

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA



BITS Pilani

Pilani Campus

Module 1 - Introduction to Computer Programming

Department of Computer Science & Information Systems

Module Overview

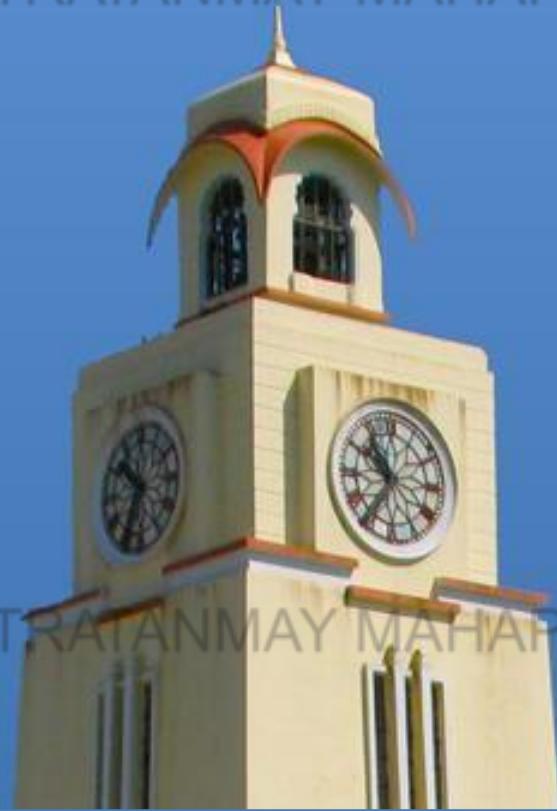
- **Objectives of the Course**
- **Computers and Computing**
- **Introduction to Programming**
- **The C language**
- **Program and Process Execution**



Course Objectives

Course Objectives

- *Learn about Computers and Computing Methodologies*
- *Learn basic concepts of programming using the C language*
- *Construct Solutions to Scientific Problems through programming*
- *Systematic techniques and approaches for constructing programs*



Computers and Computing

We have seen Computers...



**Desktop
Computer**



**Laptop
Computer**

Smart Devices (Small Computers)



Tablet

Smart Phone



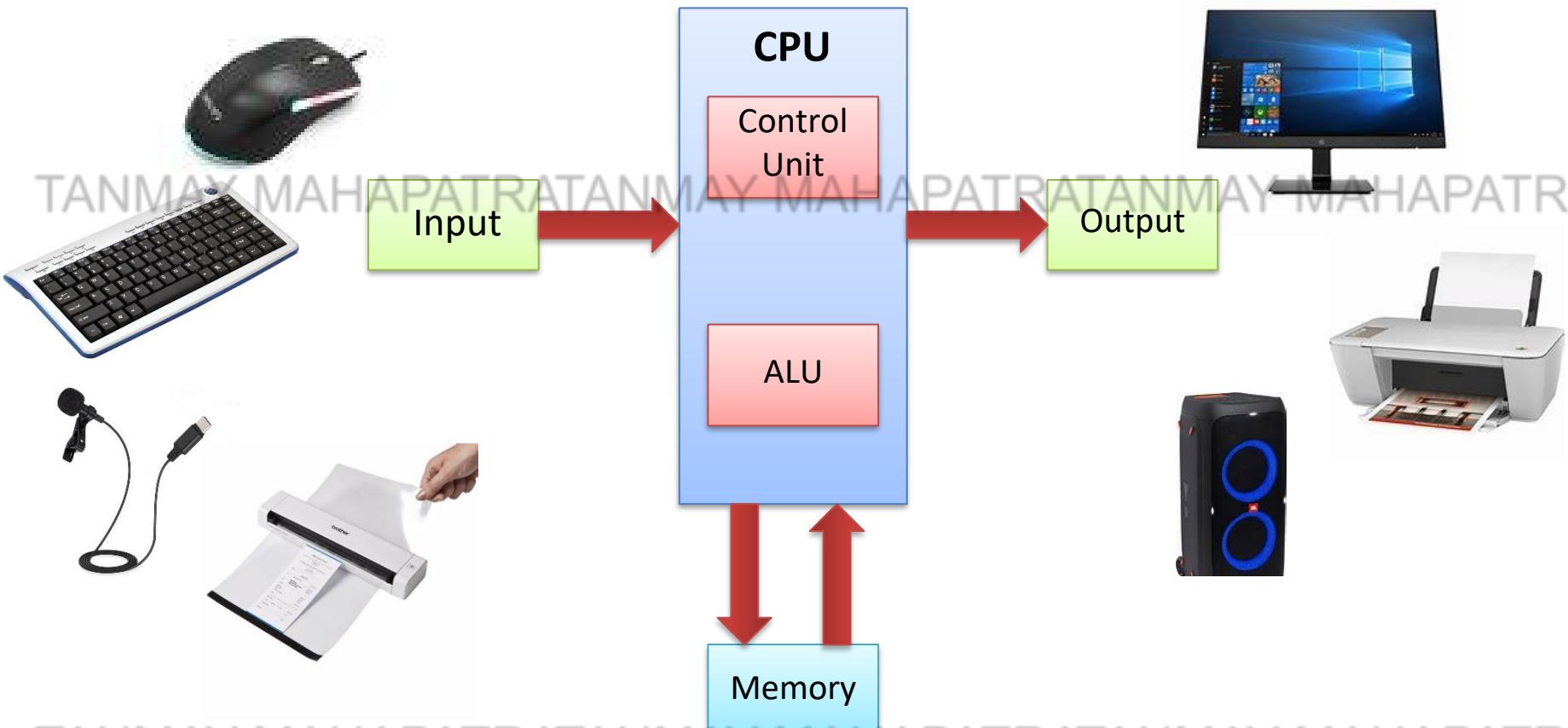
Smart Watch

Super Computer...



What is a Computer?

A device that takes data as input, does some processing and then returns an output.



What is a CPU?

- CPU stands for **Central Processing Unit**.
- The CPU is often simply referred to as the **processor**.
- The CPU is the **brain of a computer**, containing all the electronic circuitry needed to process input, store data, and output results.
 - Essentially **CPU executes computer programs**.
- It consists of an **arithmetic and logic unit** (ALU), and a **control unit**
 - ALU performs arithmetic and logical operations specified by a computer program
 - Control Unit directs the operations of the Processor



**Intel i7
Processor**

What is memory?

- Computer memory is the storage space in the computer, where “**data to be processed**” and “**instructions required for processing**” are stored.

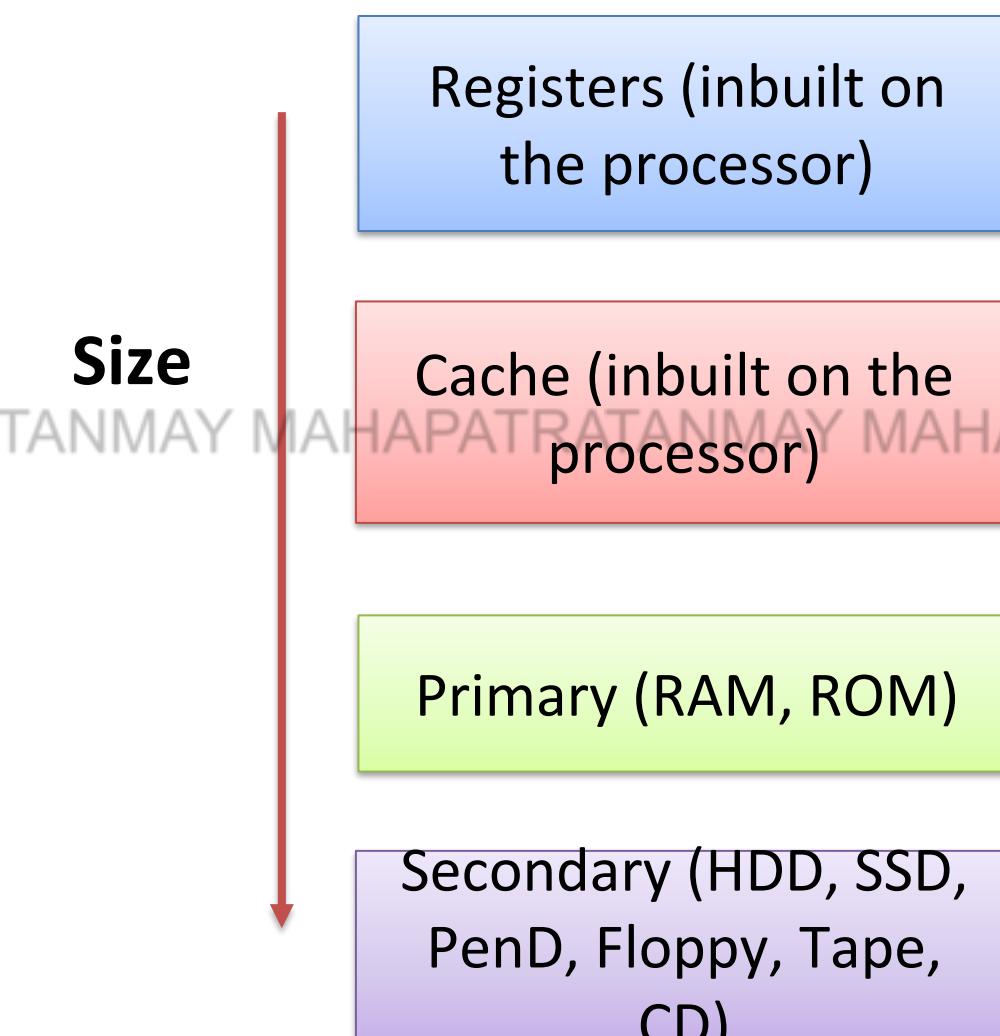


RAM chips



Hard Disk

Types of Memory



*In this course, we will deal with Primary memory (**RAM**) and Secondary Memory (**DISK**) only!*

Speed

Registers, Cache and **RAM** are volatile

- Retains data as long as the voltage is applied

ROM and **Secondary devices** are non-volatile

- Retains data permanently without any voltage

Hardware vs Software

HARDWARE

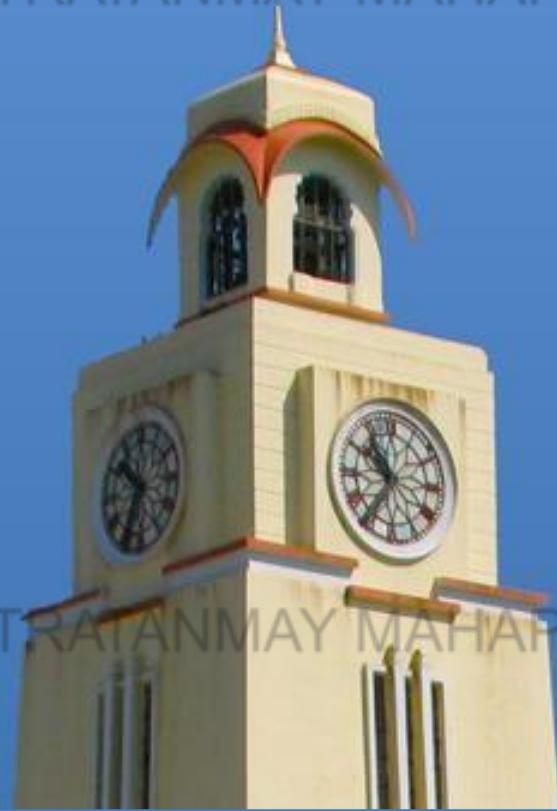


SOFTWARE

vs



- *Hardware is the physical parts of the computer: **motherboard, CPU, RAM, keyboard, monitor**, etc.*
- *Software is a collection of instructions that can be run on the hardware of the computer. These instructions tell the computer what to do.*
 - *In other words, **Software is a Computer Program**.*
 - ***Software is not a physical thing**, just a bunch of programs.*



Introduction to Programming

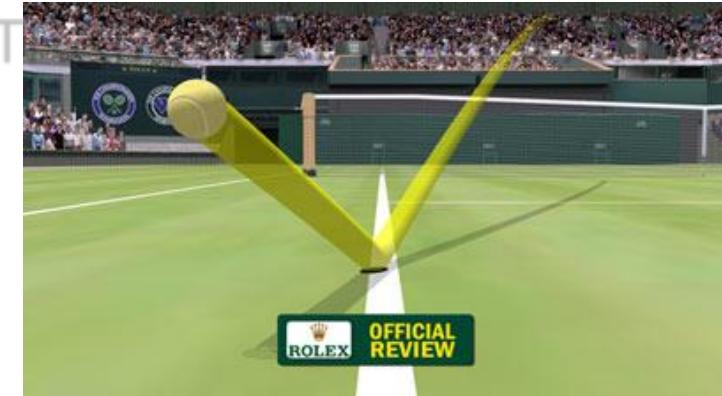
What is a Computer Program?



“A Computer Program (or simply a Program) is a collection of instructions that performs a specific task when executed by a computer”

Where are Programs Used?

- **MS Word** is a Program 
- **Google Chrome** is a Program 
- **Skype** is a Program 
- **Hawk's-eye** is a Program
- **Windows OS** is a Program 



Anything we do on a computer requires us to write a program...

Programming Language

- Programs in a computer are written using a *Programming Language*
- A programming language is a **vocabulary** and **set of grammatical rules** for instructing a computer to perform specific tasks.
- Examples: **C, C++, Java, Python**, etc.
- Each language has a **unique set of keywords** (words that it understands) and a **special syntax** for organizing program instructions.

Levels of Programming Languages

High Level Language

Assembly Language

Machine language

Ease of understanding



Machine Language

- **Language of the machine**
- **Made up of a series of binary patterns**



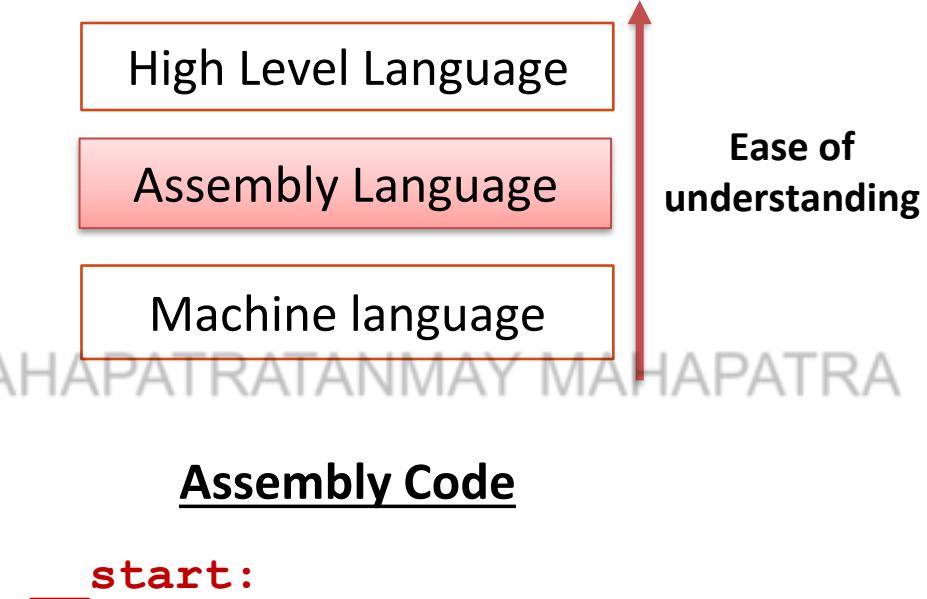
- **Can be run directly**
- **Difficult to learn**

Machine Code

```
1001110100001101000000  
0110000110100001111011  
100000010111101101110  
1111011000101101100010  
1000000010011110000110  
1001001100011100000001
```

Assembly Language

- **Uses simple mnemonic abbreviations**
- **Unique to a specific CPU architecture**
- **Requires an assembler**



```
    mov edx, len  
    add ecx, edx  
    mov ebx, 1  
    sub eax, edx  
    int 0x80
```

High level Language

- English-like
- Easier to understand
- Requires Compiler
- Compiler: Generates object/machine code

High Level Language

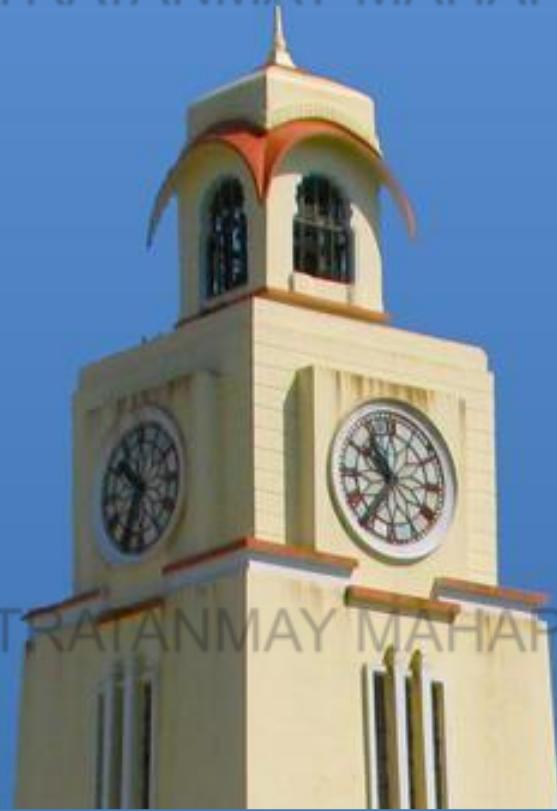
Assembly Language

Machine language

Ease of understanding

High-level Code

```
#include<stdio.h>
int main() {
    int a, b, sum;
    scanf("%d %d", &a, &b);
    sum = a + b;
    printf("The sum is %d\n", sum);
    return 0;
}
```



The C Programming Language

The C language

- General Purpose **high-level** programming language
 - Closely associated with the UNIX operating system
 - **Most versatile** as it supports direct access of memory locations through pointers
 - **Compiler** based language
 - *C programs are known for speed* **More about C in and efficiency!**
- We will Study*
this course



Program and Process Execution

Perform tasks with Programs

You want to perform a specific task using a computer. You will do the following steps:

- Write a C program specific to your task
 - Save it on the computer
 - Compile and Run your program

What happens to your program?



You write a program...

1. *Where is your program saved?*

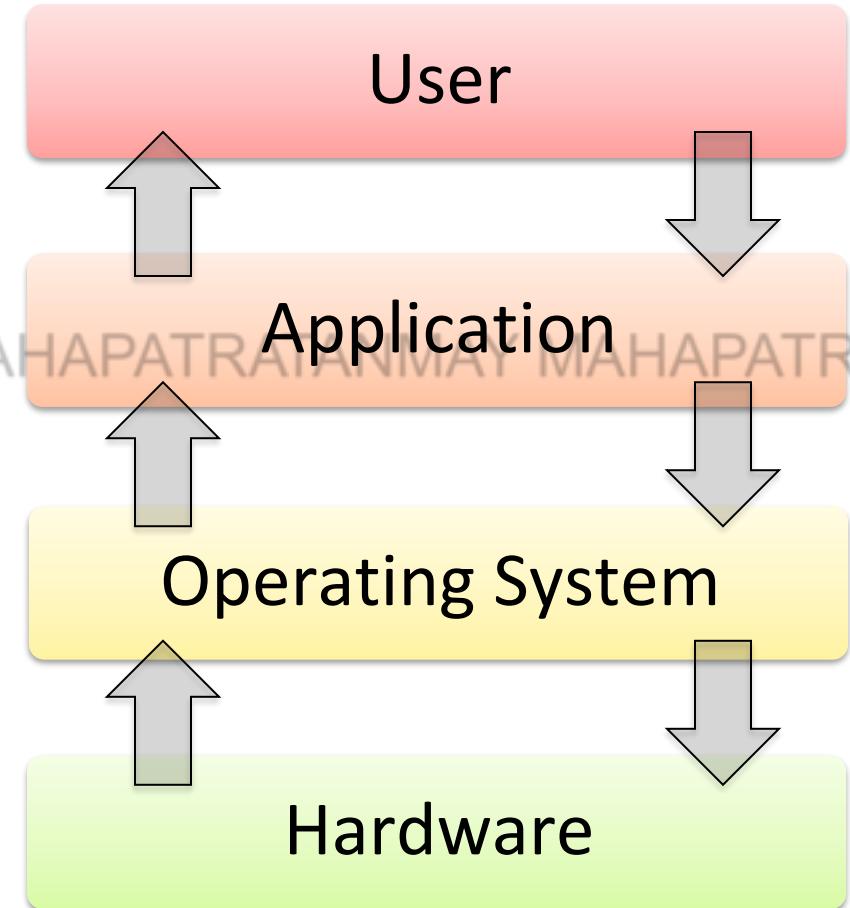
1. *What happens to your program when you try to compile and run it?*

Before we answer the above questions let us see what is an operating system...

Operating System (OS)

- An **OS** is a layer of software interposed between the application programs and the hardware

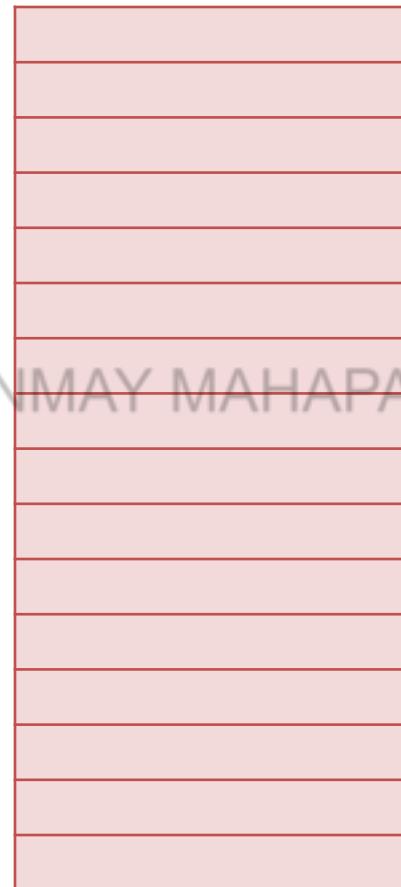
- OS manages everything on your computer including hardware
- OS is responsible for executing your program



Basic layout of Computer Hardware



CPU



Disk (Secondary
Memory)

OS Manages these
resources

What happens to your program?



Now let us answer these questions...

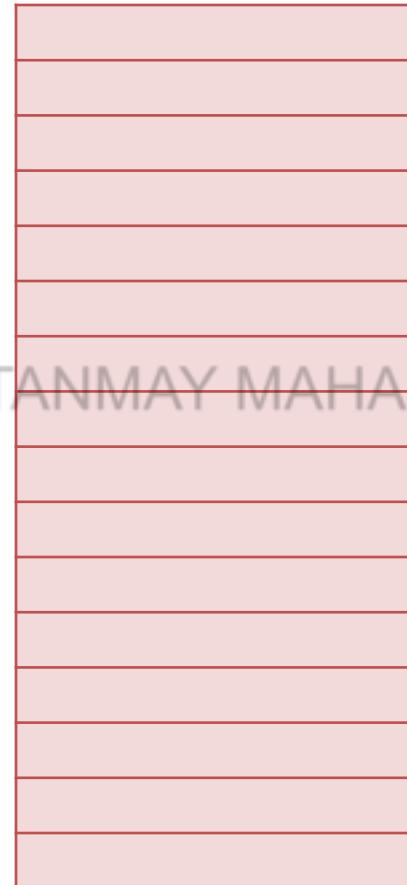
1. *Where is your program saved?*

1. *What happens to your program when you try to compile and run it?*

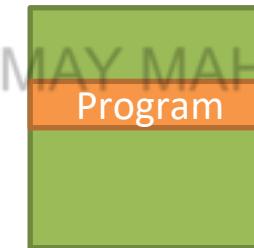
1. Where is your program saved?



CPU



Primary
Memory (RAM)



Disk (Secondary
Memory)

Your program when
saved gets stored on
the disk

What happens to your program?

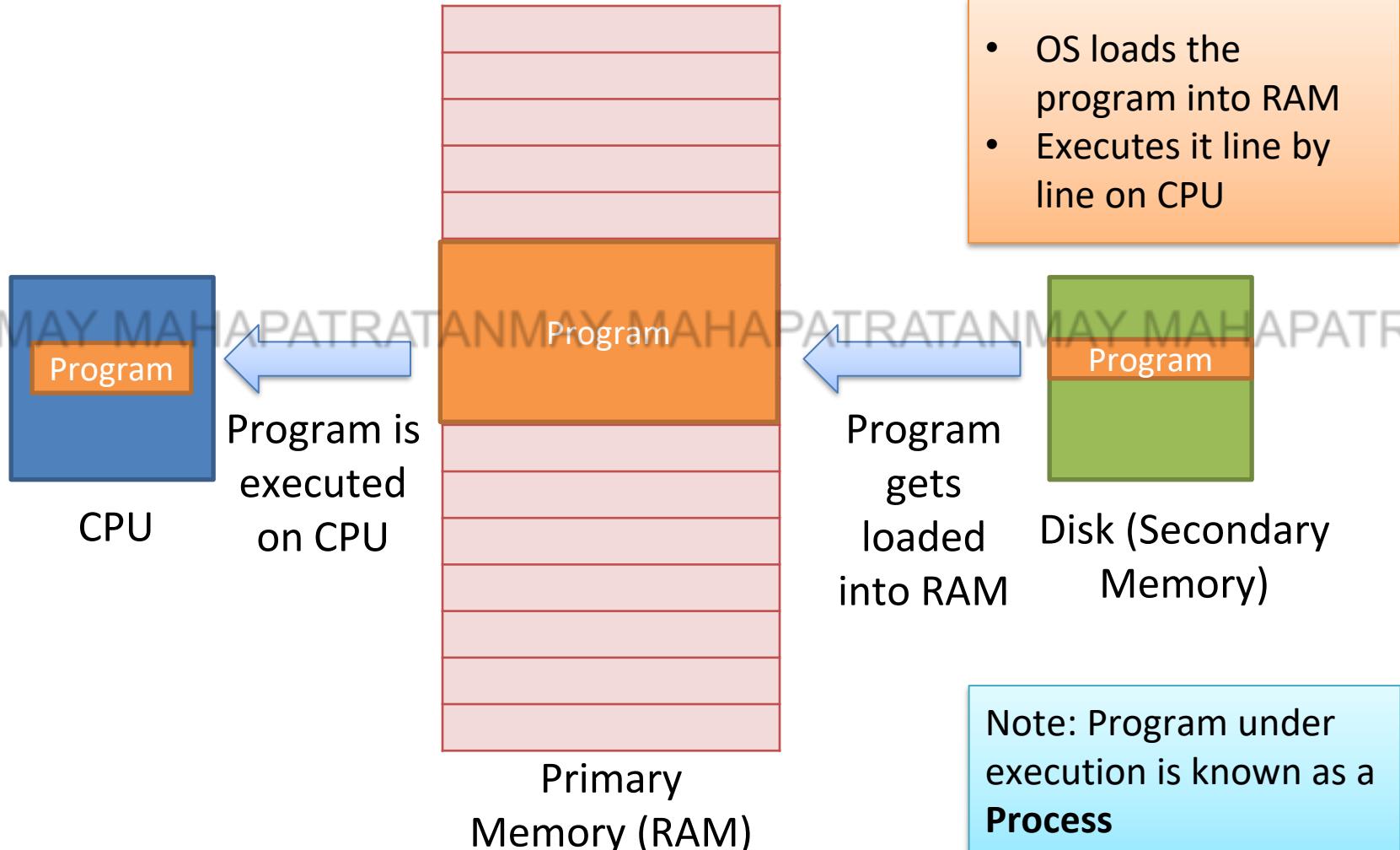


Now let us answer these questions...

1. *Where is your program saved?*

1. *What happens to your program when you try to compile and run it?*

2. What happens to your program when you try to compile and run it?





Thank you
Q & A

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA



Module 2 – Flowcharts & Algorithms

BITS Pilani

Pilani Campus

Department of Computer Science & Information Systems

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA

Module Overview

- **Steps of Programming Practices**

- **Flowcharts**

- **Algorithms**



Steps of Programming Practices

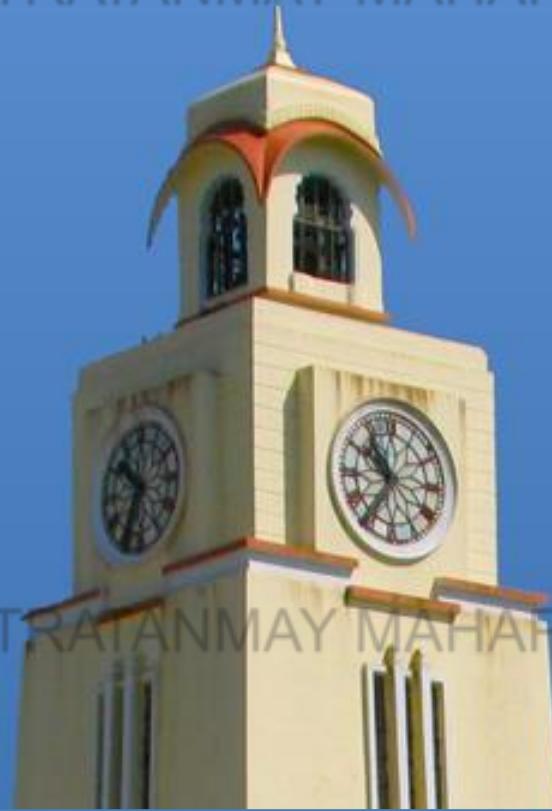
Steps of Programming Practices

- **Step1:** Requirements

- **Step2:** Creating a flow chart
- **Step3:** Creating algorithms

We are going to
look at **Flowcharts**
and **Algorithms**

- **Step4:** Writing Code
- **Step5:** Debugging
- **Step6:** Documentation



Flowcharts

Flow Chart

- A Graphical representation of a solution to a particular problem
- Created by Von Neumann in 1945

- Flowcharts help in developing logic and algorithms before writing a program

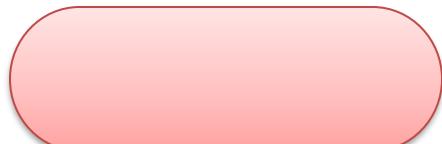
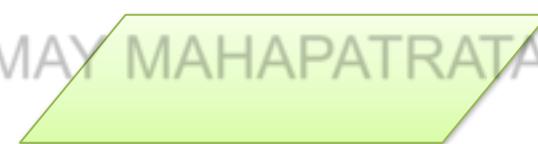
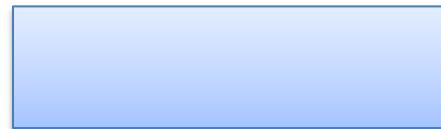
How to make a flow chart

Step1: Identify input and output.

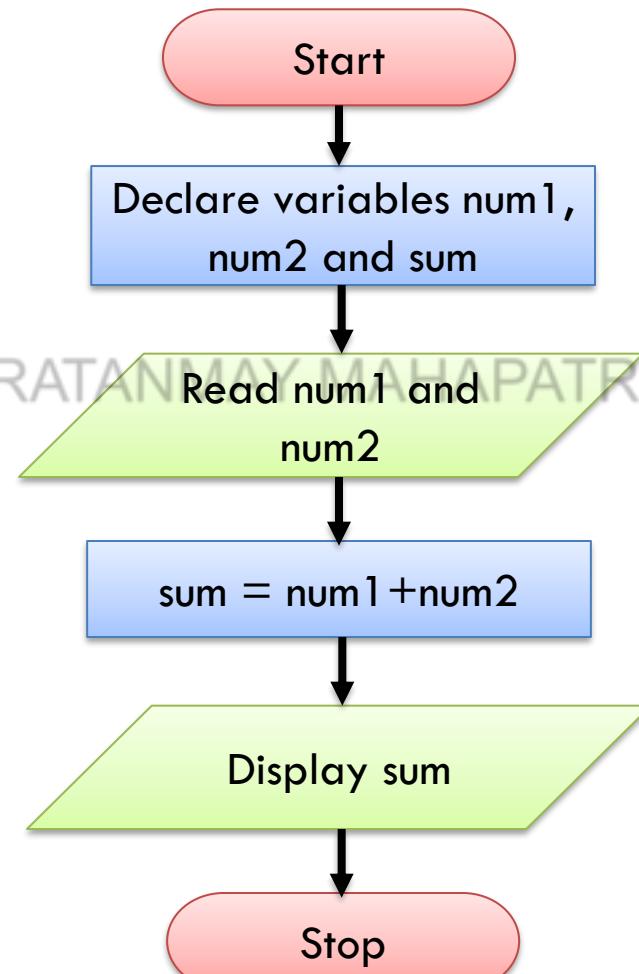
Step2: Apply reasoning skills to solve the problem

Step3: Draw the flowchart using the appropriate **symbols** and **arrows** to show the sequence of steps in solving the problem

Symbols used in Flow Charts

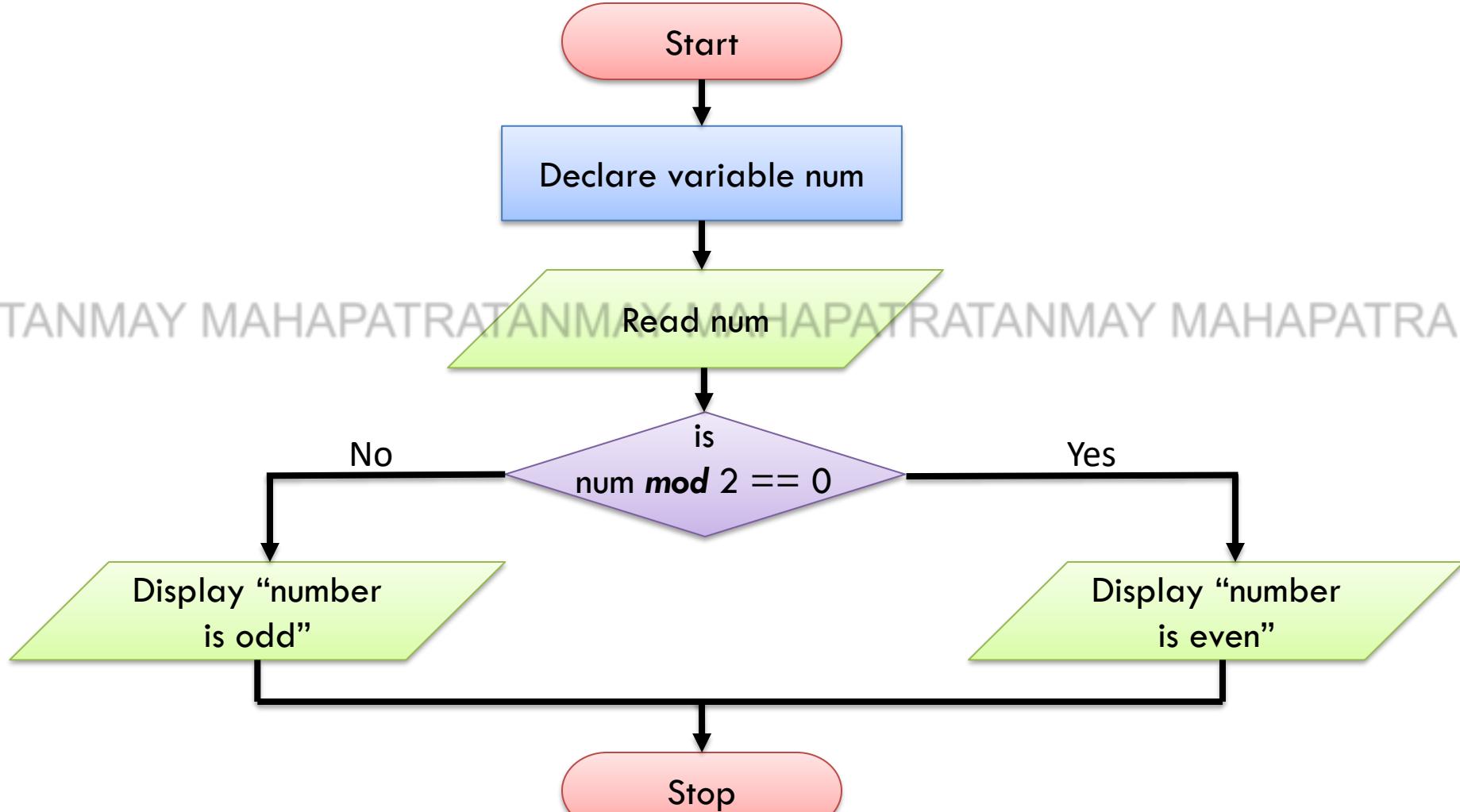
	 Flow Line Shows the logical flow of control
	 Decision Symbol One flow of line for Input and two for Output
	 Connector Connects separate portions of a flow chart

Flow Chart Example 1: Sum of two numbers

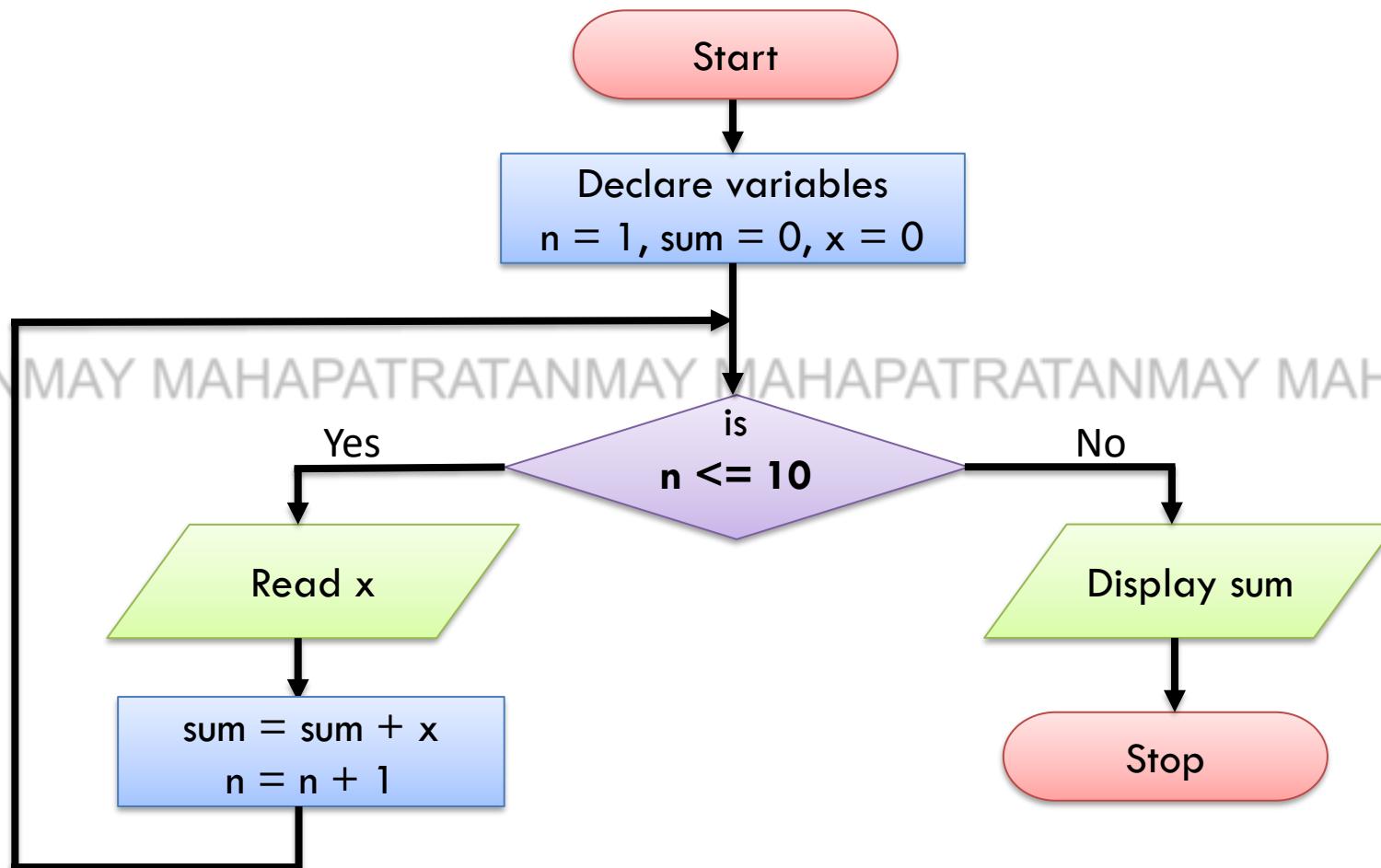


Flow Chart Example 2:

Whether a number is even or odd



Flow Chart Example 3: Sum of 10 Numbers



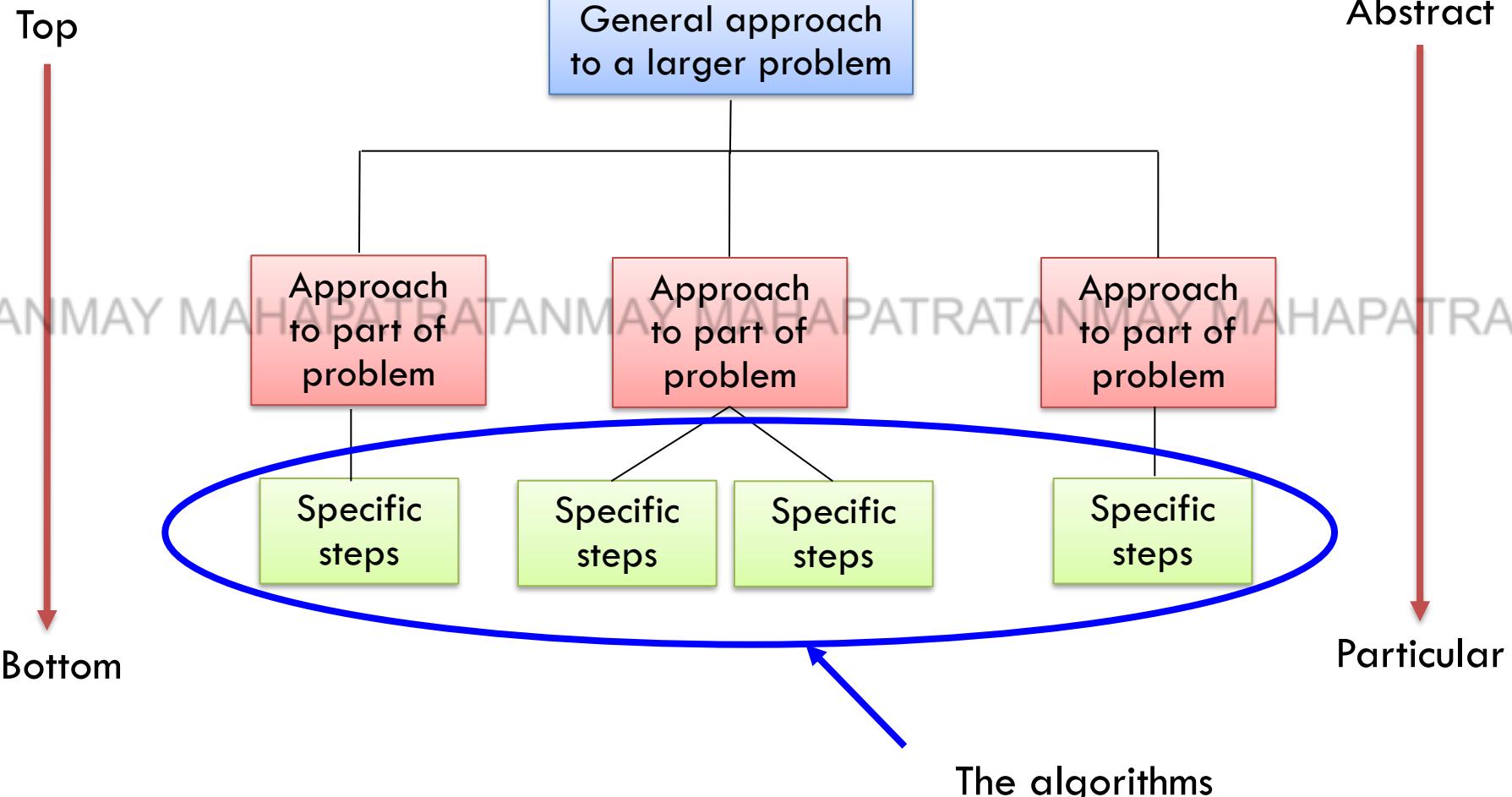


Algorithms

Algorithms

“Algorithms are a way of laying out a problem independent of the programming language”

Programming Methodologies



Top-down vs Bottom-up Programming



- **Top – Down Programming:**

- Reverse engineers the finished products by breaking it into smaller, manageable or reusable modules.
- Big picture is visualized first before breaking it up into separate components.

- **Bottom – UP Programming:**

- Low level components are designed first without fully knowing the big picture.
- The components are later integrated to form the complete system.

What is an Algorithm?

- ❖ Step by step solution to a problem in English like language.
- ❖ It is a finite set of clear and unambiguous steps that must be followed to solve a computing problem.

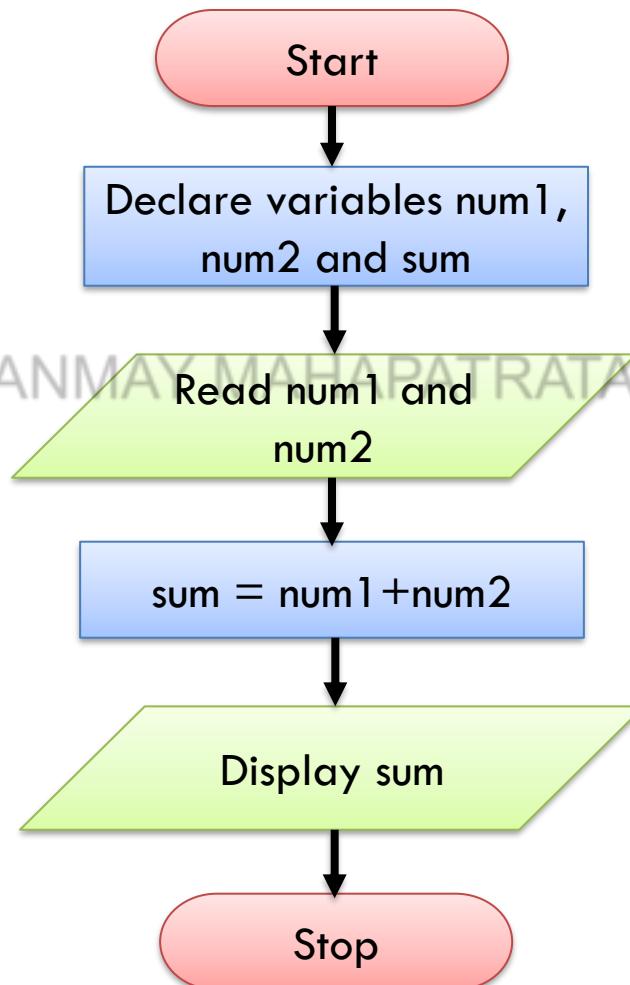
- ❖ The programmer can convert the algorithm to a program in any language.

Programming Logic Constructs

- Imperative statements (Actions)
- Conditional statements (Decision Making)
- Iterative Statements (Repetition / Loops)

Algorithm Example 1:

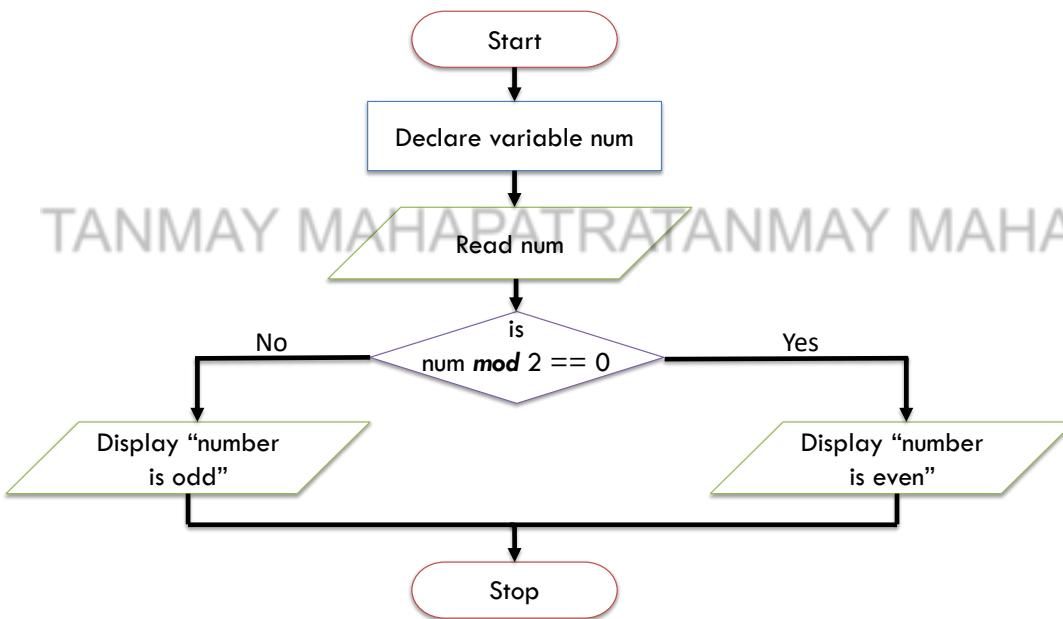
Sum of two numbers (Imperative)



1. **START**
2. Initialize sum=0
3. **INPUT** the numbers num1, num2
4. Set sum = num1 + num2.
5. **PRINT** sum
6. **END**

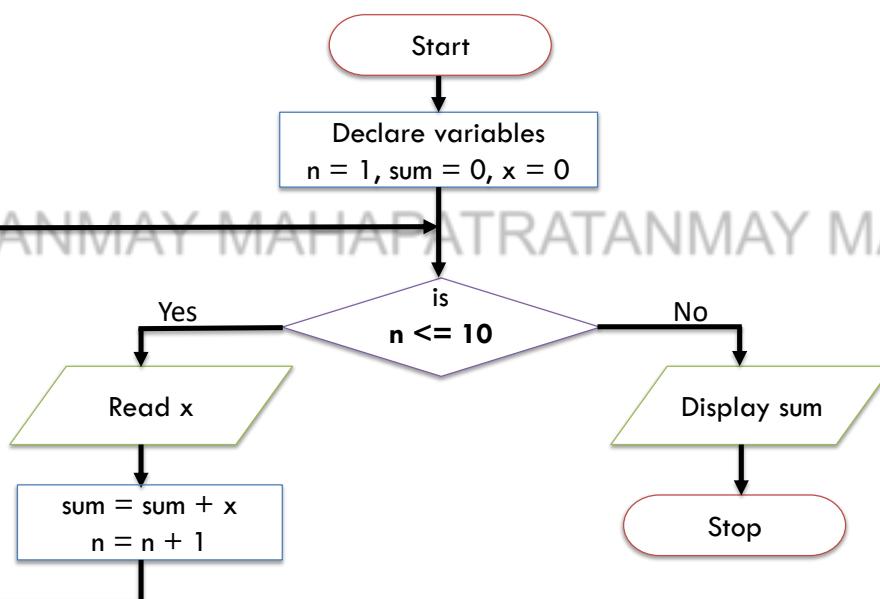
Algorithm Example 2:

Whether a number is even or odd (Conditionals)



- 1. START**
- 2. Declare num**
- 3. INPUT** the number num
- 4. IF** num mod 2 == 0 **THEN**
continue ELSE GOTO step 7
- 5. PRINT** “number is even”
- 6. GOTO** step 8
- 7. PRINT** “number is ODD”
- 8. END**

Algorithm Example 3: Sum of 10 Numbers (Iterative)



- 1. START**
- 2. Declare $n=1$, $sum=0$, $x=0$**
- 3. IF $n \leq 10$ THEN continue ELSE GOTO step 8**
- 4. INPUT x**
- 5. Set $sum = sum + x$**
- 6. Set $n = n + 1$**
- 7. GOTO step 3**
- 8. PRINT sum**
- 9. END**

Exercise

Roots of Quadratic equation

$ax^2 + bx + c = 0$ has the following roots:

$$D = b^2 - 4ac$$

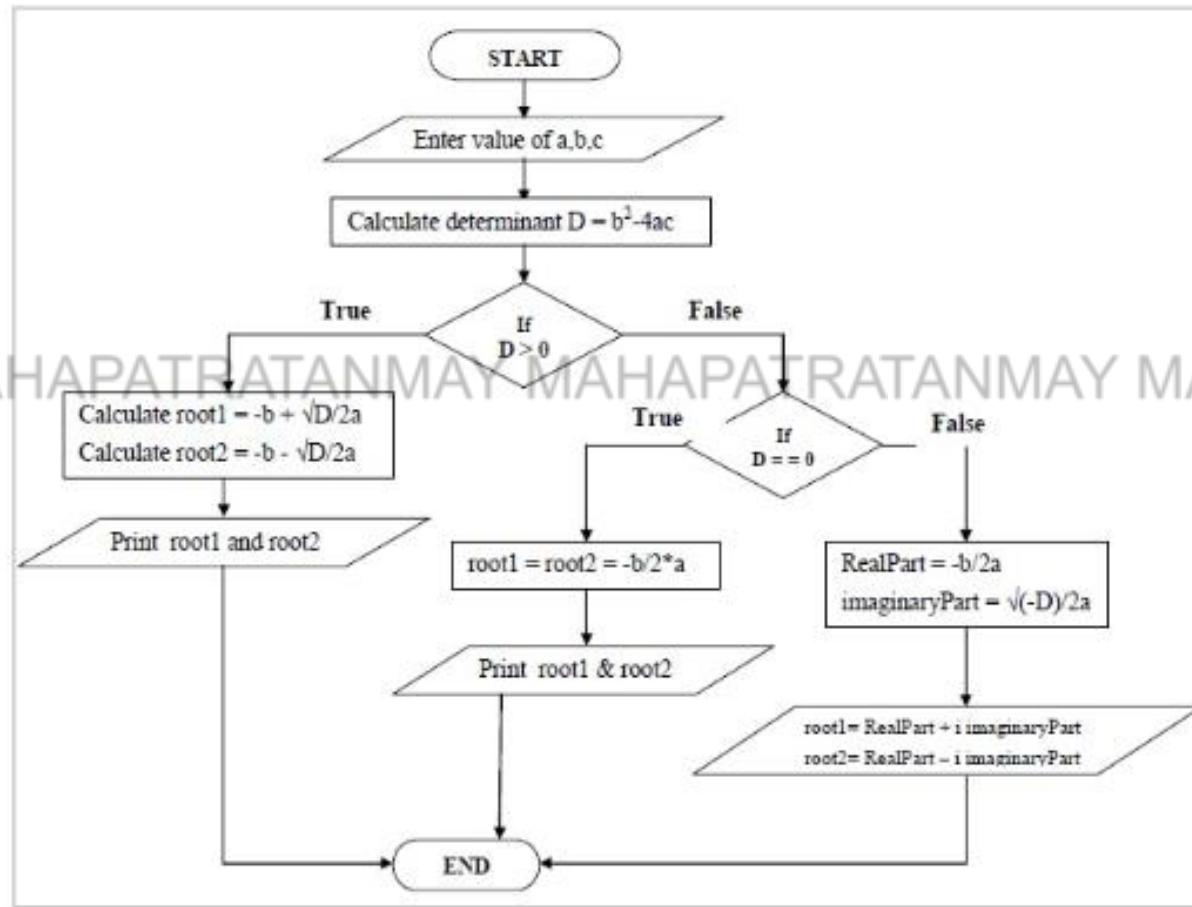
Hint: If $D > 0$ then x

$$\text{root 1} = (-b + \sqrt{D})/2a$$

$$\text{root 2} = (-b - \sqrt{D})/2a$$

- Draw a *flowchart to find roots of the above quadratic equation*
- Write an equivalent algorithm for the above flow chart

Solution



Roots of Quadratic Equation: Algorithm



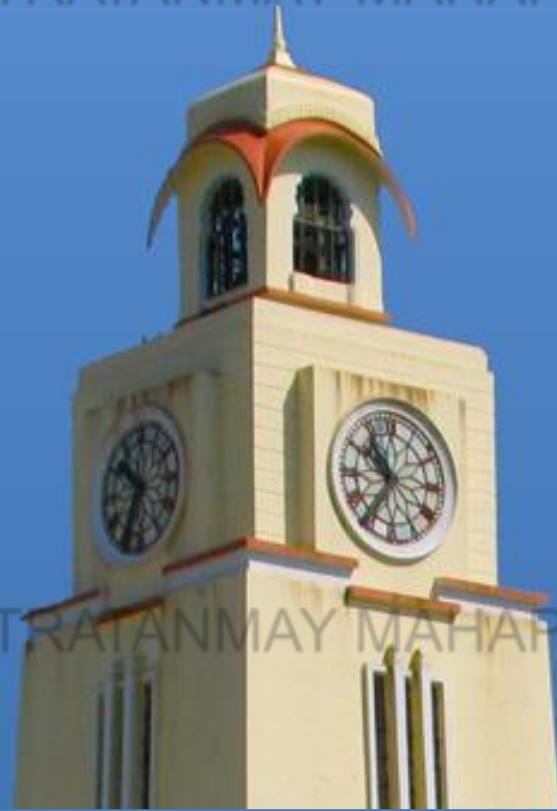
1. Start
2. Input the values of a, b and c.
3. Calculate the determinant as $D = b^2 - 4ac$
4. If ($D > 0$) then continue else GOTO Step 9
5. Calculate $\text{root1} = -b + \sqrt{D}/2a$
6. Calculate $\text{root2} = -b - \sqrt{D}/2a$
7. Print value of root1 and root2 (these are Real and different)
8. GOTO END

Roots of Quadratic Equation: Algorithm (contd.)

9. If ($D==0$) then continue else GOTO Step 13
10. Calculate $\text{root1} = \text{root2} = -b/2*a$
11. Print value of root1 and root2 (the roots are real and equal).
12. GOTO END
13. Calculate $\text{RealPart} = -b/2a$
14. Calculate $\text{imaginaryPart} = \sqrt{(-D)/2a}$
15. Print value of $\text{root1} = \text{RealPart} + i \text{ imaginaryPart}$
16. Print value of $\text{root2} = \text{RealPart} - i \text{ imaginaryPart}$ (the roots are complex and different)
17. END

More Exercises

- Draw a flowchart and write its corresponding algorithm for checking if a number n entered by the user is a prime number or not. [Hint: Check for all numbers from 1 to $n/2$ if they divide n with remainder 0 or not. If none of them divides, n is a prime number]
- Draw a flowchart and write its corresponding algorithm for computing the greatest common divisor (GCD) of two numbers entered by the user. [Hint: Follow this [link](#) to understand what is GCD]



Thank you
Q & A

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA



BITS Pilani
Pilani Campus

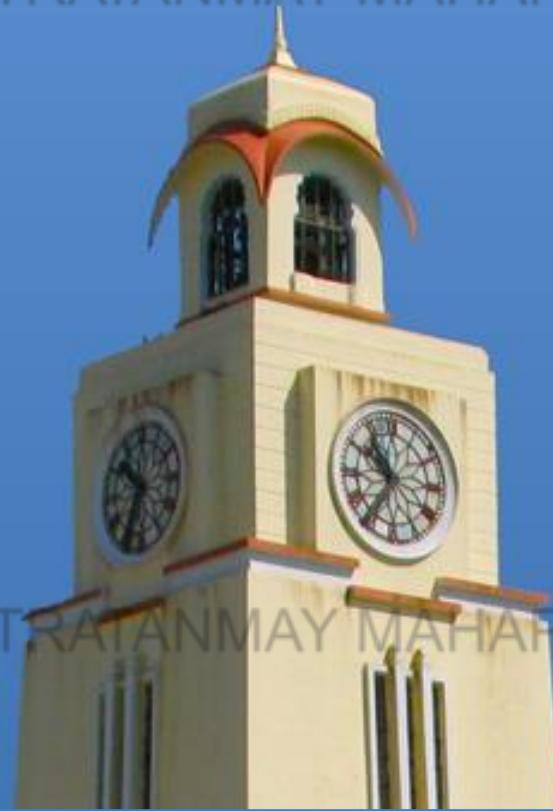
Module 3 – Basic C Program and its execution

Department of Computer Science & Information Systems

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA

Module Overview

- **Basic C Program**
- **Compilation and Execution of a C Program**
- **Errors in C Programs**

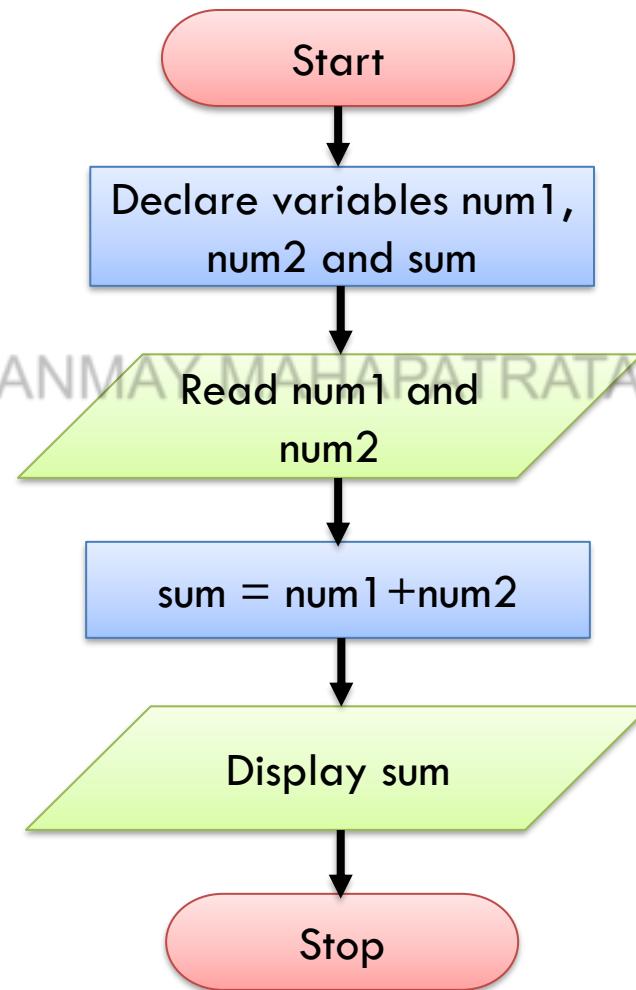


Basic C Program

Steps of Programming Practices

- **Step1:** Requirements
- **Step2:** Creating a flow chart
- **Step3:** Creating algorithms
- **Step4:** Writing Code
- **Step5:** Debugging
- **Step6:** Documentation

Example – Sum of Two Numbers



1. **START**
2. Initialize sum=0
3. **INPUT** the numbers num1, num2
4. Set sum = num1 + num2.
5. **PRINT** sum
6. **END**

C Code for Sum of Two Numbers

```
/* myfirst.c: to compute the sum of two numbers */
#include<stdio.h>           ← Preprocessor Directive
/*Program body*/
int main()                   ← The main() function where any
{                           program's execution begins
    int a, b, sum; //variable declarations
    printf("Please enter the values of a and b:\n");
    scanf("%d %d", &a, &b);
    sum = a + b; // the sum is computed
    printf("The sum is %d\n", sum);
    return 0; //terminates the program
}                           ← Block of the main() function.
                            Any block is enclosed in {...}
```

a, b, sum are variables or placeholders for values

A block consists of statements ending with ;

Header files

```
#include <stdio.h>
```

- A directive to place the contents of the header file, `stdio.h`, in the program
- A header file contains data used by multiple programs
- Processed by a preprocessor

Functions

```
printf("Please enter the values of a and b:\n");
scanf("%d %d", &a, &b);
```

- Called by its name to execute a set of statements
- Multiple programs can use the same function
- **printf** is used to print some value to the screen
- **scanf** is used to read some value from the user
- **main** is a function
- A program can consist of many more user-defined functions.

