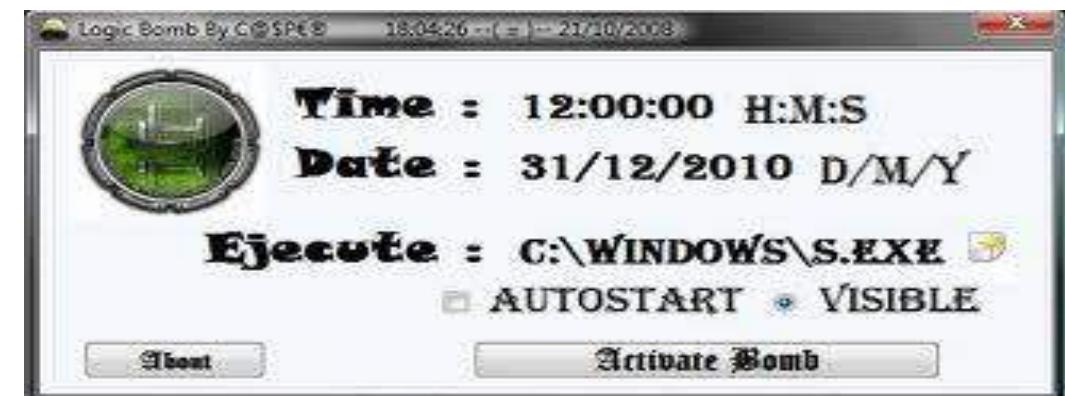
- Viruses are programs that attach themselves to a computer or a file and then circulate themselves to other files and to other computers on a network. They usually affect the data on a computer, either by altering or deleting it.





B. Logic Bombs

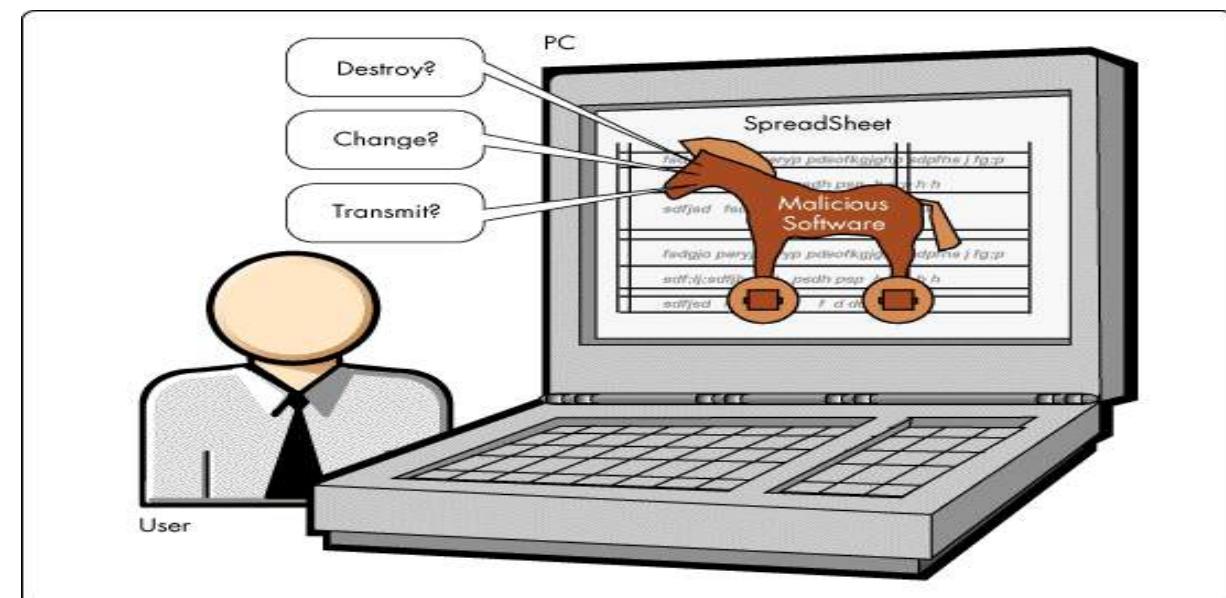
- These are event dependent programs where programs kick into action only when a certain event (known as a trigger event) occurs.
- Some viruses may be termed logic bombs because they lie dormant throughout the year and become active only on a particular date (e.g. Chernobyl virus).





B. Trojan Attacks

- An unauthorized program which functions from inside what seems to be an authorized program, thereby concealing what it is actually doing.





C. Worm

- Worms, unlike viruses do not need the host to attach themselves to. They merely make functional copies of themselves and do this repeatedly till they eat up all the available space on a computer's memory.





D. Web Jacking

- This occurs when someone forcefully takes control of a website (by cracking the password and later changing it).



E. Cyber-Terrorism

- Hacking designed to cause terror. Like conventional terrorism, 'e-terrorism' utilizes hacking to cause violence against persons or property, or at least cause enough harm to generate fear.





Computer Security

- Computer security (also known as cyber security or IT security) is information security as applied to computing devices such as computers and smartphones, as well as computer networks such as private and public networks, including the Internet as a whole.
- Computer Security is the protection of computing systems and the data that they store or access.



Computer Security

- Computer Security covers all the processes and mechanisms by which computer-based equipment, information and services are protected from unintended or unauthorized access, change or destruction.
- Computer security also includes protection from unplanned events and natural disasters.



Why is Computer Security Important?

- Enabling people to carry out their jobs, education, and research.
- Supporting critical business process.
- Protecting personal and sensitive information.



Why do I need to learn about Computer Security?

- Good Security Standards follow the "90 / 10" Rule:
- 10% of security safeguards are technical.
- 90% of security safeguards rely on the computer user ("YOU") to adhere to good computing practices
- Example: The lock on the door is the 10%. You remembering to lock the lock, checking to see if the door is closed, ensuring others do not prop the door open, keeping control of the keys, etc. is the 90%. You need both parts for effective security.

What Does This Mean for Me?

- This means that everyone who uses a computer needs to understand how to keep their computer and data secure.
- Information Technology Security is everyone's responsibility



Simple measures to be followed...

- Many cyber security threats are largely avoidable. Some key steps that everyone can take include:
 - Use good, cryptic passwords that can't be easily guessed and keep your passwords secret
 - Make sure your operating system and applications are protected with all necessary security patches and updates
 - Make sure your computer is protected with up-to-date antivirus and anti-spyware software



Simple measures to be followed...

- Don't click on unknown or unsolicited links or attachments, and don't download unknown files or programs onto your computer

- Remember that information and passwords sent via standard, unencrypted wireless are especially easy for hackers to intercept
 - To help reduce the risk, look for **https** in the URL and the little padlock that appears in the URL bar or in a corner of the browser window before you enter any sensitive information or a password.
 - Also avoid standard, unencrypted e-mail and unencrypted Instant Messaging (IM) if you are concerned about privacy

What are the consequences for security violations?

- Risk to security and integrity of personal or confidential information
 - e.g. identity theft, data corruption or destruction, unavailability of critical information in an emergency, etc.
- Loss of valuable business information
- Loss of employee and public trust, embarrassment, bad publicity, media coverage, news reports
- Costly reporting requirements in the case of a compromise of certain types of personal, financial and health information
- Internal disciplinary action(s) up to and including termination of employment, as well as possible penalties, prosecution and the potential for sanctions / lawsuits



Summary

- Definition of Cyber Crime
- Classification of Cyber crimes
- Computer Intrusions and Hacking
- Computer Security