Adding comments

```
/* This is a comment */  
// And so is this
```

Judicious use of comments helps in easy understanding of the code
(for yourself and others)

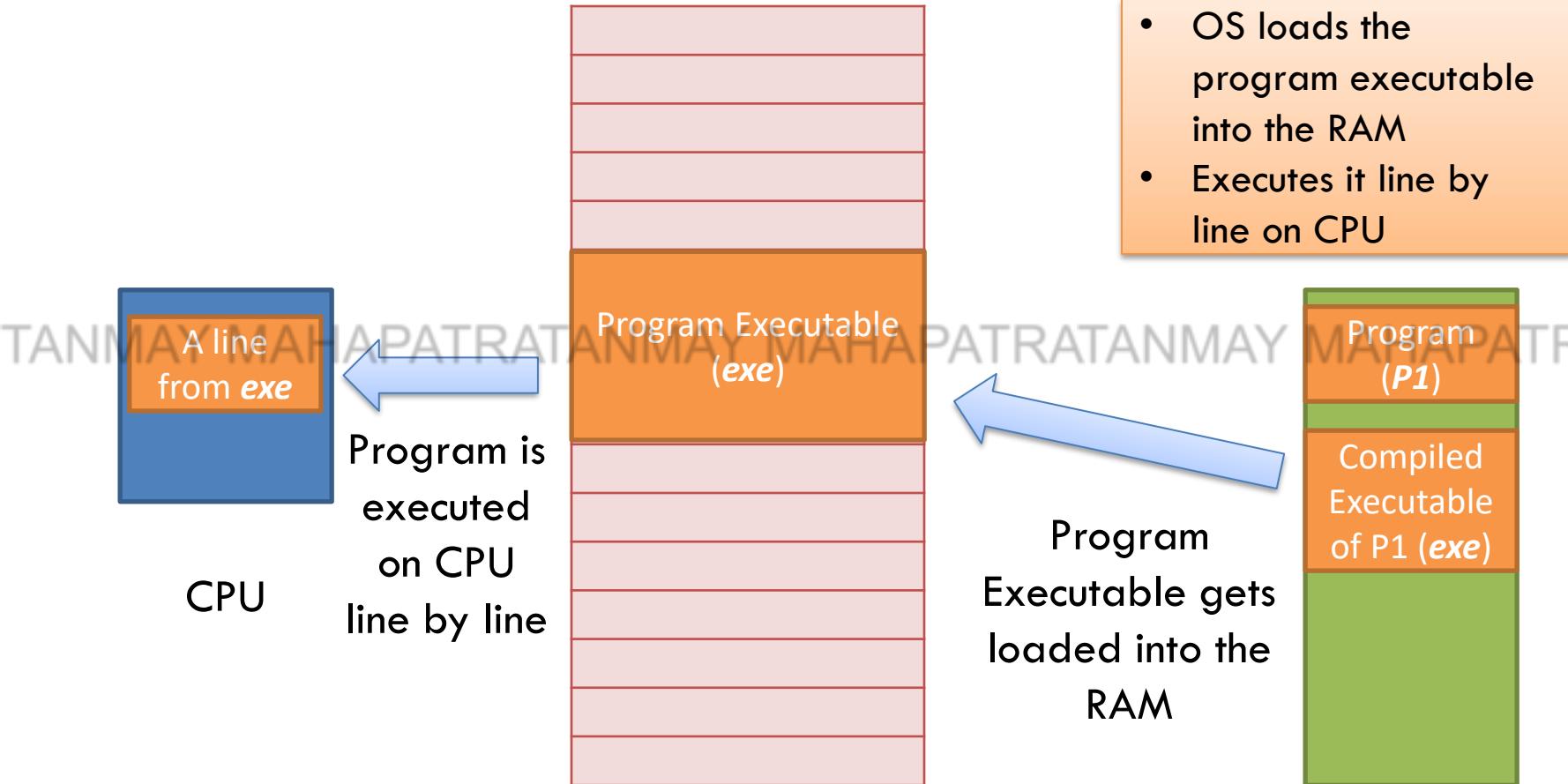


Compilation & Execution of a C Program

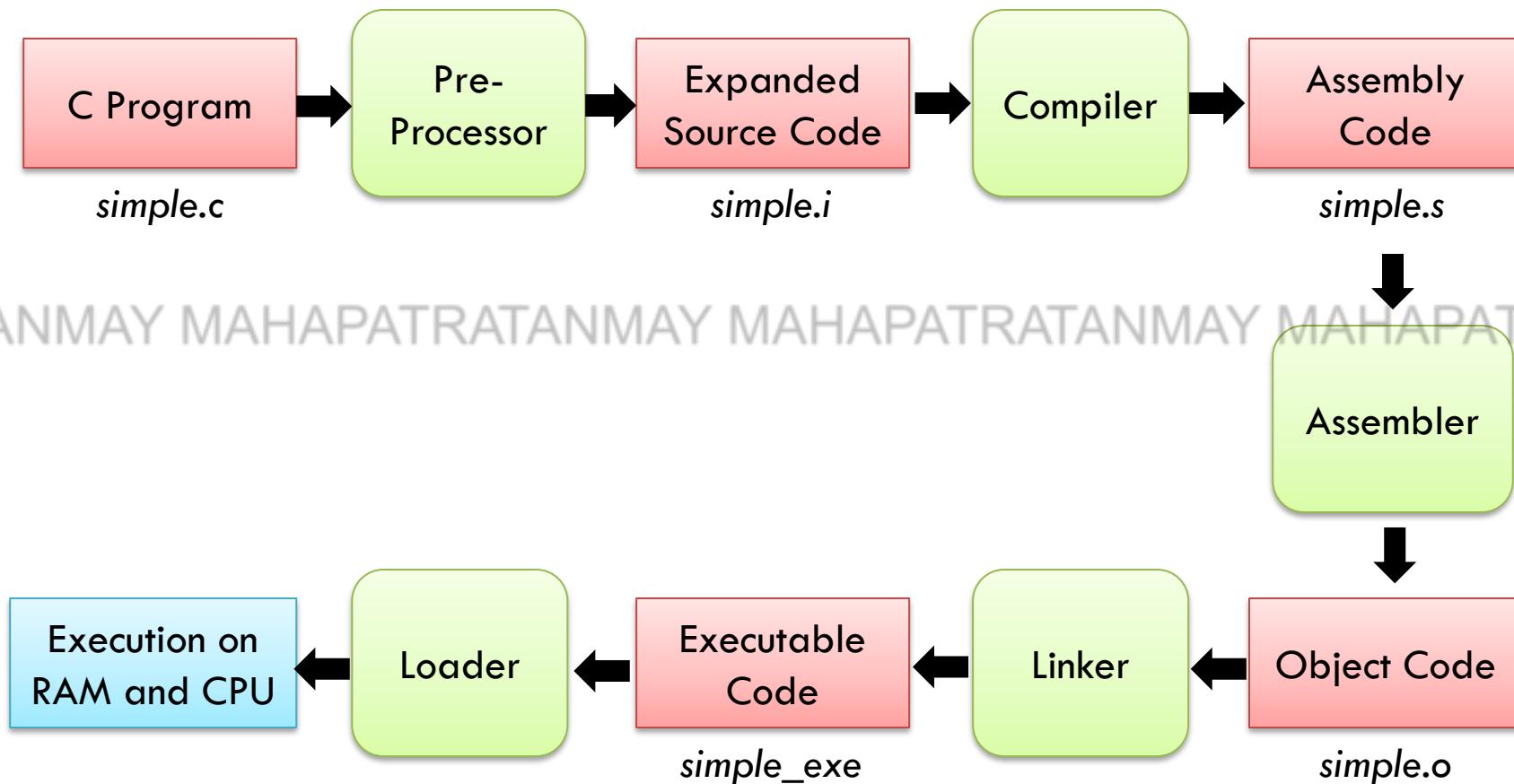
Steps to run a C Program

- 1) Write the program and save it
 - Where does it get saved? ← Disk
- 2) Compile the program to generate its executable
 - Where will the executable be saved? ← Disk
- 3) Run the executable
 - Where will the executable run? ← Executable is loaded into RAM and executed line by line on the CPU

Retrospection of Program execution (in a better way)



The compilation process (Illustrated)



The compilation process

The Preprocessor

- A Text manipulation phase that gives expanded source code to be fed into the compiler
- Removes comments
- Processes lines beginning with # (preprocessor directives)
 - `#include...` or `#define PI 3.142`

The Compiler

- Checks the source code for errors
- Creates object (or machine) code (not ready to run) (why?)

The compilation process (contd.)



The Linker

- Acts on unresolved references left by compiler
- **printf, scanf...** their machine code is added
- Combines multiple object files resultant of compiler phase
- This code is ready to run

The Assembler (optional)

- Some compilers first translate to assembly language.
- The assembler converts it to machine code

Compiling C Program

Say, our C program is stored in **myfirst.c**

To generate executable file, navigate to the directory where **myfirst.c** is stored and run the following command:

```
$ gcc myfirst.c
```

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA

Files generated (executable): **a.out**

The above process is known as **compilation** of the program to generate the executable **a.out**. To run the executable:

```
$ ./a.out
```

Compiling C Program (contd.)

C Compiler allows you to generate intermediate temporary files during its compilation. To generate them use the following command:

```
$ gcc myfirst.c -fverbose-temps -o myfirst.exe
```

Files generated:

myfirst.exe

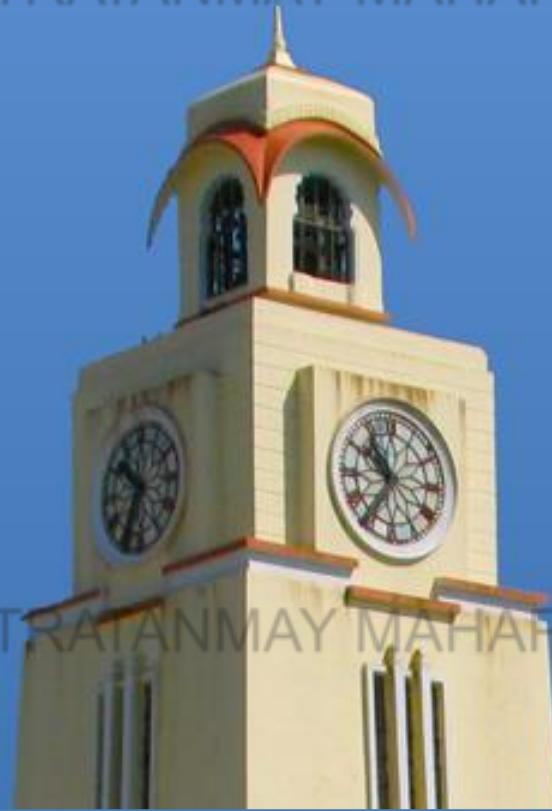
myfirst.o

myfirst.s

myfirst.i

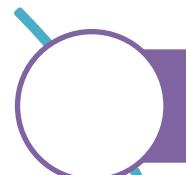
myfirst.c (our original C code)

Exercise: Open each of the above files in a suitable text editor and see what is inside...



Errors in C Programs

Errors in C Programs



Syntax Error



Run-time Error



Logical Error



Semantic Error



Linker Error

Syntax Errors

- Occur when programmers make mistakes in typing the code's syntax correctly
 - Rules of C syntax are not followed
 - Detected in the compiler phase
- Programmers can detect and rectify them easily
- Most common syntax errors:
 - Missing semi-colon (`;`)
 - Missing or open parenthesis (`{}`)
 - Assigning value to a variable without declaring it

Exercise: Find out Syntax Errors in this Program

```
#include <stdio.h>
void main()
{
    int a = 10;
    var = 5;           ← undeclared variable var
    printf("The variable is: %d", var) ← missing semi-colon
    for (int i = 0; ) { ← wrong syntax
        printf("BITSians on ROLL");
    }                   ← missing parenthesis
```

Run-time Errors

- Errors that occur during the execution of a program
- Occur after the program has been compiled successfully
- Identified only when a program is running
- Common run-time errors:
 - Arithmetic errors (e.g., division by zero)
 - calculating square root of -1
 - array index out of bounds
 - infinite loops due to missing terminating conditions
 - memory leaks

Example

```
#include <stdio.h>
void main() {
    int numerator = 10, denominator = 0;
    int result = numerator/denominator;
    //Division by zero

    printf("Result: %d\n", result);
}
```

Output:

Floating point exception (core dumped)

Logical Error

- Code successfully runs without compilation and/or run-time errors
- But output is not as intended
 - There is a logical error

Consequence:

In 1999, NASA lost a spacecraft due to a logical error

Example of Logical Error

```
#include <stdio.h>
void main() {
    // Thruster data in metric units (newton-seconds)
    double metricThrusterData = 1000.0;

    // Navigation software expects thruster data in
    // imperial units (pound-seconds)
    double navigationExpectation = 225.0; // Expected
    // imperial thruster data

    // Error: Using metric thruster data directly in
    // navigation software
    if (metricThrusterData == navigationExpectation) {
        printf("Navigation is on track.\n");
    } else {
        printf("Navigation is off track.\n");
    }
}
```

Example of Logical Error

```
#include <stdio.h>
void main() {
    float a = 10, b = 5;
    if (b = 0) // we wrote = instead of ==
    {
        printf("Division by zero is not possible");
    }
    else
    {
        printf("The output is: %f", a/b);
    }
}
```

Output:

The output is INF

Semantic Error

Errors that occur because the compiler is unable to understand the written code, although the code adheres to the syntax structure.

Example:

```
#include <stdio.h>
void main()
{
    int a, b, c;
    a * b = c; // This will generate a semantic error
}
```

Output:

error: lvalue required as left operand of assignment

Linker Error

Can occur when we try linking multiple files to create an

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA

We will study in one of the lab sheets.

Don't be errored!

Don't worry if you haven't understood everything!

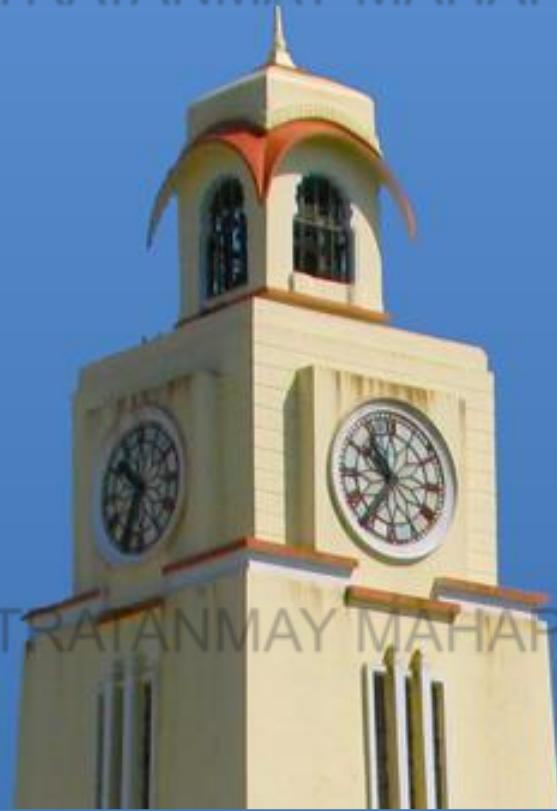
TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA

You will experience all these errors when you start coding.

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA

Write your first code

Write a program in C to compute the average of 3 numbers. Take the numbers as an input by the user.



Thank you
Q & A

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA



BITS Pilani
Pilani Campus

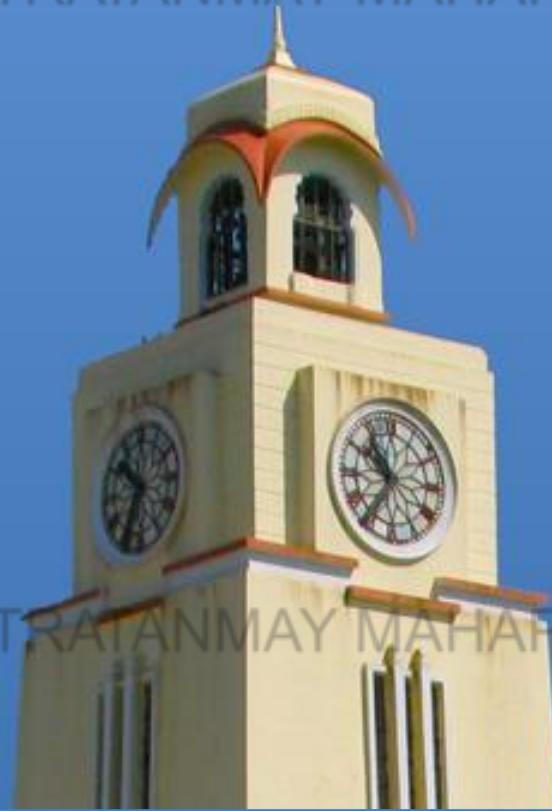
Module 4 – Number System

Department of Computer Science & Information Systems

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA

Module Overview

- **Data in Computers**
- **Binary Numbering System**
- **Binary Arithmetic**
- **IEEE Floating Point Representation**



Data in Computers

Data in Computers

- Computers understand only two things – 0 and 1
 - Data is stored only in the form of combinations of 0s and 1s
- Computers use switches to store 0 or 1
 - Stores 1 if switch is ON
 - Stores 0 if switch is OFF
- Computers consist of Integrated circuits (IC) of billions of switches; allow storage of huge amounts of information.

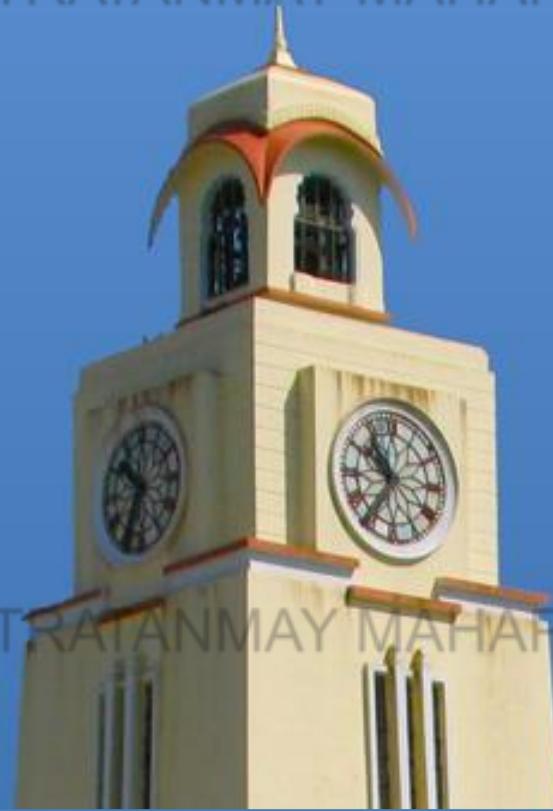
Data in Computers

- Kinds of data we store in computers:

- Integer numbers
 - 1, 3, 94, -103, etc.
- Floating point numbers
 - 1.00433, 54.9090354598, etc.
- Characters
 - 'A', 'a', '#', etc.
- Strings
 - “Delhi”, “Gopal”, etc.
- etc.

We will study more
deeply

All of them are stored as binary patterns, i.e. strings of 0s and 1s.



Binary Numbering System

The *human* numbering system

We use digits

0 to 9 -> 10 symbols (digits) (the decimal system)

What is so special about the number 10?

Nothing!



The numbering system for computers

- Computers use binary numbering system
 - i.e. it has only two symbols to represent numbers – 0 and 1
- Just like humans have 10 symbols - (0 to 9)
- Other systems
 - Octal – 8 symbols
 - 0, 1, 2, 3, 4, 5, 6, 7
 - Hexadecimal – 16 symbols
 - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Decimal vs Binary vs Octal vs Hexadecimal: An example

Decimal	Binary	Octal	Hexadecimal
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

How a decimal number is interpreted?

Eg.: 357

Digits -> 3 5 7

Weights-> 10² 10¹ 10⁰
 MSD LSD

Perform sum over Digits*Weights:

$$3*10^2 + 5*10^1 + 7*10^0
= 357$$

Range of binary numbers

One bit – up to decimal 1

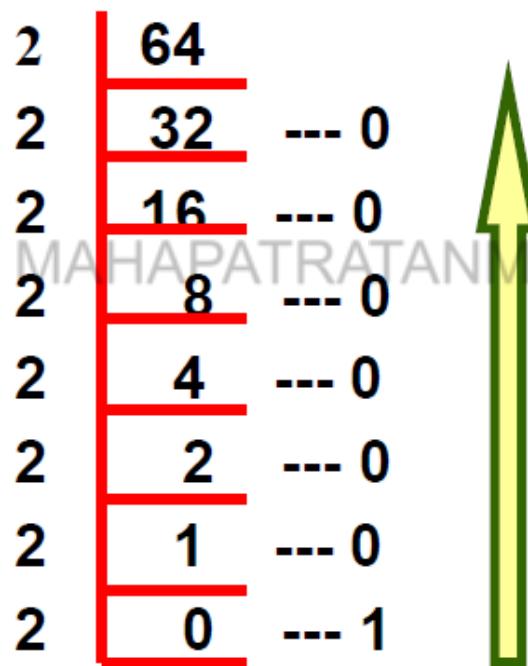
Two bits – up to 3

Three bits – up to 7

In general:

n bits - $2^n - 1$ in decimal

Example



$$(64)_{10} = (1000000)_2$$

Examples (Work out)

- Convert 10101 from binary to decimal
- Convert 16 from decimal to binary
- Convert 23 from decimal to binary

Negative Binary Numbers

- How do we signify negative numbers in arithmetic?
 - The – symbol to the left of MSD
-
- So... 1111 is 15, then -1111 should -15
 - But... computers cannot handle any symbol apart from 0 and 1

Negative Binary Numbers

- To handle negative numbers: **left-most bit is used to indicate the sign.**
- Known by various names: Most Significant Bit (**MSB**), sign-bit or high-order bit
- Positive numbers: **MSB is 0**
- Negative numbers: **MSB is 1**
- For 8 bit unsigned numbers: **range of 0 to 255 ($2^8 - 1$)**
- What about signed numbers?
 - **-127 to 127 or -128 to 127**
- How does the computer know whether to treat a number as **signed or unsigned?**
 - **It cannot.** It's the programmer's job to tell.

Negative Binary Numbers

Three schemes for representing negative binary numbers:

- Signed-magnitude representation
- One's complement representation
- Two's complement representation

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA

- We will discuss them with respect to 8-bit integers

Signed-magnitude representation

MSB is the sign as usual

7 bits for the magnitude or the value

Range: -127 to 127

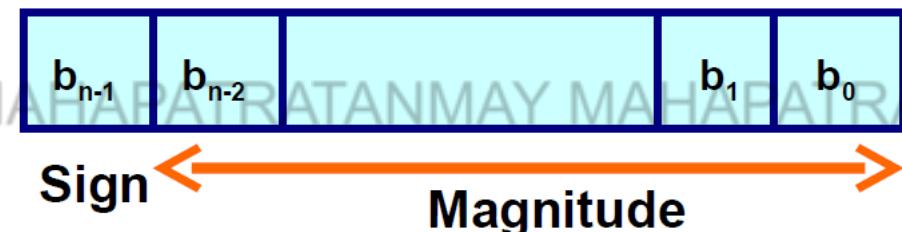
Examples:

109 01101101

-109 11101101

127 01111111

-127 11111111



What about zero?

Signed-magnitude representation

Range of numbers that can be represented:

Maximum :: $+ (2^{n-1} - 1)$

Minimum :: $- (2^{n-1} - 1)$

A problem:

Two different representations of zero.

+0 → 0 000....0

-0 → 1 000....0

1's complement

As before, MSB indicates the sign.

Negative no. = **One's complement** of the positive number

One's complement → Invert all the bits → 1s to 0s and 0s to 1s

Range: -127 to 127

Examples:

15 00001111

-15 11110000

85 01010101

-85 10101010

What about zero?

1's complement (4 bits)

0000 → +0

0001 → +1

0010 → +2

0011 → +3

0100 → +4

0101 → +5

0110 → +6

0111 → +7

1000 → -7

1001 → -6

1010 → -5

1011 → -4

1100 → -3

1101 → -2

1110 → -1

1111 → -0

1's complement

- **Range of numbers that can be represented:**

Maximum :: + $(2^{n-1} - 1)$

Minimum :: - $(2^{n-1} - 1)$

- **A problem:**

Two different representations of zero.

+0 → 0 000....0

-0 → 1 111....1

2's complement

2's complement of a no. = it's 1's complement + 1

Range: -128 to 127

Example: calculating the two's complement representation of -15

Decimal 15 00001111

In one's complement 11110000

Adding 1 +1

In two's complement 11110001

What about zero?

Most widely used!!

2's complement

2's complement of a no. = it's 1's complement + 1

Range: -128 to 127

Example: calculating the two's complement representation of -15

Decimal 15 00001111

In one's complement 11110000

Adding 1 +1

In two's complement 11110001

What about zero?

Most widely used!!

2's complement (4-bits)

0000 → +0

0001 → +1

0010 → +2

0011 → +3

0100 → +4

0101 → +5

0110 → +6

0111 → +7

1000 → -8

1001 → -7

1010 → -6

1011 → -5

1100 → -4

1101 → -3

1110 → -2

1111 → -1

2's complement

- **Range of numbers that can be represented:**

Maximum :: $+ (2^{n-1} - 1)$

Minimum :: $- 2^{n-1}$

- **Advantage:**

- ***Unique representation of zero.***



Binary Arithmetic

Binary arithmetic rules

- Addition of binary bits

- $0 + 0 = 0$
- $0 + 1 = 1$
- $1 + 0 = 1$

- $1 + 1 = 0$ 1 is carry. In binary addition carry is discarded

- Subtraction of binary bits:

- $0 - 0 = 0$
- $1 - 0 = 1$
- $1 - 1 = 0$
- $0 - 1 = 1$ Borrow 1 from next high order digit

Note: for a positive number, its binary representation in SMR, 1CT or 2CT – it is the same number itself

Subtraction using 2 CT

- Subtraction using 2CT:

1. Write the 2CT of the subtrahend
2. Add it to the minuend
3. If there is a carry over discard it, the remaining bits gives the result
4. If there is no carry over, find the 2CT of the result and add –ve sign before it

Subtraction using 2 CT: Example 1

Consider $7 - 4$:

$$1. \quad 2CT(0\ 1\ 0\ 0) = 1\ 0\ 1\ 1 + 0\ 0\ 0\ 1 \rightarrow 1\ 1\ 0\ 0$$

$$2. \quad \begin{array}{r} 0\ 1\ 1\ 1 \\ + 1\ 1\ 0\ 0 \\ \hline \end{array}$$

$$1\ 0\ 0\ 1\ 1 \rightarrow +3$$



Discard

Subtraction using 2CT: Example 2

Consider 4 – 7

1. $2CT(7) = 1000 + 0001 = 1001$

2. 0100

$+ 1001$

\hline
1101

3. $2CT(1101) = 0010 + 0001 \rightarrow 0011$

4. Final answer is -ve of the result obtained i.e., -3

Addition in 2's Complement

- Overflow

- Only possible cases:

$$\begin{array}{r}
 0111 (+7) \\
 0101 (+5) \\
 \hline
 0\ 1100 (-4) \times
 \end{array}$$

Overflow

$$\begin{array}{r}
 1111 (-1) \\
 0111 (+7) \\
 \hline
 1\ 0110 (+6)
 \end{array}$$

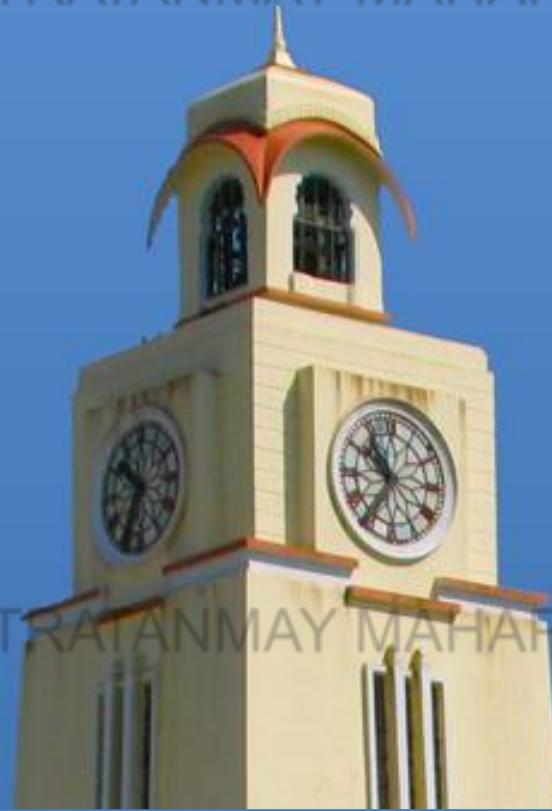
Carry

$$\begin{array}{r}
 1010 (-6) \\
 1001 (-7) \\
 \hline
 1\ 0011 (+3) \times
 \end{array}$$

Overflow

$$\begin{array}{r}
 1110 (-2) \\
 1101 (-3) \\
 \hline
 1\ 1011 (-5)
 \end{array}$$

Carry



IEEE Floating Point Representation

Converting real numbers to binary

- Real numbers: X.Y

- X – Similar to the way integers are represented (Explained in previous slides) X'
- Y
 - $Y' = Y * 2$
 - Store X' and Y = Y'
 - Repeat step 1 and 2, for the new fractional value obtained until $X' \cdot Y' = 1.0$
 - Print all the value of X'.

Example - Convert 10.6875 to binary

$$(10)_{10} = (1010)_2$$

$$(0.6875)_{10} =$$

$$0.6875 * 2 = 1.375 \rightarrow 1$$

$$0.375 * 2 = 0.75 \rightarrow 0$$

$$0.75 * 2 = 1.5 \rightarrow 1$$

$$0.5 * 2 = 1.0 \rightarrow 1$$

$$= (1010.1011)_2$$

Converting binary real numbers to decimal

- Binary number: $X.Y$

- X – Similar to the way integer values are obtained ([Explained in previous slides](#))
 - Y

- Multiply each bit with the weight of its position, where weight of positions are: $2^{-1}, 2^{-2}, 2^{-3}, \dots$
 - Take the sum of the output

▪ E.g. 1 0 1 0 . 1 0 1 1 ←
↓ ↓
Integral part Fractional part

Exercise: Convert this binary floating-point number to decimal system

10.6875

Storing Real number in Computers

Steps to convert a real number in decimal to binary:

1. First, we convert the real number in decimal to a binary real number.
 - E.g. **10.6875** is converted into **1010.1011**
2. Then we normalize the binary real number.
 - E.g. **1010.1011** can be written as **1.0101011 × 2³**
3. Encode the normalized binary real number into **IEEE 754 32-bit or 64-bit floating point format**

Let us study the IEEE 754 Floating Point Representation

IEEE 754 floating point representation

- Three pieces of information are required:
 - Sign of the number (**s**)
 - Significant value (**m**)
 - Signed exponent of 10 (**e**)
- The data type float uses IEEE 32-bit single precision format and the data type double uses IEEE 64-bit double precision format

IEEE 754 floating point representation (contd.)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
s	exponent								significand/mantissa																						
1-bit	8-bits								23-bits																						

Single precision (32 bit)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
s	exponent								significand/mantissa																							
1-bit	11-bits								20-bits																							
significand (continued)																																

Double precision (64 bit)

$$\pm \text{mantissa} * 2^{\text{exponent}}$$

IEEE floating point representation

- In single precision (32 bits):
 - The MSB indicates the sign bit (s): 1 is -ve and 0 is +ve.
 - The next 8 bits: Stores the value of the signed exponent (e)
 - Remaining 23: Stores the significand also known as *mantissa* (m).
 - Value of a floating number IN 32 bits is

$$(-1)^s \times 1.m \times 2^{e-127}$$

- $e - 127$: Excess 127 representation, meaning if the exponent value is 5, it will be stored as $5 + 127 = 132$ in the exponent field

IEEE floating point representation

Exp. Field	Fraction Field	Meaning
000...00	0000...0000	± 0
111...11	Non-Zero	Denormalized $(\pm 0.bbbb * 2^{-126})$
	0000...0000	$\pm \text{infinity}$
	Non-Zero	NaN (Not A Number) - 0/0, 0 $^{*\infty}$, SQRT(-x)

Storing 10.6875 int IEEE 754 32-bit Floating Point Format

10.6875 → 1010.1011 → 1.0101011 × 2³

binary

Normalised binary



s = 0

e = $(3+127)_{10} = (130)_{10} = 10000010$

m = 010101100000000000000000

which is

0 10000010 010101100000000000000000

IEEE 754 32-bit Floating Point representation

Conversion to decimal real number (Work out)



Convert the following number to decimal:

1 00000111 11000000000000000000000000

It is

$$-1.75 \times 2^{(7-127)} = -1.316554 \times 10^{-36}$$

IEEE floating point representation

Why excess 127?

- Helps in natural ordering in the values stored in the exponent field
- Consider 2^{-126} and 2^{+126}
 - Case 1: In absence of Excess 127:
 - $-126 = 10000010$
 - $+126 = 01111110$
 - While -126 is a smaller value the exponent field contains larger value
 $(10000010 > 01111110)$
 - Case 2: In presence of Excess 127:
 - -126 is written as 1 ($1 - 127 = 6 \rightarrow 00000001$)
 - +126 is written as 253 ($253 - 127 = 126 \rightarrow 11111101$)
 - -126 is a smaller value and so is the value stored in the exponent ($00000001 < 11111101$)

Example 1

Convert 12.375 into 32-bit IEEE 754 Floating Point Format

1. $(12)_{10} + (0.375)_{10} = (1100)_2 + (0.011)_2 = (1100.011)_2$
2. Shift $(1100.011)_2$ by 3 digits to normalize the value

$$(12.375)_{10} = (1.100011)_2 * 2^3$$

$$s = 0$$

$$\text{Exponent} = 130 \text{ (represented in excess 127)}$$

$$\text{Mantissa} = 100011$$

$$\text{FP Representation} \rightarrow 0|10000010|1000110000.....$$

Example 2

Convert the following binary number in 32-bit IEEE 754 floating point format into decimal:

1 1011 0110 011 0000 0000 0000 0000 0000

$$(-1)^1 \times 2^{10110110 - 0111111} \times 1.011$$

$$= -1.375 \times 2^{55}$$

$$= -49539595901075456.0$$

$$= -4.9539595901075456 \times 10^{16}$$

Example 3 (Work out)

Convert -10.7 into 32-bit IEEE 754 Floating Point Format

You will see a recurring pattern of bits that is never-ending.

What should you do?

Store only the bits that can fit your 23-bits mantissa. Rest are truncated.

This leads to approximation.

Practice Examples

1. Octal to Decimal

Example 1

$$372_{(8)} = (3*8^2) + (7*8^1) + (2*8^0) = (3*64) + 56+2 = 250_{10}$$

Example 2

$$24.6_8 = (2*8^1) + (4*8^0) + (6*8^{-1}) = 20.75_{10}$$

Examples

2. Decimal to Octal

Example 1: Convert 266_{10} to octal number.

$$\frac{266}{8} = 33 + \text{remainder of } 2 \text{ (LSD)}$$

$$\frac{33}{8} = 4 + \text{remainder of } 1$$

$$\frac{4}{8} = 0 + \text{remainder of } 4$$

$$266_{10} = 4\ 1\ 2_{(8)}$$

Examples

2. Decimal to Octal

Example 2: Convert 0.35_{10} to octal number.

Multiply by 8	Integer	Fraction	coefficient
$0.35*8=$	2	+ 0.80	$a_1 = 2$
$0.8*8 =$	6	+ 0.40	$a_2 = 6$
$0.4*8 =$	3	+ 0.20	$a_3 = 3$
$0.2*8 =$	1	+ 0.60	$a_4 = 1$
$0.6*8 =$	4	+ 0.80	$a_5 = 4$
$(0.35)_{10} = (0.a_1 a_2 a_3 a_4 a_5)_2 = (0.26314)_8$			

0.8 is repeated so stop.

Note: In general, you need to continue until you get 0 in the fraction



Correct order

Examples

3. Hexadecimal to Decimal

Example 1:-Convert $356_{(16)}$ to decimal:

$$356_{(16)} =$$

$$(3*16^2) + (5*16^1) + (6*16^0) = 3*256 + 80 + 6 = 854_{(10)}$$

Example 2:-Convert $2AF_{(16)}$ to decimal:

$$2AF_{(16)} =$$

$$(2*16^2) + (10*16^1) + (15*16^0) = 512+160+15 = 687_{(10)}$$

Examples

4. Decimal to Hexadecimal

Example 1: Convert 423_{10} to hex number.

$$\frac{423}{16} = 26 + \text{remainder of 7 (LSD)}$$

$$\frac{26}{16} = 1 + \text{remainder of 10}$$

$$\frac{1}{16} = 0 + \text{remainder of 1}$$

$$423_{10} = 1 A 7_{(16)}$$

Examples

5. Hexadecimal to Binary

Example 1: Convert $9F2_{(16)}$ to its binary equivalent

9	F	2
↓	↓	↓

1001 1111 0010

$$9F2_{(16)} = 100111110010_{(2)}$$

Example 2: Convert $BA6_{(16)}$ to binary equivalent

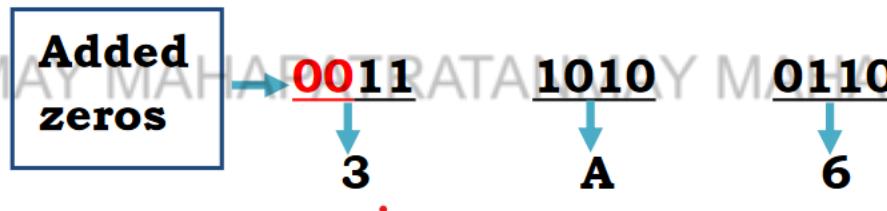
$$BA6_{(16)} = (\underline{1011} \ \underline{1010} \ \underline{0110})_2$$

Examples

6. Binary to Hexadecimal

Example 1: Convert $1110100110_{(2)}$ to hexa equivalent

Solution:



$$1110100110_{(2)} = 3A6_{(16)}$$

Example 2: Convert 101011111_2 to hexa equivalent

Solution:

$$\underline{1} \ \underline{0101} \ \underline{1111}_2 = 15F_{(16)}$$

Examples

7. Octal to Binary

Example 1: Convert $472_{(8)}$ to binary number

Solution:

$$\begin{array}{ccc} 4 & 7 & 2 \\ \downarrow & \downarrow & \downarrow \\ 100 & 111 & 010 \end{array}$$

$$472_{(8)} = 100111010_{(2)}$$

Example 2: Convert $5431_{(8)}$ to binary number

Solution:

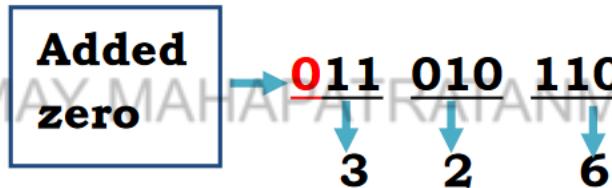
$$5431_{(8)} = \underline{101} \ \underline{100} \ \underline{011} \ \underline{001} = 101100011001_{(2)}$$

Examples

8. Binary to Octal

Example 1: Convert $11010110_{(2)}$ to octal equivalent

Solution:



$$11010110_{(2)} = 326_{(8)}$$

Note:

Zero was placed to the left of the MSB to produce groups of 3 bits.



Thank you
Q & A

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA



Module 5 – Program Data and Operators

BITS Pilani

Pilani Campus

Department of Computer Science & Information Systems

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA

Module Overview

- **Building blocks of a program**
- **Program Data, Constants and Variables**
- **Data Types**
- **Type Conversions**
- **Operators and Expressions**



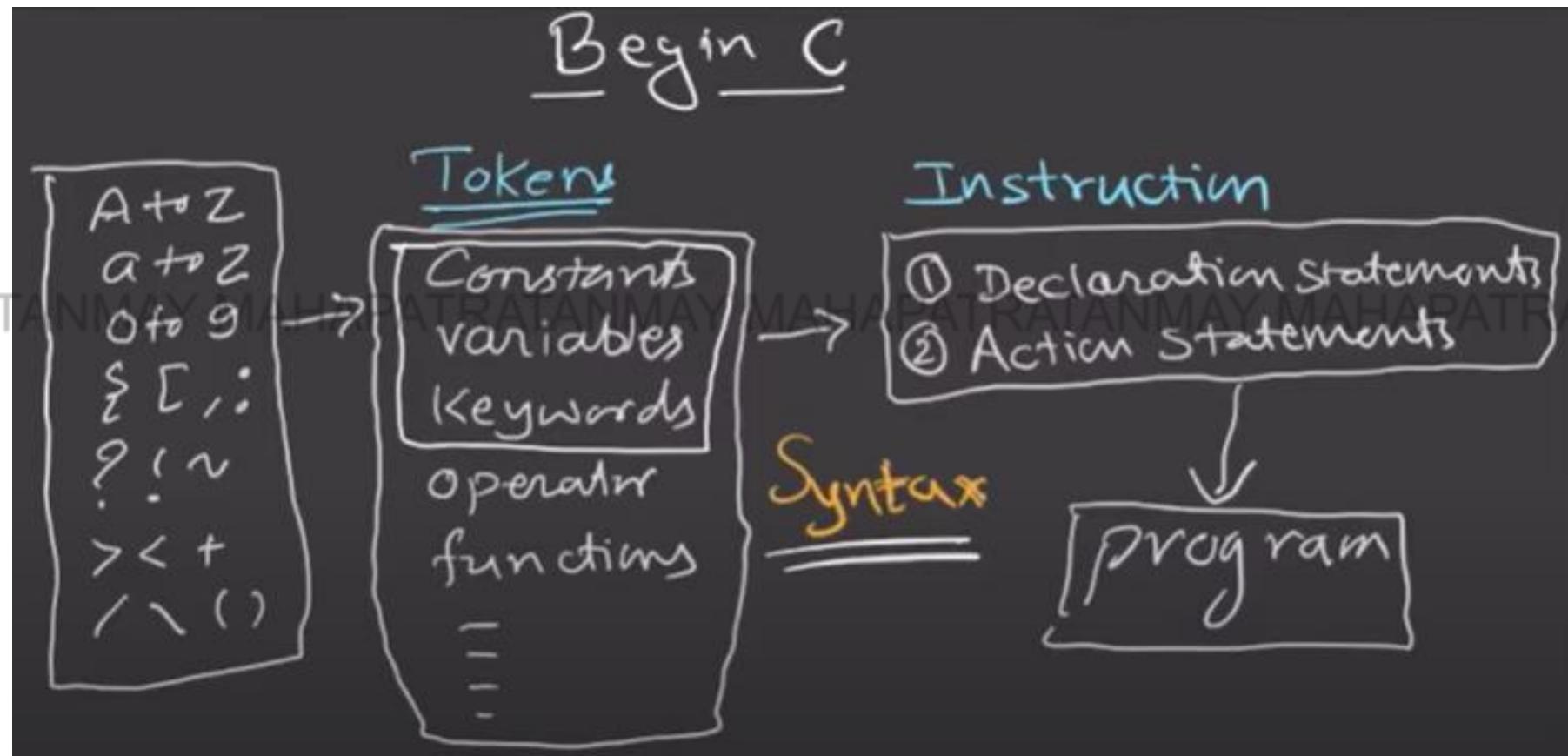
Building blocks of a program

What is a language made up of?



- Letters/digits/alphabets/symbols:
A-Z, a-z, 0-9, {}, [], (), ; ...
- Words/**tokens**:
Keywords, Constants, Identifiers, String literals, Operators, Comments,..
- Sentences/**Instructions/Statements**:
 1. *Declaration Statements*
 2. *Action Statements*
- Grammar/**Syntax**
Punctuation,...

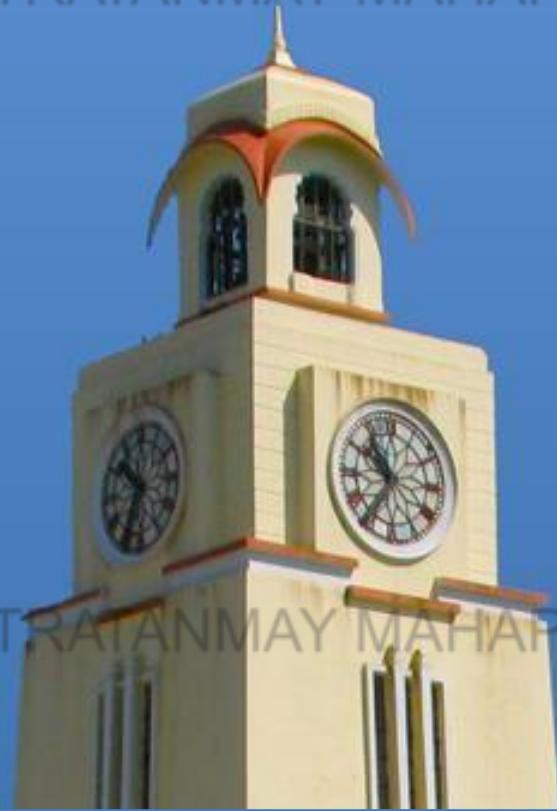
Constructing a program



Keywords

- Reserved words with some predefined meaning

auto	float	signed	_Alignas (since C11)
break	for	sizeof	_Alignof (since C11)
case	goto	static	Atomic (since C11)
char	if	struct	Bool (since C99)
const	inline (since C99)	switch	Complex (since C99)
continue	int	typedef	Decimal128 (since C23)
default	long	union	Decimal32 (since C23)
do	register	unsigned	Decimal64 (since C23)
double	restrict (since C99)	void	Generic (since C11)
else	return	volatile	Imaginary (since C99)
enum	short	while	Noreturn (since C11)
extern			Static_assert (since C11)
			Thread_local (since C11)



Program Data, Constants and Variables

Program Data

Programs deal with data!

Examples:

- Name
- Weight
- Quantity
- Price

Program Data

Program data occurs in the form of

- *Variables*
- *Constants*

Constants

Primary constants

- *Integer:* 25, -5, 42, 0x102F (*Hexadecimal*), 0703 (*Octal*), 0b10110001 (*binary*)
- *Real:* 3.7, 3.0, -0.423
- *Character:* 'a', 'Z', '+', ' ', '6', '54', '5.67', '-7'

Secondary constants

- *Arrays*
- *String literals:* "PILANI"
- *Pointers,...*

Variables

- A name given to a memory location
- The name used to access a variable is also known as an identifier
- C has strict rules for variable naming
- Variable name can begin with a letter or underscore, and comprise of letters, digits or underscore. Names are **case-sensitive**
- *No other symbol is allowed in the variable name.*
- e.g., *max2, _store, Title, ... 2max, Store\$,*
- Keywords cannot be used as variable names (e.g., *continue, if, short, union, return ...*)
- Use meaningful names!

Variable declaration and initialization

- A variable **must** be declared before its use

Examples:

```
int max; /* declares a variable max that  
can store integer values */  
  
int age, quantity, price;  
/* declares three variables that can  
store integer values */
```

- To initialize a variable, use =

Example:

```
age = 21;
```

- Declaration and initialization can be combined in one statement

Example:

```
int age = 4, quantity = 500, price = 999;
```

- An uninitialized variable is simply *junk* (in general)

More on variables...

```
int octalNum = 075; //Octal  
int hexNum = 0x1A; //Hexadecimal  
int binaryNum = 0b1101; //Binary
```

```
int result_oct = octalNum + 10;  
int result_hex = hexNum * 2;  
int result_bin = binaryNum + 5;  
printf(" result_oct: %d, result_hex: %d,  
    result_bin: %d\n", result_oct,  
    result_hex, result_bin);  
// Output values: 71, 52, 18
```

Variables in Main Memory

- Consider the following C Program:

```
#include <stdio.h>
int main()
{
    int num1, num2, num3;
    num1 = 2;
    num2 = 4;
    num3 = num1 + num2; // computing the sum of num1 and num2
    printf("The sum is: %d \n", num3); // printing the sum
    return 0;
}
```

- This program has three variables: `num1`, `num2` and `num3`.
- Where are these variables stored?*

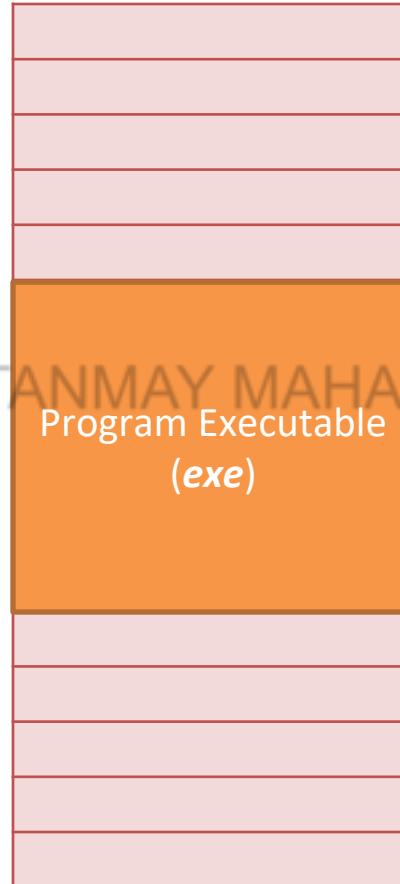
Remember this block diagram!

We are going to see this block diagram again and again!

A line from *exe*

CPU

Program is executed on CPU line by line

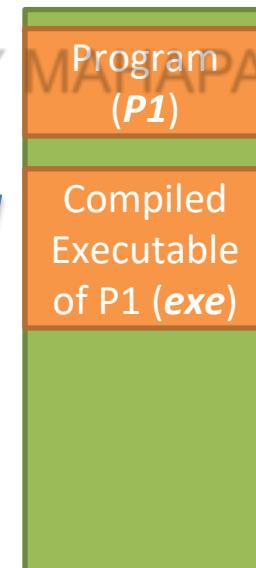


Memory (RAM)

- OS loads the program executable into the RAM
- Executes it line by line on CPU

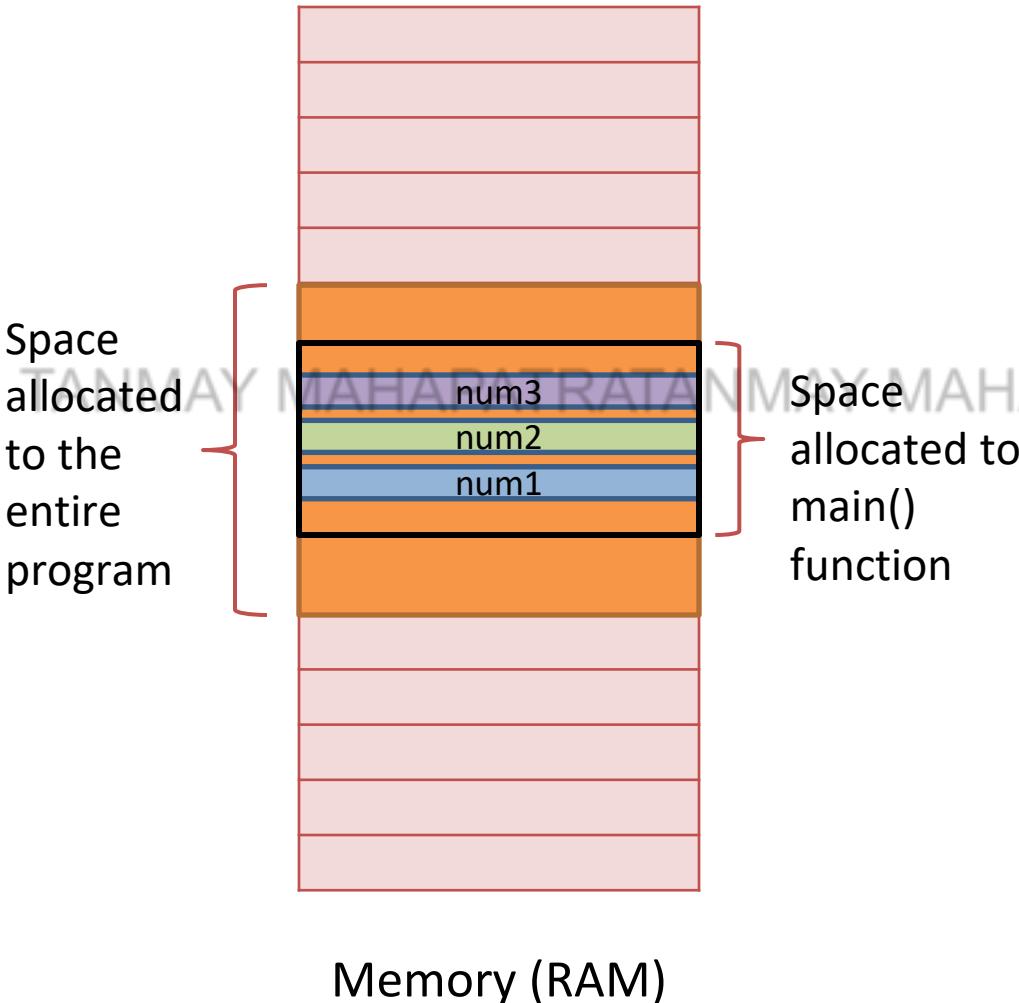


Program Executable gets loaded into the RAM



DISK

Look at the main memory only



- When we compile and execute a program, OS allocates some space in the main memory
- The declared variables are stored in that allocated space.
- In our example, `num1`, `num2` and `num3` are stored in this space.
- More specifically in the space allocated to the `main()` function, within the above space.
 - We will study about memory allocation and functions in greater detail later!



Data Types

Data types

- C is a *typed* language
- Every data item has a type associated with it.
- Examples of declaring variables:

- int num1;
- short num2;
- long num3;
- float cgpa;
- double percentage;
- long double pqr;
- char c1;

Fundamental data types in C

- Integer (`short`, `int`, `long`, `long long`)
- Floating point (`float`, `double`, `long double`)
- Character (`char`)
- Boolean data type (`bool` – *C99 onwards provided stdbool.h is included*)
- Fixed size of each data type / sub-type.

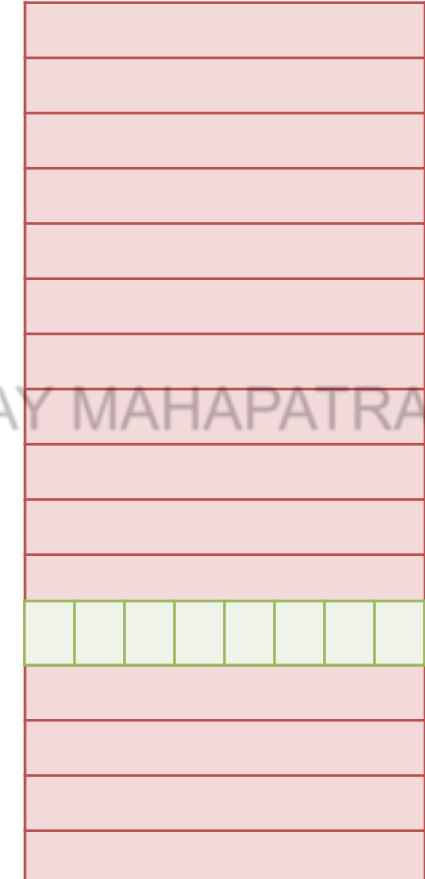
Machine Readable size of variables

- Every variable occupies space in the program allocated space of the main memory
- *How much space does each variable occupy?*
- It depends on its type!
- For example, variable of type `int` occupies either **2 or 4 bytes** depending upon the specific compiler you are using.
- *What is a **byte**?*
- *Before we answer this question, let us see how the main memory is organized!*

Organization of Main Memory

- Main memory is typically a table of memory locations of 8 bits (0/1) each
- A set of contiguous 8 bits is known as a byte
- So, each location in this main memory is of one byte
 - We call byte-addressable memory
- Each location is associated with an address.

1 byte

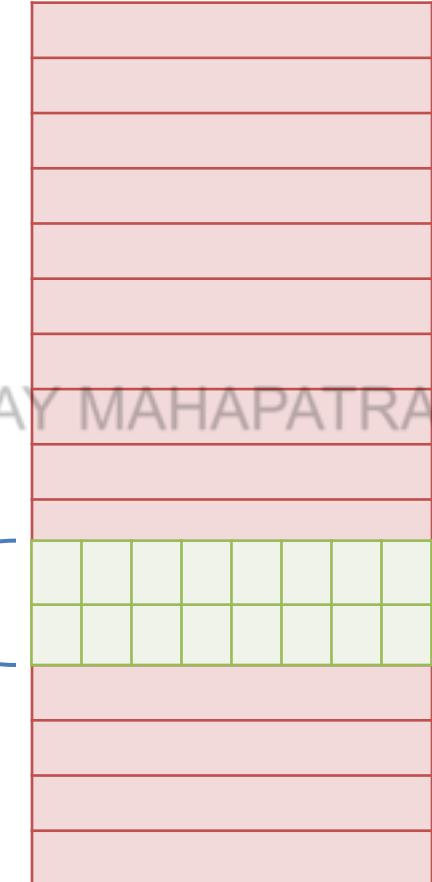


Each
location is of
8 bits

Storing int in main memory

- Remember how integer values are represented in 2's complement representation.
- Our machines use 2's complement representation to store integer variables.
- If int variables are of 2 bytes size (or 16 bits), then each int variable shall occupies 2 contiguous locations in the memory

Space occupied by int variable, which is 2 bytes or 16 bits



Types of integer variables

Type / Subtype	Minimum size (bytes)	Range	Format Specifier
short	2	-32,768 to 32,767 (-2¹⁵ to 2¹⁵-1)	%hd
unsigned short	2	0 to 65,535 (0 to 2¹⁶-1)	%hu
int	2 or 4	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647	%d
unsigned int	2 or 4	0 to 65,535 or 0 to 4,294,967,295	%u
long	4 or 8	-2,147,483,648 to 2,147,483,647 or -9223372036854775808 to 9223372036854775807	%ld
unsigned long	4 or 8	0 to 4,294,967,295 or 0 to 18446744073709551615	%lu
long long	8	-9223372036854775808 to 9223372036854775807	%lld
unsigned long long	8	0 to 18446744073709551615	%llu

Format Specifiers

- Placeholders within format strings to indicate type and format of data to be printed and read
- Help printf and scanf functions to interpret and display data correctly

- Integers (`%hd`, `%hu`, `%u`, `%d`, `%ld`, `%lld`, `%i..`)
- Floating-point numbers (`%f`, `%e`, `%lf`, `%Lf`)
- Characters and strings (`%c`, `%s`)
- Others (Octal: `%o`, hexadecimal: `%x`, pointers: `%p`)

Types of floating point numbers

While integers classified by size, floating point numbers are classified by *precision*

Type / Subtype	min precision (digits)	min size (bytes)	mantissa-exponent	Format Specifier
float	6	4	23-8	%f, %e
double	10	8	52-11	%lf
long double	10	8-16	112-15	%Lf

Input/Output formatting

Width

- A number after % specifies the minimum field width to be printed

```
int num = 42;
```

```
printf("%5d\n", num); // Prints " 42" with leading spaces to ensure width of 5
```

Precision (*this precision is different from that shown in previous slide*)

- Typically used with floating-point numbers to control the number of decimal places displayed
- A period(.) symbol separates field width with precision.

```
float pi = 3.141592;
```

```
printf("%5.2f\n", pi);
```

```
tejasv@LAPTOP-6IBMJSH8:/mnt/d/WSL$ ./a.out  
3.14
```

Sign

- A minus(-) sign tells left alignment

```
float pi = 3.141592;
```

```
printf("%-5.2f\n", pi); Code
```

```
tejasv@LAPTOP-6IBMJSH8:/mnt/d/WSL$ ./a.out  
3.14
```

Questions for you

What happens when you store a very large value in an ‘int’?

What happens when you use a wrong format specifier?

What happens when you store a large ‘double’ value in a ‘float’?

What is the size of ‘long long’ on your machine?

Character type

- If everything is in binary, how do we deal with characters?
- Use Character-encoding schemes, such as **ASCII**
- ASCII defines an encoding for representing English characters as numbers, **with each letter assigned a number from 0 to 127.**
- 8-bit ASCII used, so char data type is one byte wide

ASCII Encoding

innovate

achieve

lead

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0 000	000	NUL (null)	32	20	040	 	Space	64	40	100	@	Ø	96	60	140	`	`
1	1 001	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2 002	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3 003	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4 004	004	EOT (end of transmission)	36	24	044	$	\$	68	44	104	D	D	100	64	144	d	d
5	5 005	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6 006	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7 007	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8 010	010	BS (backspace)	40	28	050	((72	48	110	H	H	104	68	150	h	h
9	9 011	011	TAB (horizontal tab)	41	29	051))	73	49	111	I	I	105	69	151	i	i
10	A 012	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B 013	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C 014	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D 015	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E 016	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F 017	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10 020	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11 021	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12 022	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13 023	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14 024	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15 025	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16 026	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17 027	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18 030	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19 031	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A 032	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B 033	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[[123	7B	173	{	{
28	1C 034	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174	|	
29	1D 035	035	GS (group separator)	61	3D	075	=	=	93	5D	135]]	125	7D	175	}	}
30	1E 036	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	X	126	7E	176	~	x
31	1F 037	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	DEL	127	7F	177		DEL

Computations with char

```
char ch1 = 'A', ch2;  
printf("%c\n", ch1);  
ch2 = ch1 + 1;  
printf("%c\n", ch2);
```

```
char ch3 = 67;  
printf("%c\n", ch3);  
printf("%d\n", ch3);
```

Escape Sequences

- Escape sequences – backslash followed by the lowercase letter
 - Examples: `\n`, `\t`, etc.
 - `printf("number: %d\t", num);`

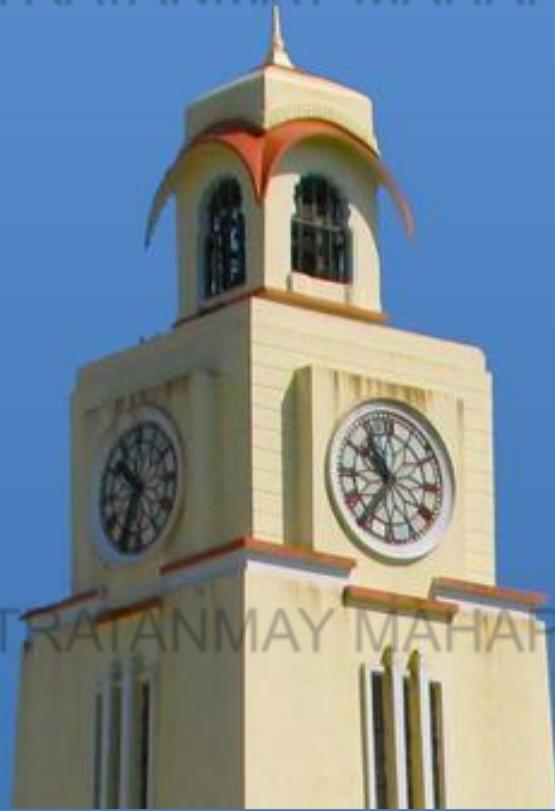
- Printing special characters like `'`, `"`, `\`, etc.
 - use a preceding `\`
 - `printf("\'Hello\'");`

Chars and Strings

- "*Internet*" is a string, which is basically a bunch of characters put together
- A string in C is an *array* of chars.

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA

- *We will study arrays and strings in greater detail later!*
- Note the use of double quotes
- What is the distinction between 'A', "A", 'Axe' and "Axe"??



Revisiting Constants

Declaring Constants

2 ways:

1. Using `const` qualifier

```
const float pi = 3.14f; //pi is read-only
```

2. Symbolic constants using `#define`

```
#define PI 3.14f //no semi-colon used
```

What is the difference between these two?

Declaring Constants

- C specifies a default type for a constant.
- We can also force a type.
- Default for an integer is int. If a constant doesn't fit in int, the next higher type is tried.
- Suffixes can be used to coerce the data type.
- U or u for unsigned. L or l for long....

e.g., `23801u, 1234567890l`

- For a floating point number, the default is double.
- Suffix F or f demotes it to float; L or l promotes it to long double.

e.g., `3.14159f, 123.4567890123456789L`

- A character constant is stored as an int (!!)

Declaring Constants

- Using `#define`
- Syntax: `#define variable_name value`
- Note: no use of semicolon at the end
- Example:

```
#include<stdio.h>
#define val 10 //always written before int main()
int main() {
    int A = val;
    val = val+10; //error: lvalue required as left operand of
    assignment
    return 0;
}
```

Constants vs Variables

Constants	Variables
A constant does not change its value over time.	A variable, on the other hand, changes its value dependent on the equation.
Value once assigned can't be altered by the program.	Values can be altered by the program.



Type Conversions

Type Conversions

Implicit

- If either operand is **long double**, convert the other into **long double**.
- Otherwise, if either operand is **double**, convert the other into **double**.
- Otherwise, if either operand is **float**, convert the other into **float**.
- Otherwise, convert **char** and **short** to **int**.
- Then, if either operand is **long**, convert the other to **long**.

Explicit (also known as **coercion** or **typecasting**)

Implicit Type Conversion

e.g.,

```
float A = 100/3; output: 33.000000
```

```
float A = 100/3.0 output: 33.333333
```

```
int A = 10;
```

```
A = A + 'B'; Output: 76
```

‘B’ is type converted to integer

```
A = 'A' + 'B'; Output ?
```

```
float X = 0.2; double A = X/0.1; Output ?
```

Explicit Type Conversion (Typecasting)

Example1:

Conversion of integer to a float variable

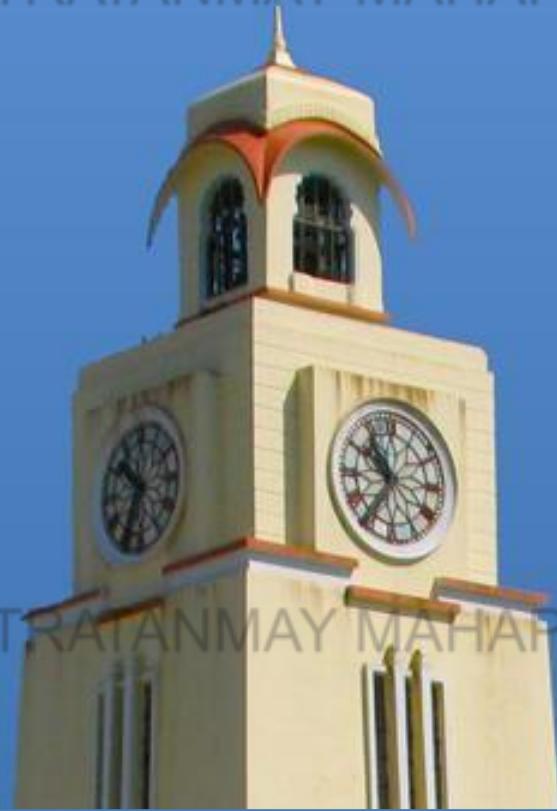
```
int a = 10;  
float f = (float) a;
```

Example2:

Conversion of integer to a char variable

```
int a = 20;  
char ch = (char) a;
```

Note: The above conversion is valid as after all characters are integer values of ASCII codes.



Operators and Expressions

Celsius to Fahrenheit Program



```
#include <stdio.h>
int main()
{
    float cel, far;          /* variable declarations */

    printf("Enter the temperature in deg. Celsius: ");
    scanf("%f", &cel);        /* getting user input */

    far = cel * 1.8 + 32;

    printf("%f degree C = %f degree F\n\n", cel, far);
                           /* printing the output */

    return 0;
}
```

Operators

- Can be unary, binary or ternary
- Used to perform some operation (e.g., arithmetic, logical, bitwise, assignment) on operands
- In an expression with multiple operators, the order of evaluation of operators is based on the **precedence** level
- Operators with the same precedence work by rules of **associativity**
- C does not define the order in which the operands of an operator will be evaluated

Operator	Description	Associativity
()	Parentheses (grouping)	left-to-right
[]	Brackets (array subscript)	
.	Member selection via object name	
->	Member selection via pointer	
++ --	Unary post-increment/post-decrement	
++ --	Unary pre-increment/pre-decrement	right-to-left
+ -	Unary plus/minus	
! ~	Unary logical negation/bitwise complement	
(type)	Unary cast (change type)	
*	Dereference	
&	Address	
sizeof	Determine size in bytes	
* / %	Multiplication/division/modulus	left-to-right
+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <=	Relational less than/less than or equal to	left-to-right
> >=	Relational greater than/greater than or equal to	
== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
	Logical OR	left-to-right
:?	Ternary conditional	right-to-left
=	Assignment	right-to-left
+= -=	Addition/subtraction assignment	
*= /=	Multiplication/division assignment	
%= &=	Modulus/bitwise AND assignment	
^= =	Bitwise exclusive/inclusive OR assignment	
<<= >>=	Bitwise shift left/right assignment	
,	Comma (separate expressions)	left-to-right

Types of Operators

- Unary operators
- Arithmetic operators
- Bitwise operators
- Relational operators
- Logical operators
- Assignment operators
- Conditional operators

Unary Operators

- Performs operation on single operand
 - `+, -, ++` (Increment operator) and `--` (Decrement operator), `sizeof()`

Increment and Decrement Operators

Increment (++) and Decrement (--) operators

- Both these operator can be applied before an operand as well as after the operand
- Can be used either as prefix or postfix
 - Prefix:
 - “Change value and then use”
 - Lower precedence than the postfix
 - Postfix:
 - “Use and then change value”
 - Higher precedence than prefix
 - Can be applied only to variables
 - Causes **side effects**

Increment and Decrement Operators



Prefix:

`++<variable_name> / --<variable_name>`

Example:

```
int A = 10;  
printf("A is %d ", ++A);  
int B = --A;  
printf("B is %d ", B);
```

First, the value is increased/decreased and then used

Output: A is 11 B is 10

Increment and Decrement Operators



Postfix:

<variable_name>++ / <variable_name>--

Example:

```
int A = 10;  
printf("A is %d", A++);  
int B = A--;  
printf("B is %d", B);
```

First, the value is used and then increased (or decreased)

Output: A is 10 B is 11

Knowing the size of data types

- How do we know the size of a data type for our C compiler?
- Use `sizeof()` operator
- Example:

```
printf("Size of int is %lu bytes", sizeof(int));  
// prints size of int in bytes. The format specifier is  
unsigned long int (%lu) on gcc.
```

- Note: By default, integer data type is *signed*.

Arithmetic Operators

- Used for performing arithmetic operations
- Follow left to right associativity
- Binary arithmetic operators
 - Takes two operands as input
 - `*`, `/`, `%`, `+` and `-`
 - `*`, `/`, `%` have higher precedence than `+` and `-`

Arithmetic Operators

- int a = 31, b = 10;
- floats c = 31.0, d = 10.0;

For integers:

$$\begin{aligned}a + b &= 41 \\a - b &= 21 \\a/b &= 3 \\a\%b &= 1 \\a * b &= 310\end{aligned}$$

For float:

$$\begin{aligned}c + d &= 41.000000 \\a - b &= 21.000000 \\c/d &= 3.100000 \\c\%d &= \text{Not valid} \\c * d &= 310.000000\end{aligned}$$

Bitwise Operators

- Performs operation at bit level using:
 $\&$, $|$, $^$, \sim , $>>$, $<<$
- $a | b$ Known as bitwise OR. Sets the bit when the bits in at least one of the operand is set
- A = 43 and B = 7
- $A | B = \begin{array}{r} 00101011(43) \\ 00000111(7) \\ \hline = 00101111(47) \end{array}$

Bitwise Operators

- $a \& b$ Known as bitwise AND. Sets the bit only when the bits in both the operands are set
- $A = 43$ and $B = 7$

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA

$$\begin{array}{r} A \& B = \quad 0010\ 1011(43) \\ \quad \quad \quad 0000\ 0111(7) \\ = \quad \quad \quad \hline 0000\ 0011(3) \end{array}$$

Bitwise Operators

- $a \wedge b$ Known as bitwise XOR. Sets the bit only when the bit is set in only one of the operand
- A = 43 and B = 7

- $A \wedge B =$
$$\begin{array}{r} 0010\ 1011(43) \\ 0000\ 0111(7) \\ \hline =\quad 0010\ 1100(44) \end{array}$$

Bitwise Operators

- $a \ll x$ ↗ Shift it left by x bits.
- $a \gg y$ ↗ Shift it right by y bits.

- $A = 7 \quad 0000\ 0111 (7)$

$A \ll 2 \quad 0001\ 1100 (28)$

- $A = 7 \quad 11000001 (-63)$

$A \gg 4 \quad 00001100 (12)$

Bitwise Operators

- $\sim a$? Flip the bits. Sets the bit only when the bit in the operand is not set.

- $A = 43 = \begin{array}{r} 00101011 \\ (43) \end{array}$

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA

- $A = -63 = \begin{array}{r} 11000001 \\ (-63) \end{array}$

$\sim A = \begin{array}{r} 00111110 \\ (62) \end{array}$

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA

Relational Operators

- `a == b` ? checks whether a is equals to b
- `a != b` ? checks whether a is not equal to b
- `a < b` ? checks whether a is less than b
- `a > b` ? checks whether a is greater than b
- `a <= b` ? checks whether a is less than equal to b
- `a >= b` ? checks whether a is greater than equal to b
- All are of **same precedence** and are **left to right** associative

Logical Operators

- `expr1 && expr2` ↗ Returns true, when both operands are non-zero
- `expr1 || expr2` ↗ Returns true, when at least one of the expression is non-zero
- `! (expr)` ↗ If `(expr)` is non zero, then `! (expr)` returns zero
- All are of same precedence and are **left to right** associative

Example

```
int a = 10, b = 4, c = 10, d = 20;
```

Evaluate the expression: $(a > b \&\& c == d)$

Evaluate the expression: $(a > b || c == d)$

Evaluate the expression: $(!a)$

Assignment Operator

- $A = B$ // assigns value of B to the variable A
- $A \text{ op} = B$? $A = A \text{ op } B$, op can be any binary arithmetic or bitwise operator

E.g., $A = 43, B = 7$

$A += B$? $A = A + B$ O/P: $A = 50, B = 7$

$A \&= B$? $A = A \& B$ O/P: $A = 3, B = 7$

Important Considerations

- C does not specify the order in which the operands of an operator are evaluated.
- Exceptions: `&&` `||` `?:` ,
- The operands of **logical-AND** and **logical-OR** expressions are **evaluated from left to right**. If the value of the first operand is sufficient to determine the result of the operation, the second operand is not evaluated. This is called "*short-circuit evaluation*".
- If conditional operator also has order of evaluation:
 - Example: `int i=(a<10)?10:20;`

Short Circuiting in C

- A short circuit in logic is when it is known for sure that an entire complex conditional is either true or false before the evaluation of whole expression
- Mainly seen in case of expressions having logical AND(`&&`) or OR(`||`)

Short Circuiting in Logical AND

Consider the expression: E1 && E2

- Output of the expression is true only if both E1 and E2 are non-zero
- If E1 is false, E2 never gets evaluated

```
a = 0, b = 3;  
int I = ++a && ++b;  
printf("%d %d %d", a, b, I);  
O/P 1, 4, 1
```

```
a = 0, b = 3;  
int I = a++ && ++b;  
printf("%d %d %d", a, b, I);  
O/P 1, 3, 0
```

Short Circuiting in Logical OR

Consider the expression: E1 || E2

Output of the expression is false if only both E1 and E2 are zero

If E1 is true, there is no need to evaluate E2

```
a = 0, b = 3;
```

```
int I = ++b || ++a;
```

```
printf("%d %d %d", a, b, I);
```

O/P 0, 4, 1

```
a = 0, b = 3;
```

```
int I = ++a || ++b;
```

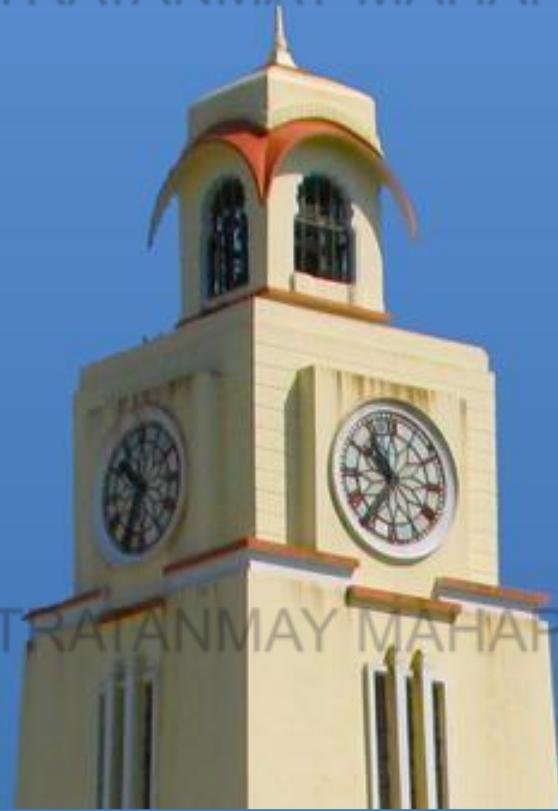
```
printf("%d %d %d", a, b, I);
```

O/P 1, 3, 1

Problems to Solve

```
int a = 1, b = 1;  
int c = a || --b; // --b is not evaluated  
int d = a-- && --b; // --b is evaluated, b becomes 0  
printf("a = %d, b = %d, c = %d, d = %d", a, b, c, d);  
O/p : a = 0, b = 0, c = 1, d = 0
```

```
int i=-1, j=-1, k=0, l=2, m;  
m = i++ && j++ && k++ || l++;  
printf("i = %d, j = %d, k = %d, l = %d, m =  
%d", i, j, k, l, m);  
O/p : i = 0, j = 0, k = 1, l = 3, m = 1
```



Thank you
Q & A

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA



Module 6 – Control Flow: Branching and Looping – Part 2



BITS Pilani

Pilani Campus

Department of Computer Science & Information Systems

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA

Module Overview

- **Control Flow: Looping**

- **while Loop**

- **do... while Loop**

- **for Loop**



Control Flow: Looping/Iteration

Looping/Iteration – basics

- Many activities repetitive in nature
- Examples:
 - *Decimal-to-binary* ↳ repeated division
 - *Computing an average* ↳ repeated addition
- We need a mechanism to perform such repetitive tasks
- The repetition framework should also include the halting mechanism

Loops in C support repetitive tasks

Loops in C

while loop

do...while loop

for loop



while loop

while loop

```
while (exp is true)
    statement;
```

OR

```
while (exp is true)
{
    statement1;
    statement2;
    ...
}
```

Control expressions: Examples



- `while (answer == 'Y')`
- `while (count)`
- `while (5)`
- `while (base>0 && power>=0)`
- `while (--num > 0)`
- `while (printf("Enter the value of N: "))`

Example

```
int var=1;  
while (var <=10)  
{  
    printf("%d \n", var++);  
}  
printf("While loop over");
```

This program prints first 10 positive integers

A null body/statement

```
while (num > 0);
```

is different from

```
while (num > 0)
{
    // Empty block: no statements here
}
```

Exercises

Write a program to calculate the sum of 10 numbers entered by the user

Nested while loops

```
while(exp1) {           int i = 10, j, count;  
...  
    while(exp2) {         j = 5;  
        ...  
        while(j) {  
            printf("Hello!\n");  
            j--;  
        }  
        i--;  
    }  
}
```

How many times is “Hello” printed?

Examples

```
int i = 10, j, count=0;  
while(i) {  
    j = 5;  
    while(j) {  
        TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA  
        printf("%d",  
        count++);  
        j--;  
    }  
    i--;  
}
```

```
int i = 10, j, count=0;  
while(i) {  
    j = 5;  
    while(j) {  
        TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA  
        count++;  
        j--;  
    }  
    i--;  
}  
printf("%d ", count);
```

Exercises (Code)

Print this pattern on your screen using loops. No. of rows are user input.

```
int i, j, rows;
printf("Enter number of rows: ");
scanf("%d",&rows);
i =1;
while(i<=rows) {
    j =1;
    while(j<=i) {
        printf("* ");
        j++;
    }
    printf("\n");
    i++;
}
```

Exercises

Print this pattern on your screen using loops.

No. of rows are user input.

```
int i, space, rows, k=0;
printf("Enter number of
rows: ");
scanf("%d", &rows);
i=1;
while(i<=rows) {
    space=1;
    while(space<=rows-i) {
        printf("  ");
        space++;
    }
    k=0;
```

```
while(k != 2*i-1) {
    printf("*  ");
    ++k;
}
printf("\n");
i++;
k=0;
```

*

* * *

* * * * *

* * * * * * *

* * * * * * * *

Practice tasks

* * * * *

* * * * *

* * * *

* *

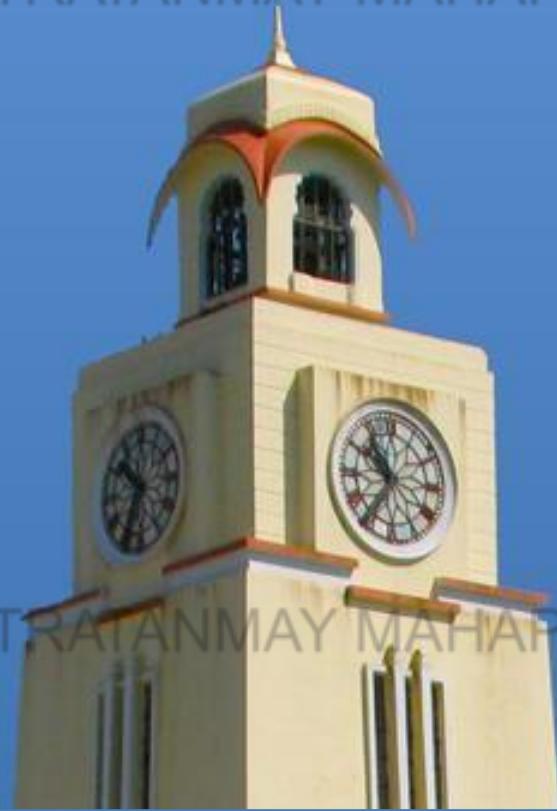
*

1

2 3

4 5 6

7 8 9 10



Break and Continue

Break and continue

- break – immediate suspension of the current loop
- continue – restart a new iteration

```
while (testExpression) {  
    // codes  
    if (condition to break) {  
        break;  
    }  
    // codes  
}
```

```
→ while (testExpression) {  
    // codes  
    if (testExpression) {  
        continue;  
    }  
    // codes  
}
```

Break and continue (Work out code)



Write a program to calculate the sum of at most 10 numbers entered by the user

- When the user enters a negative value, prompt the user to enter a positive integer
- When the user enters zero, terminate the loop
- Display the sum of the valid numbers entered by the user

Break and continue (Work out code)



```
#include <stdio.h>

#define MAX_NUMBERS 10

int main()
{
    int num, count = 0;
    int sum = 0;

    while (count < MAX_NUMBERS)
    {
        printf("Enter a positive integer (or 0 to terminate): ");
        scanf("%d", &num);

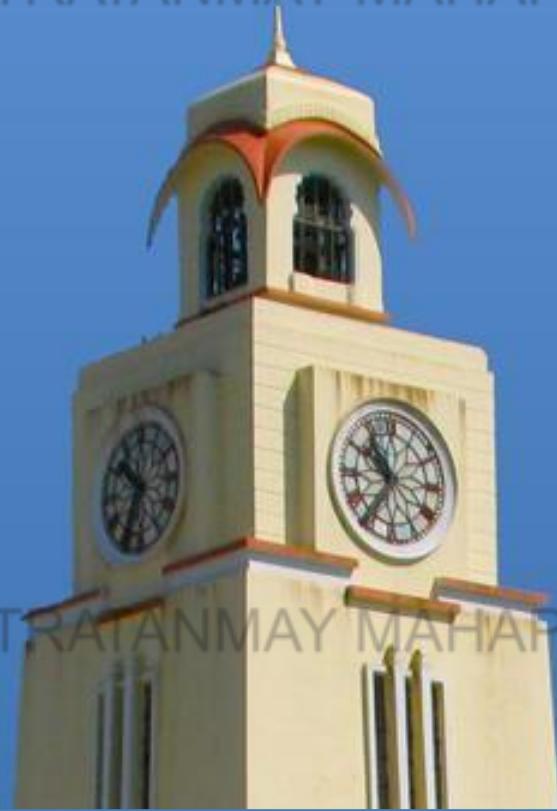
        if (num < 0)
        {
            printf("Negative values are not allowed. Please enter a positive integer.\n");
            continue;
        }

        if (num == 0)
        {
            break;
        }

        sum += num; // Add valid number to the sum
        count++; // Increment the count of valid numbers
    }

    // Display the sum of the valid numbers
    printf("The sum of the valid numbers entered is: %d\n", sum);
}

return 0;
```



The do... while loop

The do ... while loop

```
do {  
    statement(s);  
} while( condition );
```

- Conditional expression appears at the end
- Statements in the loop execute once before the condition tested
- Useful in scenarios where you want the code to execute at least once
- While loop: first satisfy the condition then proceed
- Do while loop: proceed first, then check the condition

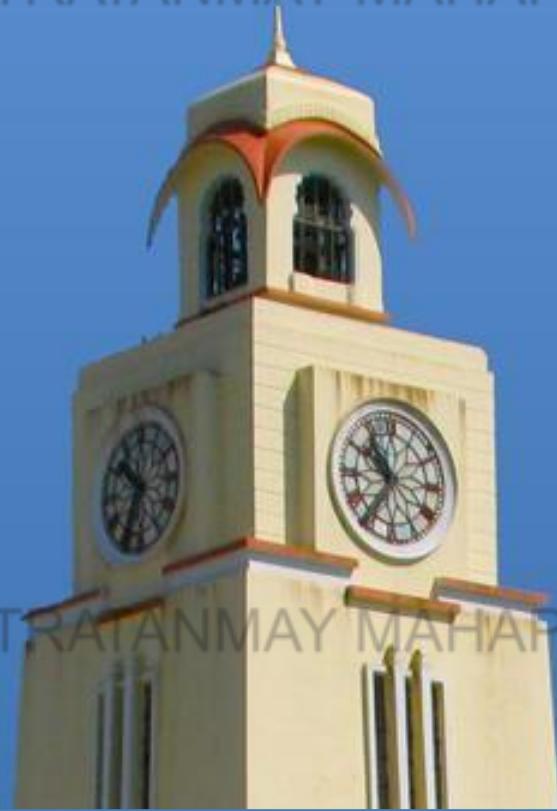
Example

```
int var=1;  
do {  
    printf("%d ", var++);  
} while (var <=10);  
printf("Do... While loop over");
```

This program prints first 10 positive integers

while vs do... while

while loop	do...while loop
This is entry-controlled loop. It checks condition before entering into loop	This is exit-controlled loop. Checks condition when coming out from loop
The while loop may run zero or more times	Do-While may run more than one times but at least once.
The variable of test condition must be initialized prior to entering into the loop	The variable for loop condition may also be initialized in the loop also.
<pre>while (condition) { //statement }</pre>	<pre>do { //statement } while (condition);</pre>



The for loop

For loop

```
for (initialization; condition; increment or decrement)
{
    //Statements to be executed repeatedly
}
```

• Initialization executed first, **and only once**

- Condition evaluated. If true, loop body executed
- Increment/decrement executed to update control variable
- Condition evaluated again. The process repeats.
- Condition false ? loop terminates

Examples

```
int main()
{
    int i;
    for (i=1; i<=3; i++)
    {
        printf("%d\n",
i);
    }
    return 0;
}
```

```
int main()
{
    int i;
    for (i=1; i<=3; i++)
        printf("%d\n", i);
    return 0;
}
```

Various forms of for loop

- ✓ `for (num=10; num<20; num=num+2)`

- ✓ `int num=10;`
 `for (;num<20; num++)`

- ✓ `for (num=10; num<20;) { num++; }`

- ✓ `for(; ;)`

- ✓ `for (i=1,j=10; i<10 && j>0; i++, j--)`

Exercises (Code)

What does this code print?

```
int count;  
for (count=1; count<=3; count++) ;  
printf("%d\n", count);
```

Exercises (Code)

Repeat this exercise using for loop:

Write a program to calculate the sum of max 10 numbers entered by the user

- When the user enters a negative value, prompt the user to enter a positive integer
- When the user enters zero, terminate the loop
- Display the sum of the valid numbers entered by the user

Exercises

Print this pattern on your screen using For loops. No. of rows are user input.

```
int i, j, rows;
printf("Enter number of rows: ");
scanf("%d",&rows);
for(i=1; i<=rows; ++i)
{
    for(j=1; j<=i; ++j)
        printf("* ");
    printf("\n");
}
```

Exercises

Homework

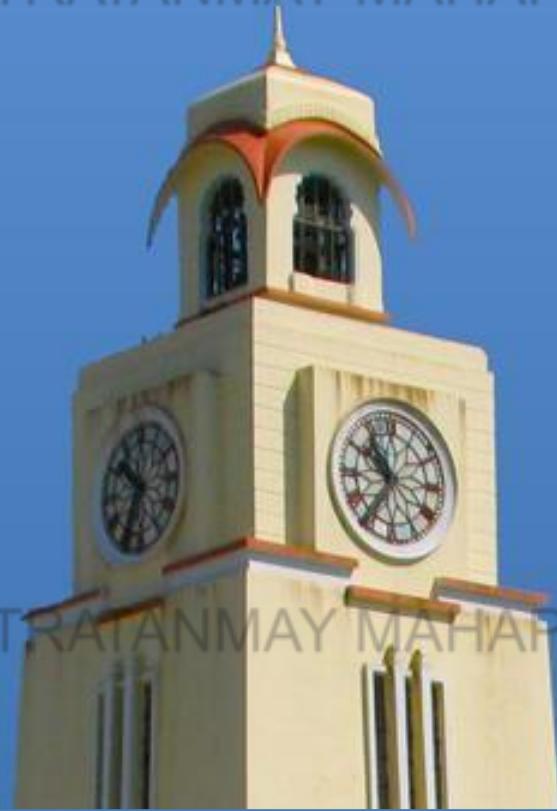
1

2 3 2

3 4 5 4 3

4 5 6 7 6 5 4

5 6 7 8 9 8 7 6 5



Thank you
Q & A

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA



TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA



Module 6 – Control Flow: Branching and Looping

BITS Pilani – Part 1

Pilani Campus

Department of Computer Science & Information Systems

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA

Module Overview

- Need for additional programming constructs
- Control Flow: Branching
 - Branching: if statement
 - Branching: if ladder
 - Branching: if-else statement
 - Branching: if-else ladder
 - Branching: nested if-else statement
- Control Flow: Ternary Operator
- Control Flow: Switch Statement



Need for additional Programming Constructs Conditionals, Statements and Blocks

Fibonacci Numbers

Definition

Consider the following mathematical definition:

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n > 1 \end{cases}$$

- What additional constructs do we need to define such things in C language?
 - A way to specify the **condition**
 - Example: *if n = 0*
 - A way to selectively choose different blocks of statements depending on the **outcomes of the condition check**
 - Example: $F_n = 0$ when $n = 0$ and $F_n = 1$ when $n = 1$

Statements and Blocks

Statement:

An expression followed by a semi-colon

Examples: `c = a + b; i++; printf("Hello World");`

Block:

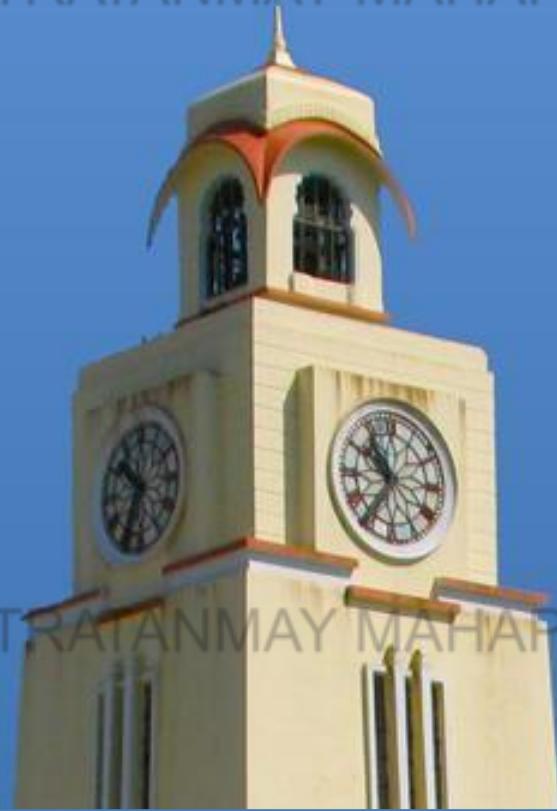
Set of statements enclosed inside a set of braces { and }

Example:

```
{  
    c = a+b;  
    c++;  
    printf("c is %d", c);  
}
```

Statements (Non-standard categorization)

- Declaration statements
 - Variable declarations (Module 5)
- Action statements
 - I/O Instructions: e.g., using *printf*, *scanf*, *getch*
 - Arithmetic instructions: using operators (Module 5)
 - Control Flow Instructions (this Module)
 - Decision control/Selection control instructions: using *if*, *if else*, *?:* (*conditional operator*)
 - Switch case control instructions
 - Iterative control instructions
 - Goto control instructions (Not recommended any more)



Control Flow: Branching

Control Flow: Branching

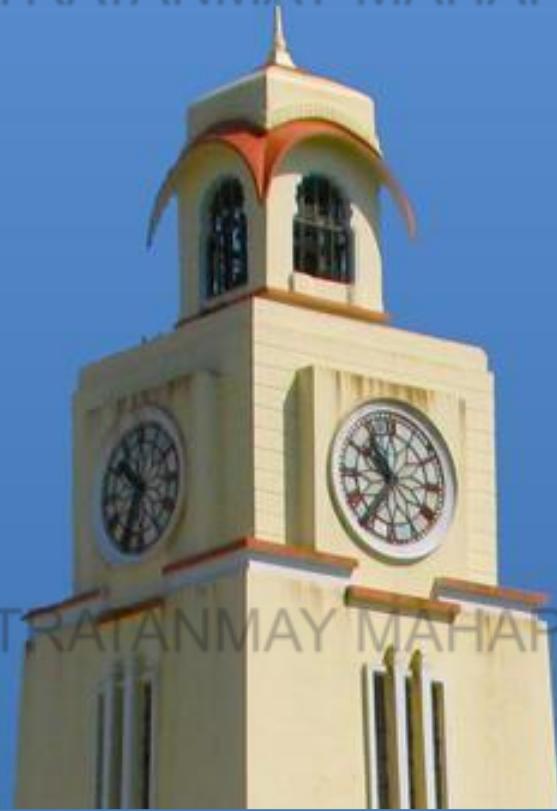
- Different set of instructions gets executed depending on the outcome of a conditional expression
- Writing **conditional expression**
 - Using relational operators such as `==`, `>=`, `<=`, `!=`, `<`, `>`
 - Using logical operators and : `&&`, `||`, `!`
- Examples of conditional expressions:
 - `(x+y >= 10)`
 - `(marks >= 90 && marks <= 100)`
 - `(no_of_transactions >= 5 && city == "Metropolitan")`

Control Flow: Branching

- Outcome of the condition
 - Non- Zero or **true**
 - Zero or **false**

Examples:

```
int x = 5, y = 10;  
(x + y <= 20)  
int Marks = 95;  
(Marks <= 100 && Marks >= 90)  
(x & y)
```



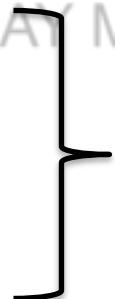
Branching: if Statement

Branching: if statement

- `if(condition)
 statement;`

- `if(condition)
{
 statement1;
 statement2;

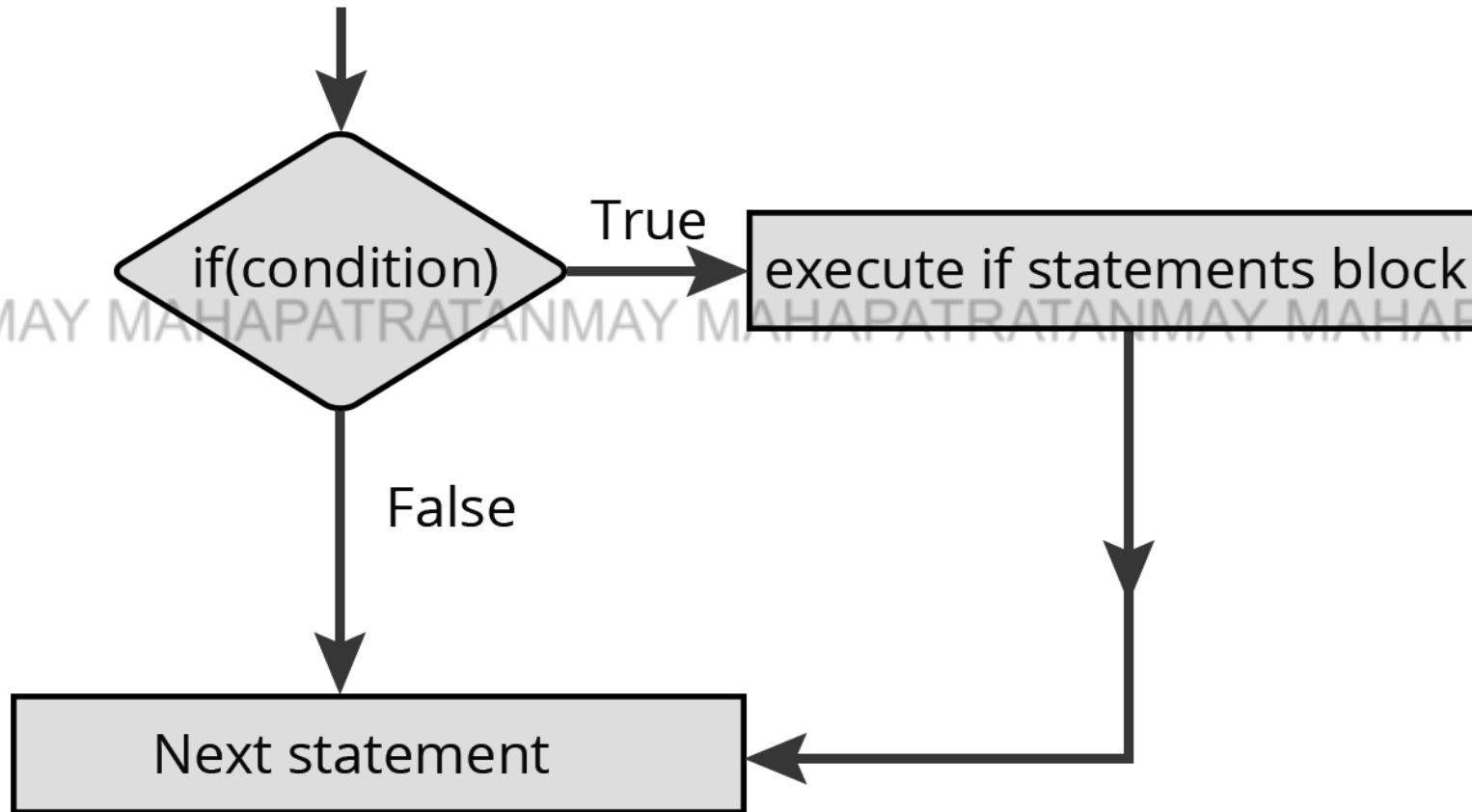
 statementn;
}`



Block of statements

- *statement or block of statements* gets executed only if the condition evaluates to *true or non-zero*

Branching: if statement



Example 1 (Check if number is positive)

```
int main()
{
    int a;
    printf("Enter a number");
    scanf("%d", &a);
    if( a > 0 )
    {
        printf("Number is positive\n");
    }
    printf("Rest of the program");
    return 0;
}
```

O/P:

Enter a number

10

Number is positive

Rest of the program

Example 1 (Check if number is positive) contd..

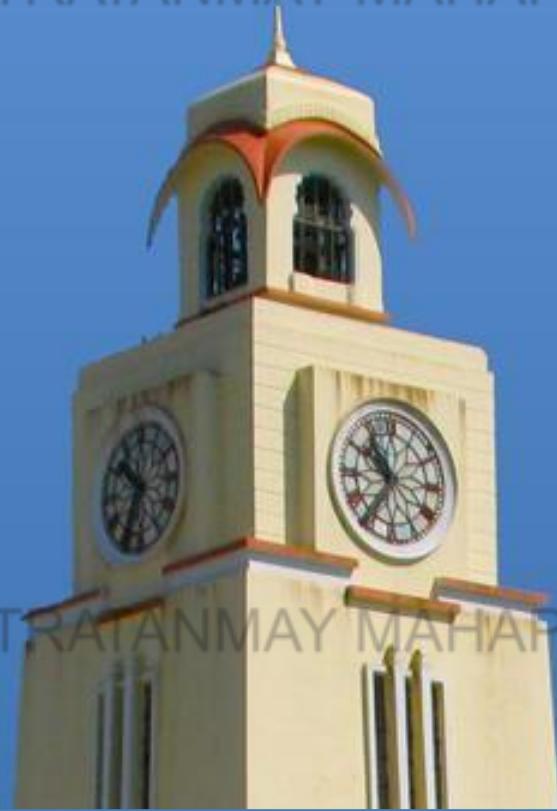
```
int a;  
printf("Enter a number");  
scanf("%d", &a);  
if( a > 0 )  
{  
    printf("Number is  
positive\n");  
}  
printf("Rest of the program");  
return 0;
```

O/P:

Enter a number

-5

Rest of the program



Branching: if ladder

Branching: if ladder

A single program can have more than one If statement

```
if (condition1) {  
    //These statements would execute if the condition_1 is  
    true  
}  
if(condition2){  
    //These statements would execute if the condition_2 is  
    true  
}  
.  
.  
.  
if (conditionn) {  
    //These statements would execute if the nth condition  
    is true  
}
```

Example 2 (Check the sign of the number)

```
int a;  
printf("Enter a number");  
scanf("%d", &a);  
  
if( a > 0 ) {  
    printf("Number is positive\n");  
}  
if( a < 0 ) {  
    printf("Number is negative\n");  
}  
if( a == 0 ) {  
    printf("You entered zero\n");  
}  
  
printf("Rest of the program");
```

O/P:

Enter a number

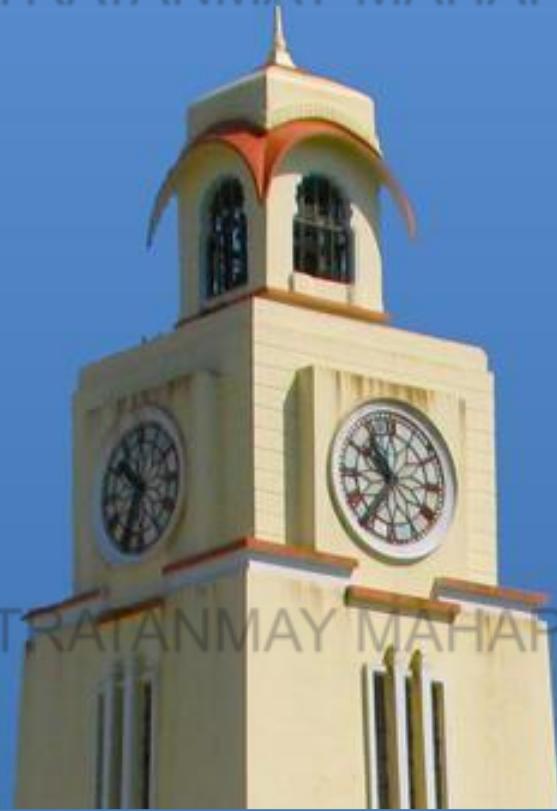
0

You entered zero

Rest of the program

Exercise

Write a C program using only If statements to find the larger value among two numbers



Branching: if-else statement

Branching: if-else Statement

```
if(condition)
{
    // If- block of statements
}
else
{
    // else-block of statements
}
```

```
if(condition)
    // statement;
else
    // statement;
```

If condition inside the *If* parentheses is true then *If- block of statements* is executed, else *else-block of statements* is executed

Example 2 (Check the sign of the number) contd..

```
int a;  
printf("Enter a number");  
scanf("%d", &a);  
if( a > 0 )  
{  
    printf("Number is positive\n");  
}  
else  
{  
    printf("Number is either negative or  
zero\n");  
}  
printf("Rest of the program");
```

O/P:

Enter a number

5

Number is Positive

Rest of the program

Example 2 (Check the sign of the number) contd..

```
int a;  
printf("Enter a number");  
scanf("%d", &a);  
if( a > 0 )  
{  
    printf("Number is positive\n");  
}  
else  
{  
    printf("Number is either negative or  
zero\n");  
}  
printf("Rest of the program");
```

O/P:

Enter a number

-5

Number is either negative
or zero

Rest of the program

Example 3

```
#include <stdio.h>
int main() {
    int a = 10, b = 4, c = 10, d = 20;

    if (a > b && c == d)
        printf("a is greater than b AND c is equal to d\n");
    else
        printf("AND condition not satisfied\n");

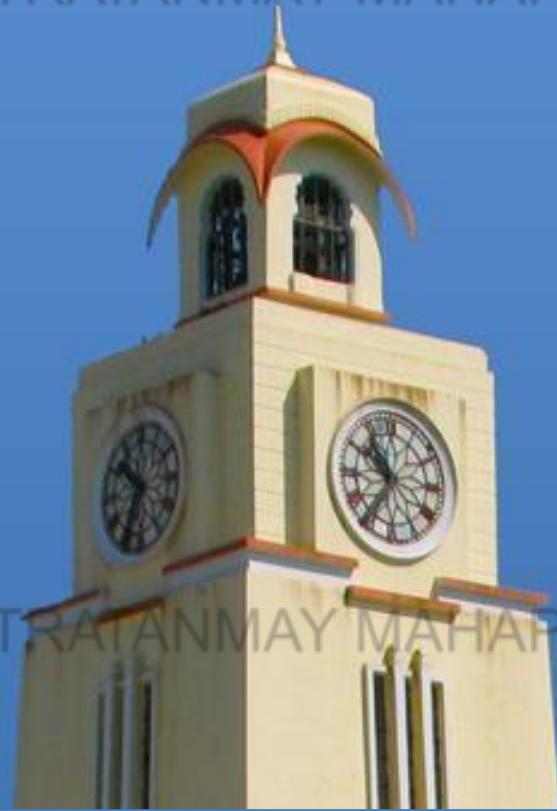
    if (a > b || c == d)
        printf("a is greater than b OR c is equal to d\n");
    else
        printf("Neither a is greater than b nor c is equal to d\n");

    if (!a)      printf("a is zero\n");
    else        printf("a is not zero");

    return 0;
}
```

Output:

AND condition not satisfied
a is greater than b OR c is equal to d
a is not zero



Branching: Nested if-else statement

Branching: Nested if-else statement

If one or more *if* and/or *else* statement is/are present inside the body of another “if” or “else”

```
if (condition1)
{
    if (condition2)
        // statement or block of statements for if
    else
        // statement or block of statements for else
}

else{
    if (condition3)
        // statement or block of statements for if
    else
        // statement or block of statements for else
}
```

Example 4

Write a C program using nested if-else statements to find the larger value among three integers a, b, c.

Example 4 contd..

```
int a, b, c;
printf("Enter the numbers");
scanf("%d%d%d", &a, &b, &c);
if(a > b)
{
    if(a > c)
        printf("a is the largest");
    else
        printf("c is the largest");
}
else
{
    if(b > c)
        printf("b is the largest");
    else
        printf("c is the largest");
}
printf("End of the program");
return 0;
```



Branching: if-else ladder

Branching: if-else ladder

```
if (condition1) {  
    //These statements would execute if the condition1 is true  
}  
else if (condition2) {  
    //These statements would execute if the condition2 is true  
}  
else if (condition3) {  
    //These statements would execute if the condition3 is true  
}  
.  
.  
else {  
    /* These statements would execute none of the previous  
    condition is true */  
}
```

Example 5

Write a C program using nested If-else to find the grade of a student based on following conditions

1. Marks greater than 90 implies grade A
2. Marks above 80 and less than or equal to 90 implies grade B
3. Marks above 70 and less than or equal to 80 implies grade C
4. Else, Grade is Fail

Example 5 contd..

```
if(Marks > 90)
    printf("Grade is A");
else if(Marks <= 90 && Marks > 80)
    printf("Grade is B");
else if(Marks <= 80 && Marks > 70)
    printf("Grade is C");
else
    printf("Grade is FAIL");
```

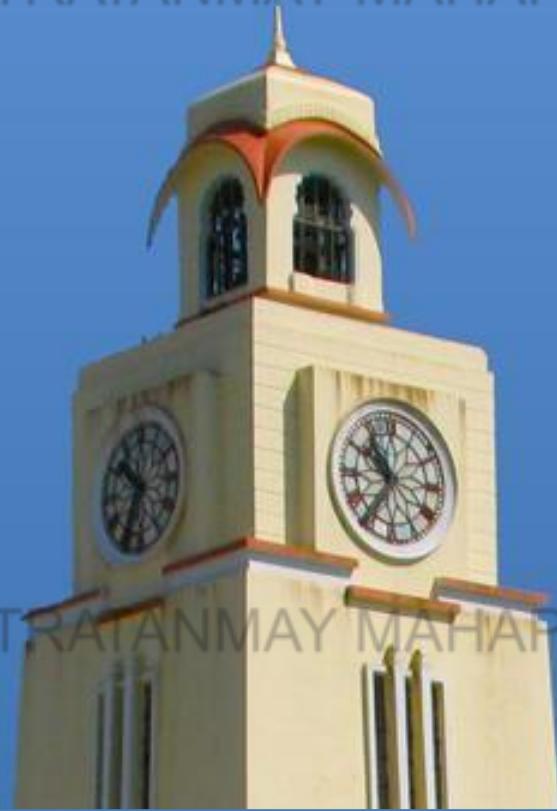
Exercise

Write a code to find whether a character entered is a digit, uppercase alphabet, lowercase alphabet or any other special character

Note: ASCII value of digits ☐ 48 to 57

ASCII value of upper alphabets ☐ 65 to 90

ASCII value of lower alphabets ☐ 97 to 122



Control Flow: Ternary Operator

Ternary Operator

- An alternative way to write if...else construct:

```
if (expr1)  
    expr2 ;  
else  
    expr3;
```

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA

- Syntax: `expr1? expr2:expr3`
- Only one of `expr2` and `expr3` is evaluated
- If `expr2` and `expr3` are of different types, the type of the result is determined by the type conversion rules.

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA

Ternary Operator (Type conversion)



```
int x = 5;  
double y = 10.5;  
double max = (x < y) ? x : y; // x is implicitly converted to double
```

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA

Example 6

A = 43, B = 7, C = 0, D = 0

A + B == 50 ? C = 10 : C = 15

– O/P: C = 10

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA

A > C ? printf("Hello") : printf("World") ;

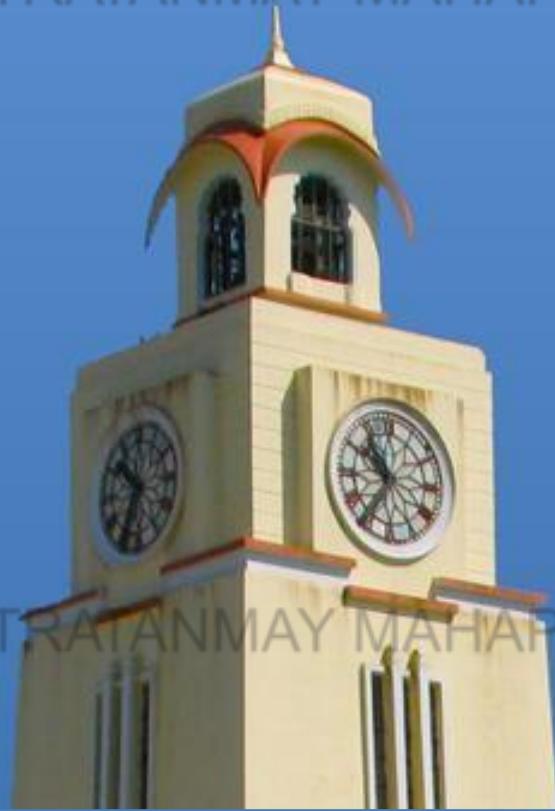
– O/P: ?

A < C ? printf("Hello") : printf("World") ;

– O/P: ?

Exercise

Write a statement (using ternary operator) to find out the largest of the three integers `a`, `b` and `c`, and store the value in `max`.



Control Flow: Switch Statement

Switch Statement

- A multi-way decision that tests whether an expression matches one of a number of **constant integer** values, and branches accordingly.
- The following statement can be introduced in the Switch statement for immediately exiting from it.

break;

Syntax:

```
switch (expr) {  
    case const-expr1 : statements  
    case const-expr2 : statements  
        break;  
    default:         statements
```

Example 7

```
#include <stdio.h>
int main() {
    int language = 10;
    switch (language) {
        case 5: printf("C#\n");
        break;
        case 2: printf("C\n");
        break;
        case 8: printf("C++\n");
        break;
    default:
        printf("Other programming language\n");
    }
}
```

What is the output?

Example 8

```
#include <stdio.h>
int main() {
    int number=5;
    switch (number) {
        case 1:
        case 2:
        case 3:
            printf("One, Two, or Three.\n");
            break;
        case 4:
        case 5:
        case 6:
            printf("Four, Five, or Six.\n");
            break;
        default:
            printf("Greater than Six.\n");
    }
}
```

What is the output?



Thank you
Q & A

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA



Module 7 – Part 1 - Functions

BITS Pilani

Pilani Campus

Department of Computer Science & Information Systems

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA

Module Overview

- **What are Functions?**
- **Function Declaration**
- **Function Definition**
- **Function Call**
- **Functions in Memory**
- **Call by Value**
- **Exercises**



Functions

Functions

We have already seen functions

main() function is the place where any C program starts its execution

```
/* myfirst.c: to compute the sum of two numbers */
#include<stdio.h> //Preprocessor directive
/*Program body*/
int main()
{
    int a, b, sum; //variable declarations
    printf("Please enter the values of a and b:\n");
    scanf("%d %d", &a, &b);
    sum = a + b; // the sum is computed
    printf("The sum is %d\n", sum);
    return 0; //terminates the program
}
```

What are functions?

- Functions are “*Self contained program segment that carries out some specific well-defined task*”

- A function

- processes information that is passed to it from the calling portion of the program, and
- returns a single value

- Every C program has one or more functions

Sum of two numbers:

```

int sum(int a, int b) ←
{
    int total;
    total = a + b;
    return total;
}
int main()
{
    int x,y,z;
    x = 5, y = 4;
    z = sum(x,y);
    printf("Sum is %d",z);
    return 0;
}

```

Two values received by the “sum” function

“sum” function computes the sum of two values and returns it

“main” function calls “sum” function passing values of x and y and receives their sum.

Types of Functions

- Predefined functions
 - e.g., `scanf()`, `printf()`, `getc()`, `getchar()`, `exit()`, ...
- User defined functions

Using Functions

Functions are

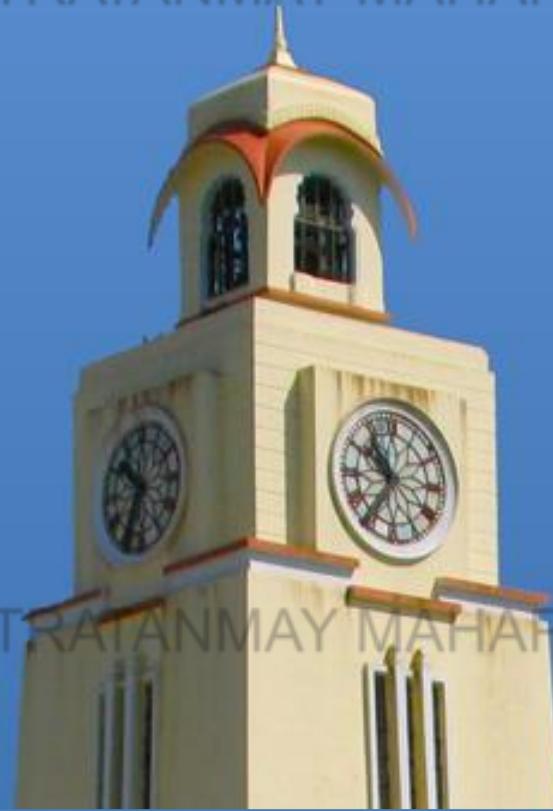
- Declared
- Defined
- Called

```
/* myProg.c */  
#include <stdio.h>  
int sum(int a, int b);  
int sum(int a, int b)  
{  
    int total;  
    total = a + b;  
    return total;  
}  
int main()  
{  
    int x, y, z;  
    x = 5, y = 4;  
    z = sum(x, y);  
    printf("Sum is %d", z);  
    return 0;  
}
```

Function Declaration

Function Definition

Function Call



Function Declaration

Function Declaration

- Gives information to the compiler that the function may later be used in the program
 - Declarations appear before definitions

- Although optional in many compilers, it is a good practice to use

- Syntax:

```
<return_type> <function_name> (list-of-typed-parameters);
```

- Example:

- **float sqrt(float x);**
 - **void average(float, float);**

Function Declaration (Contd.)

```
float average(float a, float b);
```

Return type of the function

- Any valid data type in C
- **void** in case a function does not return anything explicitly

Optional list of parameters

- A comma-separated list of type-name pairs
 - Or just types
- It can be an empty list

Name of the function

- Any valid identifier name

Another example

```
void average(float, float);
```

Note that declarations doesn't mandate specifying the variable names



Function Definition

Function Definition

Syntax:

```
return_type function_name (list_of_parameters)  
{  
    function body  
}
```

Parts of function definition:

1. Type of the value it returns
2. Name of the function
3. Optional list of typed parameters
4. Function body

Function Definition

```
float average(float a, float b)
{
    float avg;
    avg = (a + b) / 2;
    return avg;
}
```

Name of the function

- Any valid identifier name

Return type of the function

- Any valid data type in C
- **void** in case a function does not return anything explicitly

Optional list of parameters

- A comma-separated list of type-name pairs
 - Can't be just types
- Contains **formal parameters**
- It can be an empty list

Function body

- Lines of C code evaluating a program logic

Another example

```
void print_nums(float a, float b)
{
    printf("Num1, Num2 are: %f, %f", a, b);
}
```

Matching Function definition with Function declaration



```
float average(float x,float y); float average(float a,float b)
{
    float avg;
    avg = (a + b)/2;
    return avg;
```

The following should exactly match between a **Function Declaration** and a **Function Definition**:

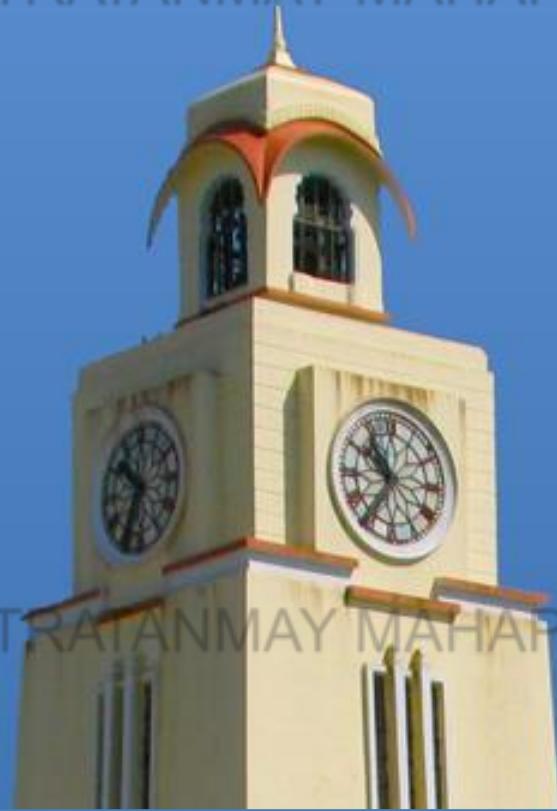
- Name of the Function
- Number of Parameters
- Type of each Parameter
 - Names of the parameters need not match!
- The return type

Function Definitions and Declarations

- Typically, functions are declared first in the program
- Then the functions are defined.
- Although **declaring functions is optional**, it is recommended to declare them before defining them otherwise **“Warning”**

- Helps in writing modular programs
 - Typically, functions are declared in “.h” files
 - “.h” files are header files
 - These “.h” files are included in your “.c” files

We will see more about writing modular programs in lab 8



Function Call

Calling a function

The functions which you have declared and (or) defined can be called from either `main()` function or any other function.

```
/* myProg.c */
#include <stdio.h>
int sum(int a, int b);
int sum(int a, int b)
{
    int total;
    total = a + b;
    return total;
}
int main()
{
    int x,y,z;
    x = 5, y = 4;
    z = sum(x,y);
    printf("Sum is %d:",z);
    return 0;
}
```

Function Declaration

Function Definition

Function Call

Example: Function “sum” being called from “main” function

Calling a function (another example)

```
#include <stdio.h>
int sum(int, int);
int sum_call(int, int);
int sum(int a, int b) {
    int total;
    total = a + b;
    return total;
}
int sum_call(int m, int n)
{
```

```
    int main() {
        int x,y,z;
        x = 5, y = 4;
        z = sum_call(x,y);
        printf("Sum is %d:",z);
        return 0;
    }
```

- *Function “sum” being called from “sum_call” function*
- *Function “sum_call” being called from “main” function*

Calling a Function (Syntax)

```
function_name (arguments values) ;
```



Function Name

- This should match with the name used in the function definition and declaration



Argument values

- comma-separated list of expressions
- contains **actual parameters**

Examples:

```
sum(a, b);
```

```
sum(a+b, c+d);
```

Matching Function Call with Function Definition

```
int sum(int a, int b); int main() {  
int sum(int a, int b){    int x,y,z;  
    int total;                x = 5, y = 4;  
    total = a + b;            z = sum(x,y);  
    return total;             printf("Sum is %d:",z);  
}return 0;  
}
```

The following should exactly match between a **Function Definition** and a **Function Call**:

- The **Name of the function**
- The **number of actual arguments** in the function call must match the **number of formal parameters** in the function definition
- **Type of each argument** in the function call must match with the **type of the corresponding parameter** in the function definition
 - *Names of parameters need not match!*
- The **return type**

Flow of program execution

```
/* myProg.c */  
#include <stdio.h>  
  
int sum(int a, int b);  
int sum(int a, int b){  
    int total;  
    total = a + b;  
    return total;  
}
```

```
int main() {  
    int x,y,z;  
    x = 5, y = 4;  
    z = sum(x,y);  
    printf("Sum is %d:",z);  
    return 0;  
}
```

Pros and Cons of using Functions



Advantages

- **Modularization**
- **Code Reusability**
- **Reduced Coding time**
- **Easier to Debug**
- **Easier to understand**

Disadvantages

- **Reduced execution speed**
 - *Every function call adds an additional overhead to the OS to create space for it in the program memory.*
 - *This slows down the program.*

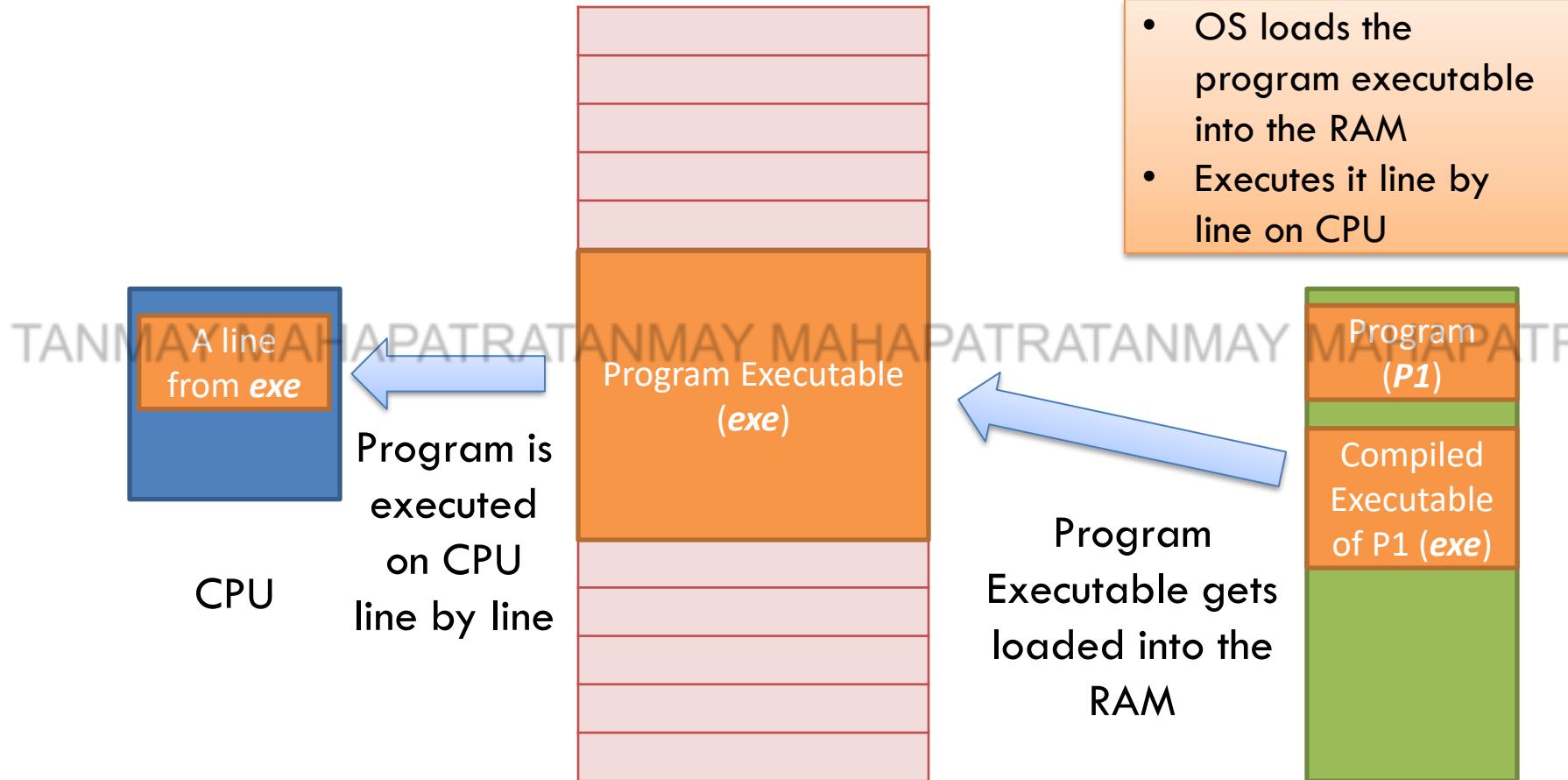


Functions in Memory

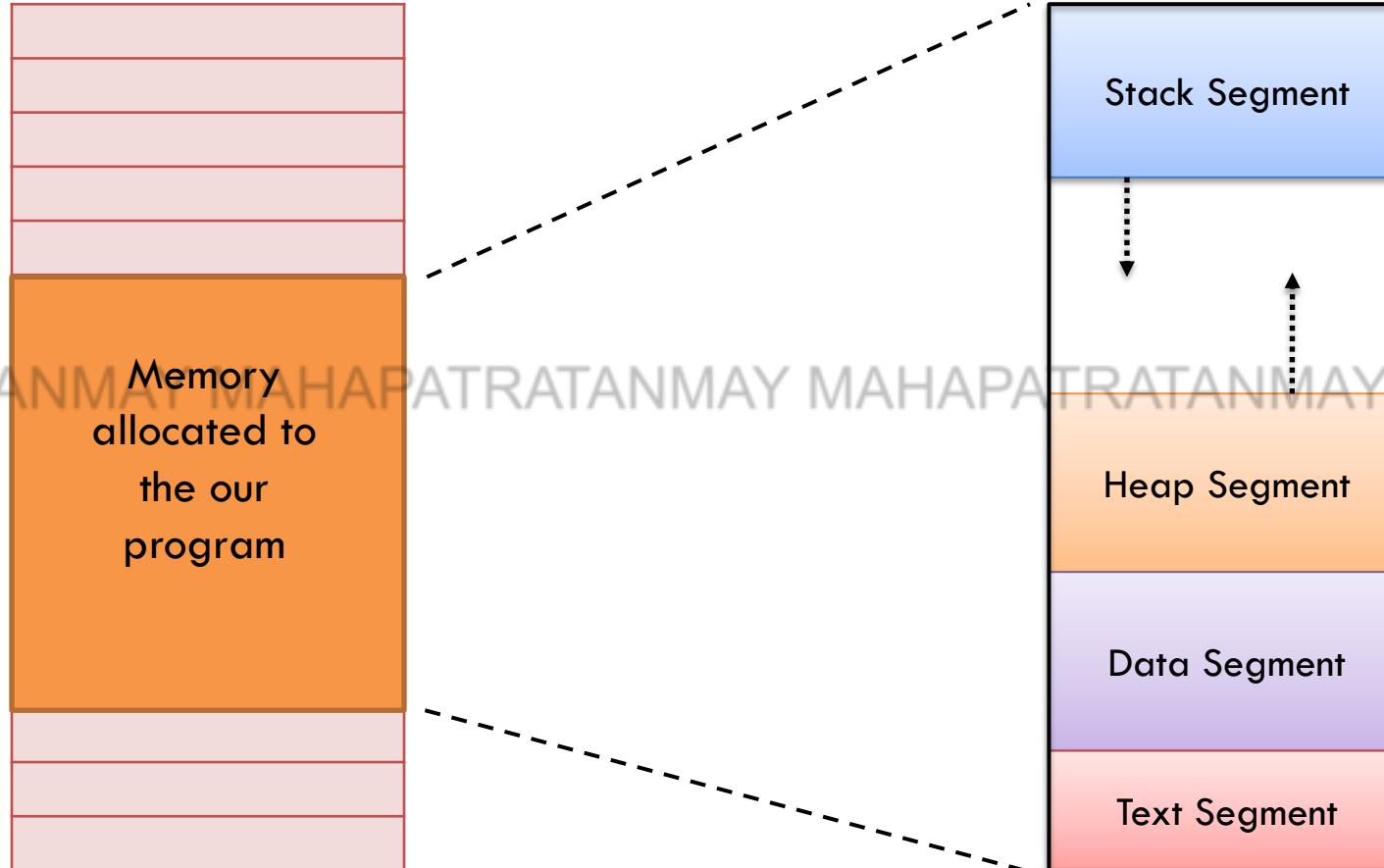
How does function execution look like in memory?

Let us recall our block diagram

Our block diagram is back again!



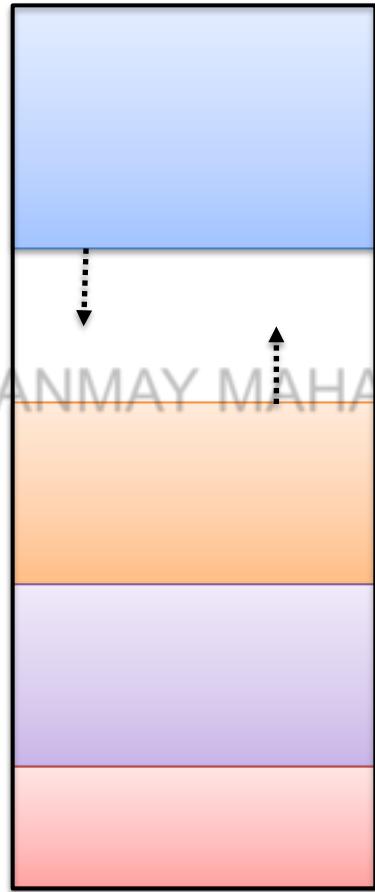
Looking at the main memory only



Functions (and variables defined in them) reside in the stack segment of the memory

Functions in Memory

Stack



Heap

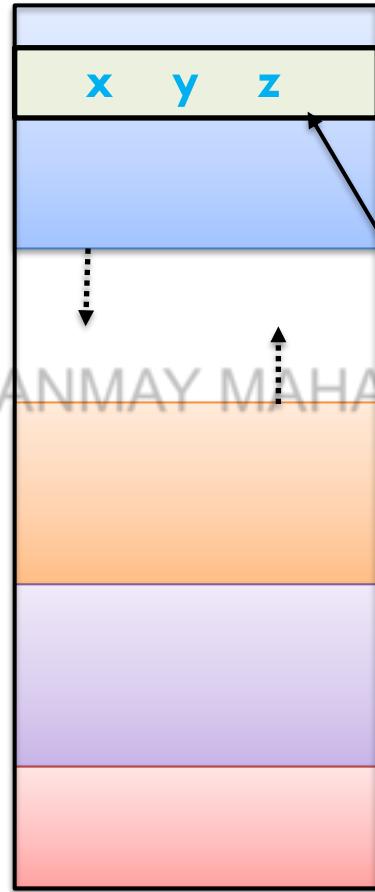
Data

Text

```
/* myProg.c */
#include <stdio.h>
int sum(int a, int b);
int sum(int a, int b)
{
    int total;
    total = a + b;
    return total;
}
int main()
{
    int x,y,z;
    x = 5, y = 4;
    z = sum(x,y);
    printf("Sum:%d",z);
    return 0;
}
```

Functions in Memory

Stack



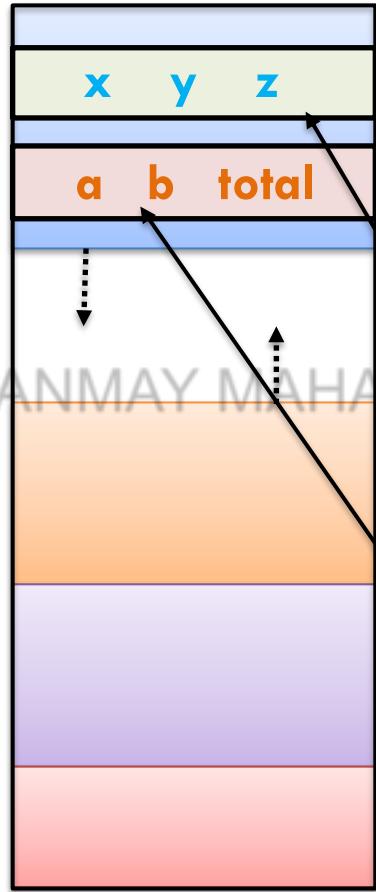
Frame allocated to `main()` in the stack

Variables `x`, `y` and `z` reside inside the stack frame allocated to the function `main()`

```
/* myProg.c */
#include <stdio.h>
int sum(int a, int b);
int sum(int a, int b)
{
    int total;
    total = a + b;
    return total;
}
int main()
{
    int x,y,z;
    x = 5, y = 4;
    z = sum(x,y);
    printf("Sum:%d",z);
    return 0;
}
```

Functions in Memory

Stack



Frame allocated to `main()` in the stack

Frame allocated to `sum()` in the stack

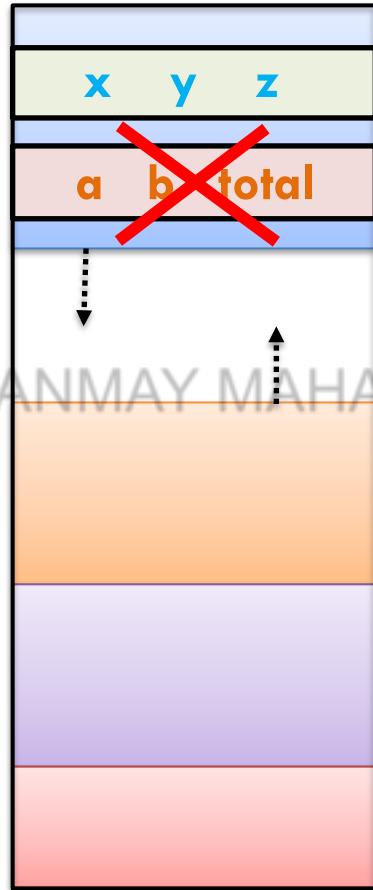
Variables `x`, `y` and `z` reside inside the stack frame allocated to the function `main()`

Variables `a`, `b` and `total` reside in the stack frame allocated to the function `sum()`

```
/* myProg.c */
#include <stdio.h>
int sum(int a, int b);
int sum(int a, int b)
{
    int total;
    total = a + b;
    return total;
}
int main()
{
    int x,y,z;
    x = 5, y = 4;
    z = sum(x,y);
    printf("Sum:%d",z);
    return 0;
}
```

Functions in Memory

Stack



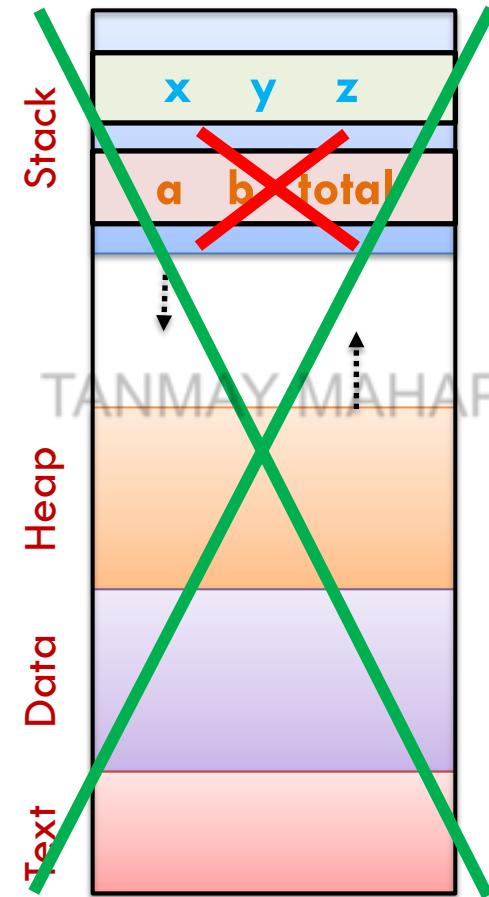
Frame allocated to `main()` in the stack

Frame allocated to `sum()` in the stack

When function `sum()` returns,
frame allocated to it is
destroyed. So, a, b and total
are destroyed

```
/* myProg.c */
#include <stdio.h>
int sum(int a, int b);
int sum(int a, int b)
{
    int total;
    total = a + b;
    return total;
}
int main()
{
    int x,y,z;
    x = 5, y = 4;
    z = sum(x,y);
    printf("Sum:%d",z);
    return 0;
}
```

Functions in Memory



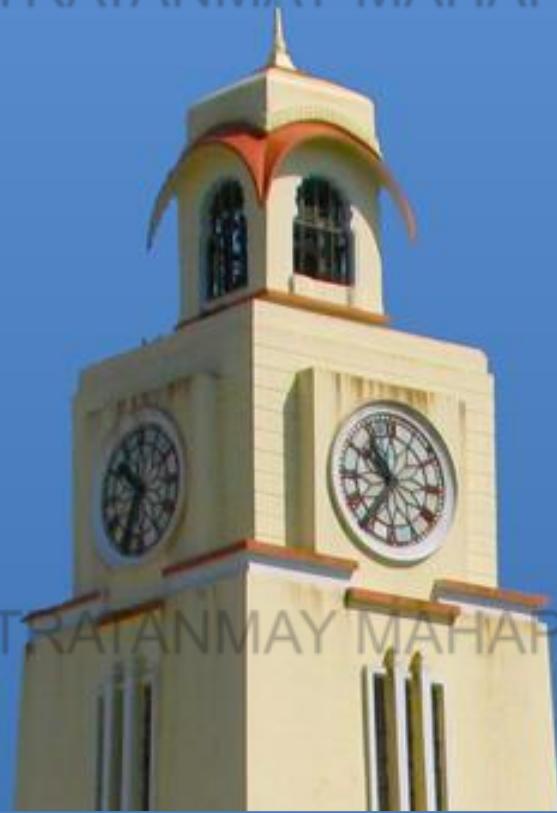
Frame allocated to **main()** in the stack

Frame allocated to **sum()** in the stack

When function **sum()** returns, frame allocated to it is destroyed. So, **a**, **b** and **total** are destroyed.

When function **main()** returns or the program terminates, the entire memory allocated to this program is destroyed.

```
/* myProg.c */
#include <stdio.h>
int sum(int a, int b);
int sum(int a, int b)
{
    int total;
    total = a + b;
    return total;
}
int main()
{
    int x,y,z;
    x = 5, y = 4;
    z = sum(x,y);
    printf("Sum:%d",z);
    return 0;
}
```



Call by Value

Swap two numbers and “Call by Value”

```
int main()
{
int x, y;
printf("Enter the numbers:\n");
scanf("%d %d", &x, &y);
swap(x, y);
printf("x=%d, y=%d\n", x, y);
return 0;
}

void swap(int a, int b)
{
int c = a;
a = b;
b = c;
printf("a=%d, b=%d\n", a, b);
}
```

Call by value:

- When a function is being called, the values of the actual arguments from caller function are copied into the formal parameters of the called function.
- Values of x and y are copied into a and b respectively
- Any change to the values of a and b is not reflected into x and y
- a and b are swapped in the swap() function. This swap is not reflected in x and y
- printf statement in swap() function prints the swapped values of a & b
- printf statement in main() function prints the old values of x and y

Program Execution:

Enter the numbers:

4 5

a=5, b=4

x=4, y=5



More Examples and Exercises

Examples (Work out)

Example 1:

Write a function `factn()` which accepts input n and computes n!

Write the relevant code in `main()` to call this function.

Example 2:

Write a C program to compute the sum of the series

$1 + 2^2 + 3^3 + \dots + n^n$ (where n is user input).

Use a function `long compute_sum(int)` to compute the sum of the series, and within it, call another function `long power(int)` to compute the term i^i for each i

Exercise: Execute both the programs and observe the output

innovate

achieve

lead

```
void f1(int);  
int f2();  
int main() {  
    f1(f2());  
    return 0;  
}  
  
void f1(int a){  
    printf("%d",a);  
}  
  
int f2(){  
    return 5;  
}
```

```
int f1(int);  
int f2(int);  
int main(){  
    printf("%d\n",f1(f2(f1(15))));  
    return 0;  
}  
  
int f1(int a){  
    return f2(a/2);  
}  
  
int f2(int a){  
    return a>5?a:5;  
}
```

Pre-defined Functions

- **printf() function**

```
int printf("format string",argument_list);
```

- **scanf() function**

```
int scanf("format string",argument_list);
```

Pre-defined Functions

- **exit() function**

```
#include<stdio.h>
#include<stdlib.h>
int main() {
    //Program code
    ...
    ...
    if (errorCondition) {
        printf("An error occurred.\n");
        exit(EXIT_FAILURE); // Terminate the entire program with a failure status
    }
    // More program code
    ...
    ...
    return 0; // Return from the main function with a success status
}
```



Thank you
Q & A

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA



Module 7 – Part 2 – Scope and Storage Classes of Variables

BITS Pilani

Pilani Campus

Department of Computer Science & Information Systems

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA

Module Overview

- **Scope and Storage Class of Variables**
- **Memory Layout of a C Program**
- **Auto Variables**
- **Global Variables**
- **Static Variables**
- **Register Variables**

Consider this example

```
#include <stdio.h>
void f1(int a) {
    printf("a = %d\n",a);
    a = 10;
    printf("a = %d\n",a);
}
int main() {
    int a = 5;
    f1(a);
    printf("a = %d",a);
    return 0;
}
```

output:

a = 5 } **a is local to the function f1**
a = 10 }

a = 5 ← **a is local to the function
main**

Scope and Storage Class of a variable



Storage Class: A storage class of a variable tells us about the following:

- the variable's scope, or which sections of the code where you can access and use it
- the location where the variable will be stored inside the memory
- the initial value of a variable
- the lifetime of a variable, or how long does the variable reside in the memory

Example - Scope

```
#include <stdio.h>
int y = 5; //scope of y is complete program
void main() {
    int a = 5; //scope of a is main()
    {
        int b = 10; //scope of b is {}
    }
    printf("a = %d, b = %d",a,b); f1();
}
void f1() {
    int x = 2, b = 5; //scope of x and b is f1()
}
```

Scope and Storage Class of a variable



- **Auto**
- **Global**
- **Static**
- **Register**



Auto Variables

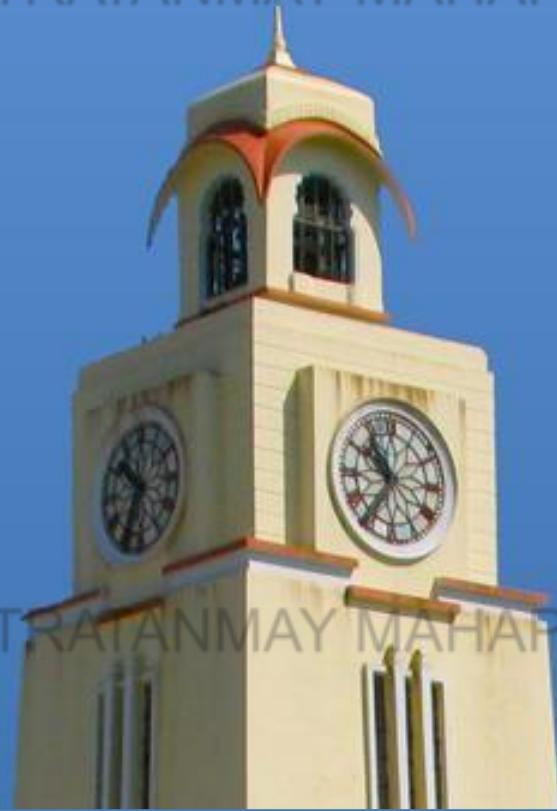
Auto Storage Class

Variables that are declared within a code block (`{ ... }`) are known as **Auto variables** or *local variables*.

- Scope:** Only the block in which it is declared can access it
- Initial Value:** By default, it contains a **garbage value**
- Storage Location:** Stored in the **stack segment**
- Lifetime:** Until the execution of the block finishes

```
#include <stdio.h>
void main() {
    int a; ← auto variable a local to block of main() function
    a = 5;
    {
        int b = 10; ← auto variable b local to the block {}
        printf("a = %d, b = %d", a, b);
        f1();
    }
    void f1() {
        int a; ← auto variable a local to the function f1()
        a = 20;
    }
}
```

Error!



Global Variables

Global Storage Class

Global variables are variables that are **declared outside all the blocks** (or outside all the functions)

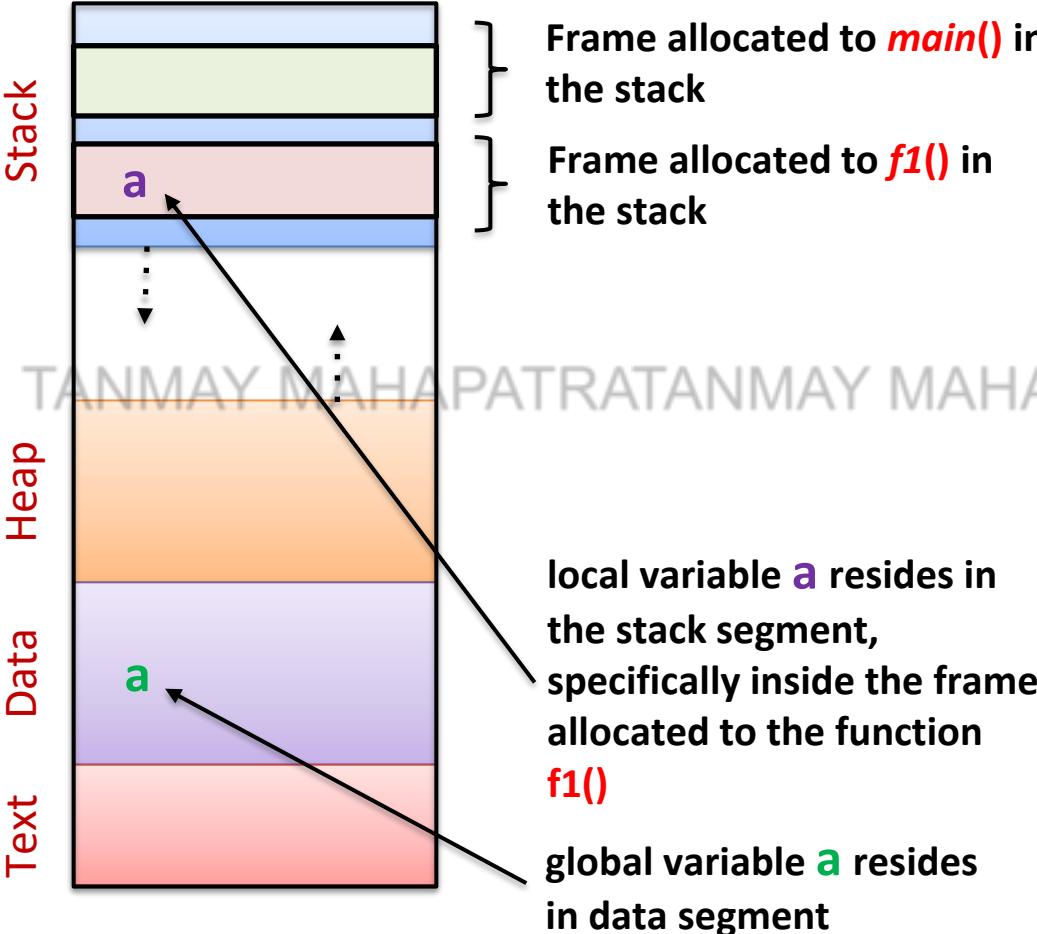
- **Scope:** Accessible to all the functions in a program
- **Initial Value:** Default value is **0**
- **Storage Location:** Stored in the **data segment**
- **Lifetime:** Accessible throughout the program execution

```
#include <stdio.h>
int a;           ← global variable a that is
void f1() {
    printf("a = %d\n", ++a);
    int a = 2;
    printf("a = %d\n", a);
}
void main() {
    a = a+3; f1();
    int a = 5;
    printf("a = %d", a);
}
```

Output:

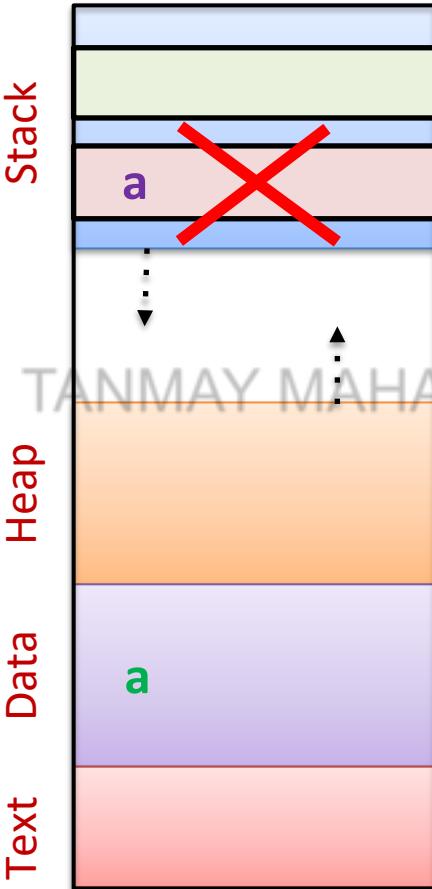
a = 4 ← *value of global variable a*
 a = 2 ← *value of variable a local to f1()*
 a = 5 ← *value of variable a local to main()*

Functions, local and global variables in main memory



```
#include <stdio.h>
int a;
void f1() {
    printf("a = %d\n", ++a);
    int a = 2;
    printf("a = %d\n", a);
}
void main() {
    a = a+3; f1();
    int a = 5;
    printf("a = %d", a);
}
```

Functions, local and global variables in main memory



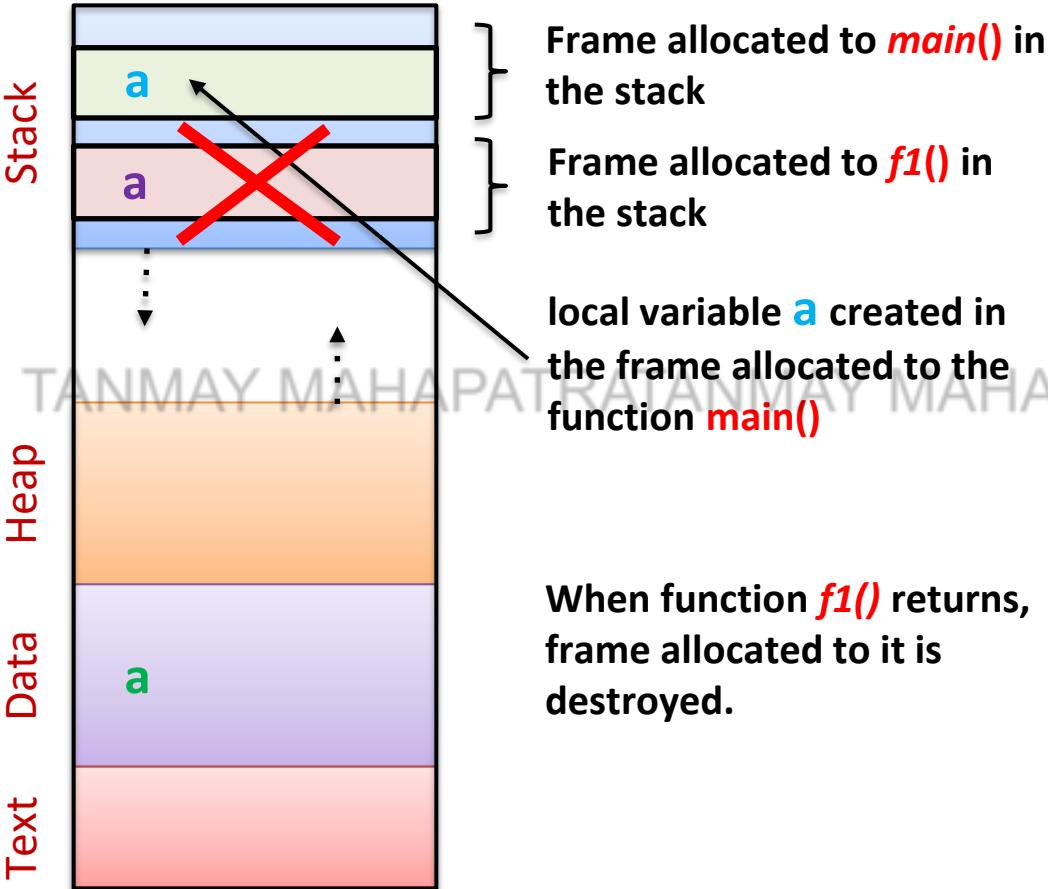
Frame allocated to `main()` in the stack

Frame allocated to `f1()` in the stack

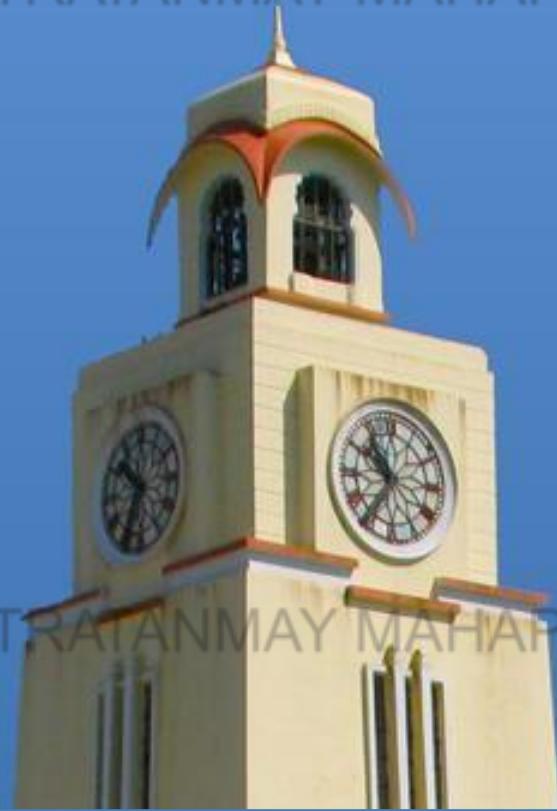
When function `f1()` returns, frame allocated to it is destroyed.

```
#include <stdio.h>
int a;
void f1() {
    printf("a = %d\n", ++a);
    int a = 2;
    printf("a = %d\n", a);
}
void main() {
    a = a+3; f1();
    int a = 5;
    printf("a = %d", a);
}
```

Functions, local and global variables in main memory



```
#include <stdio.h>
int a;
void f1() {
    printf("a = %d\n", ++a);
    int a = 2;
    printf("a = %d\n", a);
}
void main() {
    a = a+3; f1();
    int a = 5;
    printf("a = %d", a);
}
```



Static Variables

Static Variables

Are of two types:

- Local static variable
- Global static variable

Local Static Variables

Static variables are declared using the keyword **static**.

- E.g.- **static int a;**
- **Scope:** Remains **visible only to the function or the block in which it is defined**
- **Initial Value:** Default value is **0**
- **Storage Location:** Stored in the **Data Segment**
- **Lifetime: Until the program terminates.** The value assigned to it remains even after the function (or block) where it is defined terminates
- **Initialized only once**

```
#include <stdio.h>
void main() {
    int i = 0;
    for(i = 0; i < 3; i++)
    {
        static int y = 0;
        y += 10;
        printf("y = %d\t",y);
    }
}
```

Local Static Variables

```
#include <stdio.h>
void main() {
    int i = 0;
    for(i = 0; i < 3; i++) {
        int y = 0;
        y += 10;
        printf("y = %d\t",y);
    }
}
```

Output: 10 10 10

```
#include <stdio.h>
void main() {
    int i = 0;
    for(i = 0; i < 3; i++) {
        static int y = 0;
        y += 10;
        printf("y = %d\t",y);
    }
}
```

Output: 10 20 30

Local static variable

- Retains its value between function calls or block
- Remains visible only to the function or block in which it is defined.
- It remains even after the function terminates

Local Static Variables – Another Example

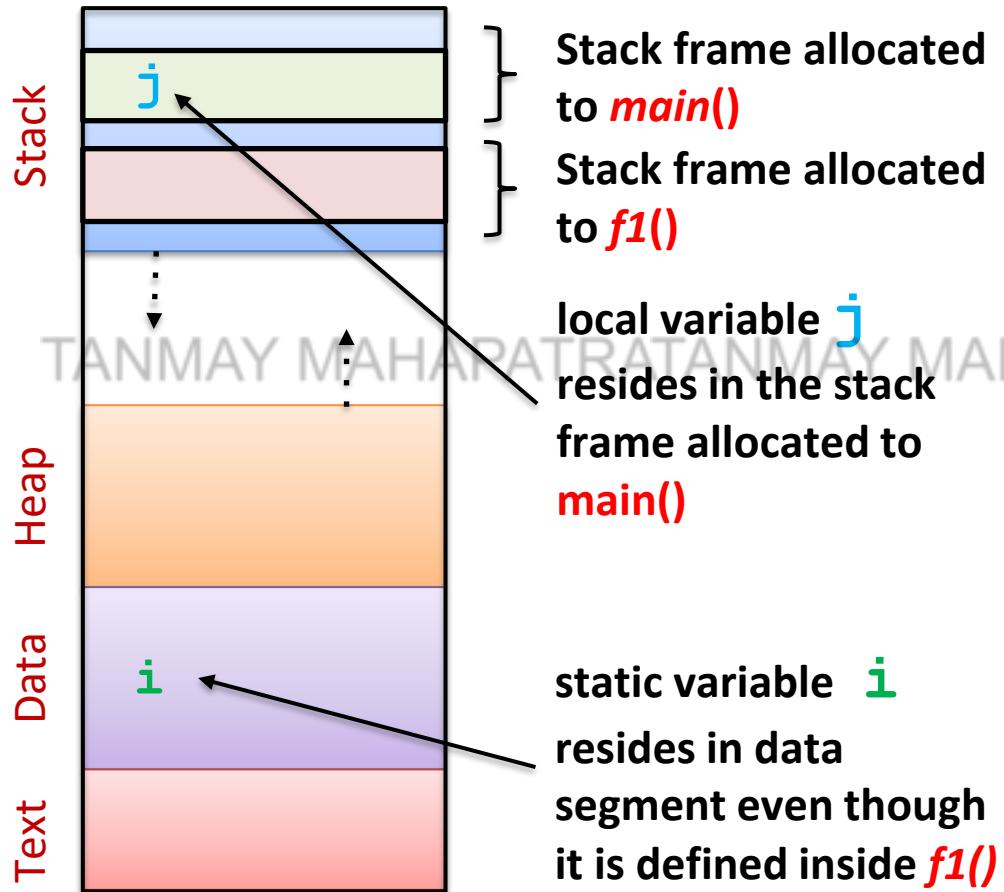
```
#include <stdio.h>
void f1() {
    int i = 0;
    i = i+10;
    printf("i = %d\t",i);
}
void main() {
    int i = 0;
    for(i = 0; i < 3; i++)
        f1();
}
```

Output: 10 10 10

```
#include <stdio.h>
void f1() {
    static int i = 0;
    i = i+10;
    printf("i = %d\t",i);
}
void main() {
    int i = 0;
    for(i = 0; i < 3; i++)
        f1();
}
```

Output: 10 20 30

Local Static Variables in Memory



```
#include <stdio.h>
void f1() {
    static int i = 0;
    i = i+10;
    printf("i = %d\t", i);
}
void main() {
    int j = 0;
    for(j = 0; j < 3; j++)
        f1();
}
```

Note: The stack frame for **f1()** is **created and destroyed 3 times**. But variable **i** is **created only once** and **stored in the data segment**.

Global Static Variables

- **Scope:** **Visible to all the functions** in a program
- **Initial Value:** By default, initialized to 0
- **Storage Location:** Stored in the **Data Segment**
- **Lifetime:** **Until the program terminates.**

• **Initialized only once**

- **Difference with global storage class:** While static global variables are visible only to the file in which it is declared, global variables can be used in other files as well.
 - ***We will study more about multi-file compilation in upcoming lab sessions.***

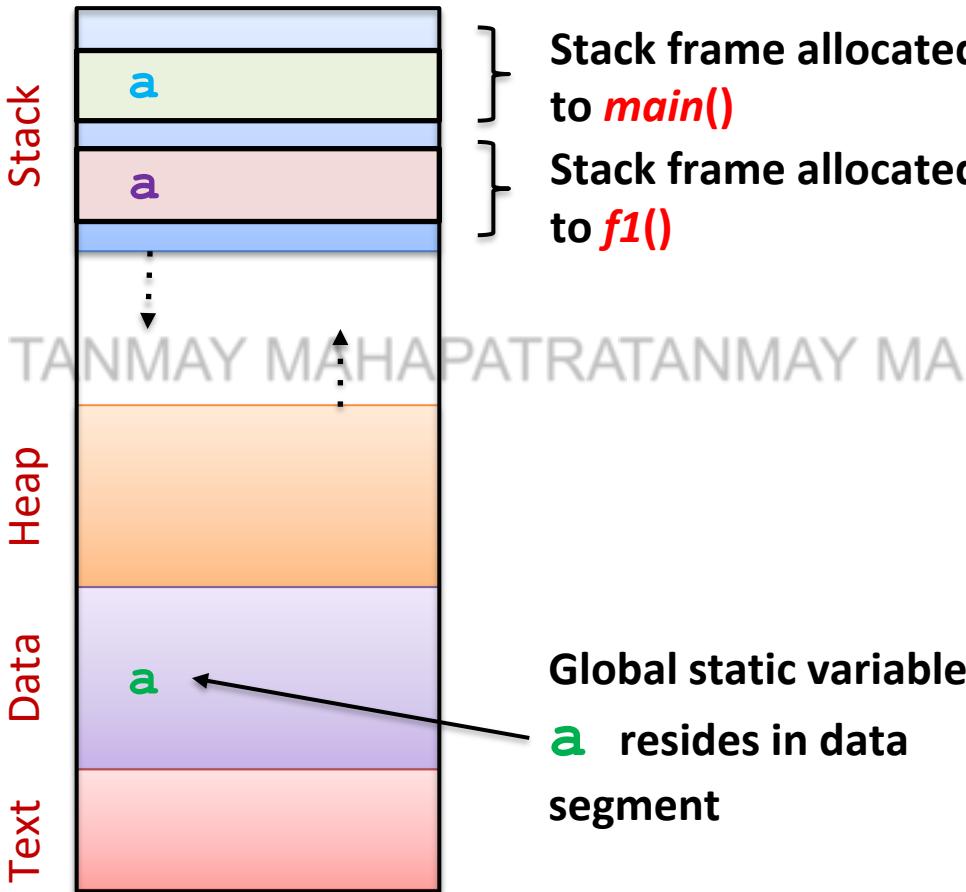
Example

```
#include <stdio.h>
static int a; //initialized
void f1(){
    printf("a = %d\n",a);
    int a = 2;
    printf("a = %d\n",a);
}
void main() {
    f1();
    int a = 5;
    printf("a = %d",a);
}
```

output:

- a = 0 ← value of **static** global variable
- a = 2 ← value of **auto** variable a local to f1
- a = 5 ← value of **auto** variable a local to main

Memory Allocation of Static global Variable



```
#include <stdio.h>
static int a; //initialized
void f1() {
    printf("a = %d\n", a);
    int a = 2;
    printf("a = %d\n", a);
}
void main() {
    f1();
    int a = 5;
    printf("a = %d", a);
}
```



Register Variables

Registers and Register variables

- A Register is very high-speed memory storage that is located in the CPU.
- Typically limited in number ([16 registers for Intel i7 Processor](#))
- They are extremely fast to access when compared to main memory (RAM).
- The variables that are most frequently accessed can be put into registers using the *register* keyword.
- The keyword *register* hints to compiler that a given variable can be put in a register.
- It's compiler's choice to put it in a register or not.
- Generally, compilers themselves do some optimizations and put the variables in register (even when declared without the register keyword).

Register Variables

- **Scope** – They are local to the function.
- **Default value** – Default initialized value is the garbage value.
- **Lifetime** – Till the end of the execution of the block in which it is defined.

Example:

```
#include <stdio.h>
int main(){
    register char x = 'S';      register int a = 10;
    int b = 8;
    printf("The value of register variable b : %c\n",x);
    printf("The sum of auto and register variable: %d", (a+b));
    return 0;
}
```

Output:

The value of register variable b : S
The sum of auto and register variable : 18

Caveats

- You can't access address of a register variable with an “&”
 - *Can't use register variables with scanf().*
- Register is a storage class, and C doesn't allow multiple storage class specifiers for a variable.
 - *register can not be used with static or extern.*
- You can't declare global register variables
 - *All register variable must be declared within the functions.*
- There is no limit on number of register variables in a C program.
 - *Compiler may put some variables in register and some not depending upon availability as number of registers is limited.*



Thank you
Q & A

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA



TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA



Module 7 – part 3 – Recursion

BITS Pilani

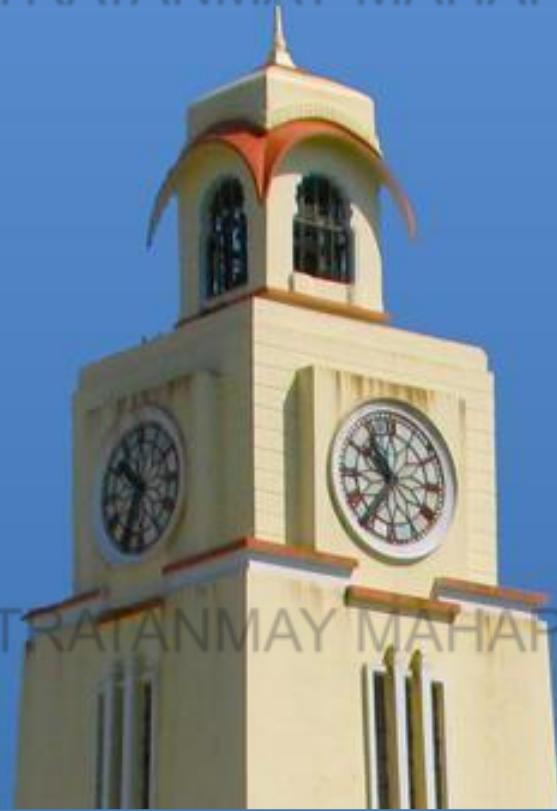
Pilani Campus

Department of Computer Science & Information Systems

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA

Module Overview

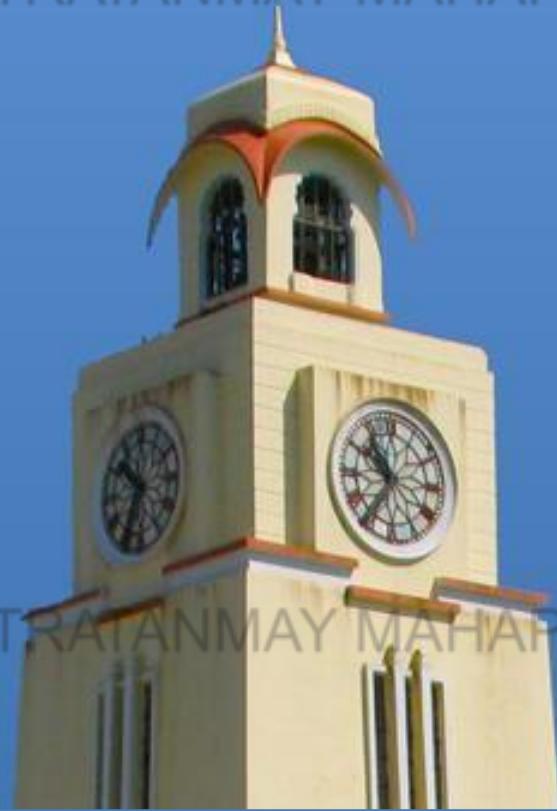
- **Recursion**
- **Recursive Functions**
- **Examples**



Recursion

Recursion

- Recursion is the process of defining something in terms of itself, and is sometimes called *circular definition*.
- A recursive process is one in which objects are defined in terms of other objects of the same type.
- The entire class of objects can then be built up from a few initial values and a small number of rules.
- Ex: Pingala sequence? - *mātrāmeru* (Fibonacci Number series)



Recursive Functions

Recursive functions

- A recursive function is one which calls itself (repeatedly)
- Recursive functions are useful in evaluating certain types of mathematical function. (we'll see examples)

Simple Example:

```
main()
{
    printf("This is an example of recursive function");
    main();
}
```

When this program is executed. The line is printed repeatedly and indefinitely. We might have to abruptly terminate the execution.

Recursive functions contd..

- Solves a problem by calling a copy of itself to work on a smaller problem.
- It is important to ensure that the recursion terminates.

Recursive vs iterative code:

- Recursive code is generally shorter and easier to write than iterative code
- Solution to some problems are easier to formulate recursively

Factorial Calculation

```
long int fact(int n)      /* iterative */
{
    int t, ans;
    ans = 1;
    for(t=1; t<=n; t++)
        ans = ans*t;
    return(ans);
}
```

```
long int factr(int n)   /* recursive */
{
    int ans;
    if(n==1)
        return(1);
    ans = n*factr(n-1);
    return(ans);
}
```

Base case

Recursive call

Power function (Calculate non negative power of a number)

```
double power(double val, unsigned pow)
{
    if(pow == 0) /*pow(x, 0) returns 1*/
        return(1.0); ←
    else
        return(val*power(val, pow-1)); →
}
```

Base case

Recursive call

Recursive Version of Fibonacci Series

```
int fib(int num)      /*Fibonacci value of a  
number */  
{  
    switch (num)  
    { case 0: return (0);  
        break;  
    case 1: return (1);  
        break;  
    default:          /*Including recursive  
calls */  
        return (fib(num - 1) + fib(num - 2));  
        break;  
    }  
}
```

The diagram features two callout boxes. A green box labeled 'Base cases' has an arrow pointing to the two `return` statements for `case 0` and `case 1`. An orange box labeled 'Recursive call' has an arrow pointing to the red-colored line of code `return (fib(num - 1) + fib(num - 2));`.

Analysis

Input Value	Number of time fib is called
0	1
1	1
2	3
3	5
4	9
5	15
6	25
7	41
8	67
9	109

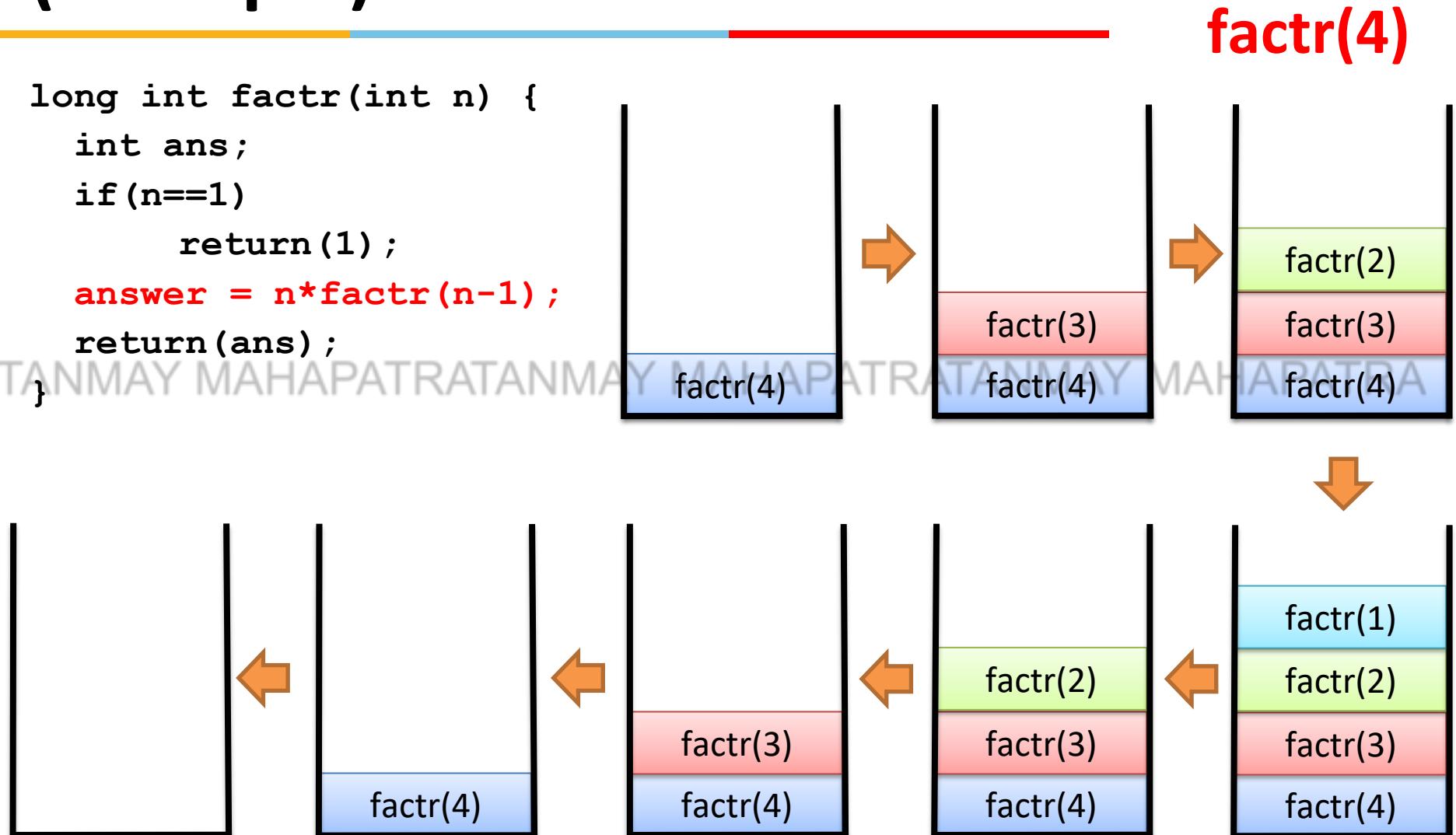
How recursive functions work???



- When a function calls itself, a new set of local variables and parameters are allocated storage on the stack, and the function code is executed from the top with these new variables.
- A recursive call does not make a new copy of the function. Only the arguments are new.
- As each recursive call returns, the old local variables and parameters are removed from the stack and execution resumes at the point of the function call inside the function.

Recursive Call Stack (Example)

```
long int factr(int n) {  
    int ans;  
    if(n==1)  
        return(1);  
    answer = n*factr(n-1);  
    return(ans);  
}
```



Another Example

```
int main()
{
    static int i=5;
    if (--i) {
        printf("%d ",i);
        main();
    }
}
```

4 3 2 1

Cons of recursion

- No significant reduction in code size as well as no improvement in memory utilization.
- Also, the recursive versions of most routines may execute a bit slower than their iterative equivalents because of the overhead of the repeated function calls.
- In fact, many recursive calls to a function could cause a stack overrun.

Pros of recursion

- The main advantage to recursive functions is that you can use them to create clearer and simpler versions of several algorithms.
- For example, the MergeSort and the Quicksort (**two popular sorting techniques**) are quite difficult to implement in an iterative way. Also, some problems, especially ones related to artificial intelligence, lend themselves to recursive solutions.
- Finally, some people seem to think recursively more easily than iteratively.

Precaution

- When writing recursive functions, you must have an **if** statement somewhere to force the function to return without the recursive call being executed.
- If you don't, the function will never return once you call it.



Thank you
Q & A

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA



TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA



Module 8 – part 1– Arrays in C

BITS Pilani

Pilani Campus

Department of Computer Science & Information Systems

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA

Module Overview

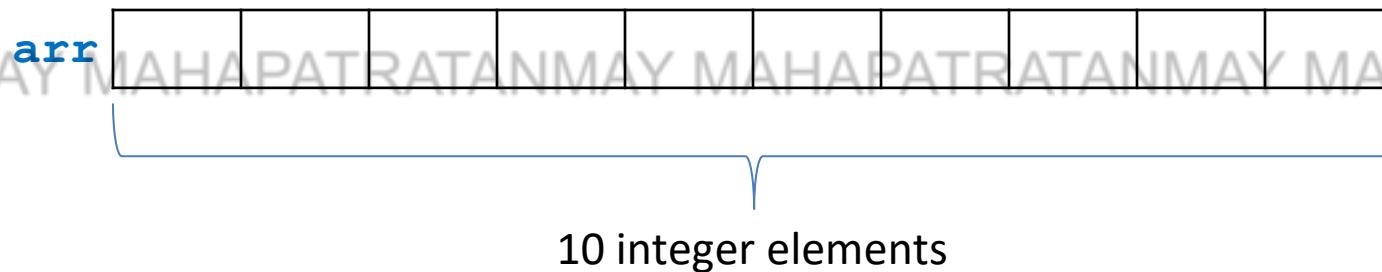
- **Introduction to Arrays in C**
- **Arrays in Memory**
- **A few examples**
- **Passing arrays to functions**
- **Searching in an Array**
- **Selection Sort**
- **Binary Search**
- **Insert/Delete in an Array**
- **Character Arrays**
- **Multi-dimensional Arrays**
- **Matrix Addition and Multiplication**



Intro to Arrays in C

What are Arrays?

- Array – fixed size sequenced collection of similar elements of the same data type.
- It is a derived data type.
- Example:
 - `int arr[10]` declares an array of 10 elements each of integer type



- Array is also known as **subscript variable**, e.g., recall m_1, m_2, \dots, m_{10} used in mathematics.
- Single name is used to represent a collection of items.
 - *arr represents 10 integer elements in the above array*
- Arrays are useful in processing multiple data items having a common characteristic
 - E.g.: set of numerical data, list of student names, etc.

Importance of Arrays

- Easier storage, access, and data management
- Easier to search
- Easier to organize data elements
- Useful to perform matrix operations
- Useful in databases
- Useful to implement other data structures

Defining Arrays

Syntax:

<Storage-class> <data-type> <array_name>[<size>]

Note: Storage-class is optional

Examples:

```
int count[100];  
char name[25];  
float cgpa[50];
```

Good practice:

```
#define SIZE 100  
int count[SIZE];
```

Initializing arrays

Few valid ways of initializing arrays in C

```
int intArray[6] = {1, 2, 3, 4, 5, 6};
```

1	2	3	4	5	6
---	---	---	---	---	---

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA

```
float floatArray[10] = {1.387, 5.45, 20.01};
```

1.387	5.45	20.01	0.0	0.0	0.0	0.0	0.0	0.0	0.0
-------	------	-------	-----	-----	-----	-----	-----	-----	-----

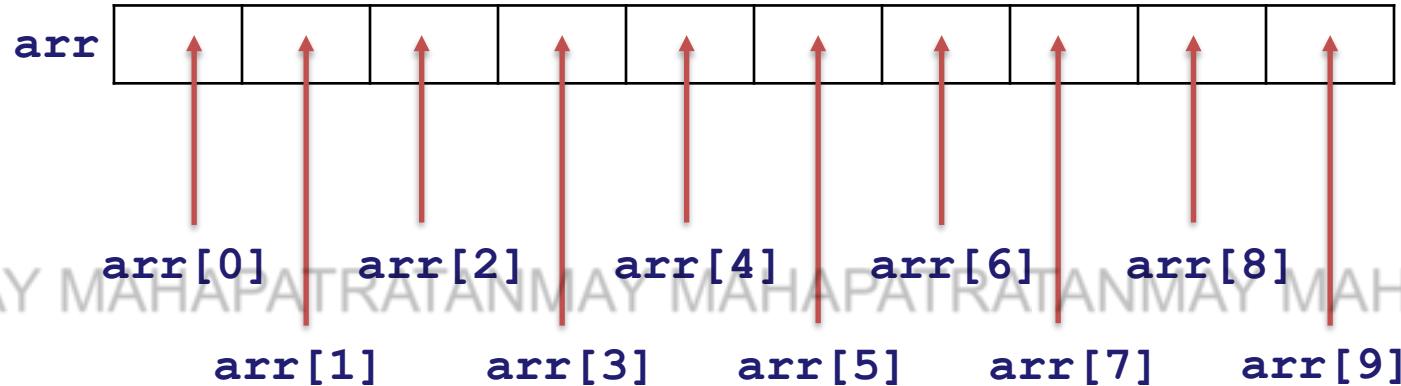
```
double fractions[] = {3.141592654, 1.570796327, 0.785398163};
```

3.141592654	1.570796327	0.785398163
-------------	-------------	-------------

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA

Accessing elements of an array

```
int arr[10];
```



For an n -element array:

- The **first element** is accessed by $\text{arr}[0]$
- The **second element** is accessed by $\text{arr}[1]$
- The **last element** is accessed by $\text{arr}[n-1]$

Initializing Arrays During Program Execution

```
#include <stdio.h>
int main()
{
    int nums[10]; int i;
    for(i=0; i<10; i++)
    {
        /* Reading an array of elements */
        scanf("%d", &nums[i]);
    }
}
```

Example

A program that takes 10 elements from the user and stores them in an array and then computes their sum.

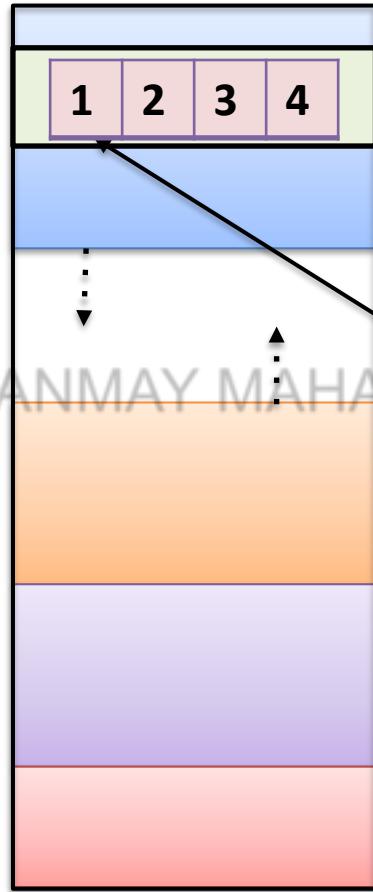
```
#include <stdio.h>
int main() {
    int nums[10]; int sum=0; int i;
    for(i=0; i<10; i++) {
        scanf("%d", &nums[i]);
    }
    for(i=0;i<10;i++) {
        sum = sum + nums[i];
    }
    printf("The sum is: %d", sum);
}
```



Arrays in Memory

Arrays in memory

Stack



} Frame allocated to
main() in the stack

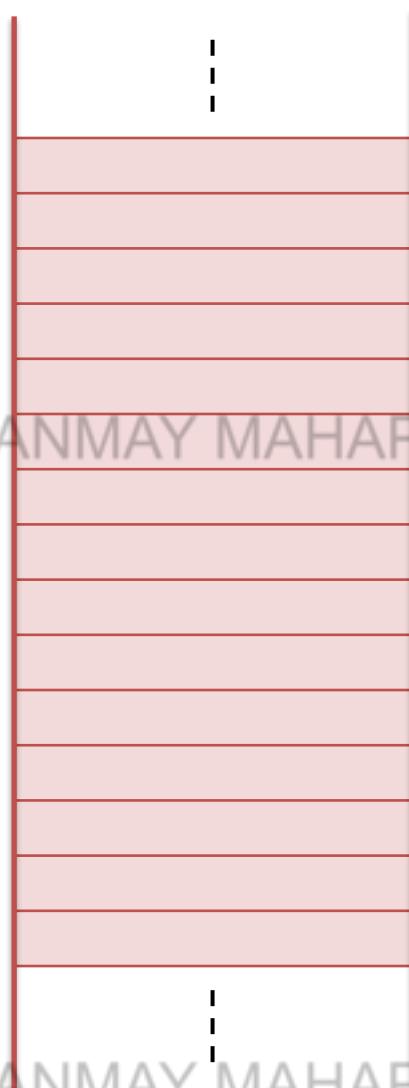
Array ***arr*** residing in
the frame allocated to
main()

Consider this program:

```
int main() {
    int arr[4] = {1, 2, 3, 4};
    ...
    return 0;
}
```

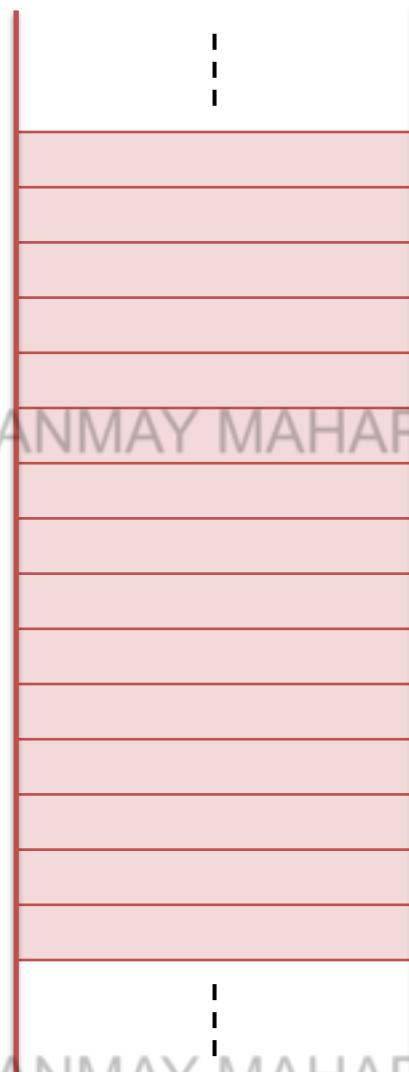
- Elements of `arr` are stored in contiguous memory locations
- `arr` references the first element in the array.
- In other words, the variable `arr` contains the *address of the first location/element* of the 4 elements array that we have just defined.
- ***What is this address?***
- ***We will see***

Addresses in Main Memory



- Remember how our memory is actually organized
- It is like a table of contiguous memory locations, each having an address
 - Each address is represented by a string of bits
 - *typically 32 or 64 bits in modern computers*

Addresses in Main Memory



- Note: our memory is byte addressable.
 - every byte in the memory has an address,
 - In other words, size of each memory location is 1 byte.
- For Simplicity let us consider each address in the main memory to be represented by 8 bits.
- The addresses of the memory locations are depicted in the figure
- *Note: The addresses of two contiguous locations differ by 1 (in binary)*

How is an array stored in main memory?

Addresses

10010000

10010001

10010010

10010011

10010100

10010101

10011101

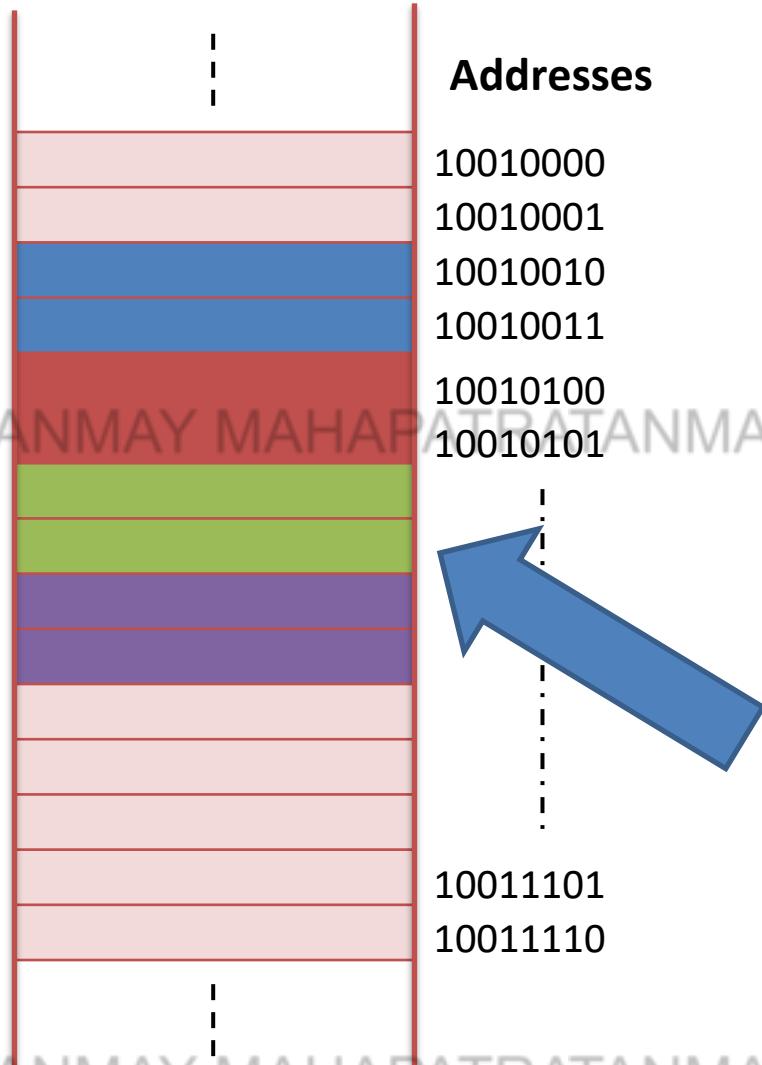
10011110

Consider this program:

```
int main() {  
    int arr[4] = {1, 2, 3, 4};  
    ...  
    return 0;  
}
```

- Say each integer variable takes 2 bytes of memory
- To store an integer array of size 4, we need $4 * 2 = 8$ bytes of memory or 8 contiguous locations in memory (given each location is of size 1 byte)

How is an array stored in main memory?



Consider this program:

```
int main() {  
    int arr[4] = {1, 2, 3, 4};  
    ...  
    return 0;  
}
```

- Say each integer variable takes 2 bytes of memory
- To store an integer array of size 4, we need $4 * 2 = 8$ bytes of memory or 8 contiguous locations in memory (given each location is of size 1 byte)

How is an array stored in main memory?

```
int arr[4] = {1, 2, 3, 4};
```

Stores first element of arr, accessed by **arr[0]**

{

Stores second element of arr, accessed by
arr[1]

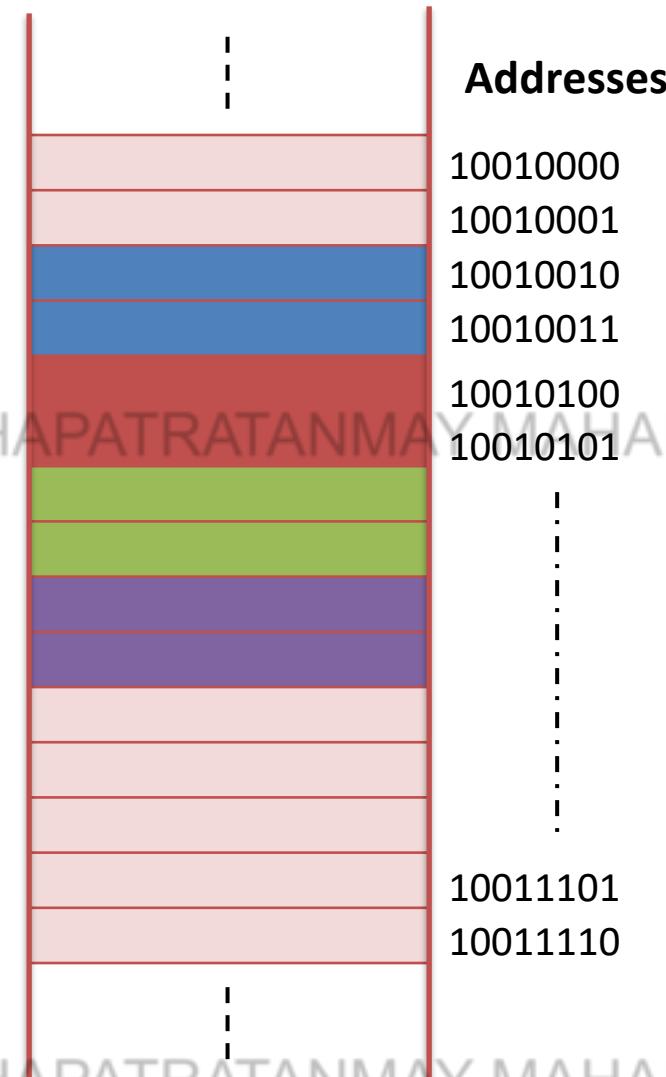
{

Stores third element of arr, accessed by **arr[2]**

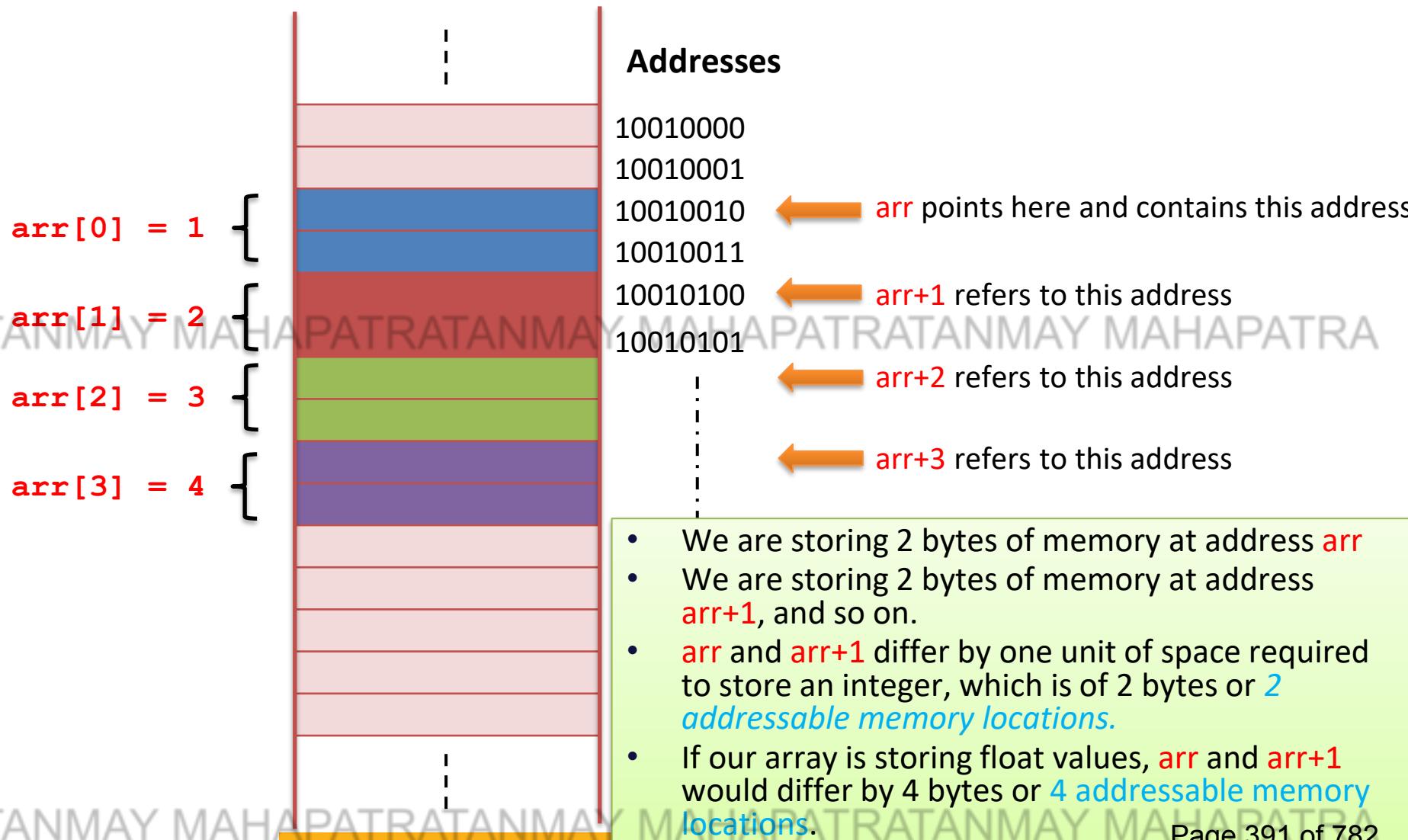
{

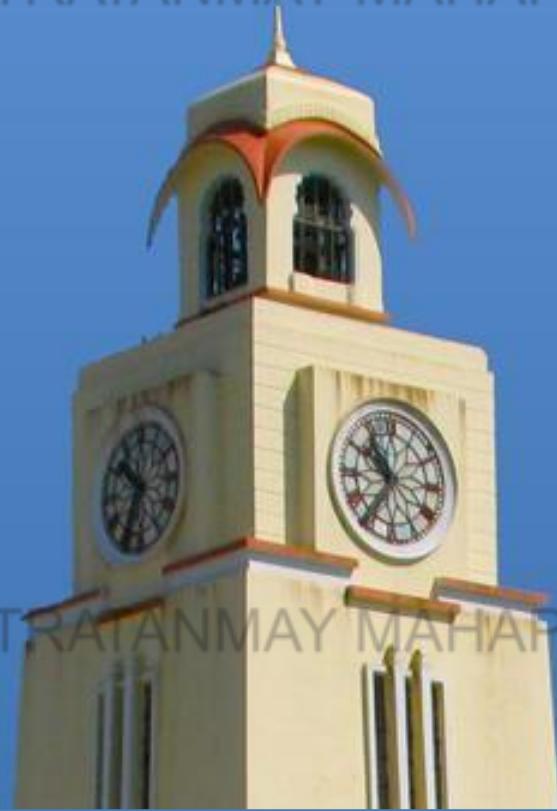
Stores fourth element of arr, accessed by **arr[3]**

{



How is an array stored in main memory?





A few examples

Example 1

```
int main() {  
    int arr[6] = {1, 2, 3, 4, 5, 6};  
    for(int i=0;i<sizeof(arr)/sizeof(arr[0]); i++)  
        printf("%d\t",arr[i]);  
  
    return 0;  
}
```

Output?

1 2 3 4 5 6

Example 2

```
int main() {  
    int arr[6] = {1, 2, 3};  
    for(int i=0;i<sizeof(arr)/sizeof(arr[0]);i++)  
        printf("%d\t",arr[i]);  
    return 0;  
}
```

Output?

1 2 3 0 0 0

Example 3

```
int main() {  
    int arr[] = {1, 2, 3};  
    for(int i=0;i<sizeof(arr)/sizeof(arr[0]);i++)  
        printf("%d\t",arr[i]);  
    return 0;  
}
```

Output?

1 2 3

Example 4

```
int main() {  
    int arr[6];  
    arr[6] = {1, 2, 3, 4, 5, 6};  
    for(int i=0;i<sizeof(arr)/sizeof(arr[0]);i++)  
        printf("%d\t",arr[i]);  
    return 0;  
}
```

Output?

Compile-time error

What is the right way in such a scenario (declaration and initialization are separated) ?

Example 5

```
int main() {  
    int arr[6];  
    for (int i=0;i<6;i++)  
        arr[i] = i;  
    for(int i=0;i<6;i++)  
        printf("%d\t",arr[i]);  
    return 0;  
}
```

Output?

0 1 2 3 4 5

Example 6

```
int main() {  
    int arr[6];  
    for (int i=0;i<6;i++)  
        arr[i] = i;  
  
    for(int i=0;i<9;i++)  
        printf("%d\t",arr[i]);  
    return 0;  
}
```

Output?

0 1 2 3 4 5 3965 -8905 4872

(junk values)

Example 7

```
int main() {  
    int arr[6];  
    for (int i=0;i<8;i++)  
        arr[i] = i;  
    for (int i=0;i<6;i++)  
        printf("%d\t",arr[i]);  
    return 0;  
}
```

Output?

```
*** stack smashing detected ***: ./a.out terminated  
0      1      2      3      4      5      Aborted (core dumped)
```

Problems with arrays – No bounds checking!!

- There is no check in C compiler to see if the subscript used for an array exceeds the size of the array.
- Data entered with a subscript exceeding the array size will simply be placed in memory outside the array limit and lead to unpredictable results.
- It's the programmer's responsibility to take care.

Example 8

```
int main() {  
    int arr[6]={0};  
    arr[6]= 1;  
    for (int i=0;i<7;i++)  
        printf("%d\t",arr[i]);  
    return 0;  
}
```

Output?

```
*** stack smashing detected ***: ./a.out terminated  
0      0      0      0      0      1  
Aborted (core dumped)
```

Example 9

Write a C code which takes an int array of size SIZE and calculates (and displays) the average of all numbers

```
#include<stdio.h>
#include<stdlib.h>

#define SIZE 9
int main()
{
    int arr[SIZE], sum=0;
    for (int i=0;i<SIZE;i++)
        arr[i] =i;
    for (int i=0;i<SIZE;i++)
        sum = sum + arr[i];
    printf("AVG=%d\n",sum/SIZE);
    return 0;
}
```

Copying Arrays

```
int old_value[5]={10,20,30,40,50};
```

```
int new_value[5];
```

How to copy the elements of **old_value** into **new_value**?

```
new_value = old_value;
```

It will not work. Why?

```
Simple.c:6:11: error: assignment to expression with array type
      6 | new_value = old_value;
          |           ^
```

Question:

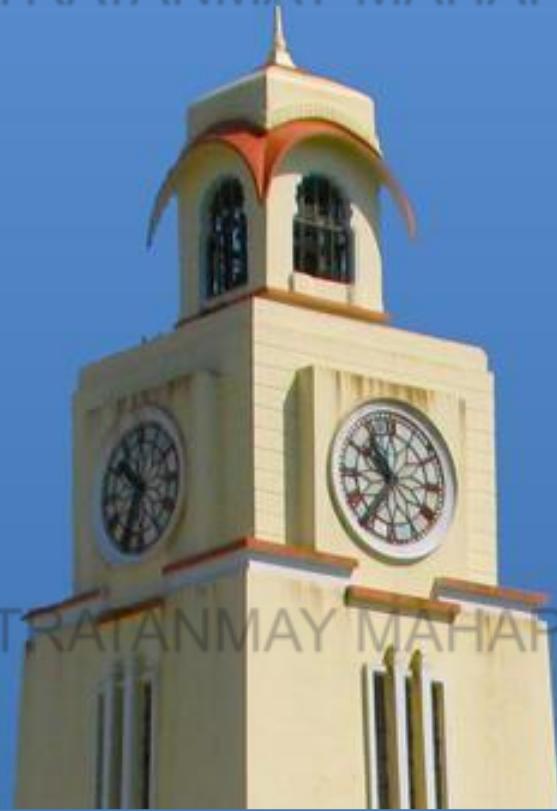
What is the correct way of copying arrays?

Sol:

Copy each individual element one by one

Home Exercise

Write a C program to find out the largest element in an array of integers. Assume N numbers are entered by the user (N is #defined)



Passing Arrays to functions

Passing arrays to functions

USING FUNCTIONS:

Write a C function that takes an int array (integers from 0 to SIZE-1) of size **SIZE** and calculates (and displays) the average of all numbers

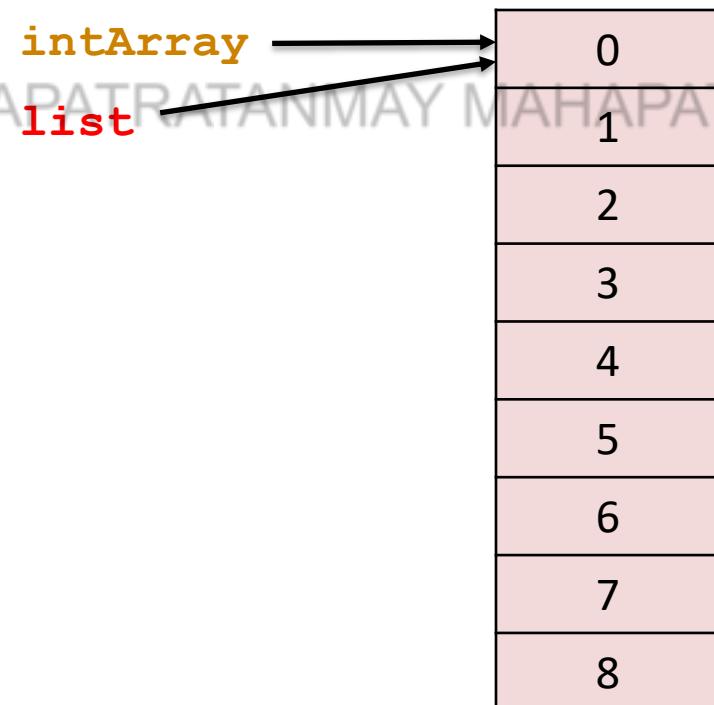
Solution

```
#define SIZE 9
int avg(int n, int list[]) {
    int sum = 0;
    for (int i=0;i<n;i++)
        sum = sum + list[i];
    return sum/n;
}
int main() {
    int intArray[SIZE];
    int average;
    for (int i=0;i<SIZE;i++)
        intArray[i] = i;
    average=avg(SIZE, intArray);
    printf("Average=%d\n", average);
    return 0;
}
```

Output:

Average=4

When **avg()** is called, the address of the first element of the array **intArr** is copied into the array **list**.



Slight change...

```

int main(){
    int intArray[SIZE], average;
    for (int i=0;i<SIZE;i++) {
        intArray[i] = i;
        printf("%d\t",intArray[i]);
    }
    average=avg(SIZE,intArray);
    printf("Average=%d\n", average);
    printf("After calling avg...\n");

    for (int i=0;i<SIZE;i++)
        printf("%d\t",intArray[i]);
    return 0;
}
  
```

```

int avg(int n, int list[])
{
    int sum = 0;
    for (int i=0;i<n;i++) {
        list[i] = list[i]*2
        sum = sum + list[i];
    }
    return sum/n;
}
  
```

Output:

0 1 2 3 4 5 6 7 8

Average = 8

After calling avg...

0 2 4 6 8 10 12 14 16

Arguments to functions

Ordinary variables are **passed by value**

- *Values of the variables passed are copied into local variables of the function*

However... when an array is passed to a function

- Values of the array are **NOT** passed
- Array name interpreted as the address of the first element of the array is passed **[Pass by reference]**
- This value is captured by the corresponding function parameter, which becomes a **Pointer** to the first element of the array that was passed to it.

Therefore, altering **list[i]** within **avg()** altered the original values of **intArray[i]**.

Can we ‘return’ an array from a function?

If the array is defined inside the function, returning the array would give run-time error.

- Why?
- Try this! Next Slide.

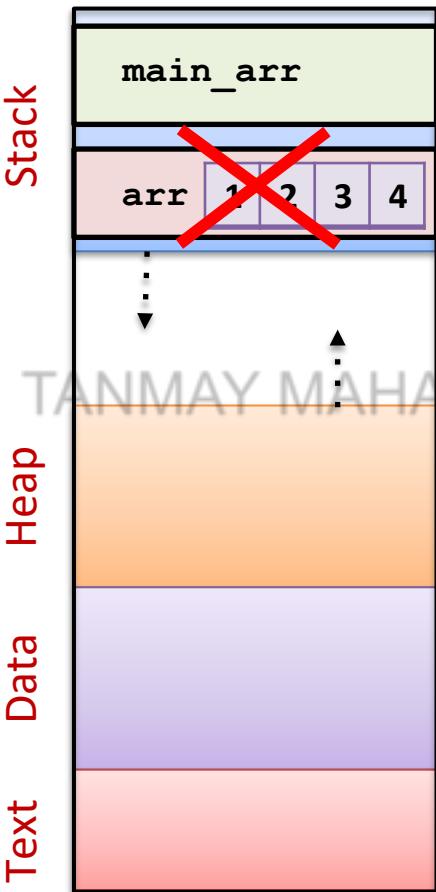
We can return dynamically allocated arrays.

- We will study about them when we study pointers.

We can also use global arrays in place of arrays defined inside functions.

- We don’t have to pass arrays into functions. Global arrays are accessible to all the functions in the program.

Let us try to explain with our memory diagram



} Stack frame allocated to
main() in stack
} Stack Frame allocated to
f1() in the stack

When f1() returns, the memory allocated to it is destroyed.
So arr declared inside f1() does not exist anymore. Accessing arr in main function now gives an error.

Consider this program:

```
int f1() {
    int arr[4] = {1, 2, 3, 4};
    return arr;
}
int main() {
    int main_arr[] = f1();
    printf("First ele is: %d", main_arr[0]);
    return 0;
}
```

Output:

Error! Invalid array declaration in main()

Memory allocated to our program

Example with global arrays

```
#define SIZE 4
int globArray[SIZE];
// int globalArray[SIZE] = {1,2,3,4} is also allowed
int avg(){
    int sum = 0;
    for (int i=0;i<n;i++)
        sum = sum + globArray[i];
    return sum/n;
}
int main(){
    int average;
    for (int i=0;i<SIZE;i++)
        globArray[i] = i;
    average=avg();
    printf("Average=%d\n", average);
    return 0;
}
```

Example (Worked on the Board)

Write a C function which accepts an array as the input and returns the index of the largest element in the array. Assume **N** numbers are entered by the user (**N is #defined**)



Searching in an Array

Linear search

Task: Search for an element **key** in the Array.

Each item in the array is examined until the desired item is found or the end of the list is reached

Algorithm:

1. Read an array of **N** elements named **arr[0...N-1]** and search element **key**
2. Repeat **for i=0 to i=N-1**
 - If **key equals to arr[i]**
 - Display element found and stop
 - 3. Display element not found
 - 4. Stop

Linear search

Array	6	3	0	5	1	2	8	-1	4
-------	---	---	---	---	---	---	---	----	---

Element to search: 8

Linear Search – Implementation

```
#include <stdio.h>

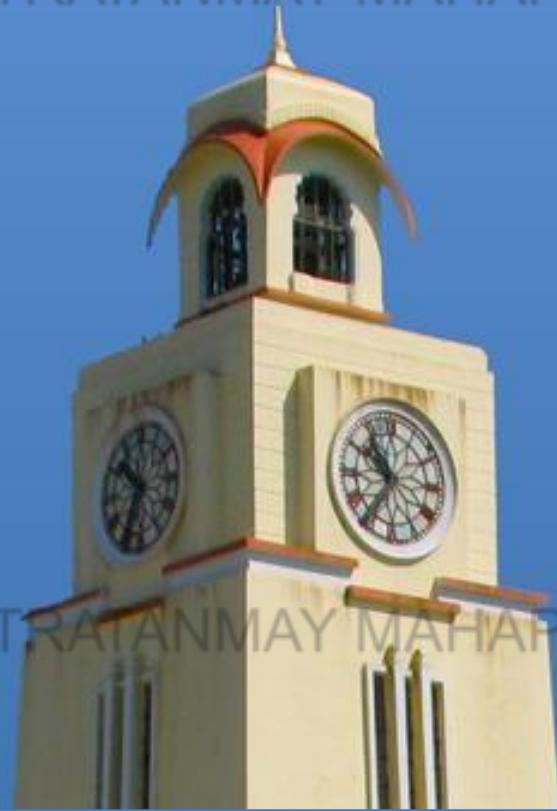
int linearSearch(int arr[], int size, int key)
{
    // Function implementing linear search of key in array
    // arr of size size
    int i = 0;

    for(i=0; i<size; i++) {
        if(key == arr[i])
            return i; // element found at index i in the array
    }

    return -1; // element not found in the array
}
```

Linear Search – Implementation (contd.)

```
int main() {  
    int arr[10],pos,key;  
    printf("Enter Array elements:");  
    for(index = 0; index<10; index++)  
        scanf("%d",&arr[index]);  
    printf("Enter search element");  
    scanf("%d", &key);  
  
    pos = linearSearch(arr,10,key);  
  
    if (pos == -1)      printf("Element not found");  
    else              printf("Element found at index %d\n",pos);  
  
    return 0;
```



Sorting – Selection Sort

Sorting

- Sorting refers to ordering data in an *increasing* or *decreasing* order
- Sorting can be done by
 - Names
 - Numbers
 - Records
 - etc.
- *Sorting reduces the time for lookup (or search for an element)*
- Example: Telephone directory
 - Time to search for someone's phone number
 - Directory sorted alphabetically vs. no ordering

Various Algorithms for Sorting

- Selection sort
- Bubble sort
- Insertion sort
- Merge sort
- Quick sort
- Shell sort
- Bucket sort
- Radix sort
- and more . . .

We would be studying
in this course!

Sorting (Selection Sort)

- Selection sort is a simple sorting algorithm.
- In this algorithm, the list (or an array) is divided into two parts:

- *The sorted part at the left*
- *The unsorted part at the right*

- *Initially, the sorted part at the left is empty, and the unsorted part at the right is the full list.*
- *In each iteration, the smallest element from the unsorted part of the array is added to the sorted part.*
- *The process continues until the unsorted part of the array is empty.*

Illustrating Selection Sort

14	33	27	10	35	19	42	44
----	----	----	----	----	----	----	----

Find the minimum element in the array and bring to the 0th index of the array

14	33	27	10	35	19	42	44
----	----	----	----	----	----	----	----

Element 10 is the minimum element in the array. To bring it the 0th index, we need to swap it with the element present at the 0th index

10	33	27	14	35	19	42	44
----	----	----	----	----	----	----	----

We can now see that the sorted portion of the array is from index 0 to 0, and the unsorted portion of the array is from index 1 to 7.

Illustrating Selection Sort (Cont.)

innovate

achieve

lead

10	33	27	14	35	19	42	44
----	----	----	----	----	----	----	----

Now, we have to look at the array excluding the first element (element at 0th index). In the remaining (unsorted) portion of the array, again we will find the minimum element.

10	33	27	14	35	19	42	44
----	----	----	----	----	----	----	----

The minimum element in the unsorted portion of the array is 14 at index 3. We should swap it with the element at index 1.

10	14	27	33	35	19	42	44
----	----	----	----	----	----	----	----

We can now see that the sorted portion of the array is from index 0 to 1, and unsorted portion of the array is from index 2 to 7.

Illustrating Selection Sort (Cont.)

In the same way, we can continue this process. In the end, the sorted portion will be the entire array, and the unsorted portion will be empty.

10	14	27	33	35	19	42	44
----	----	----	----	----	----	----	----

10	14	19	33	35	27	42	44
----	----	----	----	----	----	----	----

10	14	19	27	35	33	42	44
----	----	----	----	----	----	----	----

10	14	19	27	33	35	42	44
----	----	----	----	----	----	----	----

10	14	19	27	33	35	42	44
----	----	----	----	----	----	----	----

10	14	19	27	33	35	42	44
----	----	----	----	----	----	----	----

10	14	19	27	33	35	42	44
----	----	----	----	----	----	----	----

Selection Sort: Implementation

```
#include <stdio.h>
void selectionSort(int arr[], int n) {
    int i, j, min;

    // One by one move the boundary of the unsorted subarray
    for (i = 0; i < n-1; i++) {
        min = i; //minimum element in unsorted array

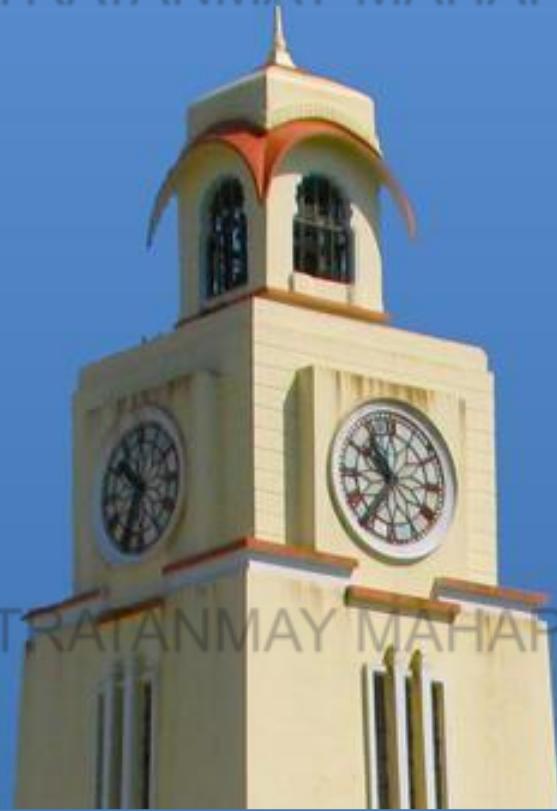
        for (j = i+1; j < n; j++) {
            if (arr[j] < arr[min]) min = j;
        }

        // Swap the min element with the first element in the unsorted subarray
        int temp = arr[min];
        arr[min] = arr[i];
        arr[i] = temp;
    }
}
```

Does this function need to return anything?

Implementation (contd.)

```
int main()
{
    int a[] = {24, 36, 20, 7, 42, 19};
    int size = sizeof(a) / sizeof(a[0]);
    selectionSort(a, size);
    return 0;
}
```



Binary Search in an array

Binary Search

- Useful when the array is already sorted
- More efficient than linear search
- It performs lesser number of comparisons with the elements in the array, when compared to linear search
- Hence, very fast!

Binary Search Algorithm

Algorithm // Pre-condition: List must be sorted

- The desired item is first compared to the element in the middle of the list
- If the desired item is equal to the middle element:
 - No further searches are required
- If the desired item is greater than the middle element:
 - The left part of the list is discarded from any further search
- If the desired item is less than the middle element:
 - The right part of the list is discarded from any further search
- This process continues either until element is found or list reaches to singleton element

Illustrating Binary Search

1. The array in which searching is to be performed is:



Let $x = 4$ be the element to be searched.

2. Set two pointers **low** and **high** at the lowest and the highest positions respectively.



Illustrating Binary Search

- Find the position of the middle element $mid = (\text{low}+\text{high})/2$. The middle element is $\text{arr}[mid] = 6$.

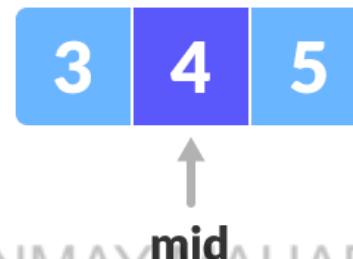


- If $x==\text{arr}[\text{mid}]$, then **return mid**.
- Else If $x > \text{arr}[\text{mid}]$, **compare x with the middle element of the elements on the right side of mid**. This is done by setting low to $\text{low} = \text{mid} + 1$.
- Else If $x < \text{arr}[\text{mid}]$, **compare x with the middle element of the elements on the left side of mid**. This is done by setting high to $\text{high} = \text{mid} - 1$.



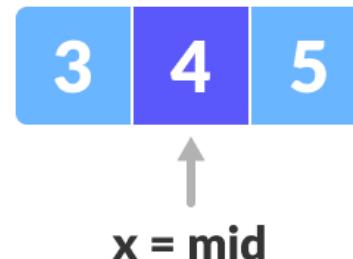
Illustrating Binary Search

7. Repeat steps 3 to 6 until low meets high.



TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA

7. $x = 4$ is found.



TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA

Binary Search - Code

```
#define Size 10
main() {          // Binary search implementation
    int arr[Size], index, upper, lower, key, mid;
    printf("Enter Array elements:");
    for(index = 0; index<Size; index++)
        scanf("%d", &arr[index]);
    printf("Enter search element");
    scanf("%d", &key);
    upper=Size-1; lower=0;
    while(lower<=upper) {
        mid=(lower+upper)/2;
        if(key>arr[mid])
            lower=mid+1;
        else if(key<arr[mid])
            upper=mid-1;
        else{
            printf("Element found at location %d", mid);
            return;
        }
    }
    printf("Element no found");
}
```

Exercise: Re-write this program to do the searching part with a function call.



Insert/Delete in an Array

Handling changing number of elements in the array

Insert/Delete in an Array

- *Arrays once declared they are of fixed size.*
- Example:
`int arr1[10];` declares an array of size 10.
We can't store 11 elements to this array.
- To enable insertion and deletion in an array
 - Choose **MAX_SIZE** and declare the array with it.
 - Insertions can be done as long as the number of elements of the array does not exceed **MAX_SIZE**.
 - Keep a **count** of actual number of elements present in the array
 - **count <= MAX_SIZE**
 - Some positions will remain vacant. We shall need to store a **DEFAULT_VALUE** in those positions.
 - **Keep all occupied positions contiguous (Unoccupied positions as well)**
 - If **MAX_SIZE** is 10, **count** is 6, then first 6 positions of the array should be occupied with some values and remaining 4 positions should have **DEFAULT_VALUE**.

Insert/Delete in an Array

- Choose **MAX_SIZE** and declare the array with it.
- Insertions can be done as long as the number of elements of the array does not exceed **MAX_SIZE**.
- Keep a **count** of actual number of elements present in the array
 - count <= MAX_SIZE**
- Some positions will remain vacant. We shall need to store a **DEFAULT_VALUE** in those positions.
- Keep all occupied positions contiguous (Unoccupied positions as well)
 - If **MAX_SIZE** is 10, **count** is 6, then first 6 positions of the array should be occupied with some values and remaining 4 positions should have **DEFAULT_VALUE**.

```
#define MAX_SIZE 10
#define DEFAULT_VALUE -1
int main() {
    // declare array with MAX_SIZE
    int arr[MAX_SIZE];

    // declare a variable to keep count of
    // number of elements in the array
    int count = 0;

    // fill all the positions with the
    // DEFAULT_VALUE
    for(int i=0;i<MAX_SIZE;i++) {
        arr[i] = DEFAULT_VALUE;
    }
    ...
}
```

arr at this stage



Insert in an Array

```
#define MAX_SIZE 10
#define DEFAULT_VALUE -1
int main() {
int arr[MAX_SIZE];
int count = 0;

for(int i=0;i<MAX_SIZE;i++) {
    arr[i] = DEFAULT_VALUE;
}
```

// Insert 6 elements into the array.
Insert squares of their respective
index.

```
for (int i=0;i<6;i++){
    arr[i] = i*i;
    count+=1;
}
```

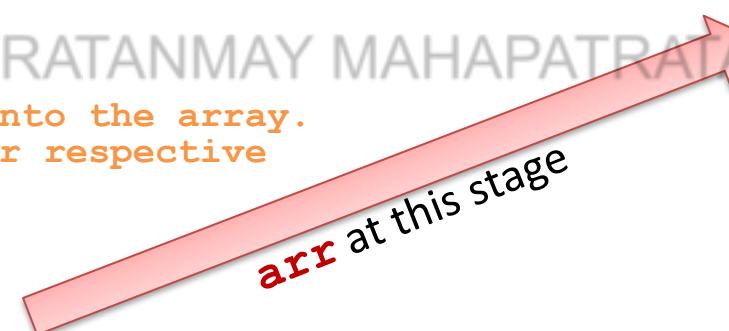
// Append an element at the end of
the array and increment count

```
arr[count]=12;
count+=1;
```

```
}
```

0	1	4	9	16	25	-1	-1	-1	-1
---	---	---	---	----	----	----	----	----	----

arr at this stage



Element inserted here

0	1	4	9	16	25	12	-1	-1	-1
---	---	---	---	----	----	----	----	----	----

arr at this stage

Insertion in a sorted array

```
#define MAX_SIZE 10
#define DEFAULT_VALUE -1
int main() {
int arr[MAX_SIZE];
int count = 0;
```

```
for(int i=0;i<MAX_SIZE;i++) {
    arr[i] = DEFAULT_VALUE;
}
for (int i=0;i<6;i++) {
    arr[i] = i*i;
    count+=1;
}
```

*// Insert 10 into arr, while keeping arr sorted
 // It should get inserted between 9 and 16.
 // What should be done ?
 // Shift 25, 16, by one place towards right and
 // insert 10 in the place of 16.*

```
}
```

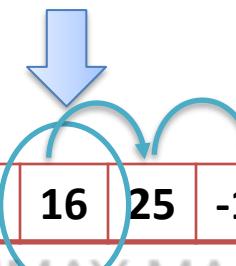
0	1	4	9	16	25	-1	-1	-1	-1
---	---	---	---	----	----	----	----	----	----



arr at this stage

Insert 10 here after shifting elements to the right

0	1	4	9	16	25	-1	-1	-1	-1
---	---	---	---	----	----	----	----	----	----



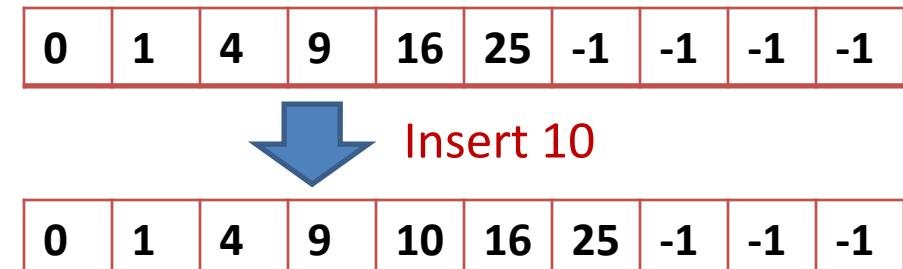
Insertion in a sorted array

```
#define MAX_SIZE 10
#define DEFAULT_VALUE -1
int main() {
...
...
int x=10; //element to be inserted
int i=0;

// Find the position to insert x
for (i=0; i<count; i++) {
    if (arr[i]>=x)
        break;
}
// i is the position in arr where
// x should be inserted
}
```

```
// Now shift elements from last
// occupied position until i, one
// position to right
for (int j=count-1; j>i;j--) {
    arr[j+1] = arr[j];
}
// loop exits when j=i, the
// position to insert x.

arr[j] = x; count++;
// x inserted at its position
}
```



Delete in Array (Sorted or unsorted)

Delete is similar to insert.

Example: *Delete 9 from arr*

0	1	4	9	10	16	25	-1	-1	-1
---	---	---	---	----	----	----	----	----	----



Delete 9 from arr:

- shift 10,16,25 to one position towards left
- `arr[count-1] = DEFAULT_VAL`
- `count = count - 1`

0	1	4	10	16	25	-1	-1	-1	-1
---	---	---	----	----	----	----	----	----	----

*Exercise: Implement this
operation*



Multi-dimensional Arrays in C

Multi-dimensional Arrays

- C supports arrays of multiple dimensions
- A basic multi-dimensional array is a **2-D array**
 - Also known as a **matrix**

Declaring 2-D Arrays - Syntax:

```
type variable_name [row_size] [column_size];
```

row_size

❑ *Number of rows in the matrix*

column_size

❑ *Number of columns in the matrix*

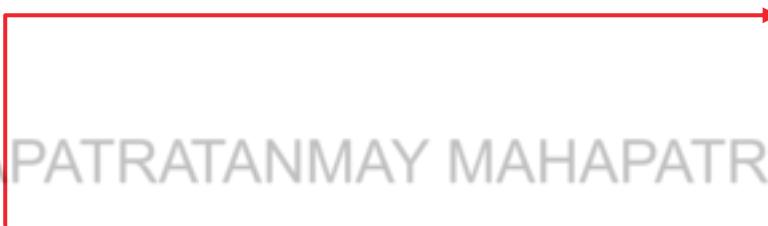
Examples of 2-D Arrays

Examples:

```
int number[3][4]; /* 12 elements */  
float number[3][2]; /* 6 elements */  
char name[10][20]; /* 200 chars */
```

column_size=4

row_size=3



Initializing a 2-D Array

```
int a[2][3]={1,2,3,4,5,6};  
int a[2][3]={{1,2,3}, {4,5,6}};  
int a[][3]={{1,2,3}, {4,5,6}}
```



All are equivalent
and produce the
following array:

1	2	3
4	5	6

How will the values will be assigned in each case?

Incorrect Ways of Initializing 2-D Arrays



Following initializations are not allowed

```
int a[3][]={2,4,6,8,10,12};
```

```
int a[][]={2,4,6,8,10,12};
```

Note:

- If the first bracket pair is empty, then the compiler takes the size from the number of inner brace pairs
- If the second bracket pair is empty, the compiler throws a compilation error!

Accessing a 2-D Array

```
int a[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};  
for(i=0;i<3;i++)  
{  
    for(j=0;j<4;j++)  
    {  
        printf("%d",a[i][j]);  
    }  
    printf("\n");  
}
```

*Run time initialization of an array can be done in a similar way by changing **printf()** to **scanf()** inside the j loop.*



Memory Maps of 2-D Arrays

Storing 2-D Arrays in Memory

- 2-D arrays are stored in the memory as a linear sequence of variables
- Two methods for storing:
 - **Row major**
 - **Column major**

TANMAY MAHAPATRA TANMAY MAHAPATRA TANMAY MAHAPATRA

TANMAY MAHAPATRA TANMAY MAHAPATRA TANMAY MAHAPATRA

Row Major vs. Column Major

Example:

```
int a[3][3];
```

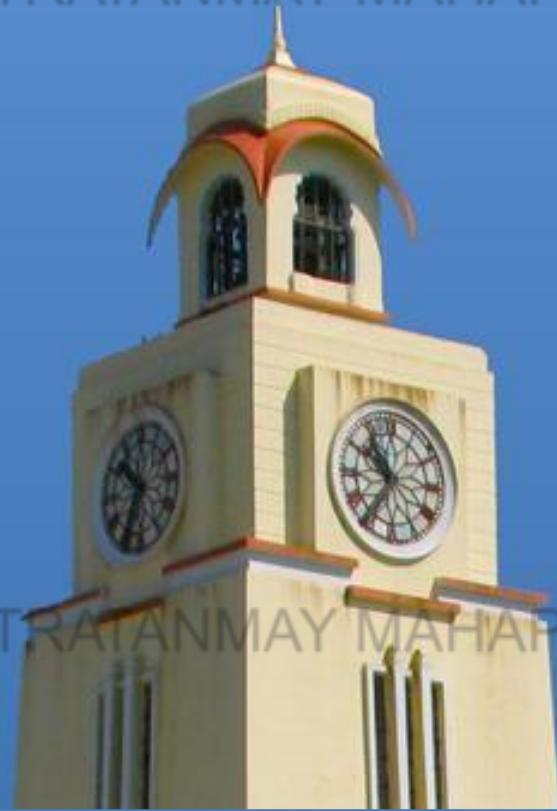
Row major storage will have this sequence in main memory:

$a[0][0]$, $a[0][1]$, $a[0][2]$, $a[1][0]$, $a[1][1]$,
 $a[1][2]$, $a[2][0]$, $a[2][1]$, $a[2][2]$

Indicates
addresses

Column major storage will have this sequence in main memory:

$a[0][0]$, $a[1][0]$, $a[2][0]$, $a[0][1]$, $a[1][1]$,
 $a[2][1]$, $a[0][2]$, $a[1][2]$, $a[2][2]$



Matrix Addition

Matrix Addition and Subtraction

Matrix Addition:

$$\begin{bmatrix} 1 & 3 \\ 1 & 0 \\ 1 & 2 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 7 & 5 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 + 0 & 3 + 0 \\ 1 + 7 & 0 + 5 \\ 1 + 2 & 2 + 1 \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ 8 & 5 \\ 3 & 3 \end{bmatrix}$$

Matrix Subtraction:

$$\begin{bmatrix} 1 & 3 \\ 1 & 0 \\ 1 & 2 \end{bmatrix} - \begin{bmatrix} 0 & 0 \\ 7 & 5 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 - 0 & 3 - 0 \\ 1 - 7 & 0 - 5 \\ 1 - 2 & 2 - 1 \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ -6 & -5 \\ -1 & 1 \end{bmatrix}$$

Working with two dimensional Arrays (Matrix Addition & Subtraction)

Let $\mathbf{A}[m][n]$ and $\mathbf{B}[p][q]$ be two matrices.

Precondition: m equals to p and n equals to q .

Algorithm Steps:

1. Read two matrices \mathbf{A} and \mathbf{B} , and initialize \mathbf{C} matrix to zero
2. Repeat (3) **for** $i=0$ to $m-1$
3. Repeat (3.a) **for** $j=0$ to $n-1$
 - 3.a) $\mathbf{C}[i][j] = \mathbf{A}[i][j] + \mathbf{B}[i][j]$
4. Display \mathbf{C} matrix

Matrix Addition: Code

```
#define ROW 10
#define COL 10
int main(){
    int M1 [ROW] [COL] ,M2 [ROW] [COL] ,M3 [ROW] [COL] ,i,j;
    int row1,col1,row2,col2;
    printf("Enter row value for M1\n");
    scanf("%d",&row1);
    printf("Enter column value for M1\n");
    scanf("%d",&col1);
    printf("Enter row value for M2\n");
    scanf("%d",&row2)
    printf("Enter column value for M2\n");
    scanf("%d",&col2)
```

Matrix Addition: Code (Contd.)

```
if(row1!=row2 || col1!=col2)
{
    printf("Invalid Input: Addition is not possible");
    return;
}
printf("Enter data for Matrix M1\n");
for(i=0;i<row1;i++)
{
    for(j=0;j<col1;j++)
    {
        scanf("%d", &M1[i][j]);
    }
    printf("\n");
}
```

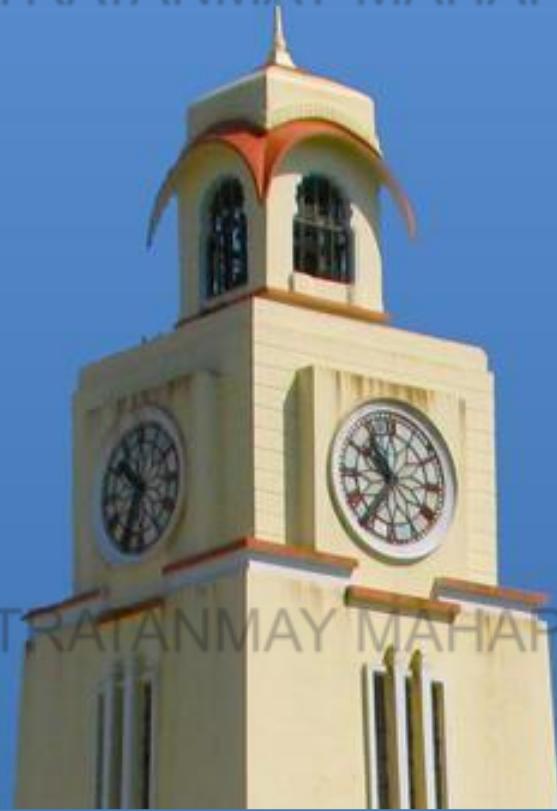
Matrix Addition: Code (contd.)

```
printf("Enter data for Matrix M2\n");
for(i=0;i<row2;i++)
{
    for(j=0;j<col2;j++)
    {
        scanf("%d",&M2[i][j]);
    }
    printf("\n");
}
printf("Addition of Matrices is as follows\n");
for(i=0;i<row2;i++)
{
    for(j=0;j<col2;j++)
        M3[i][j]= M1[i][j] + M2[i][j];
```

Matrix Addition: Code (contd.)

```
// display the new matrix after addition
for(i=0;i<row2;i++)
{
    for(j=0;j<col2;j++)
    {
        printf("%d",M3[i][j]);
    }
    printf("\n");
}
```

Matrix Subtraction can be done in a similar way



Matrix Multiplication

Matrix Multiplication

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 10 & 11 \\ 20 & 21 \\ 30 & 31 \end{bmatrix}$$

$$= \begin{bmatrix} 1 \times 10 + 2 \times 20 + 3 \times 30 & 1 \times 11 + 2 \times 21 + 3 \times 31 \\ 4 \times 10 + 5 \times 20 + 6 \times 30 & 4 \times 11 + 5 \times 21 + 6 \times 31 \end{bmatrix}$$

$$= \begin{bmatrix} 10+40+90 & 11+42+93 \\ 40+100+180 & 44+105+186 \end{bmatrix} = \begin{bmatrix} 140 & 146 \\ 320 & 335 \end{bmatrix}$$

Matrix Multiplication

M1xM2



Pre-condition: Number of columns in M1 should be equal to number of rows in M2

Algorithm Steps:

1. Read two matrices **M1 [m] [n]** and **M2 [n] [r]** and initialize another matrix **M3 [m] [r]** for storing result
2. Repeat for **i=0 to m-1**
 - Repeat for **j=0 to r-1**
$$M3[i][j] = 0$$
 - Repeat for **k=0 to n-1**
$$M3[i][j] += M1[i][k] * M2[k][j]$$
3. Print matrix **M3**

Matrix Multiplication: Code

```
#include <stdio.h>
#define row1 4
#define col1 3
#define row2 3
#define col2 4
#define row3 4
#define col3 4

void main()
{
    int M1[row1][col1],M2[row2][col2],M3[row3][col3];
    int i,j,k;
```

Matrix Multiplication: Code (contd.)

```
printf("Enter data for Matrix M1\n");
for(i=0;i<row1;i++)
{
    for(j=0;j<col1;j++)
        scanf("%d", &M1[i][j]);
    printf("\n");
}
```

Matrix Multiplication: Code (contd.)

```
printf("Enter data for Matrix M2\n");
for(i=0;i<row2;i++)
{
    for(j=0;j<col2;j++)
        scanf("%d",&M2[i][j]);
    printf("\n");
}
```

Matrix Multiplication: Code (contd.)

```
if (col1!= row2) {  
    printf("Multiplication is not possible");  
    return;  
}  
for(i=0;i<row1;i++) {  
    for(j=0;j<col2;j++) {  
        M3 [i] [j] = 0;  
        for(k=0;k<col1;k++) {  
            M3 [i] [j] += M1 [i] [k] * M2 [k] [j];  
        }  
    }  
}
```

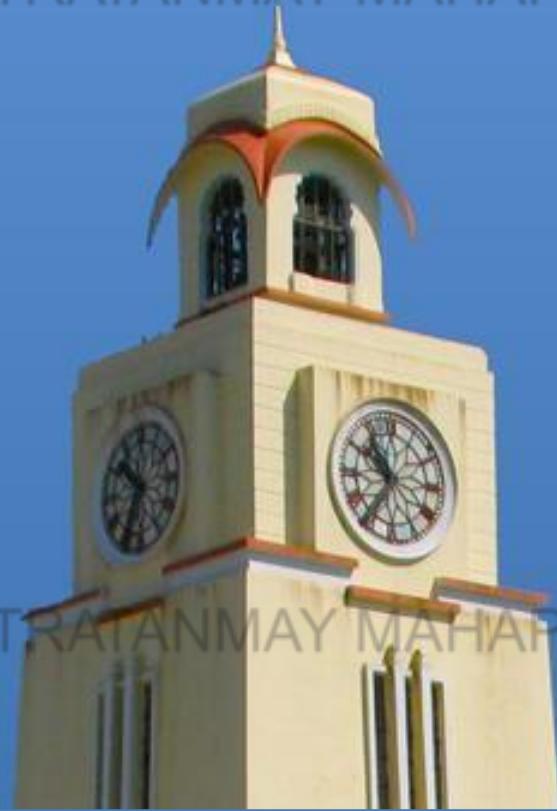
The diagram shows two matrices being multiplied. The first matrix, on the left, is a 2x3 matrix with elements 1, 2, 3 in the top row and 4, 5, 6 in the bottom row. The second matrix, on the right, is a 3x2 matrix with elements 10, 11 in the top row, 20, 21 in the middle row, and 30, 31 in the bottom row. Red arrows point from the top row of the first matrix to the first column of the second matrix, indicating the calculation of the first element of the resulting 2x2 matrix.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 10 & 11 \\ 20 & 21 \\ 30 & 31 \end{bmatrix}$$

Matrix Multiplication: Code (contd.)



```
printf("RESULT MATRIX IS\n ");
for(i=0;i<row3;i++) {
    for(j=0;j<col3;j++) {
        printf("%d",M3[i][j]);
    }
    printf("\n");
}
return;
}
```



n-dimensional arrays: A glimpse

N-dimensional arrays

3D array: `int arr[2][2][3];`

4D array: `int arr[2][2][2][2];`

5D array: `int arr[2][2][2][2][2];`

...

Note:

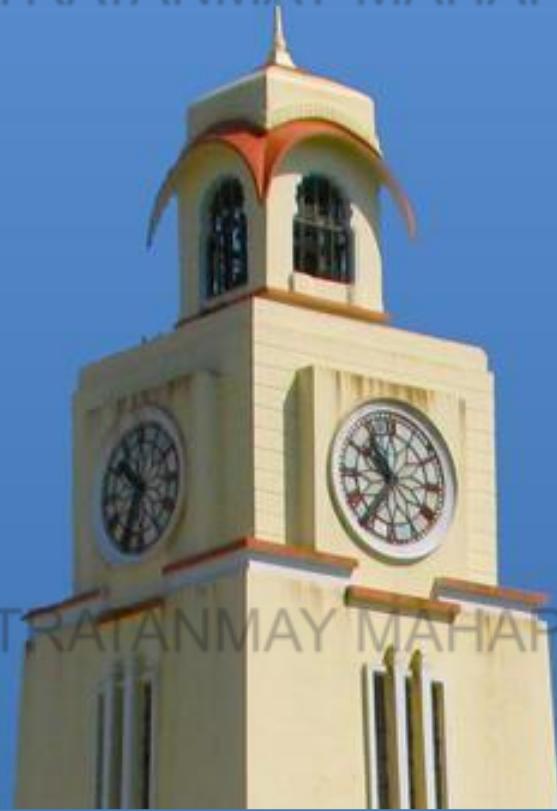
- Elements are stored and accessed in a similar way as that of 2D arrays.
- They are also stored in Row Major format.

Exercises

Q. 1 Generate Fibonacci series using Array.

Q. 2 Write a program to find a binary equivalent of a decimal number using an array.

Q. 3 Write a program to find the transpose of a matrix.



Thank you
Q & A

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA



Module 8 – part 2 – String operations

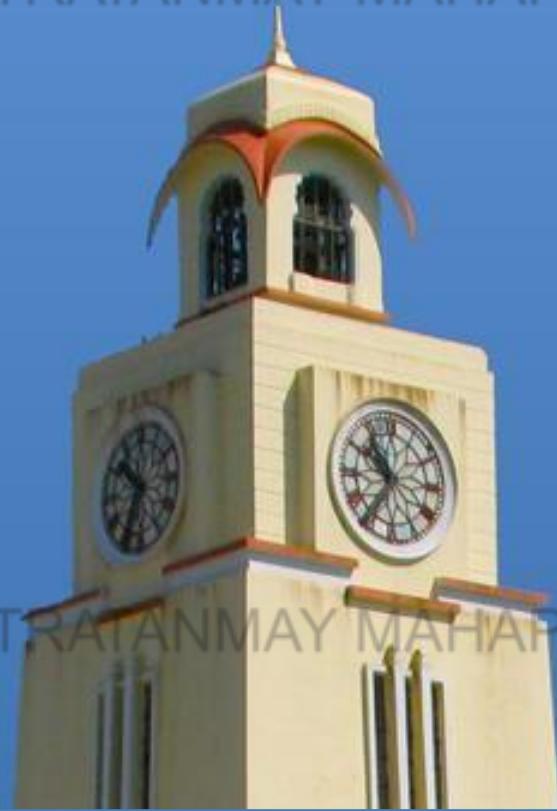
BITS Pilani

Pilani Campus

Department of Computer Science & Information Systems

Module Overview

- **Character Arrays**
- **Strings**
- **Strings Operations**



Character Arrays

Character (Char) Arrays

```
char color[3] = "RED";
```



```
char color[] = "RED";
```



Are they the same?

Character arrays are the way to represent **strings** in C.
Each string typically ends with a NULL character "**\0**".

Strings

- Strings in C are represented by arrays of characters
- End of the string is marked with a special character NULL.
- The corresponding escape sequence character is \0.
- C does not have string data type.

Declaration of strings:

```
char str[30];  
char line[80];
```

String Initialization

```
char str[9] = "I like C";
```

same as

```
char str[9]={'I',' ','l','i','k','e',' ','C','\0'};
```

Q. Is there any difference between following Initialization?

```
char str[] = "BITS";
```

```
char str[4] = "BITS";
```

Ans: Yes, in second declaration there is no null character

Printing Strings

```
char text[]="C Programming";
printf("%s\n",text);
```

Output???

C Programming

Important Character functions in <ctype.h>

isdigit(c) /*Returns a nonzero if c is a digit*/

islower(c) /* Returns a nonzero if c is a lower case alphabetic character */

isalpha(c) /*Returns a nonzero if c is an alphabet*/

isspace(c) /*Returns a nonzero for blanks */

isupper(c) /*Returns a nonzero if c is capital letter*/

toupper(c) /* Returns upper case of c */

tolower(c) /* Returns lower case of c */

Char Arrays contd...

Write a C program which reads a 1D char array, converts all elements to uppercase, and then displays the converted array.

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA

Hint: use toupper(ch) of <ctype.h>

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA

Lower Case to Upper Case using char array

```
#include <stdio.h>
#include <ctype.h>
int main(){
    int size,i=0;
    char name[50];
    name[0]=getchar();
    while(name[i]!='\n')      }
    {
        i++;
        name[i]=getchar();
    }

    name[i]='\0';
    size = i;
    printf("\nName is %s", name);
    for(i=0;i<size;i++)
        putchar(toupper(name[i]));
    return 0;
}
```

getchar() and putchar()

- **int getchar(void);**
 - *getchar* is a standard C library function that reads a single character from the standard input (usually the keyboard) and returns the character as an integer (ASCII value). It's commonly used for basic character input in console-based programs.

- **int putchar(int character);**
 - *putchar* is a standard library function in C and C++ used for output operations. It is used to write a single character to the standard output, typically the console or terminal.

Char Arrays contd...

Modify the previous code to use scanf()/printf() in place of

getchar()/putchar()

Lower Case to Upper Case using char array

```
#include <stdio.h>
#include <ctype.h>
int main(){
    int size,i=0;
    char name[50];
    scanf("%c",&name[0]);
    while(name[i]!='\n')
    {
        i++;
        scanf("%c",&name[i]);
    }
    name[i]='\0';
    size = i;
    printf("\nName is %s",name);
    for(i=0;i<size;i++)
        printf("%c",toupper(name[i]));
    return 0;
}
```

Reading a String (2-1)

Using scanf()

```
char text[30];
printf("Enter a string: ");
scanf("%s",text);
printf("The string is : %s",text);
```

Sample output:

Enter a string: hello

The string is: hello

Enter a string: hello how are you

The string is: hello

Note: scanf() takes string without blank space

Reading a String (2-2)

```
char text[30];  
printf("Enter a string: ");  
scanf("%[a-z]s", text);  
printf("The string is : %s", text);
```

Sample output:

Enter a string: hello

The string is: hello

Enter a string: hello123

The string is: hello

Single Line input

```
char text[80];  
printf("Enter a string: ");  
scanf("%[^\\n]s",text); /*newline terminated string */  
printf("The string is : %s",text);
```

Sample output:

Enter a string: hello how are you

The string is: hello how are you

Multi line Input (using custom delimiter for scanf)

```
char text[180];
printf("Enter a string terminating with ~: ");
scanf("%[^~]s",text);
printf("The string is : %s",text);
```

Note: After ^ any character can be used to terminate the input.

Sample output:

Enter a string terminating with ~: hello how
are you. ~

The string is: hello how are you.

Using gets/puts

```
#include<stdio.h>
int main()
{
    char str[20];
    printf("Enter a string: ");
    gets(str);
    printf("The string is %s",str);
    puts(str);
    return 0;
}
```

Note:

- It is not required to explicitly insert '\0' character at the end of each character array while using **gets()**.
- It automatically adds so.

Output

Enter a string :C programming
The string is : C programming

Input String using gets() with a larger number of characters

```
char str[20];  
printf("Enter a string :");  
gets(str);  
printf("The string is %s",str);  
puts(str);
```

Output

```
tejasv@LAPTOP-6IBMJSH8:/mnt/d/wsl$ ./a.out  
Enter a string :SitaramSitaramSitaramSitaram  
The string is SitaramSitaramSitaramSitaramSitaramSitaramSitaram  
*** stack smashing detected ***: terminated  
Aborted
```

Warning!! gets() is deprecated.. Use fgets() instead

Input String using fgets()

```
char str[20];
printf("Enter a string: ");

if (fgets(str, sizeof(str), stdin) != NULL) {
    for (int i = 0; str[i] != '\0'; i++) {
        if (str[i] == '\n') {
            str[i] = '\0';
            break;
        }
    }

    printf("The string is: %s\n", str);
} else {
    printf("Error reading input.\n");
}
```

Output:

```
tejasv@LAPTOP-6IBMJSH8:/mnt/d/wsl$ ./a.out
Enter a string: SitaramSitaramSitaramSitaram
The string is: SitaramSitaramSitar
```

Character Manipulation in the String – eliminate spaces

```
int main()
{
    char s[80],ws[80];
    int i,j;
    printf("Enter the text:\n");
    gets(s); /* reading text from user */

    for(i=0,j=0; s[i]!='\0'; i++)
    {
        if(s[i]!=' ')
            ws[j++] = s[i];
    }
    ws[j]='\0';

    printf("The text without blank space is:\n");
    puts(ws); /* printing text on monitor */
    return 0;
}
```

Character Manipulation in the String

– eliminate spaces – using fgets



```
int main()
{
    char s[80],ws[80];
    int i,j;
    printf("Enter the text:\n");
    fgets(s, sizeof(s), stdin); /* reading text
from user */

    for(i=0,j=0; s[i]!='\0'; i++)
    {
        if(s[i]!=' ' && s[i]!='\n')
            ws[j++] = s[i];
    }
    ws[j]='\0';

    printf("The text without blank space is:\n");
    puts(ws); /* printing text on monitor */
    return 0;
}
```

String Manipulation functions in <string.h>

`strcpy(s1,s2) /* copies s2 into s1 */`

`strcat(s1,s2) /* concatenates s2 to s1 */`

`strlen(s) /* returns the length of s */`

`strcmp(s1,s2)/*returns 0 if s1 and s2 are same
returns less than 0 if s1<s2
returns greater than 0 if s1>s2 */`

Implementation of strlen()

```
int n = 0;  
char text[100],c;  
while((c = getchar()) != '\n' && n<99)  
    text[n++] = c;
```

```
text[n++]= '\0';  
printf("Length of text : %d",n);
```

Implementation of strcat()

```
void main() /* s2 is concatenated after s1 */
{ char s1[100],s2[100];
int i = 0,j = 0;
printf("Enter first string\n");
scanf("%[^\\n]s",s1); getchar();
printf("Enter second string\n");
scanf("%[^\\n]s",s2);
while(s1[i++] != '\0');
i--;
while(s2[j] != '\0')
s1[i++] = s2[j++];
s1[i] = '\0';
printf("\n Final string is:%s",s1);
}
```

Palindrome problem

```
void main()
{ char str[80];
int left,right,i,len,flag = 1;
printf("Enter a string");
for(i = 0;(str[i] = getchar()) != '\n';++i);
len = i-1;
for(left = 0,right = len; left < right; ++left,--right)
{ if(str[left] != str[right])
    { flag = 0;
      break;
    }
}
if(flag)printf("\n String is palindrome");
else
printf("\n String is not a palindrome");
}
```

Word counting Problem

```
void main()
{ char text[40];
  int i = 0, count = 0;
  printf("Enter a string:");
  gets(text);
  while(text[i] != '\0')
  {
    while(ispace(text[i]))
      i++; /* Repeat till first non blank character */
    if(text[i] != '\0')
      { count++;
        while(!ispace(text[i]) && text[i] != '\0')
          i++; /* Repeat till first blank character */
      }
  }
  printf("The number of words in the string is %d", count);
}
```

Homework

Try replacing gets with fgets in the previous example!!

Other functions in string.h/stdlib.h

atof(): Converts an ASCII string to its floating-point equivalent (type double)

```
char s1[] = "+1776.23";
double my_value = atof(s1);
```

atoi(): Converts an ASCII string to its integer equivalent

```
char s2[] = "-23.5";
int my_value = atoi(s2);
```

strncat(): Works like strcat, but concatenates only a specified number of characters.

```
char s1[50] = "Hello, world!";
char s2[] = "Bye now!";
strncat (s1, s2, 3);
```

Other functions in string.h

strncmp(): Works like strcmp, but compares only a specified number of characters of both strings.

```
char s1[] = "dogberry";
char s2[] = "dogwood";
int comp = strncmp (s1, s2, 3);
```

strncpy(): Works like strcpy, but copies only a specified number of characters.

```
char dest[50];
char src[] = "C Program";
strncpy (dest, src, 3);
```

strstr(): Tests whether a substring is present in a larger string. Returns a pointer to the first occurrence of the substring in the larger string, or zero if the substring is not present.

```
char s1[] = "Got food?";
char s2[] = "foo";

if (strstr (s1, s2))
printf ('%s' is a substring of
'%s'.\n", s2, s1);
```

Arrays of Strings

Declaration:

```
char name[5][30];
```

Five strings each contains maximum thirty characters.

Initialization:

```
char [5] [10]={ "One", "Two", "Three", "Four", "Five" };
```

Other valid declarations

```
char [] []={ "One", "Two", "Three", "Four", "Five" };
```

```
char [5] []={ "One", "Two", "Three", "Four", "Five" };
```

Array of Strings

```
char city[4][12] = {  
    "Chennai",  
    "Kolkata",  
    "Mumbai",  
    "New Delhi"  
};
```

	0	1	2	3	4	5	6	7	8	9	10	11
0	C	h	e	n	n	a	i	\0				
1	K	o	l	k	a	t	a	\0				
2	M	u	m	b	a	i	\0					
3	N	e	w		D	e	l	h	i	\0		

String Arrays: Reading and Displaying

```
void main()
{ char name[5][30];
  printf("\n Enter five strings");
  /* Reading strings */
  for(i=0;i<5; i++)
    scanf("%s",name[i]);
  /* Printing strings */
  for(i=0;i<5; i++)
    printf("\n%s",name[i]);
}
```

Problem 3: Array of Strings

Problem Statement: Write a C program that will read and store the details of a list of students in the format

ID	NAME	MARKS
----	------	-------

And produce the following output

1. Alphabetical list of Names, ID's and Marks.
2. List sorted on ID's
3. List sorted on Marks

Implementation (1-3)

```
#define N 5
#include<stdio.h>
#include<string.h>
int main()
{
    char names[N][30],marks[N][10];
    char id[N][12],temp[30];
    int i,j;
    /* Reading Student Details */
    printf("Enter Student ID NAME and MARKS
        \n");
    for(i = 0; i<N; i++)
        scanf("%s %s %s",id[i],names[i],marks[i]);
```

Implementation (2-3)

```
/* Alphabetical Ordering of Names */
for(i=1; i<=N-1; i++)
    for(j=1; j<=N-i; j++)
        if(strcmp(names[j-1], names[j])>0)
            {
                strcpy(temp, names[j-1]);
                strcpy(names[j-1], names[j]);
                strcpy(names[j], temp);
            /* Swapping of marks */
                strcpy(temp, marks[j-1]);
                strcpy(marks[j-1], marks[j]);
                strcpy(marks[j], temp);
```

Implementation (3-3)

```
/* Swapping of ID's */  
    strcpy(temp,id[j-1]);  
    strcpy(id[j-1], id[j]);  
    strcpy(id[j], temp);  
}  
  
printf("ALPHABETICAL LIST OF ID NAME &  
MARKS");  
  
for(i=0;i<N;i++)  
printf("%s\t%s\t %s\n",id[i],names[i],marks[i]);  
return;  
}
```

ID wise Sorting (ID is a string)

```
for(i=1;i<=N-1;i++)
{
    for(j=1; j<=N-i; j++)
    {
        if(strcmp(id[j-1],id[j])>0)
        {
            strcpy(temp,id[j-1]);
            strcpy(id[j-1], id[j]);
            strcpy(id[j], temp);
            /* Swaping of marks */
            strcpy(temp,marks[j-1]);
            strcpy(marks[j-1], marks[j]);
            strcpy(marks[j], temp);
            /* Swaping of names */
            strcpy(temp,names[j-1]);
            strcpy(names[j-1], names[j]);
            strcpy(names[j], temp);
        }
    }
}
```

Home Exercise 1

- 1) Write a C program to implement strcpy()
 - 1) Write a C program to implement strcmp()
 - 1) Write a C program to search a string from an array of strings.
-
- 1) Write a C program that copies the unique words among the set of words into another memory location.
 - 1) Write a C program which will read a line of text and rewrite it in the alphabetical order.
 - 1) Write a C program to replace a particular word by another word in a given string. Both the words are provided by the user at run time.

Home Exercise 2

- 1) Write a C program that counts the number of vowels, consonants, digits and other symbols in a given line of text.

- 1) Write a C program to reverse a string. Try not to use an extra string and modify the source string to store the reversed string. Number of exchanges should be minimal.

- 1) Write a C Program to separate a given string into two strings. All the odd positioned characters are stored in the first string and even positioned characters in the second string.



Thank you
Q & A

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA



BITS Pilani
Pilani Campus

Module 9 – *Structures in C*

Department of Computer Science & Information Systems

Module Overview

- **Tuple Data**
- **Structures**
- **Nested Structures and Array of Structures**
- **Passing Structure Variables to Functions**
- **Union**
- **Enumerator**



Tuple Data

Tuple Data

Consider student database. Every student has a few attributes:

- *Name*
- *ID*
- *Group*

This is known as tuple data where **{Name, ID No, Group}** is a tuple

Example: The below table contains 3 records of the above tuple type

ID	Name	Group
0910	Ram Sharma	B4
0313	Alex Mathew	A8
0542	Vijay Kumar	A7

Tuple Data is used in Database Systems

Representing Tuple Data

First attempt: 3 lists (i.e. arrays)

- **Name array, ID array, and Group array.**
- Problems: Consider add / delete operations.
 - Shifting to be done in 3 arrays separately.
 - 3 arrays are the 3 element values passed as parameters.
 - Data representation “does not say” the 3 things are related.

Better Solution: 1 list of triples

- Each triple is of the form **(ID, Name, Group)**
- add / delete operation: Shifting for one array only.
 - *Can use Structures in C to do this.*



Structures in C

Structures in C

- **Structure**
 - Group of logically related data items
 - Example: **ID, Name, Group**
- A single structure may contain data of one or more data types
 - ID: **int ID**
 - Name: **char Name[]**
 - Group: **char Group[]**
- The individual structure elements are called **members**
- We are essentially defining a **new data type**
- ***Then we can create variables of the new data type***
 - These variables can be **global, static** or **auto**

Defining a Structure

```
struct struct_name {  
    member_1;  
    member_2;  
    ...  
};
```

- **struct** is the required keyword
- **struct_name** is the name of the structure
- **member_1, member_2, ...** are individual member declarations

Example: Student structure containing **ID, Name, Group**

```
struct student {  
    int ID;  
    char Name[20];  
    char Group[3];  
}
```

Members

Declaring Variables of the Structure type

```
struct struct_name var_1, var_2, ..., var_n;
```

Example 1:

```
struct student {  
    int ID;  
    char Name[20];  
    char Group[3];  
};
```

```
// declaring variable of  
// the type struct student  
struct student s1, s2;
```

Example 2:

```
struct book {  
    char Name[20];  
    float price;  
    char ISBN[30];  
};
```

```
// declaring variable of  
// the type struct book  
struct book b1, b2;
```

Combining Structure Definition and Variable Declaration

```
struct struct_name{  
    data_type member_1;  
    data_type member_2;  
    ...  
}var1, var2, ... varn;
```

Example:

```
struct student {  
    int ID;  
    char Name[20];  
    char Group[3];  
} s1, s2;
```

```
struct {  
    data_type member_1;  
    data_type member_2;  
    ...  
}var1, var2, ... varn;
```

Example:

```
struct {  
    int ID;  
    char Name[20];  
    char Group[3];  
} s1, s2;
```

Accessing members of a structure

- A structure member can be accessed by writing

structure_variable.MemberName

Example: Consider the following structure definition:

```
struct student {  
    int ID;  
    char Name[20];  
    float Marks[5];  
}  
s1, s2;
```

- | | |
|--------------------|---|
| s1.Name | → Value stored in the Name field of s1 variable |
| s2.Name | → Value stored in the Name field of s2 variable |
| s2.Marks[i] | → Value stored at the i^{th} position of the Marks array of s2 |

Example

```
#include <stdio.h>
int main()
{
    struct complex {
        float real;
        float cmplex;
    } a, b, c;

    scanf ("%f %f", &a.real, &a.cmplex);
    scanf ("%f %f", &b.real, &b.cmplex);
    c.real = a.real + b.real;
    c.cmplex = a.cmplex + b.cmplex;
    printf ("\n %f +%f j", c.real, c.cmplex);
    return 0;
}
```

Where can we define structures?

- **Structures can be defined in:**
 - **Global Space**
 - ***Outside all function definitions***
 - Available everywhere inside and outside all functions to declare variables of its type
 - **Local Space**
 - ***Inside a function***
 - Variables of this structure type can be declared and used only inside that function.

Examples

Structure defined in global space:

```
#include <stdio.h>
struct student{
    int ID;
    char Name[20];
    float marks[5];
};

int f1() {
    struct student s1;
    ... // operations on s1
}
int f2() {
    struct student s2;
    ... // operations on s2
}
int main() {
    ...
}
```

Structure defined in local space:

```
#include <stdio.h>
int f1() {
    struct student{
        int ID;
        char Name[20];
        float marks[5];
    };
    struct student s3;
    ... // operations on s3
}
int f2() {
    // can't define a variable
    // with struct student here
}
int main() {
    ...
}
```

Structure Initialization

```
struct book {  
    char Name[20];  
    float price;  
    char ISBN[30];
```

```
};
```

```
struct book b1 = {"Book1", 550.00, "I1"},  
                 b2 = {"Book2", 650.00, "I2"};
```



b1.Name = Book1	b2.Name = Book2
b1.price = 550.00	b2.price = 650.00
b1.ISBN = I1	b2.ISBN = I2

typedef

Allows users to define new data-types

Syntax:

```
typedef type new-type
```

Example 1:

```
typedef int Integer;  
Integer I1, I2;
```

Example 2:

```
typedef struct{  
    float real;  
    float imag;  
} COMPLEX;
```

```
COMPLEX c1, c2;
```

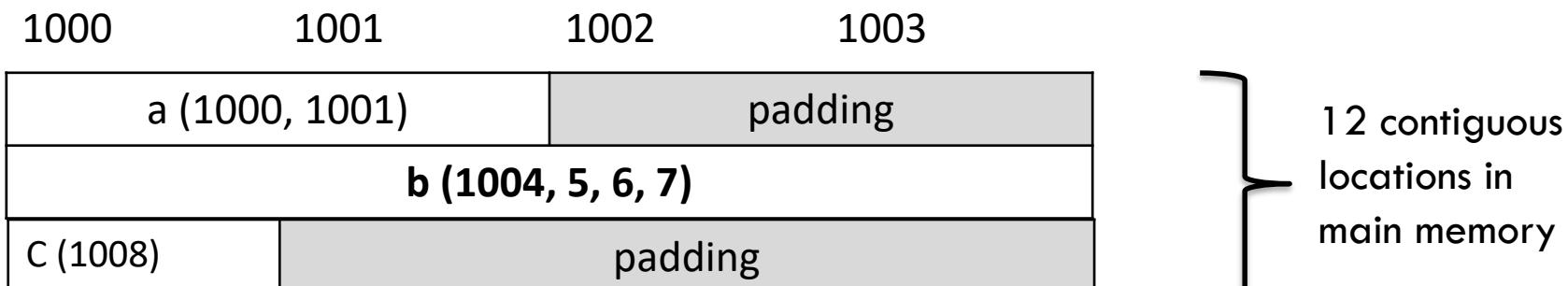
- It is a common practice to define structures using **typedef**
- It simplifies the syntax and increases readability of the program

sizeof() for struct variables

```
struct test_struct {
    short a; //2bytes
    int b; //4bytes
    char c; //1byte
} test;
```

```
printf("a= %lu\n", sizeof(test.a)); = 2
printf("b= %lu\n", sizeof(test.b)); = 4
printf("c= %lu\n", sizeof(test.c)); = 1
printf("%lu\n", sizeof(test)); = 12
```

- We can notice that the `sizeof(test) > sizeof(test.a) + sizeof(test.b) + sizeof(test.c)`
- This is because the compiler adds (may add) padding for alignment requirements
- Padding means to append empty locations towards the end (or beginning)



Memory view of storing a struct variable

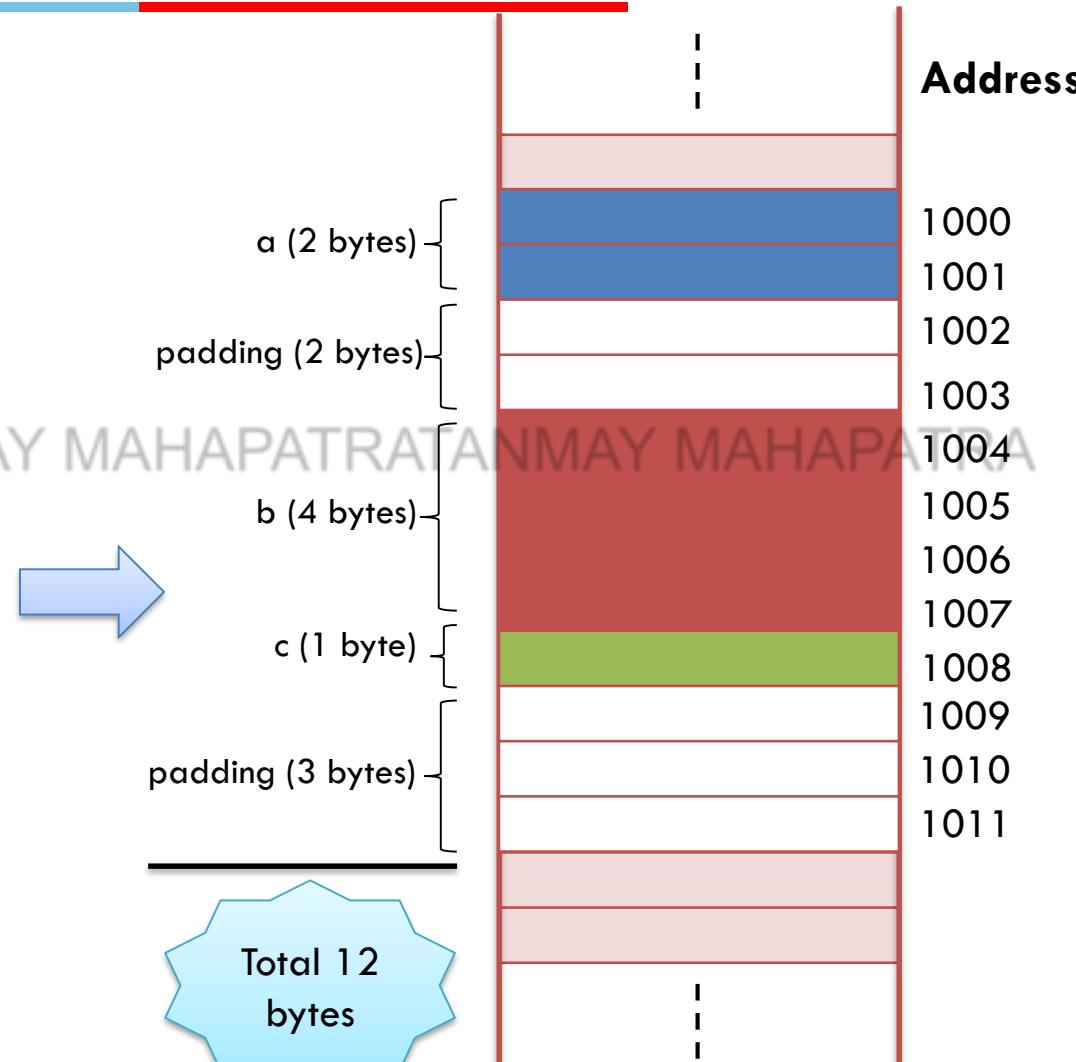
```
struct test_struct {  
    short a; //2bytes  
    int b; //4bytes  
    char c; //1byte  
} test;
```

1000 1001 1002 1003

a (1000, 1001)	padding
b (1004, 5, 6, 7)	
C (1008)	padding

Note: This illustration considers that each address is represented by 4 bits.

Also note that in this example **int** is occupying 4 bytes. Some compilers use 2 bytes for **int**, and some use 4 bytes.



Another Example

```

struct test_struct {
    char a; //1byte
    short b; //2bytes
    int c; //4bytes
} test;
printf("a= %lu\n", sizeof(test.a)); = 1
printf("b= %lu\n", sizeof(test.b)); = 2
printf("c= %lu\n", sizeof(test.c)); = 4
printf("%lu\n", sizeof(test)); = 8
  
```

a (1000)	padding (1 byte)	b(1002-1003)
c(1004-1007)		

- We can notice that the padding added by compiler is less (**only 1 byte**), for the same set of variable types, arranged with a different order in the structure.
- The total number of bytes to be added for padding is decided by the compiler depending upon the arrangement of members inside the structure.

Example

```
#include <stdio.h>
#pragma pack(1)
struct base {
    int a;    char b;    char c;
} s;
```

The size of the var is : 6

```
int main()
{
    printf("The size of the var is : %d", sizeof(s));
    return 0;
}
```

Operations on struct variables

- Unlike arrays, group operations can be performed with structure variables
- A structure variable can be directly assigned to another structure variable of the same type

a1 = a2;

- The contents of members of a2 are copies to members of a1.
- You can't do this for two arrays, without using “**pointers**”
 - We will study about pointers in the next module
 - You still can't do equality check for two structure variables, i.e.
if (a1 == a2) is not valid in C
 - You shall have to check equality for individual members

Swap two structure variables

```
struct book {  
    char Name[20];  
    float price;  
    char ISBN[30];  
};  
int main(){  
    struct book b1 = {"Algorithms", 550.00, "I1"};  
    struct book b2 = {"Programming", 650.00, "I2"};  
    // swapping b1 and b2  
    struct book b3;  
    b3 = b1;  
    b1 = b2;  
    b2 = b3;  
    printf("The name of Book in b1 is %s\n", b1.Name);  
    printf("The name of Book in b2 is %s\n", b2.Name);  
    return 0;  
}
```

Output:

The name of the Book in b1 is Programming
The name of the Book in b2 is Algorithms

Incorrect way of using structure variables

```
struct book {  
    char Name[20];  
    float price;  
    char ISBN[30];  
};  
  
int main(){  
    struct book b1 = {"Algorithms", 550.00, "I1"};  
    struct book b2 = {"Programming", 650.00, "I2"};  
    // Incorrect way of using structure variables  
    // if (b1 == b2)  
        if(b1.price == b2.price) // Correct way  
            printf("Both the books have the same price\n");  
    else  
        printf("Both the books DON'T have the same price");  
    return 0;  
}
```



Nested Structures and Array of Structures

Nested Structures

```
struct student{  
    float CGPA;  
    char dept[10];  
    struct address  
    {  
        int PINCODE;  
        char city[10];  
    } add;  
} s1;
```

```
struct address  
{  
    int PINCODE;  
    char city[10];  
};  
struct student  
{  
    float CGPA;  
    char dept[10];  
    struct address add;  
} s1;
```

To access PINCODE → s1.add.PINCODE

Example of nested structures

```
struct address
{
    int PINCODE;
    char city[10];
};

struct student
{
    float CGPA;
    char dept[10];
    struct address add;
};

int main()
{
    struct student s1;
    s1.add.PINCODE = 333031;
    strcpy(s1.add.city , "Vizag");
    s1.CGPA = 10.0;
    strcpy(s1.dept , "CS");
    printf("s1's city is %s\n",s1.add.city);
    printf("s1's CGPA is %f\n",s1.CGPA);
    return 0;
}
```

Array of Structures

Once a structure has been defined, we can declare an array of structures

Example:

```
struct book {  
    char Name[20];  
    float price;  
    char ISBN[30];  
};  
struct book bookList[50];
```

The individual members can be accessed as:

bookList[i].price → returns the price of the i^{th} book.

Example

Program to store information of 5 students

```
#include <stdio.h>

struct student{
    char Name[10];
    float CGPA;
    float marks[5];
};

struct student s[5];
```

```
int main() {
    printf("Enter the details:\n");
    for(int i = 0; i < 5; i++)
    {
        printf("Enter name\n");
        scanf("%s", s[i].Name);
        printf("Enter the marks\n");
        for(int j = 0; j< 5; j++)
            scanf("%f", &s[i].marks[j]);
        printf("Enter the CGPA\n");
        scanf("%f", &s[i].CGPA);
    }
    return 0;
}
```

Storing database records in Array of Structures

- Remember our motivation for structures to store tuple data...

ID	Name	Group
0910	Rama Sarma	B4
0313	Alex Mathew	A8
0542	Vijay Kumar	A7

- The above table can be stored in a **struct student** array – **arr**
- Now its easy to add/delete records:
 - Simply create a new variable of the type **struct student** and append it to **arr**
 - If the array is sorted on ID numbers, it is sufficient to do necessary shiftings in **arr** itself and store the new record in its appropriate position
 - Deletion is also similar

Refer to slides of Module 8 to understand add/append/delete in an array.



Passing Structures to Functions

Passing struct variables to functions

- A structure can be passed as argument to a function
- Structures can be passed both by pass by value and pass by reference

• *We will study about pass by reference later with “pointers”*

- A function can also return a structure
- Array of structures are by default passed by reference
 - Just like any other arrays.
 - *We will study later with pointers*

Example

```
#include <stdio.h>
struct student {
char name[50];  int age;
};
void display(struct student s);
int main()
{
    struct student s;
    printf("Enter your name: ");
    scanf("%s", s.name);
    printf("Enter your age: ");
    scanf("%d", &s.age);
    display(s); // passing struct as an argument
    return 0;
}
```

```
void display(struct student s)
{
    printf("Display info\n");
    printf("Name: %s", s.name);
    printf("\nAge: %d", s.age);
}
```

Example – Pass by value

```
#include <stdio.h>
typedef struct {
    float re;
    float im;
} cmplx;

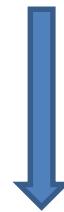
int main() {
    cmplx a, b, c;
    scanf ("%f %f", &a.re, &a.im);
    scanf ("%f %f", &b.re, &b.im);
    c = add (a, b) ;
    printf ("\n%f+%f j", c.re, c.im);
    return 0;
}
```

```
cmplx add(cmplx x,cmplx y) {
    cmplx t;
    t.re = x.re + y.re ;
    t.im = x.im + y.im ;
    return (t) ;
}
```

- When the function **add()** is **called**, the **values of the members of variables a and b are copied into members of variables x and y**.
- When the function **add()** **returns**, the **values of the members of variable t are copied into the members of variable c**.

Example: Passing structure by value illustrated

```
cmplx add(cmplx x,cmplx y) and c = add(a, b);
```



```
return (t)
```

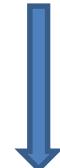
```
x.re = a.re, x.im = a.im  
y.re = b.re, y.im = b.im
```



```
c = t
```

```
c.re  
c.im
```

```
t.re  
t.im
```



```
t.re ← x.re + y.re  
t.im ← x.im + y.im
```

Previous Example (modified)

Program to store information of 5 students, compute total marks for each student and print it.

```
#include <stdio.h>
```

```
struct student{  
    char Name[10];  
    float CGPA;  
    float marks[5];  
};
```

```
struct student s[5];
```

```
void computeSum(struct student[], int);  
  
int main(){  
    printf("Enter the details:\n");  
    for(int i = 0; i < 5; i++)  
    {  
        printf("Enter name\n");  
        scanf("%s", s[i].Name);  
        printf("Enter the marks\n");  
        for(int j = 0; j < 5; j++)  
            scanf("%f", &s[i].marks[j]);  
        printf("Enter the CGPA\n");  
        scanf("%f", &s[i].CGPA);  
    }  
    computeSum(s, 5);  
    return 0; }
```

Example (contd.)

```
void computeSum(struct student sArr[], int size){  
    int i,j,sum;  
    for(i=0;i<size;i++)  
    {  
        sum = 0;  
        for(j=0;j<5;j++)  
        {  
            sum += sArr[i].marks[j];  
        }  
        printf("Total marks of %s are %d\n", sArr[i].Name, sum);  
    }  
    return;  
}
```



Unions

Union

- Also a user defined data type, but unlike Structures, Union members share the same memory location

```
struct name {  
    member_1;  
    member_2;  
    ...  
};
```

```
union name {  
    member_1;  
    member_2;  
    ...  
};
```

- struct** is the required keyword
- name** is the name of the structure
- member_1, member_2, ...** are individual member declarations
- union** is the required keyword
- name** is the name of the union
- member_1, member_2, ...** are individual member declarations

Structure vs Union

- Union members share the same memory location.
- Making changes in one will be reflected to other members as well.

```
struct student {  
    int ID;  
    char Name[20];  
    char Group[3];  
};
```

```
union student {  
    int ID;  
    char Name[20];  
    char Group[3];  
};
```

- If A's addr: A
- B's addr: A+4
- C's addr: A+24

- If A's addr: A
- B's addr: A
- C's addr: A

Example

```
#include <stdio.h>
int main()
{
    union num {
        int a;
        char b;
    } var;
    var.a = 65;
    printf ("a = %d\n", var.a);
    printf ("b = %c\n", var.b);
    printf ("Size = %d\n", sizeof(var));
    return 0;
}
```

OUTPUT:

a = 65
b = A
Size = 4

Some applications of Unions

- To create an array containing mixed type data?

```
typedef struct{  
    int a;  
    char b;  
    double c;  
}data;  
  
int main(){  
    data arr[10];  
    arr[0].a = 10;  
    arr[1].b = 'a';  
    arr[2].c = 13.76;  
    return 0;  
}
```

Size of data = 16 bytes

```
typedef union{  
    int a;  
    char b;  
    double c;  
}data;  
  
int main(){  
    data arr[10];  
    arr[0].a = 10;  
    arr[1].b = 'a';  
    arr[2].c = 13.76;  
    return 0;  
}
```

Size of data = 8 bytes

Applications of Unions contd..

- Size reduction

```
struct library{  
    char Name[20];  
    int ID;  
    char Group[3];  
    float price;  
    char ISBN[30];  
};
```

Size of data = 64 bytes

```
struct library{  
    char Name[20];  
    union{  
        struct{  
            int ID;  
            char Group[3];  
        } student;  
        struct {  
            float price;  
            char ISBN[30];  
        } book;  
    } item;  
};
```

Size of data = 56 bytes



Enumerator in C

Enumerator in C

- Consider this structure definition:

```
struct student {  
    int ID;  
    char Name[20];  
    char Group[3];  
};
```

- The member **Group** of the above structure has been declared as a character array of 3 characters, which can take any values.

- However, we know that there are only limited number of groups (or disciplines) offered in our Campus. They include:-
{A1, A2, A3, A4, A5, A7, A8, B1, B2, B3, B4, B5, D2}
- Can we do something to restrict the values assigned to the member **Group** to always be from the above set?
- enum** is the solution!

Enumerator in C

```
typedef enum {A1,A2,A3,A4,A5,A7,A8,B1,B2,B3,B4,B5,D2}  
Group_lbl;
```

New definition to struct student:

```
struct student {  
    int ID;  
    char Name[20];  
    Group_lbl Group;  
};
```

Now the member variable Group can take a value only from the above list.

Each Value inside enum is actually an integer, i.e., A1=0, A2=1, ... and so on.
We can check by printing them using %d.

We can also assign custom numbers to each of the values of enum, like:-

```
typedef enum {A1=1,A2=2,A3=5,A4=10,...} Group_lbl;
```

Example

```
#include<string.h>

typedef enum {A1,A2,A3,A4,A5,A7,A8,B1,B2,B3,B4,B5,D2} Group_lbl;

struct student {
    int ID;
    char Name[20];
    Group_lbl Group;
};

int main() {
    struct student s1 = {876, "Karthik", A1};
    struct student s2;
    s2.ID = 233;
    strcpy(s2.Name, "Ganesh");
    s2.Group = B5;
    if(s2.Group == B5) printf("s2 is studying Physics\n");
    return 0;
}
```

Same Example

```
#include<stdio.h>
#include<string.h>

typedef enum {A1,A2,A3,A4,A5,A7,A8,B1,B2,B3,B4,B5,D2} Group_Lbl;
struct student {
    int ID;
    char Name[20];
    Group_Lbl Group;
};

int main(){
    struct student s1 = {876, "Karthik", A1};
    struct student s2;
    s2.ID = 233;
    strcpy(s2.Name,"Ganesh");
    s2.Group = B5;
    if(s2.Group == B5)    printf("%s is studying Physics \n", s2.Name);
    else    printf("%s is studying CSE \n", s1.Name);
    return 0;
}
```

Exercise

```
#include<stdio.h>
enum week{Mon, Tue, Wed, Thu, Fri, Sat, Sun};
int main()
{
    enum week day;
    day = Wed;
    printf("%d", day);
    return 0;
}
```

Output?

```
#include<stdio.h>
enum week{Mon, Tue, Wed, Thu, Fri, Sat, Sun};
int main()
{
    enum week CP_day;
    CP_day = Mon+Wed+Fri;
    printf("%d",CP_day);
    return 0; }
```

```
#include<stdio.h>
enum week{Sun, Mon, Tue, Wed, Thu, Fri, Sat};
int main()
{
    int i;
    for (i=Sun; i<=Sat; i++)
        printf("%d ", i);
    return 0;
}
```



Thank you
Q & A

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA



BITS Pilani

Pilani Campus

Department of Computer Science & Information Systems

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA

Pointers in C

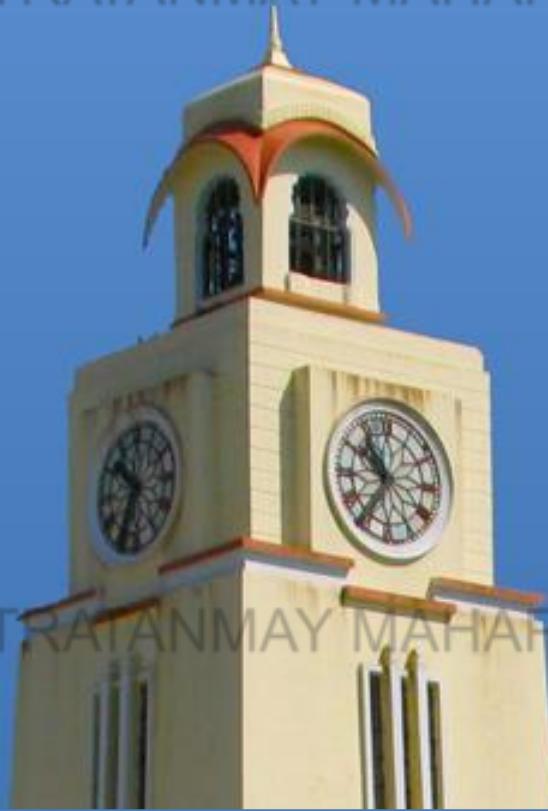
Pointers are the beauty of C

Why?

- ✓ **Allows memory level operations**
 - ✓ **Very few Programming languages allow this: C, C++, Fortran**
- ✓ **Enables “Pass by Reference”**
- ✓ **Enables us to return multiple data items from functions**
 - ✓ **Like arrays, large complex (data) structures**
- ✓ **Enables dynamic memory allocation at run-time**
 - ✓ **You don't need to fix the input size at the time of programming**
- ✓ **Many More...**

Module Overview

- **Pointers in C**
- **Pointer Arithmetic**
- **Arrays and Pointers**
- **Structures and Pointers**
- **Pass by Reference**



Pointers in C

Addresses and Pointers

Consider a variable declaration in the following program:

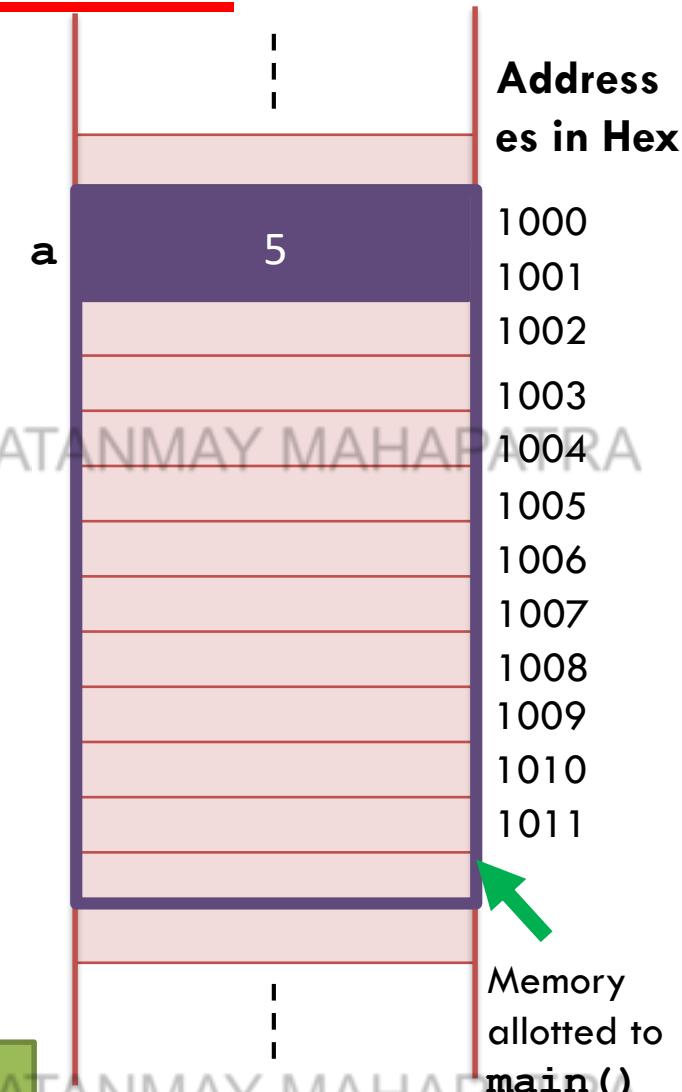
```
int main() {
    int a = 5;
}
```

Say, the variable “**a**” occupies 2 bytes starting at memory location whose address is **1000 (in hexa-decimal)**. (Assuming 16-bit addresses)

This address can be accessed by “**&a**”

```
printf("Value of a: %d", a);      5
printf("Address of a: %p", &a);  1000
```

%p used to print addresses



Addresses and Pointers

- The address of this variable **a**, can be stored in a variable called a **pointer variable**

```
int * ptr = &a;
```

- ptr** is a pointer variable of integer type. It is capable of storing the address of an integer variable.

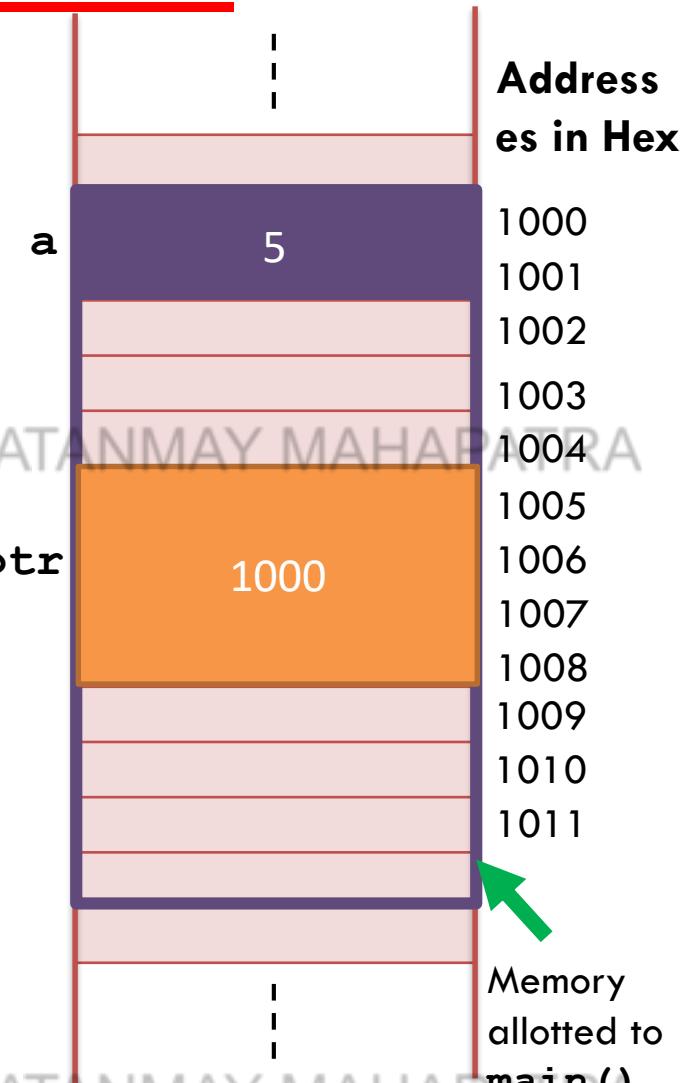
- We can access the value stored in the variable **a** by ***ptr**

```
printf("Value of a: %d", *ptr); 5
```

- *ptr** translates to **value at ptr**.

(**de-referencing**)

- Pointer variable of any type typically occupies 4 bytes (or 8 bytes) in memory depending upon the compiler.



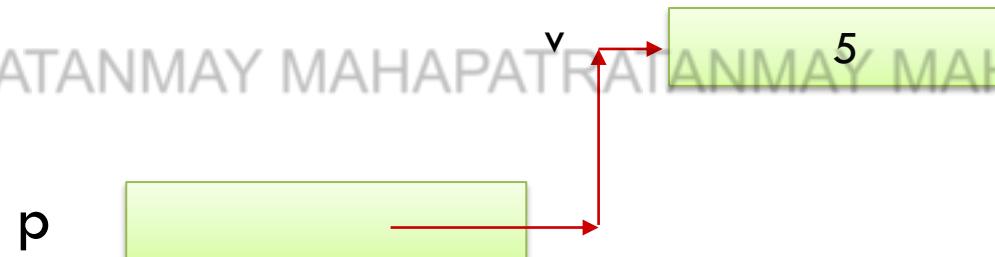
Simplifying with an example

Given declarations

int v;

int *p;

p = &v; is a valid assignment statement.



What is the effect of the following?

v = 5;

```
#include <stdio.h>
int main(){
    int a=10, *p; p=&a;
    printf("%d\t%p\t%u\t%x\t%lu\n", *p, p, p, p,
    sizeof(p)); return 0;}
```

10

0x7ffe303d47ac

809322412

303d47ac

8

Example

```
#include <stdio.h>
int main()
{
    int i=5, *p;
    p=&i;
    printf("Address= %p, %p %p", &i, p, &p);
    printf("\nsize of = %d, %d", sizeof(i), sizeof(p));
    printf("\nvalue of = %d, %d %d", i, p, *p);
    return 0;
}
```

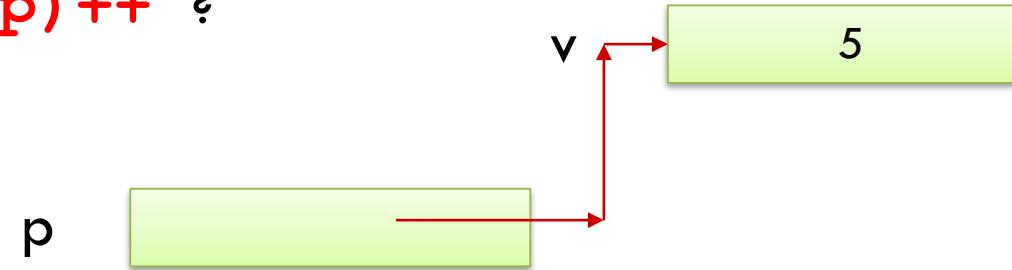
```
Address= 0x7ffebd977bac, 0x7ffebd977bac 0x7ffebd977bb0
size of = 4, 8
value of = 5, -1114145876 5
```

Simplifying with an example...

Now, what is the effect of $(*p)++$?

$(*p)++$ is the same as

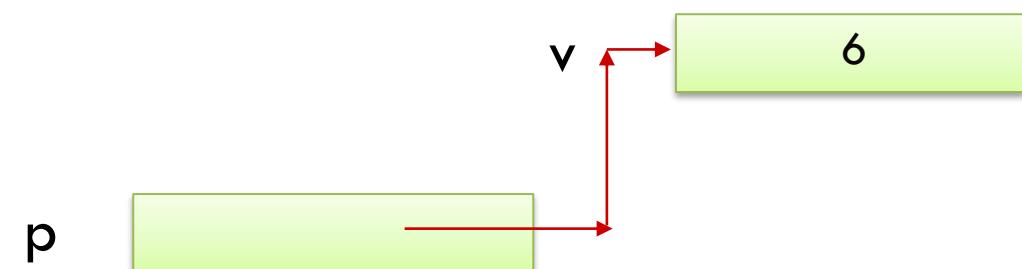
$*p = *p + 1;$



$*p$ (i.e., contents of p) is 5;

And it is changed to 6;

So v is also 6



Example with float

```
float u,v;           // floating-point variable declaration  
float * pv;          // pointer variable declaration  
.....  
pv = &v;              // assign v's address to pv
```

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA

u and **v** are floating point variables

pv is a pointer variable which points to a floating-point quantity

Another Example

```
int main() {
    int v = 3, *pv;
    pv = &v;

    printf("v=%d, pv=%p, *pv=%d\n", v, pv, *pv);

    v=v+1;

    printf("v=%d, pv=%p, *pv=%d\n", v, pv, *pv);
}
```

Note: **%p** prints contents of a pointer variable which is the address

Output:

v=3, pv=0x7ffc57c45a78, *pv=3
v=4, pv=0x7ffc57c45a78, *pv=4

64-bit addresses



Pointer Arithmetic

Size of a Pointer

	Size
Borland C / Turbo C	2 bytes
32 – bit architecture	4 bytes
64 – bit architecture	8 bytes

Modern Intel and AMD Processors are typically 64-bit architectures

Pointer Arithmetic

- **Incrementing a pointer**
 - $\text{NewPtr} = \text{CurrentPtr} + N \text{ bytes}$
 - Where N is size of pointer data type.

Example:

```
int a = 5;  
int * p = &a; // assume &a = 1000 (assume 16-bit addresses)  
int * q = p + 1;  
printf("printing ptr: %p", p);           1000  
printf("printing ptr: %p", q);           1002
```

Incrementing **ptr** will increase its value by 2 as int is of 2 bytes (assume)

What will be printed by the above print statements?

Pointer Arithmetic

- **Adding K to a pointer**

- NewPtr = CurrentPtr + K * N bytes.
- Where K is a constant integer
- N is size of pointer data type.

We will see its application when we study arrays with pointers!

Example:

```
int a = 5;  
int * p = &a; // assume &a = 1000 (assume 16-bit addresses)  
int * q = p + 4;  
printf("printing ptr: %p", p);  
printf("printing ptr: %p", q);
```

1000
1008

ptr will increase its value by 8 as int is of 2 bytes and $4*2 = 8$

What will be printed by the above print statements?

Pointer Arithmetic

- What does $*p++$ and $*p+q$ do?
 - Need Precedence and Associativity Rules to decide
- Rule 1: Unary operators have higher precedence than binary operators

— So, $*p+q$ is the same as $(*p) + q$

- Rule 2: All Unary operators have the same precedence.
 - So, we still need associativity to decide $*p++$
- Rule 3: All unary operators have right associativity.
 - So, $*p++$ is the same as $* (p++)$
- What if you want to increment the contents?
 - Use $(*p)++$

Operator Precedence

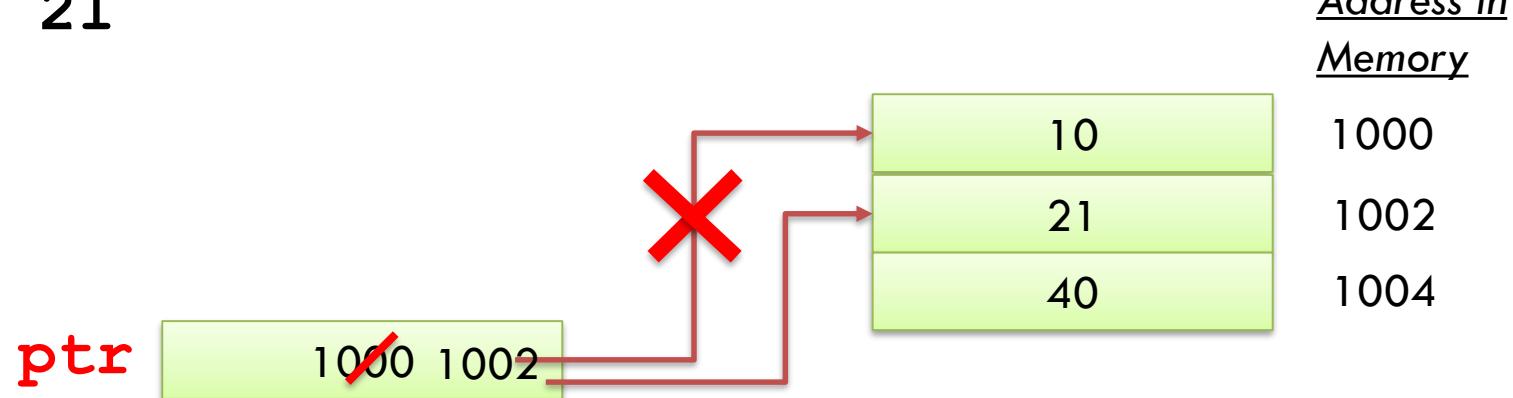
```
() [] -> .
++ -- + - ! ~          (unary)
(type) * & sizeof
* / %
+ -
<< >>
<<= >=
== !=
&
^
|
&&
||
?:
= += -= *= /= %=
&= ^= |= <<= >>=
,
```

Now, let us see various cases of incrementing a pointer variable, based on this precedence

Incrementing a pointer variable: Case 1

```
int c1 = *++ptr;  
// c1 = *(++ptr);  
// increment ptr and dereference its (now  
incremented) value
```

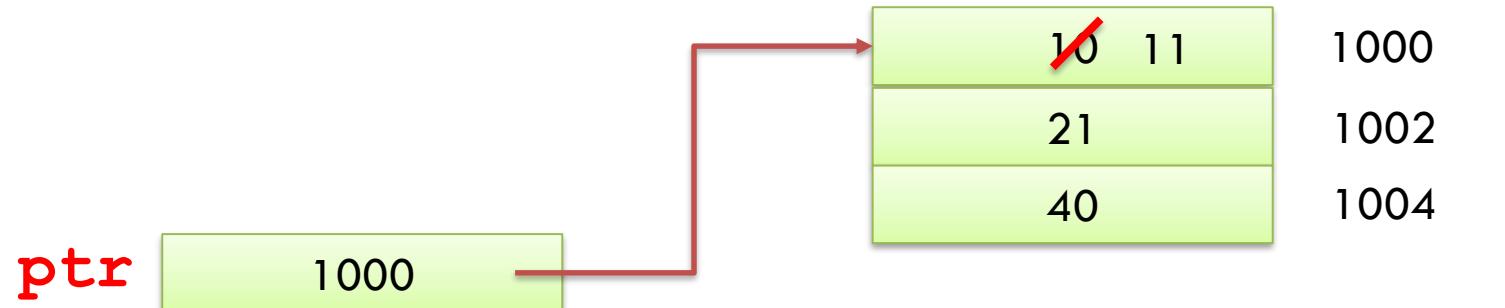
```
// c1 = 21
```



Incrementing a pointer variable: Case 2

```
int c2 = ++*ptr;  
// c2 = ++(*ptr);  
// dereference ptr and increment the  
dereferenced value
```

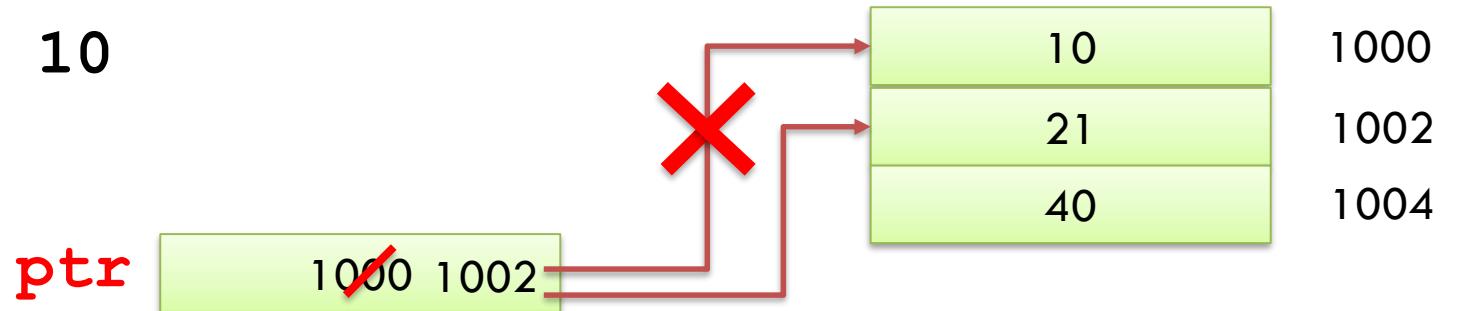
```
// c2 = 11
```



Incrementing a pointer variable: Case 3

```
int c3 = *ptr++; // or int c3 = *(ptr++);  
// both of the above has same meaning  
// c3 = *ptr; ptr = ptr+1;  
// dereference current ptr value and  
increment ptr afterwards
```

// c3 = 10



Example

```
#include<stdio.h>
void main ()
{
    int c3, a =10, *ptr;
    ptr=&a;
    printf("\nAddress %u", ptr);
    c3=*ptr++;
    printf("\nc3=%d ptr=%d", c3, *ptr);
    printf("\nAddress %u", ptr);
    return 0;
}
```

```
Address 1344471400
c3=10 ptr=10
Address 1344471404
```

Example

```
#include<stdio.h>
void main ()
{
    int c3, a =10, *ptr;
    ptr=&a;
    printf("\nAddress %u", ptr);
    c3=*ptr++;
    printf("\nc3=%d ptr=%d", c3, *ptr);
    printf("\nAddress %u\t %d", ptr, *ptr);
    printf("\nAddress %u\t %d", --ptr, *ptr);
    return 0;
}
```

Address	3624318680	
c3=10	ptr=10	
Address	3624318684	10
Address	3624318680	10

Explained Example

```
#include <stdio.h>

int main()
{
    int a[5]={11,22,33,44,55}, *p;
    p=a;

    printf(" %d", *p++);
    printf("\n %d", *p);

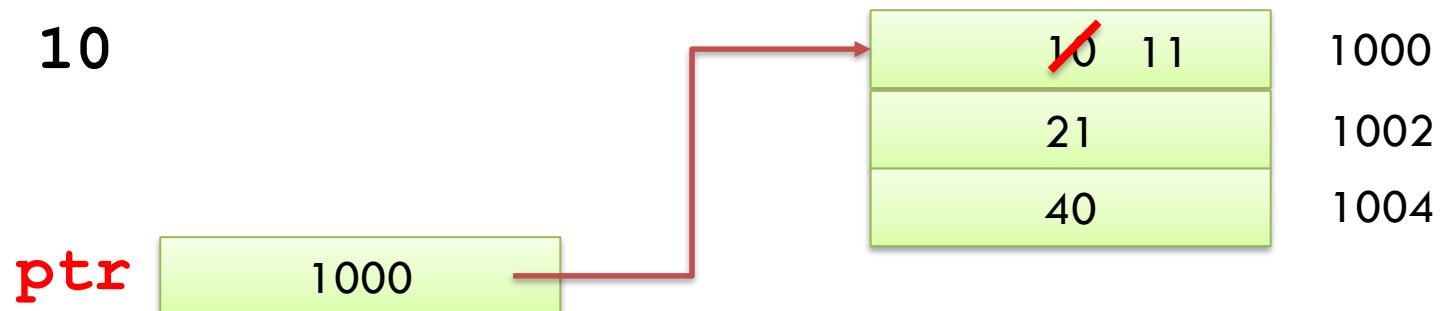
    return 0;
}
```

11
22

Incrementing a pointer variable: Case 4

```
int c4 = (*ptr)++;  
// c4 = *ptr; *ptr = *ptr + 1;  
// dereference current ptr value and increment  
the dereferenced value - now we need  
parentheses
```

```
// c4 = 10
```



Example

```
#include<stdio.h>
int main ()
{
    int c3, c4, a =10, *ptr;
    ptr=&a;
    printf("\nAddress %u", ptr);

    c3=*ptr++;
    printf("\nc3=%d ptr=%d", c3, *ptr);
    printf("\n c3 address=%u ptr Address %u", ptr,c3);

    c4=(*ptr)++;
    printf("\nc4=%d ptr=%d", c4, *ptr);
    return 0;
}
```

```
Address 1490786660
c3=10 ptr=10
    c3 address=1490786664 ptr Address 10
c4=10 ptr=11
```

Example

```
int main() {  
    int v = 3, *pv;  
  
    pv = &v;
```

Note the difference of 4 bytes in the addresses. In this example, size of int is assumed to be 4 bytes.

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA

```
*pv++;
```

```
printf("v=%d, pv=%p, *pv=%d\n", v, pv, *pv);  
}
```

Output:

v=3, pv=0x7ffeb75a749c, *pv=3

v=3, pv=0x7ffeb75a74a0, *pv=-1218808672

garbage

Example – Variation 1

```
int main() {  
    int v = 3, *pv;  
  
    pv = &v;  
  
    printf("v=%d, pv=%p, *pv=%d\n", v, pv, *pv);  
    (*pv)++;  
  
    printf("v=%d, pv=%p, *pv=%d\n", v, pv, *pv);  
}
```

Output:

```
v=3, pv=0x7ffde009df1c, *pv=3  
v=4, pv=0x7ffde009df1c, *pv=4
```

No change to the address stored in the pointer variable. The value stored is incremented.

Example – Variation 2

```
int main() {  
    int v =3, *pv;  
  
    pv = &v;  
  
    printf("v=%d, pv=%p, *pv=%d\n", v, pv, *pv);  
    ++*pv;  
  
    printf("v=%d, pv=%p, *pv=%d\n", v, pv, *pv);  
}
```

Output:

```
v=3, pv=0x7ffffbb21f63c, *pv=3  
v=4, pv=0x7ffffbb21f63c, *pv=4
```

No change to the address stored in the pointer variable. The value stored is incremented.



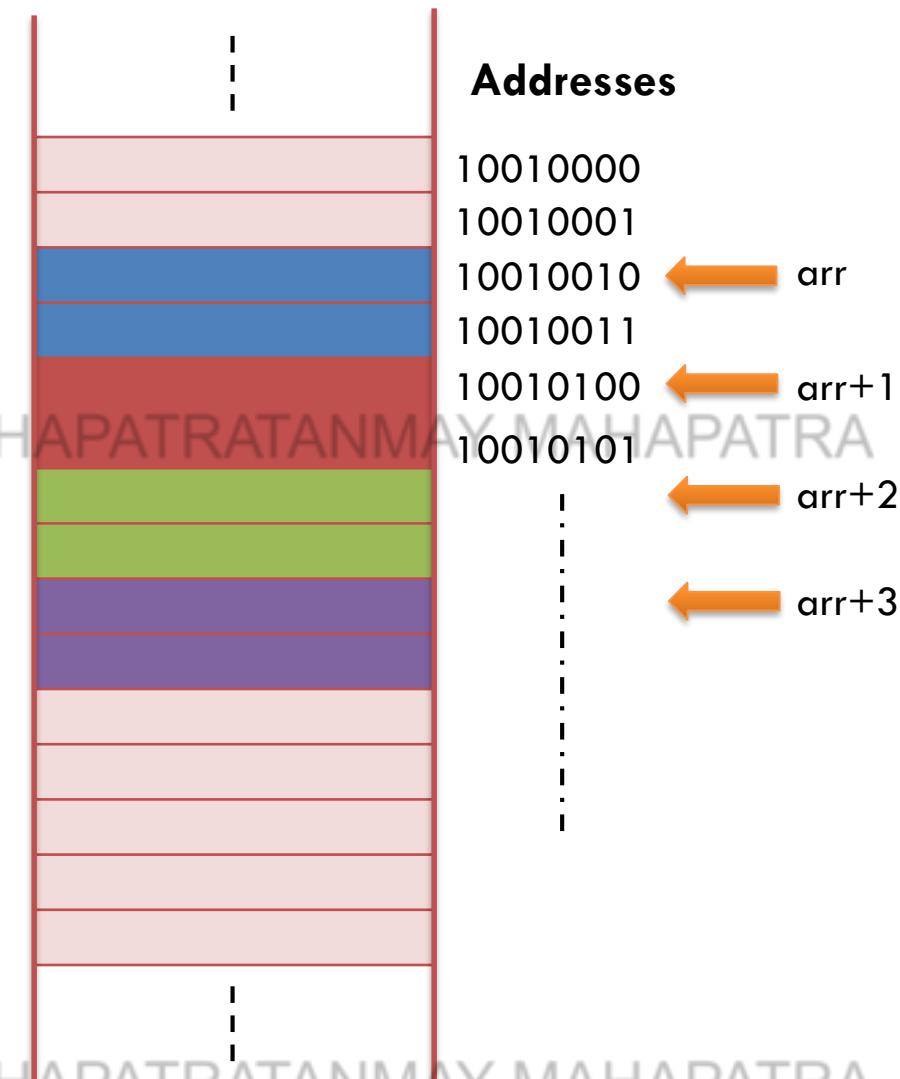
Arrays and Pointers

Arrays and Pointers

```
int arr[4] = {1,2,3,4};
```

- **arr** stores the address of the first element of the array
- **arr** is actually a pointer variable
- **arr+1** gives the address of the second element
- difference between **arr** and **arr+1** is actually 2 bytes
- **arr+2** gives the address of the third element and so on...

```
arr[0] = 1 {  
arr[1] = 2 {  
arr[2] = 3 {  
arr[3] = 4 {
```



Arrays and Pointers

Accessing elements of arrays

`arr[0]` is same as `*arr`

`arr[1]` is same as

`* (arr+1)`

`arr[2]` is same as

`* (arr+2)`

...

`arr[0] = 1` {

`arr[1] = 2` {

`arr[2] = 3` {

`arr[3] = 4` {

Address of first element:

`arr` or `&arr[0]`

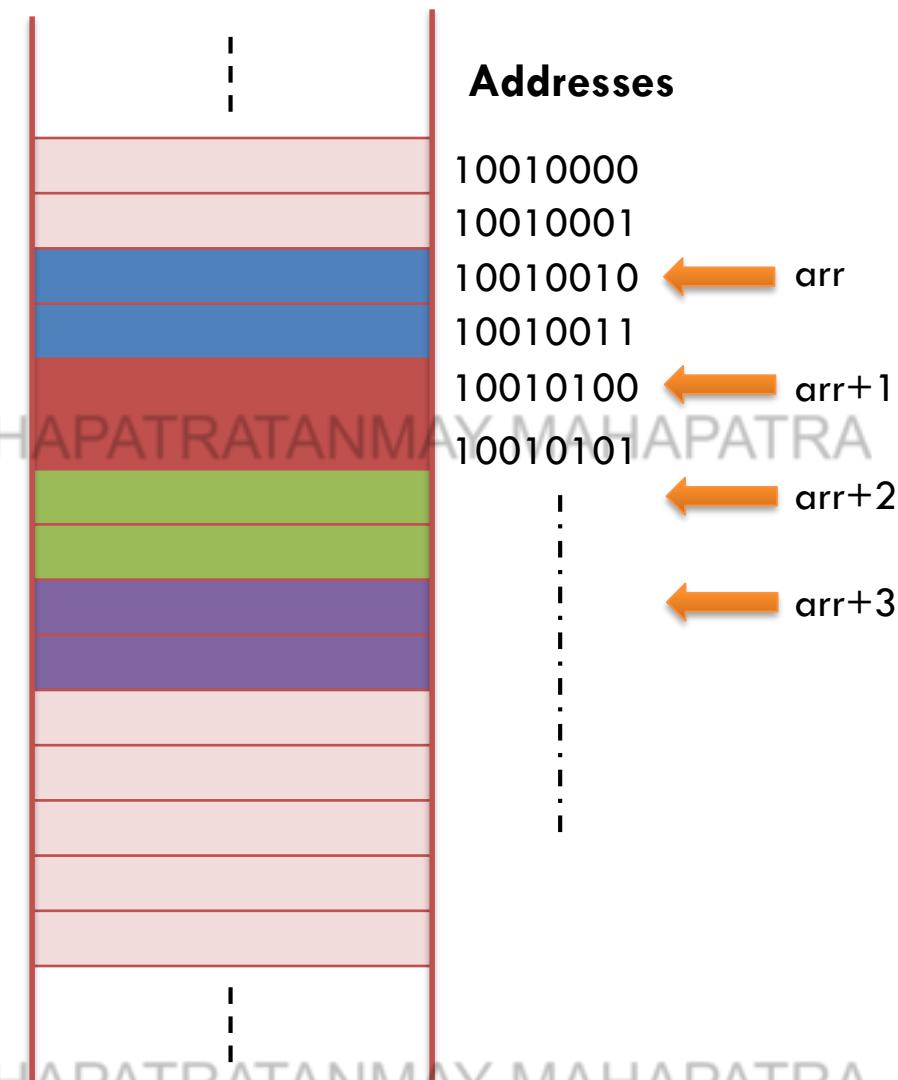
Address of second element:

`arr+1` or `&arr[1]`

Address of third element:

`arr+2` or `&arr[2]`

and so on...

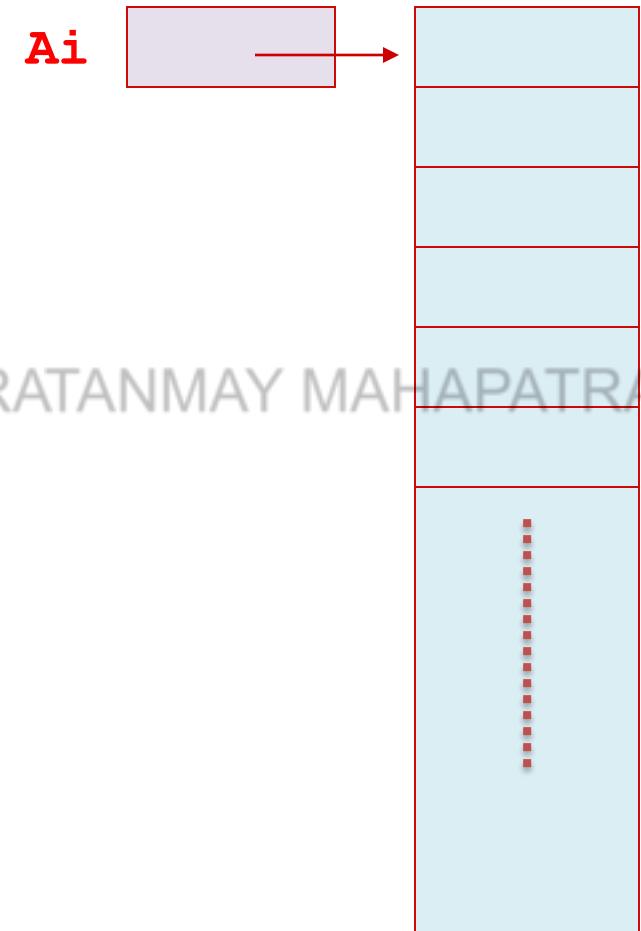


Simplifying with an example

Given

```
int Ai[100]; // array of 100 ints
```

Ai is the starting address of the array.



So, **Ai[5]** is same as ***(Ai+5)**

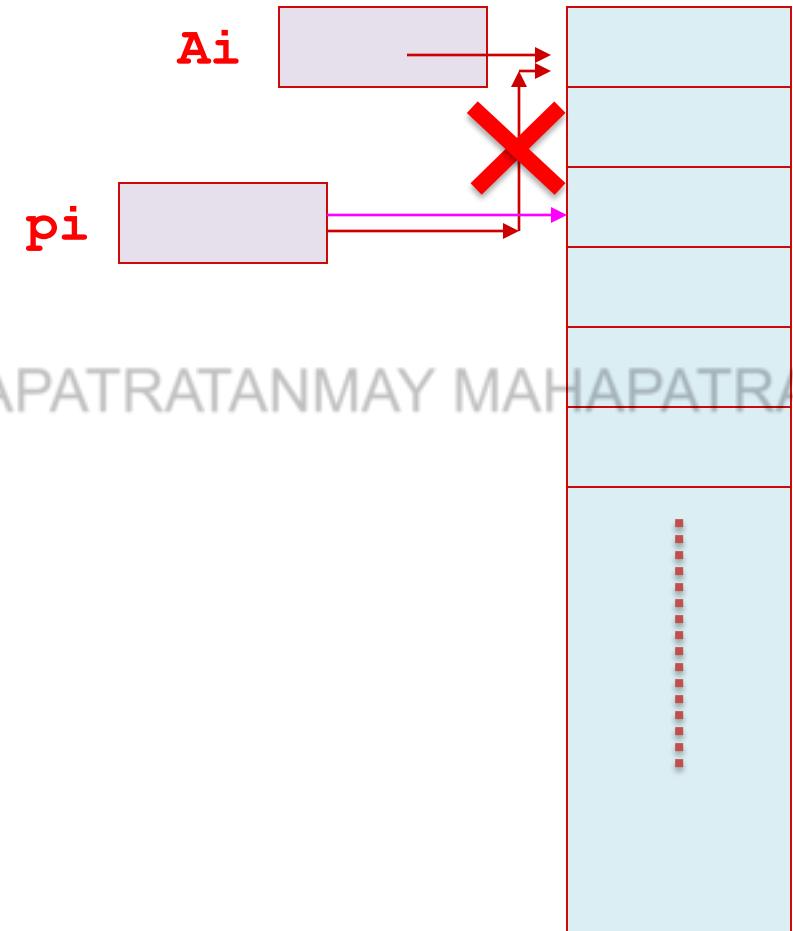
Observe that **Ai+5** is “address arithmetic”:

address **Ai** is added to int **5**
to obtain an address.

Simplifying with an example

Given

```
int Ai[100];  
int *pi;
```



the following are valid:

```
pi = Ai
```

```
pi = Ai +2
```

```
pi - Ai which will evaluate to 2
```

Example

```
#include <stdio.h>
int main()
{
    int a[100], *p, *q;
    a[90]=10;
    p=&a;
    q=a+90;
    printf("a[90]=%d, %d, %d %d %d", a[90], *p, *(p+90), *q, q-p);
    return 0;
}
```

=10, 0, 10 10 90

Example

```
int main(){
int *ptr, i, iA[5]={5,10,15,20,25};
for(i=0;i<5;i++)
    printf("iA[%d] : address=%p
           data=%d", i, &iA[i], iA[i]);
```

*This program
assumes sizeof(int)
to be 4 bytes*

```
// Accessing the Arrays using
// pointer
ptr = iA;
for(i=0;i<5;i++){
    printf("\npointer address = %p
           data = %d ", ptr, *ptr);
    ptr++;
}
return 0;
}
```

Output:

```
iA[0] : address=0x7ffd2adfbf10 data=5
iA[1] : address=0x7ffd2adfbf14 data=10
iA[2] : address=0x7ffd2adfbf18 data=15
iA[3] : address=0x7ffd2adfbf1c data=20
iA[4] : address=0x7ffd2adfbf20 data=25
pointer address = 0x7ffd2adfbf10 data = 5
pointer address = 0x7ffd2adfbf14 data = 10
pointer address = 0x7ffd2adfbf18 data = 15
pointer address = 0x7ffd2adfbf1c data = 20
pointer address = 0x7ffd2adfbf20 data = 25
```

Example: Array of Pointers

```
int main() {
    int *ptr[3], i, iA[]={5,10,15}, iB[]={1,2,3}, iC[]={2,4,6};
    ptr[0]=iA;
    ptr[1]=iB;
    ptr[2]=iC;
    for(i=0;i<3;i++) {
        printf("iA[%d] :addr=%p data=%d ",i,ptr[0]+i,* (ptr[0]+i));
        printf("iB[%d] :addr=%p data=%d ",i,ptr[1]+i,* (ptr[1]+i));
        printf("iC[%d] :addr=%p data=%d ",i,ptr[2]+i,* (ptr[2]+i));
    }
    return 0;
}
```

*This program assumes
sizeof(int) to be 4 bytes*

Output:

```
iA[0] :addr=0x7ffe7213707c data=5
iB[0] :addr=0x7ffe72137088 data=1
iC[0] :addr=0x7ffe72137094 data=2
iA[1] :addr=0x7ffe72137080 data=10
iB[1] :addr=0x7ffe7213708c data=2
iC[1] :addr=0x7ffe72137098 data=4
iA[2] :addr=0x7ffe72137084 data=15
iB[2] :addr=0x7ffe72137090 data=3
iC[2] :addr=0x7ffe7213709c data=6
```

Output ?

$\text{++}(*\text{ptr}) \Rightarrow ?$

$\text{*}(\text{++ptr}) \Rightarrow ?$

$\text{++*++ptr} \Rightarrow ?$

printf(" %d", ++*++p);

23

```
#include <stdio.h>
int main()
{
    int a[5]={11,22,33,44,55}, *p;
    p=a;
    printf(" %d", ++*p++);
    printf("\n %d", *p);
    return 0;
}
```

12
22

Another example

```
int main() {  
    int line[]={10,20,30,40,50};
```

```
    line[2]=*(line + 1);  
    *(line+1) = line[4];
```

```
    int *ptr;      ptr = &line[5];      ptr--;  
    *ptr = line[3];  
    *line=*ptr;
```

```
    for(int i =0;i<5;i++)  
        printf("%d ", *(line+i));  
    return 0;
```

Output:

40 50 20 40 40

Explained

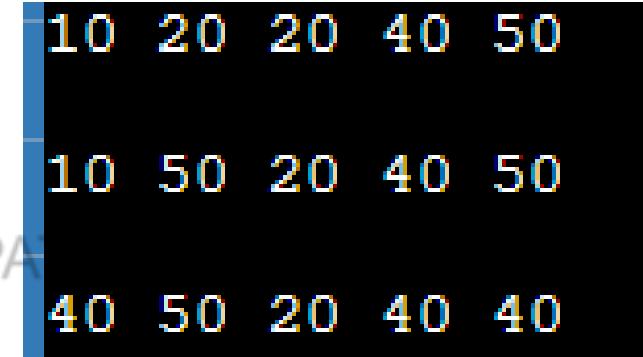
```
#include <stdio.h>
int main(){
    int line[]={10,20,30,40,50};

    line[2]=*(line + 1);
    for(int i =0;i<5;i++)
        printf("%d ", *(line+i));
    printf("\n\n") ;

    *(line+1) = line[4];
    for(int i =0;i<5;i++)
        printf("%d ", *(line+i));
    printf("\n\n") ;

    int *ptr;    ptr = &line[5];    ptr--;
    *ptr = line[3];
    *line= *ptr;

    for(int i =0;i<5;i++)
        printf("%d ", *(line+i));
    return 0; }
```



10	20	20	40	50
10	50	20	40	50
40	50	20	40	40

Example

```
#include <stdio.h>
int main()
{
    int a[5]={11,22,33,44,55}, *p;
    p=a;
    printf(" %d", ++p++);
    printf("\n %d", *p);
    return 0;
}
```

Compilation failed due to following error(s).

```
main.c: In function ‘main’:
main.c:6:19: error: lvalue required as increment operand
   6 |     printf(" %d", ++p++);
      |           ^~
```

Example

```
#include <stdio.h>
#define MAX 3
int main () {
    char *names[] = {
        "Pilani",
        "Goa",
        "Hyderabad"
    };

    int i = 0;
    char *p[3];
    for ( i = 0; i < MAX; i++) {
        p[i]=names[i];
        printf("Value of names[%d] = %s // p=%s\n", i, names[i], p[i] );
    }
    return 0;
}
```

Value of names[0] = Pilani // p=Pilani
Value of names[1] = Goa // p=Goa
Value of names[2] = Hyderabad // p=Hyderabad

Output?

```
#include <stdio.h>
#pragma pack(0)
```

```
struct base
{
    int a;
    char b;
};
```

```
int main()
{
    struct base var;
    printf("size of var: %ld\n", sizeof(var));
    return 0;
}
```

Pack(0) => 8
Pack(1) => 5



Pointers to Structures

Pointers to Structure

Consider the following structure definitions:

```
struct stud {  
    int roll;  
    char dept_code[25];  
    float cgpa;  
} class, *ptr;
```

ptr = &class ✓
br = &b; ✓

```
struct book {  
    char Name[20];  
    float price;  
    char ISBN[30];  
};  
struct book b, *br;
```

ptr = &b ✗
br = &class; ✗

Accessing members in pointers to Structures



- Once **ptr** points to a structure variable, the members can be accessed through dot(.) or arrow operators:

`(*ptr).roll, (*ptr).dept_code, (*ptr).cgpa`

OR

`ptr->roll, ptr->dept_code, ptr->cgpa`

Syntactically, `(*p).a` is equivalent to `p->a`

Illustration

```
struct stud {  
    int roll;  
    char dept_code[25];  
    float cgpa;  
} class, *ptr;
```

	roll	Dept_code	CGPA
class			
1000			

```
ptr = &class // ptr = 1000 (assumes 16-bit addresses)
```

Caveats

- When using structure pointers, we should take care of operator precedence.
- Member operator `". "` has higher precedence than `"*"`
- `ptr->roll` and `(*ptr).roll` mean the same thing.
`*ptr.roll` will lead to error
- The operator `"->"` has the highest priority among operators.
- `++ptr->roll` will increment `roll`, not `ptr`
- `(++ptr)->roll` will do the intended thing.

Explained

```
#include <stdio.h>
struct stud{
    int roll;
    char dept_code[25];
    float cgpa;
};
struct stud cla[2]={{7,"IT", 8.75f}, {17,"CSIS", 7.50f}};
int main() {
    struct stud *ptr=cla;
    printf("\n1 roll=%d", ptr->roll);
    printf("\n2 roll=%d", ++ptr->roll);
    printf("\n3 roll=%d", (++ptr)->roll);
    return 0;
}
```

Compilation failed due to following error(s).

```
main.c: In function 'main':
main.c:14:29: error: 'ptr' is a pointer; did you mean to use '>'?
    14 |     printf("\n1 roll=%d", *ptr.roll);
           | ^                                     ->
```

```
1 roll=7
2 roll=8
3 roll=17
```

Example

```
#include <stdio.h>
struct stud{
    int roll;
    char dept[25];
    float cgpa;
};
struct stud cla[2]={{7,"EEE", 8.75f}, {17,"CSIS", 7.50f}};
```

```
int main()
{
    struct stud *ptr=cla;

    printf("\n1 roll=%d", ptr->roll);

    printf("\n2 roll=%d", ++ptr->roll);

    printf("\nAddress cla=%p\t roll=%p\n", cla, &ptr->roll);

    printf("\n3 roll=%d", (++ptr)->roll);

    printf("\nAddress cla1=%p\t roll=%p\n", &cla[1], &ptr->cgpa);

    return 0;
}
```

```
1 roll=7
2 roll=8
Address cla=0x562e19ad7020      roll=0x562e19ad7020

3 roll=17
Address cla1=0x562e19ad7044     roll=0x562e19ad7064
```

Pointers to Array of Structures

```
struct stud {  
    int roll;  
    char dept_code[25];  
    float cgpa;  
} class[3], *ptr;
```

	roll	Dept_code	CGPA
1000 Class[0]			
1036 Class[1]			
1072 Class[2]			

```
ptr = class; // ptr = 1000, ptr+1 = 1036, ptr+2 = 1072
```

The assignment **ptr = class** assigns the address of **class[0]** to **ptr**

Example

```
struct stud {  
    int roll;  
    char dept_code[25];  
    float cgpa;  
};  
int main()  
{  
    int i=0;  
    struct stud sArr[3];  
    struct stud * ptr;  
    ptr = sArr;  
    printf("Enter the details");  
    for(i=0;i<3;i++)  
    {  
        scanf("%d", &(sArr[i].roll));  
        scanf("%s", sArr[i].dept_code);  
        scanf("%f", &(sArr[i].cgpa));  
    }  
    int avgCGPA = 0;  
    for(i=0;i<3;i++)  
    {  
        avgCGPA += (ptr+i)->cgpa;  
    }  
    avgCGPA /=3;  
    printf("AvgCGPA=%d",avgCGPA);  
    return 0;  
}
```



Call/Pass by reference Variable, Arrays and Structures

Swapping two variables using a function: Attempt 1 – Pass by value

First Attempt: Pass by value

```
void swap(int x, int y){  
    int temp = x;  
    x = y;  
    y = temp;  
}
```

```
int main(){  
    int a = 10, b = 20;  
    printf("Before Swapping %d %d\n", a, b);  
    swap(a, b);  
    printf("After Swapping %d %d\n", a, b);  
    return 0;  
}
```

Output:

10 20

10 20

- The values of **a** and **b** get copied into **x** and **y**
- The swapping of **x** and **y** doesn't get reflected back in **a** and **b** when **swap()** function returns
- Also, we can't return **x** and **y** to **main()** function as C supports return of a single variable.

Alternative Solution

```
#include <stdio.h>
int a = 10, b = 20;

void swap(){

    int temp =a;
    a= b;
    b = temp;  }

int main()
{
printf("Before Swapping %d %d\n", a, b);
swap();
printf("After Swapping %d %d\n", a, b);
return 0;
}
```

```
Before Swapping 10 20
After Swapping 20 10
```

Swapping two variables using a function:

Attempt 2 – Pass by reference

Second Attempt: Pass by reference

```
void swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}
```

```
int main()
{
    int a = 10, b = 20;
    printf("Before Swapping %d %d\n", a, b);
    swap(&a, &b);
    printf("After Swapping %d %d\n", a, b);
    return 0;
}
```

- The addresses of **a** and **b** get copied into **x** and **y**
- The swapping of ***x** and ***y** gets reflected back in **a** and **b** when **swap()** function returns

Passing Arrays into functions

Passing arrays into functions is by default **call by reference**.

```
void sort(int a[]) {
    int temp, i , j, sorted = 0;
    for(i = 0; i < SIZE-i-1; i++){
        for(j = 0; j<SIZE-1-i; j++){
            if(a[j] > a[j + 1])
            {
                temp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = temp;
            }
        }
    }
}
```

When arrays are passed as parameters, you pass the address of the first location which the array variable name

```
int main() {
    int arr[8] = {2,5,9,7,1,5,4,6};
    int SIZE = 8;
    printf("Array before sort: \n");
    for (i = 0; i < SIZE; i++)
        printf("%d ", arr[i]);
    printf("\n");
    sort(arr);
    printf("Array after sort: \n");
    for (i = 0; i < SIZE; i++)
        printf("%d ", arr[i]);
    printf("\n");
    return 0;
}
```

sort() function implements bubble sort which is one of the sorting algorithms. Don't worry about it.

Arrays as input parameter

```
void sort(int a[]) {  
    int temp, i , j, sorted = 0;  
    for(i = 0; i < SIZE-i-1; i++){  
        for(j = 0; j<SIZE-1-i; j++){  
            if(a[j] > a[j + 1])  
            {  
                temp = a[j];  
                a[j] = a[j + 1];  
                a[j + 1] = temp;  
            }  
        }  
    }  
}
```

```
void sort(int * a) {  
    int temp, i , j, sorted = 0;  
    for(i = 0; i < SIZE-i-1; i++){  
        for(j = 0; j<SIZE-1-i; j++){  
            if(a[j] > a[j + 1])  
            {  
                temp = a[j];  
                a[j] = a[j + 1];  
                a[j + 1] = temp;  
            }  
        }  
    }  
}
```

Both are equivalent...

Selection sorting

```
#include <stdio.h>
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
void selectionSort(int *array, int size) {
    for (int stp = 0; stp < size - 1; stp++) {
        int min_idx = stp;
        for (int i = stp + 1; i < size; i++)
            { // Select the min element in each loop.
                if (array[i] < array[min_idx])
                    min_idx = i;
            }
        // put min at the correct position
        swap(&array[min_idx], &array[stp]);
    }
}
```

Sorted array in Ascending Order using Selection Sorting:

10 12 15 16 19

```
void printArray(int *array, int size) {
    for (int i = 0; i < size; ++i) {
        printf("%d ", *(array+i));
    }
    printf("\n");
}

int main() {
    int data[] = {16, 12, 10, 15, 19};
    int size = sizeof(data) / sizeof(int);
    selectionSort(data, size);
    printf("Sorted array in Acsending Order using
Selection Sorting:\n");
    printArray(data, size);
}
```

Example: Computing length of a string

- Applications of Pointer arithmetic:

```
int strlen(char *str) {  
    char *p;  
    for (p=str; *p != '\0'; p++);  
    return p-str;  
}
```

- Observe the similarities and differences with arrays.

```
int strlen(char str[]) {  
    int j;  
    for (j=0; str[j] != '\0'; j++);  
    return j;  
}
```

Explained

```
#include <stdio.h>
int strlen1(char *str) {
    char *p; int cnt=0;
    for (p=str; *p != '\0'; p++);
    printf("%u, %u", p,str);
    return p-str;
}
int main()
{
    char *ptr="Hello World";
    printf("\nlen1=%d", strlen1(ptr));
    return 0;
}
```

```
1629057046, 1629057035
len1=11
```

Character Arrays and Pointers: Example 2



```
Boolean isPalindrome(char *str) {  
  
    char *fore, *rear;  
    fore = str; rear = str + strlen(str) - 1;  
  
    for (; fore < rear; fore++, rear--)  
        if (*fore != *rear) return FALSE;  
  
    return TRUE;  
}
```

Pass by reference using structures

```
typedef struct{  
    int a;  
    float b;  
} ST;  
  
void swap(ST * p1, ST * p2){  
    ST temp;  
    temp.a = (*p1).a;  
    temp.b = (*p1).b;  
    (*p1).a = (*p2).a;  
    (*p1).b = (*p2).b;  
    (*p2).a = temp.a;  
    (*p2).b = temp.b;  
}
```

```
int main() {  
  
    ST s1, s2;  
    s1.a=10; s1.b=10.555;  
    s2.a=3; s2.b=3.555;  
  
    printf("s1.a:%d, s1.b:%f\n",s1.a,s1.b);  
    printf("s2.a:%d, s2.b:%f\n",s2.a,s2.b);  
  
    swap(&s1, &s2);  
  
    printf("s1.a: %d, s1.b:%f\n",s1.a,s1.b);  
    printf("s2.a: %d, s2.b:%f\n",s2.a,s2.b);  
}
```

Be careful of the precedence of “*” and “.” , with the latter having higher precedence.
You must use **()** if you want the operation to be correct

Pass by reference using structures (equivalent)

innovate

achieve

lead

```
typedef struct{  
    int a;  
    float b;  
} ST;  
  
void swap(ST * p1, ST * p2){  
    ST temp;  
    temp.a = p1->a;  
    temp.b = p1->b;  
    p1->a = p2->a;  
    p1->b = p2->b;  
    p2->a = temp.a;  
    p2->b = temp.b;  
}
```

```
int main() {  
  
    ST s1, s2;  
    s1.a=10; s1.b=10.555;  
    s2.a=3; s2.b=3.555;  
  
    printf("s1.a:%d, s1.b:%f\n",s1.a,s1.b);  
    printf("s2.a:%d, s2.b:%f\n",s2.a,s2.b);  
  
    swap(&s1, &s2);  
  
    printf("s1.a: %d, s1.b:%f\n",s1.a,s1.b);  
    printf("s2.a: %d, s2.b:%f\n",s2.a,s2.b);  
}
```

(*p1).a is equivalent to **p1->a**

Pass by reference using structures (another equivalent)

innovate

achieve

lead

```
typedef struct{  
    int a;  
    float b;  
} ST;  
  
void swap(ST * p1, ST * p2)  
{  
    ST temp;  
    temp = *p1;  
    *p1 = *p2;  
    *p2 = temp;  
}
```

```
int main() {  
  
    ST s1, s2;  
    s1.a=10; s1.b=10.555;  
    s2.a=3; s2.b=3.555;  
  
    printf("s1.a:%d, s1.b:%f\n",s1.a,s1.b);  
    printf("s2.a:%d, s2.b:%f\n",s2.a,s2.b);  
  
    swap(&s1, &s2);  
  
    printf("s1.a: %d, s1.b:%f\n",s1.a,s1.b);  
    printf("s2.a: %d, s2.b:%f\n",s2.a,s2.b);  
  
}
```



Pointers Revisited

Null Pointer

- Initialize a pointer variable to **NULL** when that pointer variable isn't assigned any valid memory address yet.
 - `int *p = NULL;`

- Check for a **NULL** pointer before accessing any pointer variable. By doing so, we can perform error handling in pointer related code.
- **NULL** pointer is also useful in implementation of linked lists.
 - *Next to next module.*

Generic or Void Pointer

Do we need different types of pointers to store the addresses of the variable of different types?

No!

```
void *void_ptr; int x = 5; float y = 2.5f;
```

```
void_ptr = &x;  
printf("\n x=%d",*(int *)void_ptr);
```

```
void_ptr = &y;  
printf("\n y=%f",*(float *)void_ptr);
```

Pointer to a Pointer

A pointer to a pointer is a form of multiple indirection, or a chain of pointers.

Eg. `int **ptr;`



Example

```
int main () {  
    int var, *ptr, **pptr;  
  
    var = 3000;  
    ptr = &var;  
    pptr = &ptr;
```

“pointer to a pointer” is useful in creating 2-D arrays with dynamic memory allocation (next module)

```
printf("Value of var = %d, *ptr = %d, **pptr = %d\n",  
    var,*ptr,**pptr );  
  
return 0;  
}
```

Output:

Value of var = 3000, *ptr = 3000, **pptr = 3000

Review Question

```
int main()
{
    int (*x1)[3];
    int y[2][3]={{1,2,3},{4,5,6}};
    x1 = y;
    for (int i = 0; i<2; i++)
        for (int j = 0; j<3; j++)
            printf("\n The X1 is %d and Y is %d",*(x1+i)+j), y[i][j]);
            // printf("\n The X1 is %d and Y is %d", x1[i][j], y[i][j]);
            // would also work
    return 0; }
```

Output:

```
The X1 is 1 and Y is 1
The X1 is 2 and Y is 2
The X1 is 3 and Y is 3
The X1 is 4 and Y is 4
The X1 is 5 and Y is 5
The X1 is 6 and Y is 6
```

Review Question

```
int main()
{
    int arr[3][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}};
    int i = 1, j = 2;

    printf("\n Data at * (arr+i)+j = %d", *(arr+i)+j);
    printf("\n Data at * (arr+i+j) = %d", *(arr+i+j));

    return 0;
}
```

Output:

```
Data at * (arr+i)+j = 7
Data at * (arr+i+j) = 1970957920
```

garbage

Review Question

```
int main()
{
    int x[] = {10,12,14};
    int *y, **z;

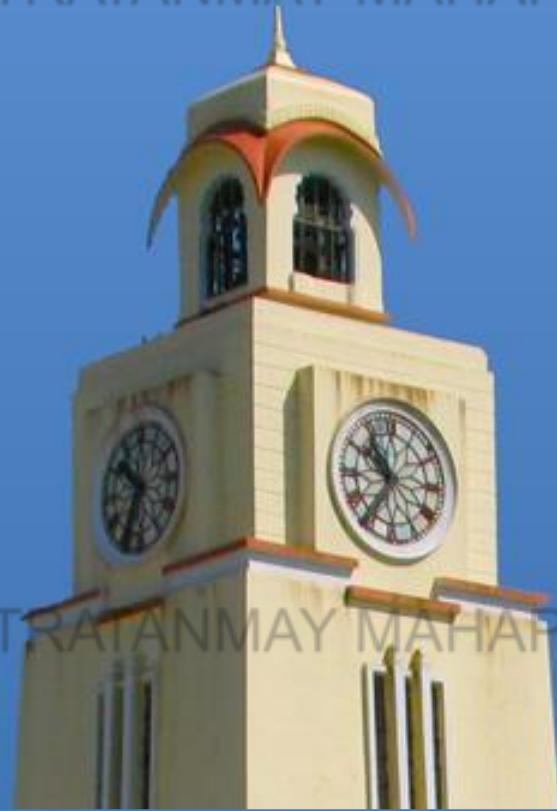
    y = x;
    z = &y;

    printf("x = %d, y = %d, z = %d\n", x[0], *(y+1), *(*z+2));
    printf("x = %d, y = %d, z = %d\n", x[0], *y+1, **z+2);

    return 0;
}
```

Output:

```
x = 10, y = 12, z = 14
x = 10, y = 11, z = 12
```



Thank you
Q & A

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA



Module 11 – Dynamic Memory Allocation

BITS Pilani

Pilani Campus

Department of Computer Science & Information Systems

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA

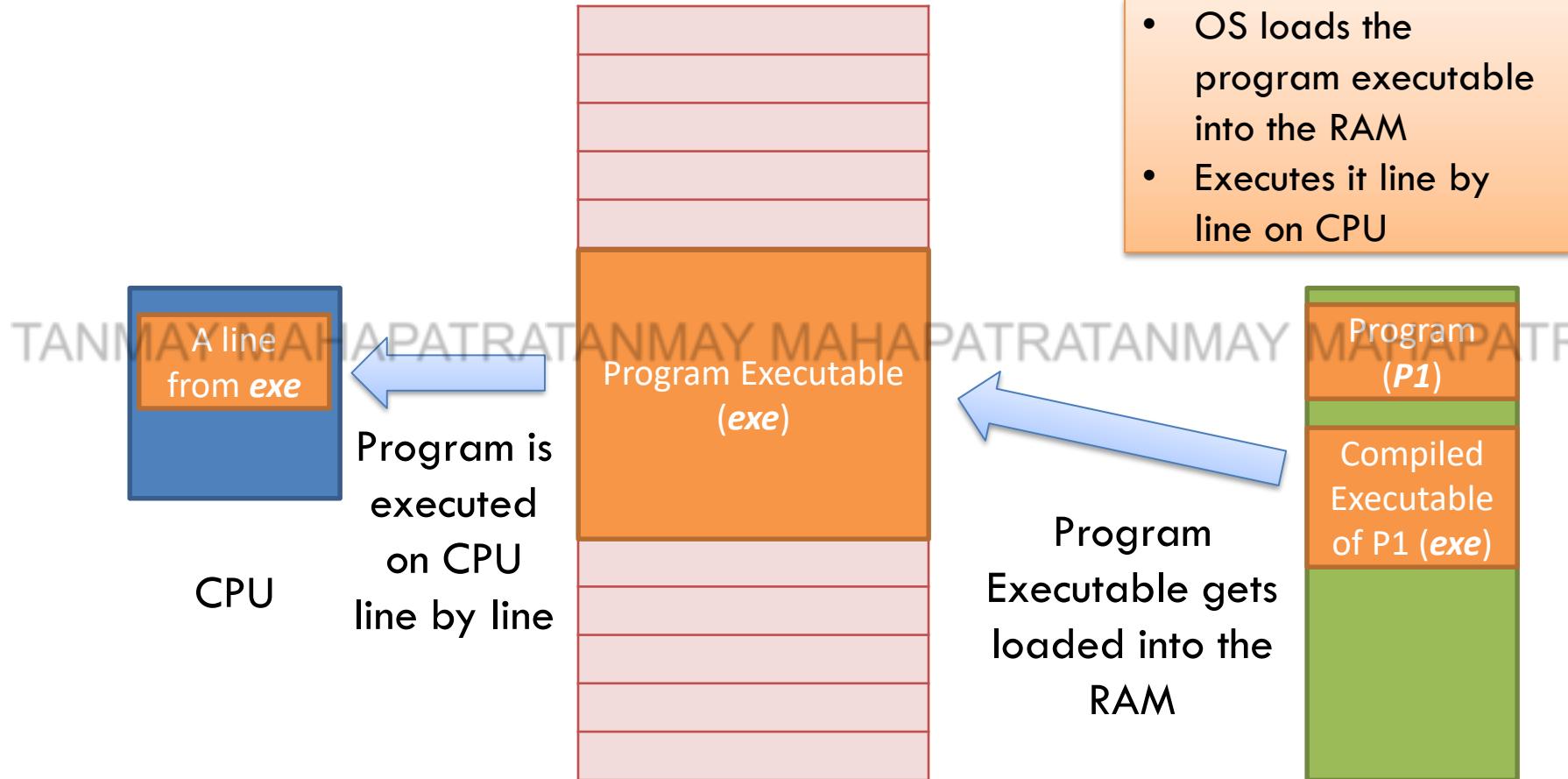
Module Overview

- **Stack vs Heap memory**
- **Dynamic Memory Allocation**
- **Dynamically allocated arrays**
- **Memory Management Issues**
- **Dynamically Allocated Arrays of Structures**
- **Multi-dimensional arrays using dynamic allocation**
- **Command Line Arguments**



Stack vs Heap Memory

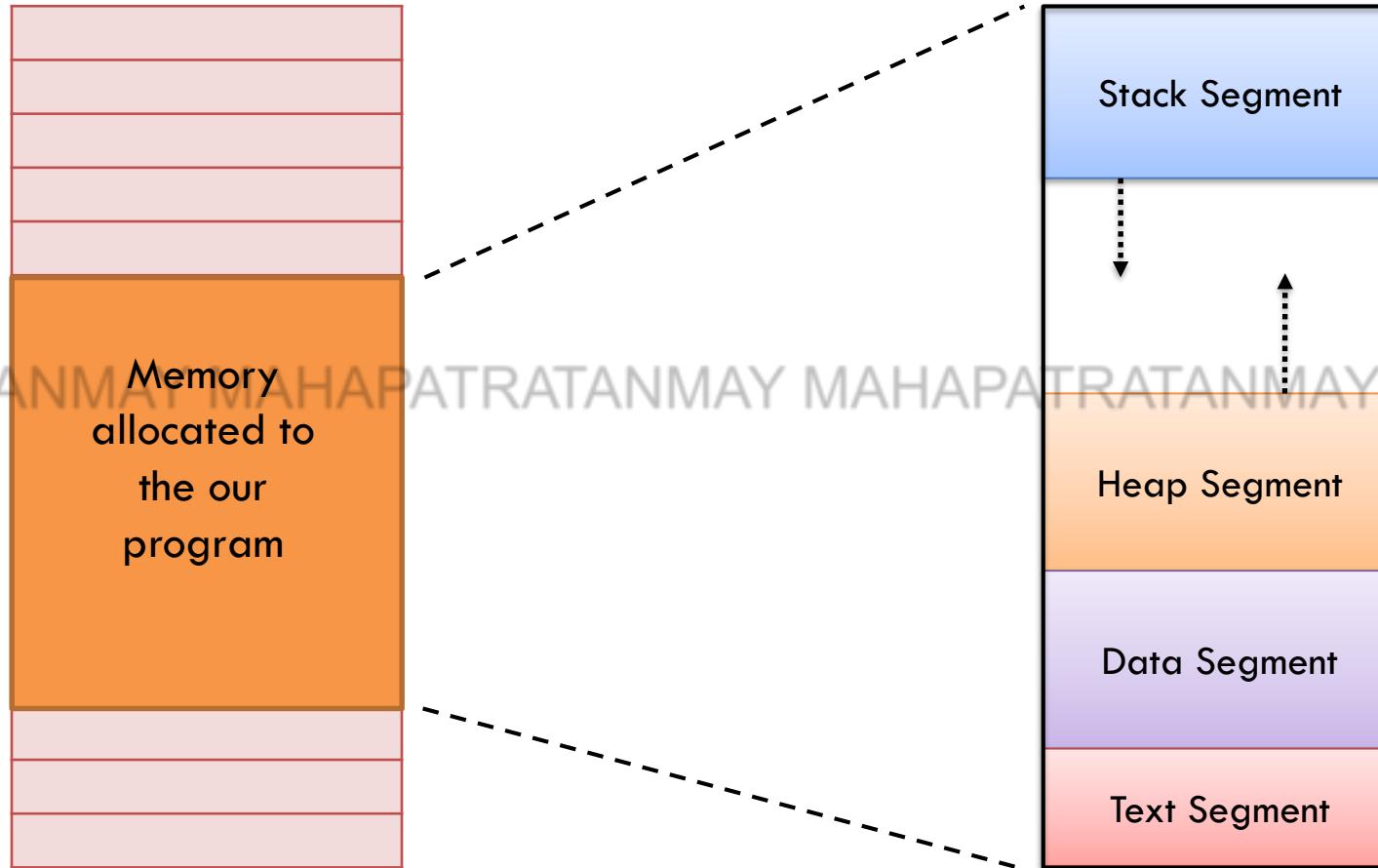
Our block diagram is back again!



Memory (RAM)

DISK

Looking at the main memory only



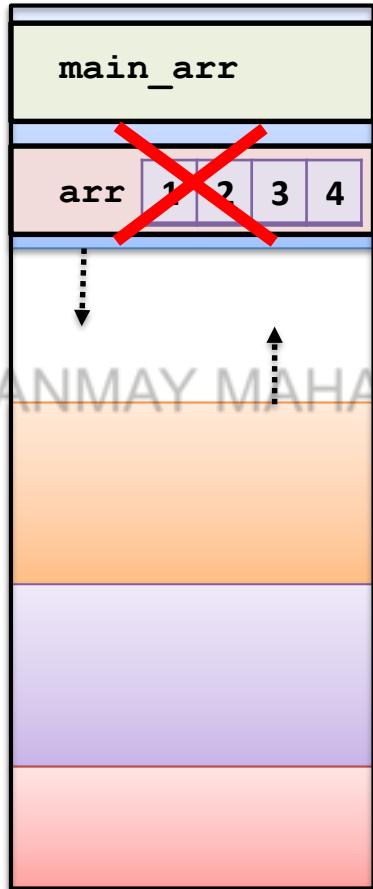
Stack vs Heap

Stack:

- One frame allocated to each function call
- Auto (or local) variable defined inside the frame allocated for a function
- Memory allocated to each frame is recalled after the function execution finishes

Let us try to explain with our memory diagram

Stack



} Stack frame allocated to
`main()` in stack
} Stack Frame allocated to
`f1()` in the stack

When `f1()` returns, the memory allocated to it is destroyed.
So `arr` declared inside `f1()` does not exist anymore. Accessing `arr` in `main` function now gives an error.

Consider this program:

```
int f1(){
    int arr[4] = {1, 2, 3, 4};
    return arr;
}
int main(){
    int main_arr[] = f1();
    printf("First ele is: %d", main_arr[0]);
    return 0;
}
```

Output:

Error!

Memory allocated
to our program

Advantage of Heap over Stack

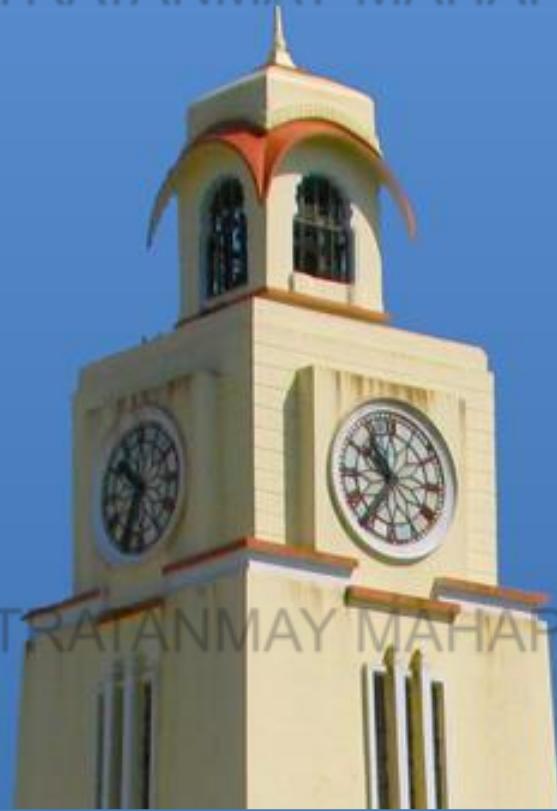
Stack:

- One frame allocated to each function calls
- Auto (or local) variable defined inside the frame allocated for a function
- Storage allocated for each frame is recalled after the function execution finishes

Heap:

- Heap is dynamically and explicitly allocated by programmer
- So allocated storage is not recovered on function return
- Programmer has to deallocate (at his/her convenience)

The memory allocated in heap is accessible globally to all functions



Dynamic Memory Allocation

Motivation

- **Limitation of Arrays**
 - Amount of memory is fixed at compile time and cannot be modified.
 - We tend to oversize Arrays to accommodate our needs.
 - We can't return an array from a function as the memory allocated to the function gets destroyed on return.
- **Solution:**
 - Allocate memory dynamically at run time.
 - The size of the array can be taken as user input
 - We can re-allocate the size of the array if it gets full
 - Array gets memory in heap, hence can be accessed by all functions of the program
- **Flexibility comes with a price:**
 - Programmer should free up memory when no longer required.

Using `malloc()` and `calloc()`

Using `realloc()`

Using `free()`

Allocating memory dynamically – malloc(): Syntax

`<stdlib.h>` is the header file for using `malloc()` and other dynamic memory allocation related operations

```
#include <stdlib.h>
void fun1()
{
```

Convert pointer to byte(s) (generic pointer) to the type pointer to int

```
    int * pt = (int *) malloc (sizeof(int));
}
```

Ask Operating System to allocate memory

Number of bytes of memory required

Creating a dynamically allocated array

```
#include <stdlib.h>
#define MAX_SIZE 10

void fun2()
{
    int * arr = (int *) malloc( MAX_SIZE * sizeof(int));
}
```

Number of bytes of memory required

malloc()

Observations:

- Library **stdlib** provides a function malloc
- **malloc** accepts number of bytes and returns a pointer (of type void) to a block of memory
- Returned pointer must be “**type-cast**” (i.e. type converted) to required type before use.
- **malloc** doesn’t initialize allocated blocks, each byte has a random value.
- The block of memory returned by **malloc** is allocated in heap

Reading between the lines:

- Number of bytes is dynamic – unlike in arrays where size is constant.
- Can number of bytes be arbitrary?
 - *No! Memory is of finite and fixed size!*

Example

```
short * p1;  
int * p2;  
float * p3;
```

}

Pointer variable
declaration

```
p1 = (short*) malloc(sizeof(short));  
p2 = (int*) malloc(sizeof(int));  
p3 = (float*) malloc(sizeof(float));
```

}

Allocating
memory to
pointer variable

```
*p1=256;  
*p2=100;  
*p3=123.456;
```

}

Assigning values

```
printf("p1 is %hd\n", *p1);  
printf("p2 is %d\n", *p2);  
printf("p3 is %f\n", *p3);
```

```
*p1 = 256  
*p2 = 100  
*p3 = 123.456001
```

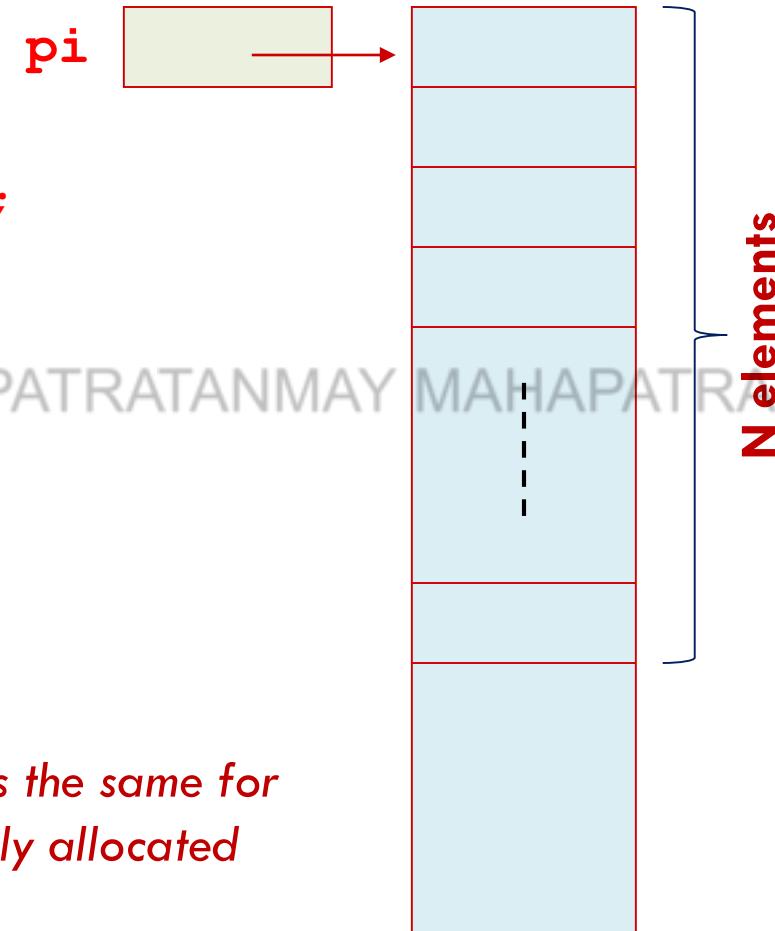


Dynamically Allocated Arrays

Dynamically Allocated Arrays

Given

```
int *pi;  
pi = (int *) malloc(N*sizeof(int));
```



Where does storage get allocated?

Not inside frames i.e., not inside call stack

But inside the heap

NOTE:

- ✓ Usage syntax for the array and its elements is the same for static arrays (arrays on stack) and dynamically allocated arrays.

Example

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    int *p,*q,i;

    p = (int*) malloc(5*sizeof(int));
    q = p; // assigning array p to q

    for (int i=0;i<5;i++)
        *p++ = i*i;
    for(int i=0;i<5;i++)
        printf("Element at index %d is %d\n",i,q[i]);

    // for(i=0;i<5;i++)
    // printf("Element at index %d is %d\n",i,* (q + i));
    return 0;
}
```

- This is not copying of array p into q.
- This means that the address contained in p is copied into q.
- Which means that now q also points to first location of the array p.

Both are equivalent

Dynamically Allocated Arrays

- We saw that arrays created inside a function could not be returned.
 - Since, arrays declared within a function are allocated in a frame and they are gone when function returns.
- Solutions:
 - Declare array outside function (in some other function) and pass as parameter (only starting address is passed)
 - Passed by reference, so changes get reflected in the calling function.
 - Or declare a pointer inside the function, allocate memory and return the pointer.
 - Since, memory is allocated in the heap, it will remain in the calling function as well.

Example: copy arrays - 1

The following program calls a function **copy()**, that copies the contents of one array into another.

```
#include <stdio.h>
#include <stdlib.h>

void copy(int c[], int n, int d[])
{
    for (int i=0; i<n; i++)
    {
        d[i]=c[i];
        // or *(b+i)=*(a+i) ;
    }
}

int main()
{
    int a[5] = {1,2,3,4,5};
    int b[5];

    copy(a,5,b);

    for(int i=0;i<5;i++)
    {
        printf("%d\t",b[i]);
    }
}
```

Example: copy arrays – 2 (using pointers)

```
#include <stdio.h>
#include <stdlib.h>

int * copy(int a[], int n)
{
    int * b = (int*) malloc(sizeof(int)*n);
    for (int i=0; i<n; i++)
    {
        b[i]=a[i];
        // or *(b+i)=*(a+i);
    }
    return b;
}
```

Can be replaced with
int * a

```
int main(){
    int a[5] = {1,2,3,4,5};
    int * b;
    b = copy(a,5);

    for(int i=0;i<5;i++){
        printf("%d\t",b[i]);
    }
}
```

Example: copy arrays – that is incorrect (using pointers)

```
#include <stdio.h>
#include <stdlib.h>

void copy(int * a, int n, int * c)
{
    c = (int*) malloc(sizeof(int)*n);
    for (int i=0; i<n; i++)
    {
        c[i]=a[i];
    }
}
```

Will lead to segmentation fault (run-time error). Why?

The contents of **b** were copied into **c** during function call. **b** was empty (or garbage value) when it was passed. The memory that was allocated in **copy()** function, its first element's address is copied to **c** during the **malloc()** call. But this was not copied to **b**, when **copy()** returned.

Solution:

- Allocate memory to **b** in **main()** and then pass. Try This!
- Allocate memory inside **copy()** and return. (previous slide)
- Use double pointer! (we will see later)

```
int main() {
    int a[5] = {1,2,3,4,5};
    int * b;
    copy(a,5,b);
    for(int i=0;i<5;i++) {
        printf("%d\t",b[i]);
    }
}
```

calloc()

- **calloc ()** is used specifically for arrays and structures.
- It is more intuitive than **malloc ()**.
- Example:

```
int *p = calloc (5, sizeof(int));  
// 2 arguments in contrast to malloc()
```

- *The bytes created by calloc () are initialized to 0.*

Note:

- After **malloc ()**, the block of memory allocated contains garbage values.
- **malloc ()** is faster than **calloc ()**

realloc()

What if the allocated number of bytes run out?

- We can reallocate!
- Use “realloc” from stdlib

```
int * pt = (int *)  
realloc(pt, 2*MAX_SIZE*sizeof(int));
```

realloc() tries to

- Enlarge existing block
- Else, it creates a new block elsewhere and move existing data to the new block. It returns the pointer to the newly created block and frees the old block.
- If neither could be done, it returns a **NULL**.

Example

```
int *p, *q, *r, i;  
p = malloc(5*sizeof(int));  
q = p;
```

Array address 0xa27010

```
for (i=0;i<5;i++)  
    *p++ = i;  
p = q;
```

Relocated address 0xa27010

```
printf("Array address %p \n", p);  
r = realloc(p, 10*sizeof(int));  
printf("Relocated address %p\n", r);  
for(i=0; i<5; i++)  
printf("The elements are %d\n", r[i]);
```

The elements are 0
The elements are 1
The elements are 2
The elements are 3
The elements are 4

Example

```
r = realloc(p, 10*sizeof(int));
```

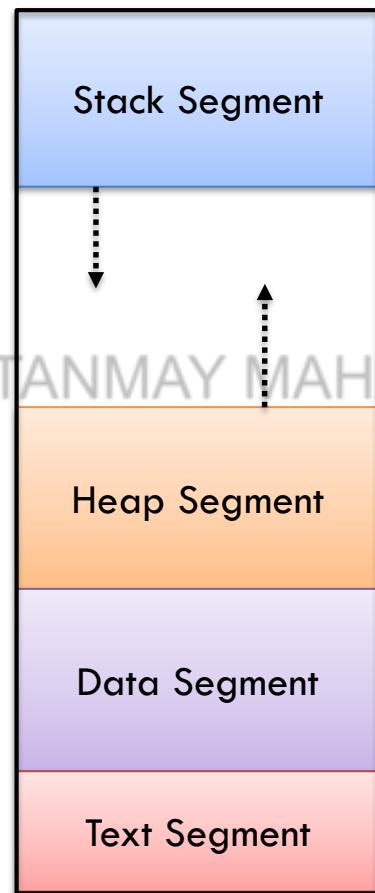


```
r = realloc(p, 9000000*sizeof(int));
```

Array address 0x21c5010

Relocated address 0x7f72f15bc010

free()



- Block of memory freed using **free()** is returned to the heap
- Failure to free memory results in heap depletion.
 - which means that **malloc()** or **calloc()** can return **NULL** saying that no more memory is left to be allocated.
- **free()** syntax:

```
int * p = malloc(5*sizeof(int));  
...  
free(p);
```

Releases all **5*sizeof(int)** bytes allocated



Memory Management Issues

Memory Management Issues

Dangling Pointer:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int * p, *q, *r, i;
    p = (int*) malloc(sizeof(int));
    printf("Address pointed by p = %p \n", p);
    free(p); // p becomes dangling pointer
    printf("Address pointed by p = %p \n", p);
    p = NULL;
    printf("Address pointed by p = %p \n", p);
    return 0;
}
```

Address pointed by p = 0x8ff010

Address pointed by p = 0x8ff010

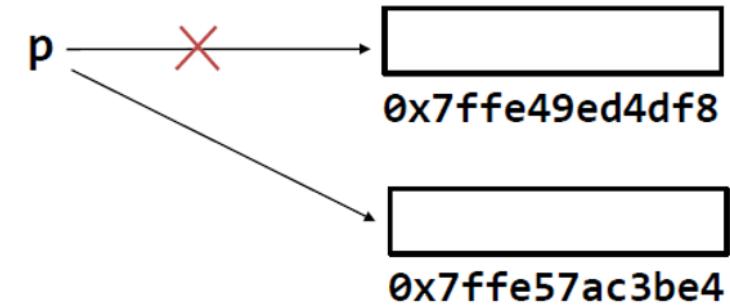
Address pointed by p = (nil)

Memory Management Issues

Memory Leaks:

- **p** is made to point to another memory block without freeing the previous one.
- **1000*sizeof(int)** bytes previously allocated to **p** are now inaccessible.
- **This is a memory leak.** Leaked memory can be recovered only after the program terminates.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int * p, *q, *r, i;
    p = (int*) malloc(1000*sizeof(
    q = (int*) malloc(sizeof(int));
    p = q;
    return 0;
```



Another example of a memory leak



```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int * p, *q, *r, i;
    for (i=0;i<1000000;i++)
    {
        p = (int*) malloc(10000*sizeof(int));
        q = (int*) malloc(sizeof(int));
    }
    // some more lines of code
    ...
}
```

**40,000 bytes are allocated and wasted for each iteration of this loop
(considering sizeof(int) is 4 bytes)**

Memory Leak (Consequences)

- Memory leaks reduces the performance of the computer by reducing the amount of available memory.
- In the worst case, too much of the available memory may become allocated and all or part of the system or device stops working correctly, the application fails, or the system slows down vastly.
- Memory leaks are particularly serious issues for programs like daemons and servers which by definition never terminate.

Memory Management Practices

In C programming it is the responsibility of the programmer to:

- keep freeing the allocated memory after its usage is over so that `malloc()` and `calloc()` don't fail.
- prevent memory leaks by careful programming practices
- keep track of dangling pointers and re-use them.



Dynamically Allocated Array of Structures

Dynamically allocated struct variable and array of structures

```
struct stud {  
    int roll;  
    char dept_code[25];  
    float cgpa;  
} class, *ptr;
```

Rest of the operations are the same as structures and array of structures. Only difference is that we have dynamically allocated memory in the heap.

```
struct stud * s1 = (struct stud*) malloc(sizeof(struct stud));
```

```
struct stud * studentArray = (struct stud*) malloc(sizeof(struct stud)*100);
```



Grocery Store Case

Grocery Store Case

You have to maintain a set of grocery items. Each grocery item has the following attributes: **ID (Integer)**, **Name (Char array)**, **Price (float)**, **Quantity (Integer)**. Implement the following functions:

- **readGroceryList()**: receives a **count** of grocery items as a function parameter, and reads the details of that many grocery items from the user and stores them in a dynamically allocated array, and returns it
- **printGroceryList()**: that receives an **array of grocery items** and its **count** as parameters, and prints the details of all the grocery items present in that array.
- **findItem()**: searches for the first grocery item in an **array of grocery items**, whose quantity is equal to **qVal** (function parameter) and returns that item
- **findMaxPriceltem()**: searches for the grocery item in an **array of grocery items** that has maximum price and returns it

Grocery Store Case: Structure Definition



```
struct item
{
    int ID;
    char name[50];
    float price;
    int quantity;
};
```

Grocery Store Case: readGroceryItems()

```
struct item * readGroceryList(int count) {
    struct item * gItems = (struct item *) malloc(sizeof(struct
item)*count);

    int uniqNum = 1;

    for (int i = 0; i < count; i++)
        gItems[i].ID = uniqNum++;
    printf("\nEnter details for item %d:\n",i+1);
    printf("Name:");
    scanf("%s",gItems[i].name);
    printf("Price:");
    scanf("%f",&gItems[i].price);
    printf("Quantity:");
    scanf("%d",&gItems[i].quantity);
}
return gItems;
}
```

Grocery Store Case: printGroceryList()



```
void printGroceryList(struct item * gItems, int count)
{
    for (int i = 0; i < count; i++)
    {
        printf("Item ID: %d, ", gItems[i].ID);
        printf("Name: %s, ", gItems[i].name);
        printf("Price: %f, ", gItems[i].price);
        printf("Quantity: %d\n", gItems[i].quantity);
    }
}
```

Grocery Store Case: findItem()



```
struct item findItem(int qVal, struct item * gItems, int count)
{
    int i = 0; int index = -1;
    while(i<count)
    {
        if(gItems[i].quantity == qVal)
        {
            index = i;
            return gItems[index];
        }
        i++;
    }

    struct item emptyItem;
    emptyItem.ID = -1;
    return emptyItem;
}
```

Grocery Store Case: **findMaxPriceItem()**



```
struct item findMaxPriceItem(struct item * gItems, int count)
{
    int maxIndex = -1;
    int maxPrice = -1;
    int i = 0;

    while(i < count)
    {
        if(gItems[i].price > maxPrice)
        {
            maxPrice = gItems[i].price;
            maxIndex = i;
        }
        i++;
    }

    return gItems[maxIndex];
}
```

Grocery Store Case: main()



```
int main() {
    int num_g;
    printf("Enter number of unique grocery items in the store:");
    scanf("%d", &num_g);
    struct item * gItems = readGroceryList(num_g);
    printGroceryList(gItems, num_g);

    ... // contd. next page
}
```

Grocery Store Case: main() (contd.)



```
int main() {
    ... // contd. from previous page
    int qVal;
    printf("\nEnter the quantity of the item you wish to find: ");
    scanf("%d", &qVal);
    struct item fItem= findItem(qVal,gItems,num_g);

    if(fItem.ID == -1) printf("\nItem Not Found!\n");
    else
    {
        printf("Item with quantity %d is %s\n", qVal, fItem.name);
    }

    struct item maxItem = findMaxPriceItem(gItems,num_g);
    printf("The item with maximum price is %s\n", maxItem.name);
}
```

Sample Execution

Enter number of unique grocery items in the store:**3**

Enter details for item 1:

Name: **Hamam**

Price: **20.5**

Quantity: **15**

Item ID: 1, Name: Hamam, Price: 20.500000, Quantity: 15

Item ID: 2, Name: Sugar, Price: 90.250000, Quantity: 20

Item ID: 3, Name: Milkmaid, Price: 210.649994, Quantity: 6

Enter details for item 2:

Name: **Sugar**

Enter the quantity of the item you wish to find: **20**

Price: **90.25**

Quantity: **20**

Item with quantity 20 is Sugar

The item with maximum price is Milkmaid

Enter details for item 3:

Name: **Milkmaid**

Price: **210.65**

Quantity: **6**



Multi-dimensional Arrays using dynamic memory allocation

Multi-dimensional arrays

2D arrays in stack memory:

```
#define NUM_ROWS 3  
#define NUM_COLS 2  
int arr2D [NUM_ROWS] [NUM_COLS];
```

Notice the double pointer

This array of 3 rows and 2 columns resides in stack and has all its problems that we discussed earlier for 1D arrays

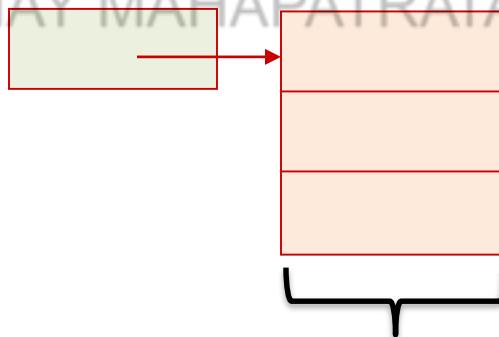
We can declare this array in heap dynamically at run-time by using a pointer to a group of pointers:

```
int ** arr2D = (int **) malloc(NUM_ROWS*sizeof(int*));  
for (int i=0; i<NUM_ROWS; i++)  
{  
    arr2D[i] = (int*) malloc(NUM_COLS*sizeof(int));  
}  
arr2D[1][2] = 10; // is a valid assignment
```

2D Array

```
#define NUM_ROWS 3  
#define NUM_COLS 2  
int ** arr2D = (int **) malloc(NUM_ROWS*sizeof(int*));  
...
```

TANMAY MAHAPATRA TANMAY MAHAPATRA TANMAY MAHAPATRA



Each of this blocks if the type **int ***, i.e., each can hold address of an **int** variable or the address of the first location of an **int** array

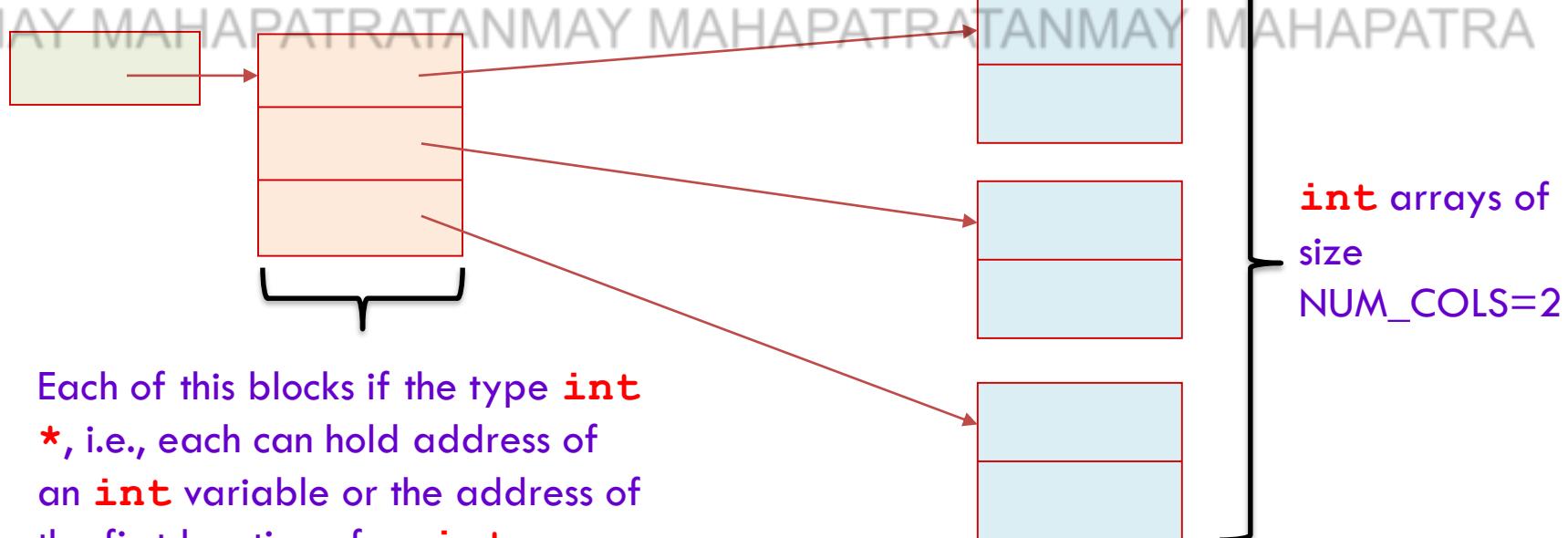
TANMAY MAHAPATRA TANMAY MAHAPATRA TANMAY MAHAPATRA

2D Array (contd.)

...

```
for (int i=0; i<NUM_ROWS; i++)  
{  
    arr2D[i] = (int*) malloc(NUM_COLS*sizeof(int));  
}
```

arr2D

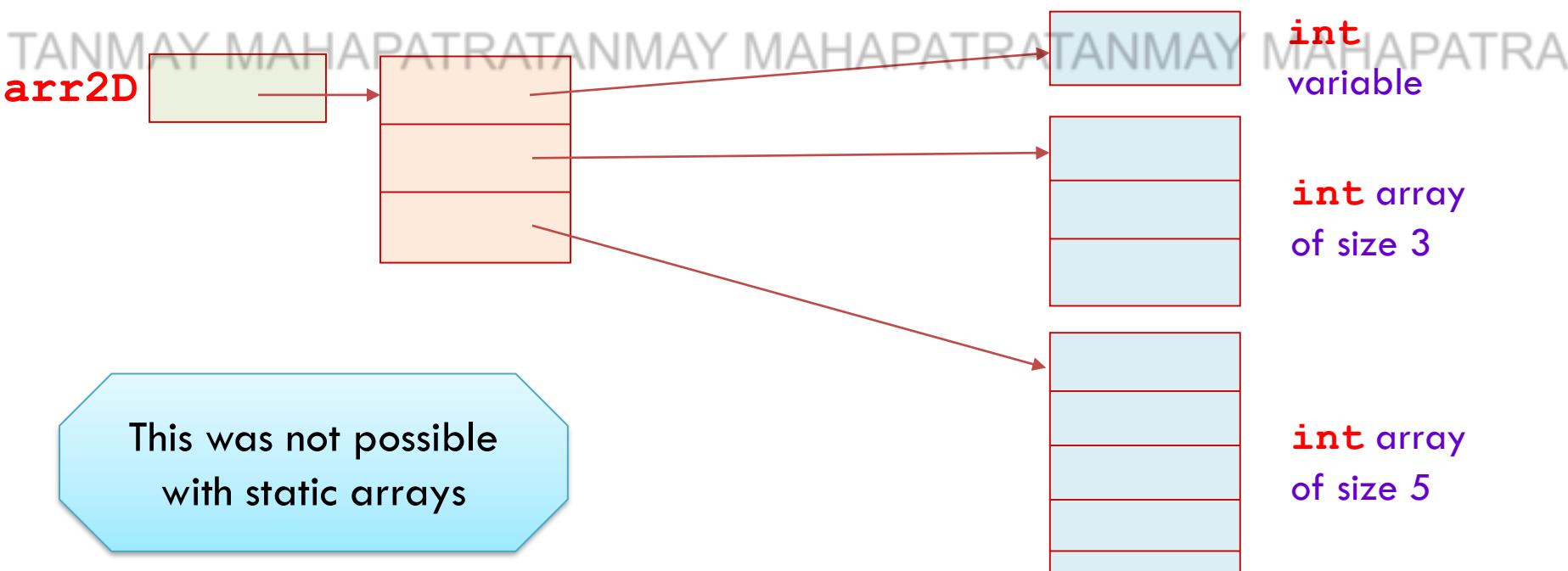


Each of this blocks if the type `int *`, i.e., each can hold address of an `int` variable or the address of the first location of an `int` array

2D Array (variable size)

One can define each array of variable size as well

```
arr2D[0] = (int*) malloc(sizeof(int)); // single int variable  
arr2D[0] = (int*) malloc(sizeof(int)*3); // int array of size 3  
arr2D[0] = (int*) malloc(sizeof(int)*5); // int array of size 5
```



Static vs Dynamic Arrays

Arrays (static) are Pointers in C?

- Well, not exactly ...
- Arrays cannot be reallocated (starting address and size are fixed).
- Multi-dimensional arrays (static) are not pointers to pointers.
 - They are contiguous memory locations

```
int a[3][3];
```

a^a , a^{a+1} , a^{a+2} , a^{a+3} , a^{a+4} ,
a^{a+5} , a^{a+6} , a^{a+7} , a^{a+8}
a[0][0] , a[0][1] , a[0][2] , a[1][0] , a[1][1] ,
a[1][2] , a[2][0] , a[2][1] , a[2][2]

Exercises

- Write a C program that computes the transpose of a 2D array.
- Write a C program that receives a 2D array and sorts the elements by accessing the 2D array as a 1D array.

Remember our **incorrect** copy arrays function...

```
#include <stdio.h>
#include <stdlib.h>

void copy(int * a, int n, int * c)
{
    c = (int*) malloc(sizeof(int)*n);
    for (int i=0; i<n; i++)
    {
        c[i]=a[i];
    }
}
```

Will lead to segmentation fault (run-time error). Why?

The contents of b were copied into c during function call. b was empty (or garbage value) when it was passed. The memory that was allocated in copy() function, its first element's address is copied to c during the malloc() call. But this was not copied to b, when copy() returned.

Solution:

- Allocate memory to b in main() and then pass. *Try This!*
- Allocate memory inside copy() and return. (*last slide*)
- **Use double pointer!**
(we will see NOW)

```
int main(){
    int a[5] = {1,2,3,4,5};
    int * b;
    copy(a,5,b);

    for(int i=0;i<5;i++){
        printf("%d\t",b[i]);
    }
}
```

Example: copy arrays – that is incorrect (using pointers)

innovate

achieve

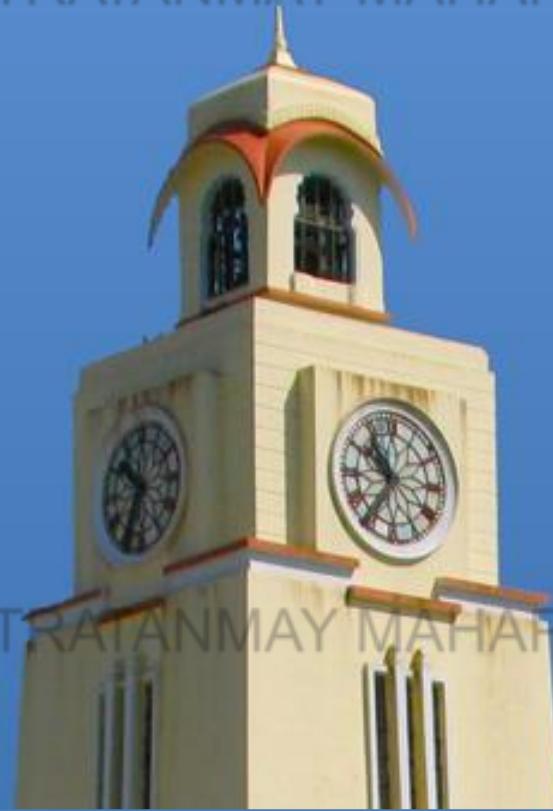
lead

```
#include <stdio.h>
#include <stdlib.h>
void copy(int c[], int n, int ** d) {
    *d = (int *) malloc(n*sizeof(int));
    for (int i=0; i<n; i++)
    {
        (*d)[i]=c[i];
    }
}

int main(){
    int a[5] = {1,2,3,4,5};
    int ** b = (int **) malloc(sizeof(int *));
    copy(a,5,b);

    for(int i=0;i<5;i++) {
        printf("%d\t", (*b)[i]);
    }
}
```

Use of double pointer enables us to not return anything, yet change getting reflected in main(). This is call by reference for dynamically allocated arrays



Command Line Arguments

Command Line Arguments

```
#include <stdio.h>
/* echo */
int main(int argc, char **argv)
{
    // argc - number of arguments passed
    // argv[0] is the name of the
    // (command) file being executed
    for (j=1; j < argc; j++)
        printf("%s ", argv[j]);
    return 0;
}
```

- gcc echo.c -o echoTest
- ./echoTest How do you do?
- Output: **How do you do?**
- Observe that we printing argv[j] from j=1
 - argv[0] refers to the name of the executable (in this case “echoTest”).
 - Each other argv[j] refers to one word (separated by blank spaces) of command line argument.
 - argc has the count of arguments

Example 1

```
int main(int argc, char *argv[])
{
    printf("Number of arguments: %d \n", argc);
    int i = 0;
    while(i < argc){
        printf("Argument %d: %s \n", i, argv[i]);
        i++;
    }
    return 0;
}
```

Notice the change

```
amitesh@Prithvi:~$ gcc test1.c
amitesh@Prithvi:~$ ./a.out hello world
Number of arguments is 3
Argument 0: ./a.out
Argument 1: hello
Argument 2: world
```

Example 2

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
    int a, b, c;
    a = atoi(argv[1]);
    b = atoi(argv[2]);
    c = atoi(argv[3]);
    printf("\n %d",a+b+c);
    return 0;
}
```



Thank you
Q & A

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA



BITS Pilani
Pilani Campus

Module 12 – part 1 – Linked Lists

Department of Computer Science & Information Systems

Module Overview

- **Motivation**
- **Linked Lists Implementation**



Motivation

Random Access List vs Sequential Access List

Random Access List:

- Given a list of elements, you should be able to access any element of the list:
 - quickly*
 - easily*
 - without traversing any other element of the list*
- Example: Using **arrays** to represent the list.

```
int arr[100] = {5, 8, 34, 98, 13, 25, 73, 88, 28, 30};
```

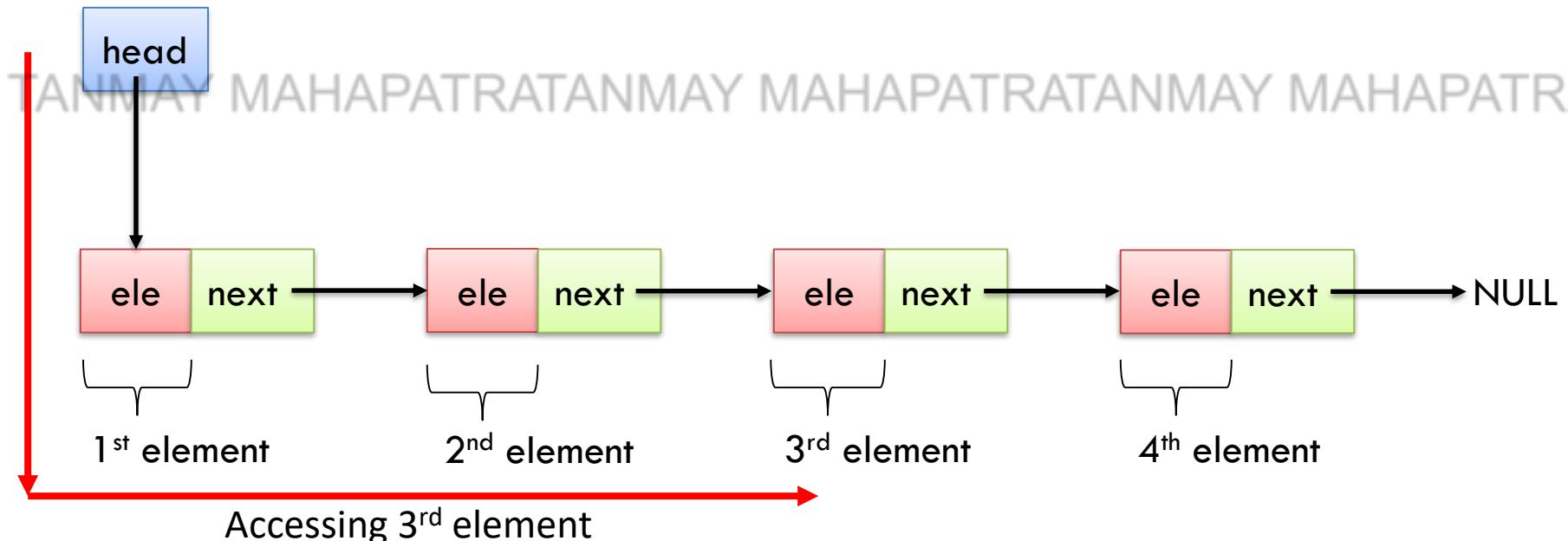
5	8	34	98	13	25	73	88	28	30
---	---	----	----	----	----	----	----	----	----

- You can access 3rd element of the array by **arr[2]**
 - This is quick, easy and doesn't need one to traverse the entire list to read the 3rd element

Random Access List vs Sequential Access List

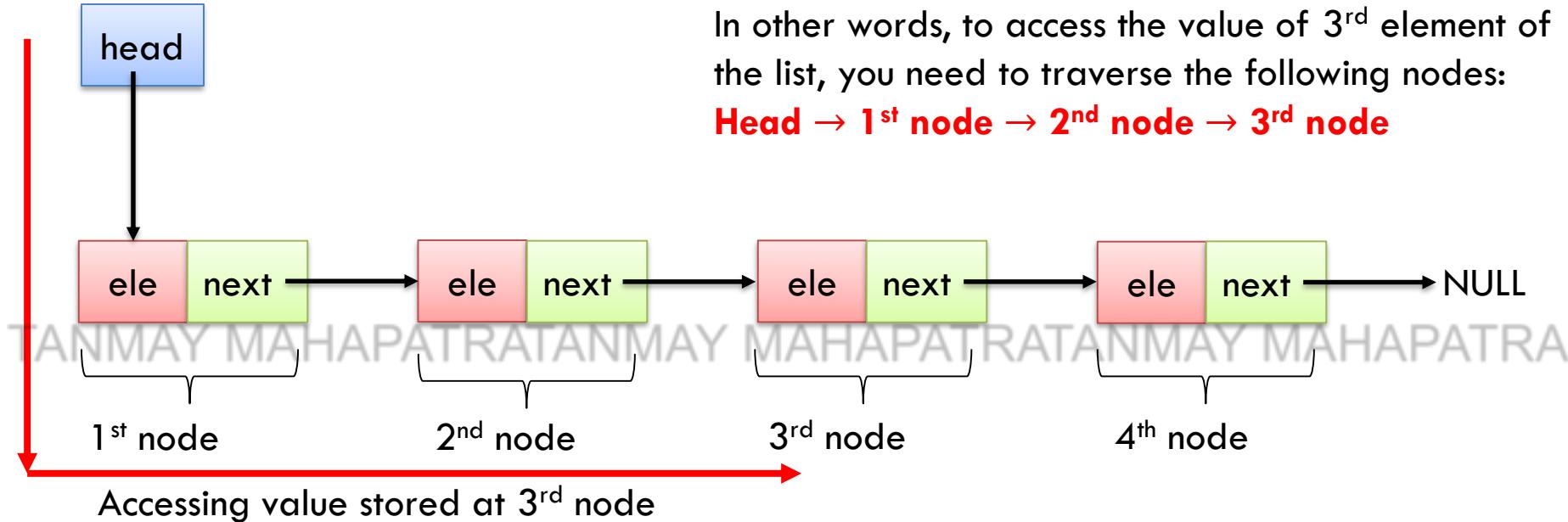
Sequential Access List

- There is another way of representing lists where you should traverse the list to reach any element of the list.



- To access 3rd element of the list you need to traverse 1st, 2nd elements

Linked Lists

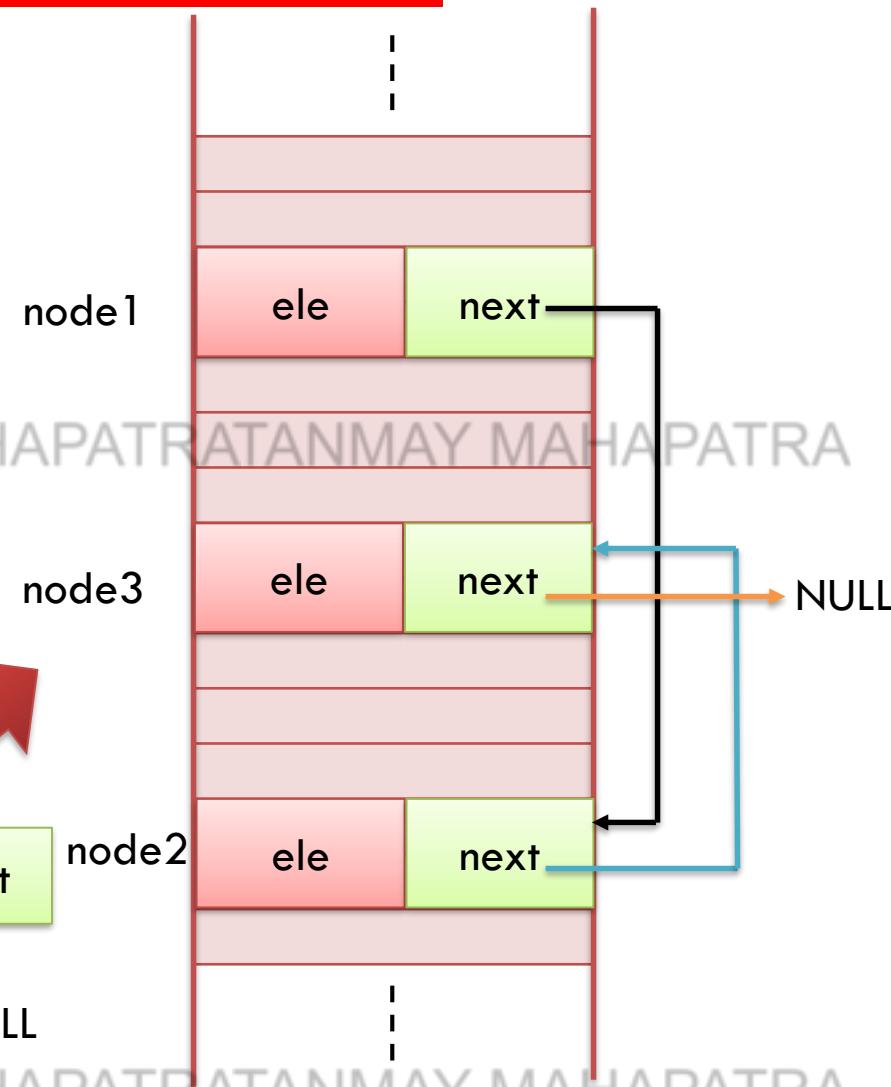
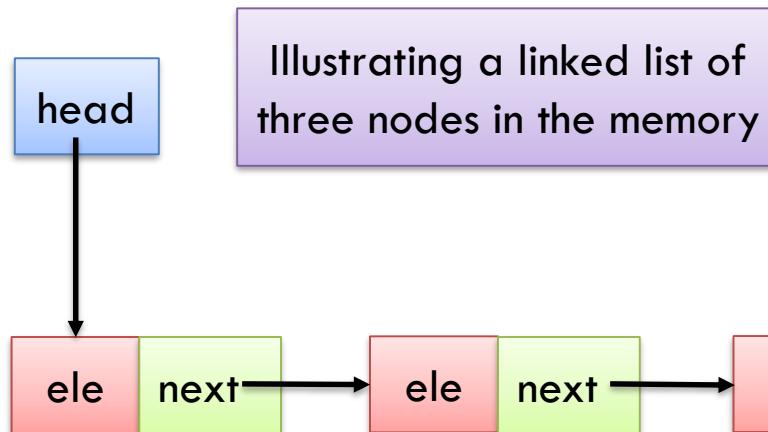


- Lists are now organized as a sequence of nodes, each containing a
 - value of the element stored at that node: ele*
 - address of the next node: next*
- The **head node** contains the **address of the first node**
- The **last node points to NULL**, meaning end of the list

All the nodes together represent a list or a sequence

Linked List in Memory

- The nodes are not sequentially arranged in the memory
- They are logically connected with links



Sequential Access Lists - Uses

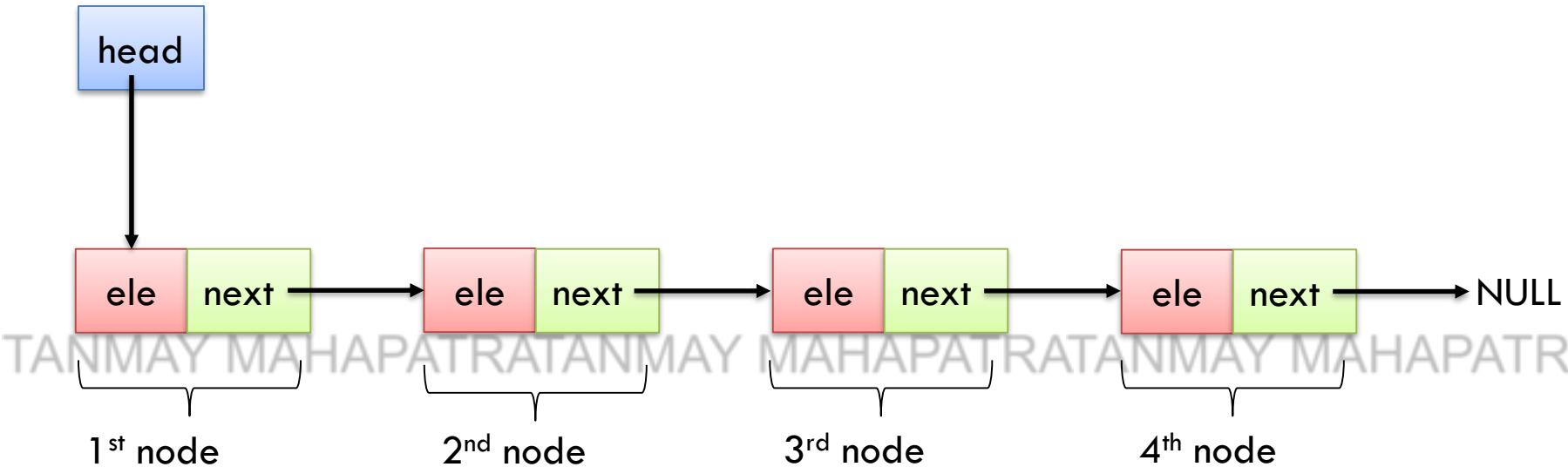
Where are Sequential Access Lists useful?

- Create dynamic lists on run-time
 - *you can keep on adding nodes to the list, without bothering about resizing the list, like in arrays when the number of elements exceed their size*
- Efficient insertion and deletion
 - *Without any shift operations*
- Used to implement
 - *Stacks*
 - *Queues*
 - *Other user-defined data types*



Linked lists Implementation

Linked Lists



To create linked lists we need two kinds of structures:

- One for storing the **head**
- The other to represent each **node** in the list

Let us see how each of these can be defined...

Self referential structures

Before we see the structure definition of linked lists, let us see what self-referential structures are:

“Self-referential structures contain a pointer member that points to a structure of the same structure type”

Wrong Declaration

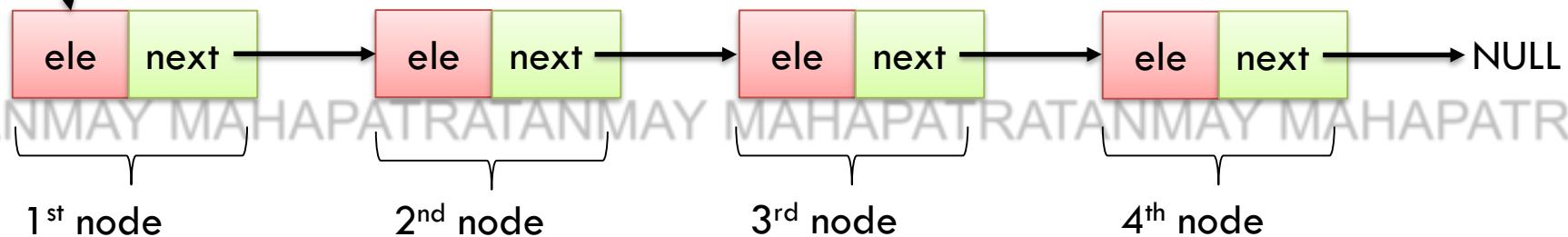
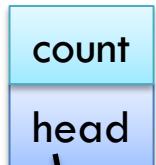
```
struct self_ref
{
    int data;
    struct self_ref b;
};
```

Correct Declaration

```
struct self_ref
{
    int data;
    struct self_ref *b;
};
```

Self-referential structures essentially store a pointer variable to a variable of its own type to reference to another structure variable of its kind.

Linked Lists



head stores the address of the first node
count stores the number of nodes/elements in the list
ele in each node stores the element (integer in this case)
next stores the address of the next node in the list

Consider that our linked list stores integer elements.

```
struct node{
    int ele;
    struct node * next;
};
```

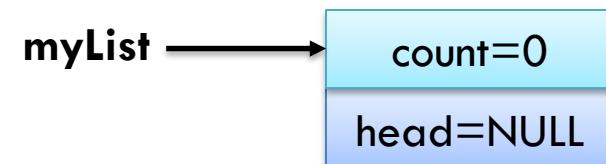
```
struct linked_list{
    int count;
    struct node * head;
};
```

Creating linked list using malloc() (on heap segment)

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

typedef struct linked_list * LIST;
struct linked_list{
    int count;
    NODE head;
};
```

```
LIST createNewList() {
    LIST myList;
    myList = (LIST) malloc(sizeof(struct linked_list));
    // myList = (LIST) malloc(sizeof(*myList));
    myList->count=0;
    myList->head=NULL;
    return myList;
}
```



Creating new node

```

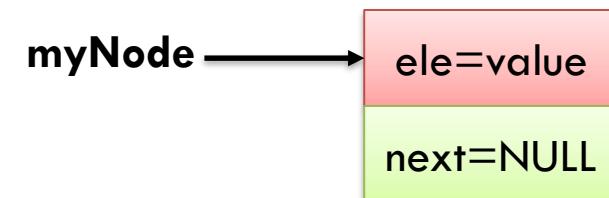
typedef struct node * NODE;           typedef struct linked_list * LIST;
struct node{
    int ele;
    NODE next;
};
struct linked_list{
    int count;
    NODE head;
};

```

```

NODE createNewNode(int value){
    NODE myNode;
    myNode = (NODE) malloc(sizeof(struct node));
    // myList = (NODE) malloc(sizeof(*myNode));
    myNode->ele=value;
    myNode->next=NULL;
    return myNode;
}

```



main()

```
int main() {  
    LIST newList = createNewList();  
    NODE n1 = createNewNode(10);  
    NODE n2 = createNewNode(20);  
    NODE n3 = createNewNode(30);
```

```
    insertNodeAtBeginning(n1,newList);  
    insertNodeAtBeginning(n2,newList);  
    insertNodeAtEnd(n3,newList);
```

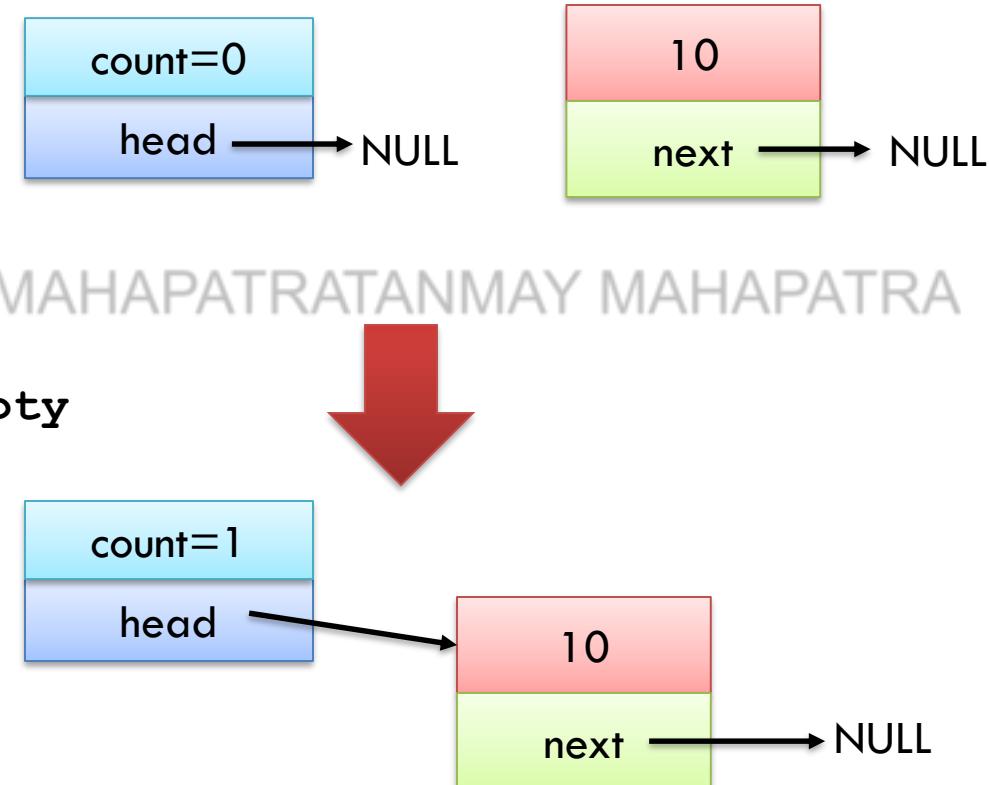
```
    NODE n4 = createNewNode(40);  
    insertAfter(10,n4,newList);
```

```
    removeFirstNode(newList);  
    removeLastNode(newList);  
    return 0;
```

```
}
```

Inserting a node at the beginning of the list

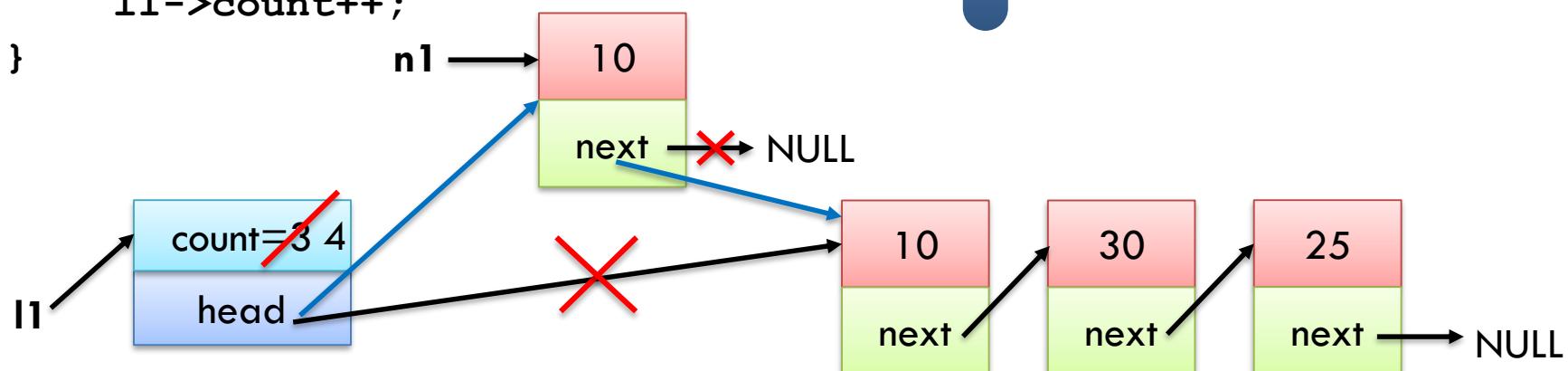
```
void insertNodeAtBeginning(NODE n1, LIST l1){  
    // case when list is empty  
    if(l1->count == 0) {  
        l1->head = n1;  
        n1->next = NULL;  
        l1->count++;  
    }  
    // case when list is non empty  
    else {  
        ... ...  
    }  
}
```



Inserting a node at the beginning of the list

(contd.)

```
void insertNodeAtBeginning(NODE n1, LIST l1) {
    // case when list is empty
    if(l1->count == 0) {
        ...
    }
    // case when list is non empty
    else {
        n1->next = l1->head;
        l1->head = n1;
        l1->count++;
    }
}
```



Inserting a node at the end of the list

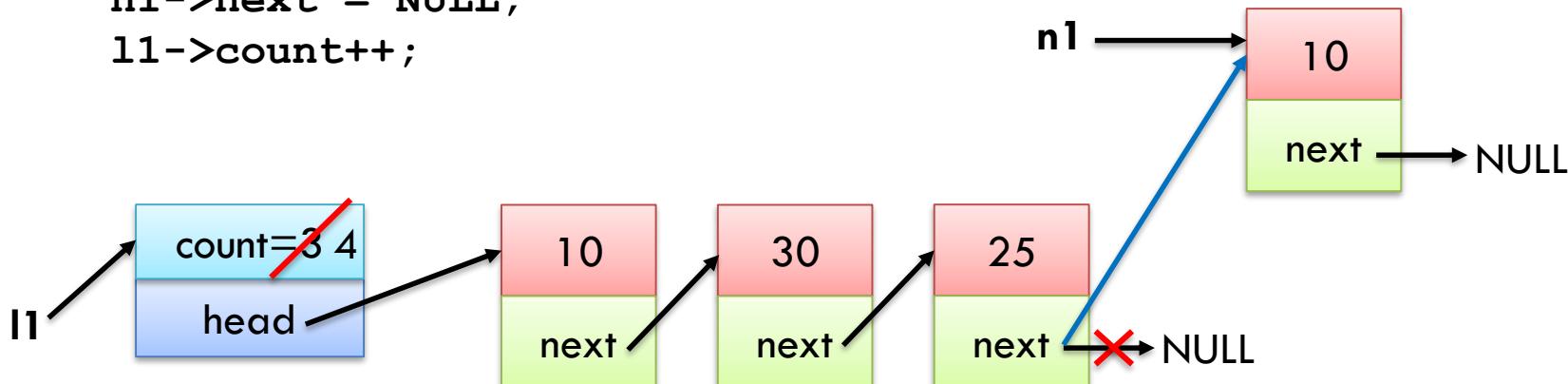
```
void insertNodeAtEnd(NODE n1, LIST l1) {  
    // case when list is empty  
    if(l1->count == 0) {  
        l1->head = n1;  
        n1->next = NULL;  
        l1->count++;  
    }  
    // case when list is non empty  
    else {  
        . . . . .  
    }  
}
```

This case is same as insert at the beginning of an empty list.

Inserting a node at the end of the list

```
void insertNodeAtEnd(NODE n1, LIST l1) {  
    ...  
    // case when list is non empty  
    else {  
        NODE temp = l1->head;  
        while(temp->next!=NULL)  
        {  
            temp = temp->next;  
        }  
        temp->next = n1;  
        n1->next = NULL;  
        l1->count++;  
    }  
}
```

- Traverse the list until the end.
- Insert new node at the end

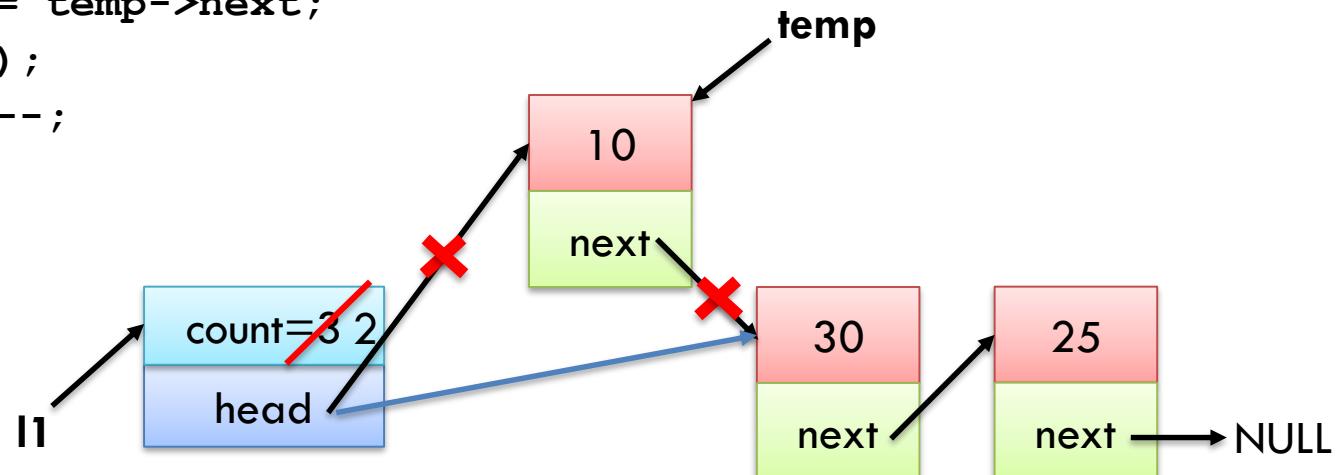


Inserting a node after a given node

```
void insertAfter(int searchEle, NODE n1, LIST l1){  
    // case when list is empty  
    ... ...  
    // case when list is non-empty  
    else {  
        NODE temp = l1->head;  
        NODE prev = temp;  
        while(temp!=NULL) {  
            if (temp->ele == searchEle)  
                break;  
            prev = temp;  
            temp = temp->next;  
        }  
        if(temp==NULL) {  
            printf("Element not found\n");  
            return;  
        }  
        else{  
            if(temp->next == NULL) {  
                temp->next = n1;  
                n1->next = NULL;  
                l1->count++;  
            }  
            else {  
                prev = temp;  
                temp = temp->next;  
                prev->next = n1;  
                n1->next = temp;  
                l1->count++;  
            }  
            return;  
        }  
    }  
}
```

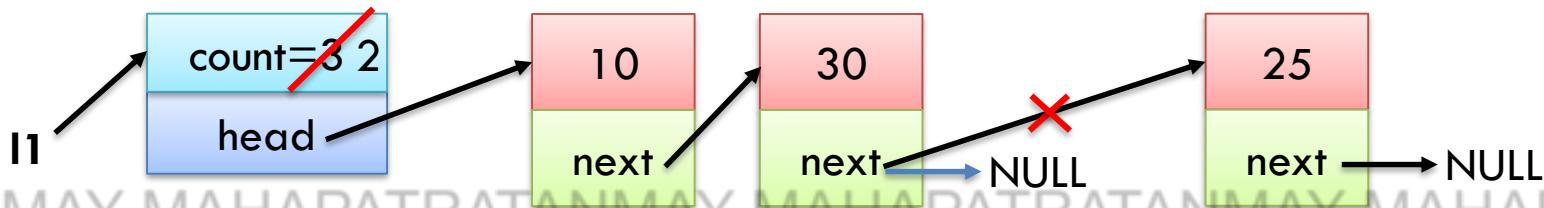
Removing a node from the beginning of the list

```
void removeFirstNode(LIST l1)
{
    if (l1->count == 0)
    {
        printf("List is empty. Nothing to remove\n");
    }
    else
    {
        NODE temp = l1->head;
        l1->head = temp->next;
        free(temp);
        l1->count--;
    }
    return;
}
```



Removing a node from the end of the list

```
void removeLastNode(LIST l1)
{
    if (l1->count == 0)
    {
        printf("List is empty\n");
    }
    else if(l1->count == 1)
    {
        l1->count--;
        free(l1->head);
        l1->head = NULL;
    }
    else
    {
        NODE temp = l1->head;
        NODE prev = temp;
        while((temp->next) != NULL)
        {
            prev=temp;
            temp = temp->next;
        }
        prev->next = NULL;
        l1->count--;
        free(temp);
    }
    return;
}
```



Other functions

Exercise: Implement the following functions for a linked list:

- **search(int data, LIST mylist)**: returns the node that contains its ele=data
- **printList(LIST mylist)**: prints the elements present in the entire list in a sequential fashion
- **removeElement(int data, LIST mylist)**: removes the node that has its ele=data
- **isEmpty(LIST mylist)**: checks if the list is empty or not
- Modify the insert/delete functions to first check whether the list is empty using **isEmpty()** function.

In each of the above, you must have to decide which one is an appropriate datatype for the same.



Thank you
Q & A

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA



Module 12 – part 2 – Circular and Doubly Linked Lists

BITS Pilani

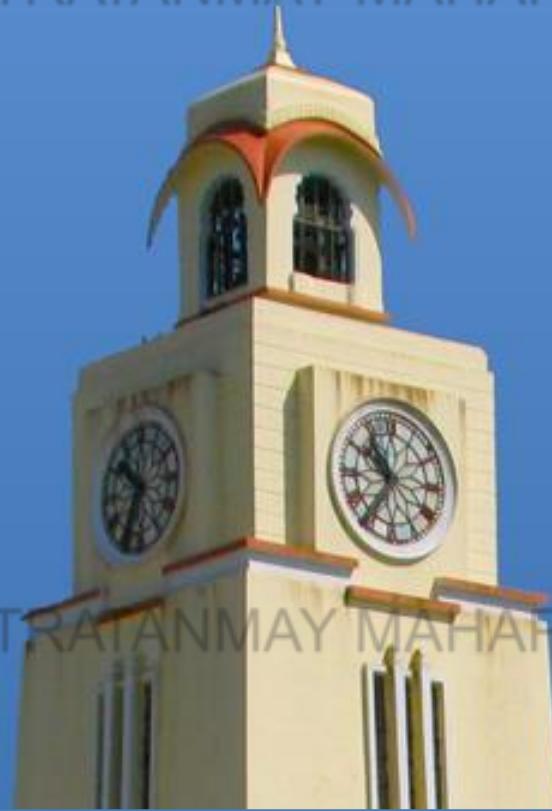
Pilani Campus

Department of Computer Science & Information Systems

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA

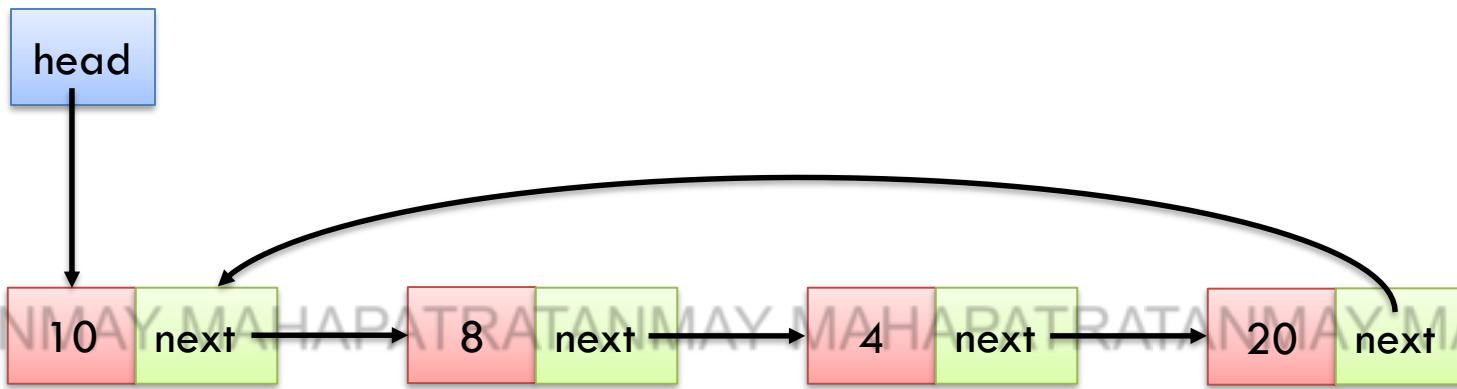
Module Overview

- **Circular Linked Lists**
- **Doubly Linked Lists**



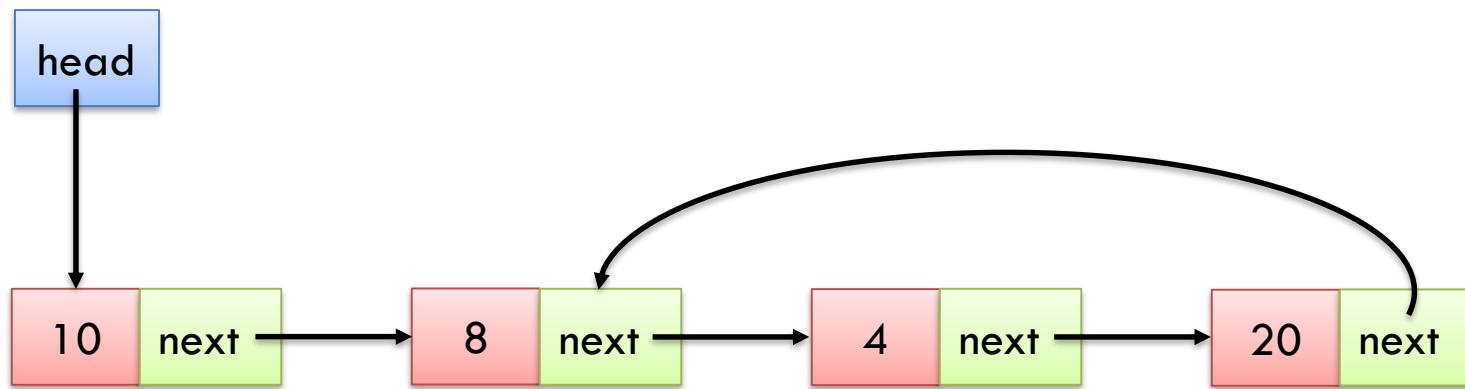
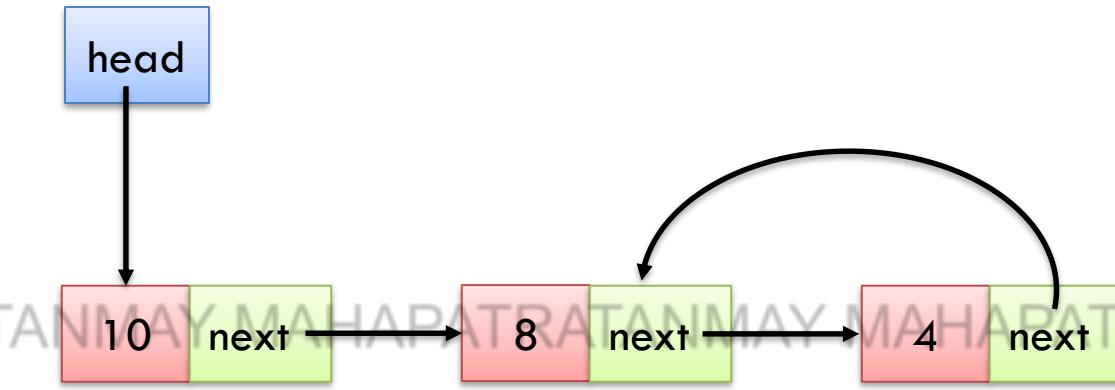
Circular Linked Lists and cycles in a Linked List

Circular Linked List



- The **next** of the last node points to the first node
- Result:
 - *Traversing the list is an infinite operation as you will never encounter a NULL pointer*

More examples





Detecting cycle in a linked list

Cycle in a linked list

Solution 1:

- Traverse the list. While traversing, store the addresses of all the visited nodes in an array/table.
 - When we traverse from **node1** to **node2**, check if the address of node2 already exists in the table. If **YES**, a cycle is detected. If not, add address of **node 2** to the table and go forward.
 - If you encounter a **NULL** in the traversal, then the list doesn't have a cycle.

Cycle in a linked list

Solution 2: This solution requires modifications to the basic linked list data structure.

- Have a **visited flag** with each node
- Traverse the linked list and keep marking visited nodes
- If you see a visited node again then there is a cycle
- If you encounter a **NULL** in the traversal, then the list doesn't have a cycle.

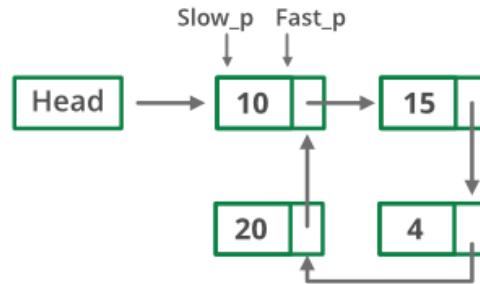
Cycle in a linked list

Solution 3: This is the fastest method

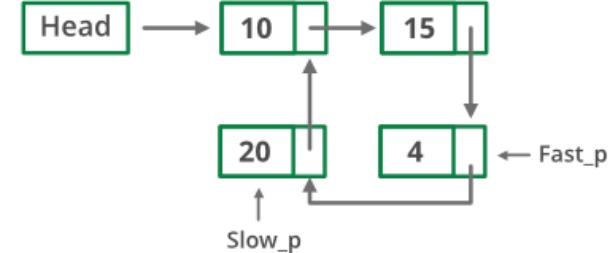
- Traverse linked list using two pointers (**slow_p** & **fast_p**)
- Move (**slow_p**) by one node and **fast_p** by two.
- If these pointers meet at the same node then there is a cycle.
- If pointers do not meet then linked list doesn't have a cycle. OR if either pointer encounters a **NULL** in the traversal, then the list doesn't have a cycle.

Solution 3 illustrated

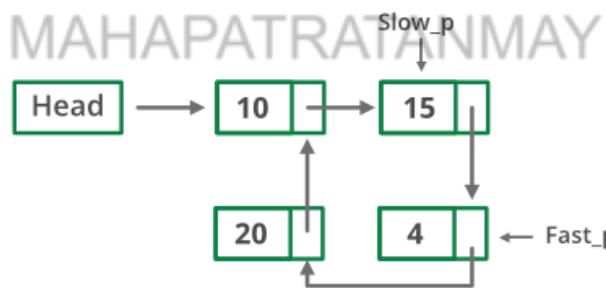
Initially :



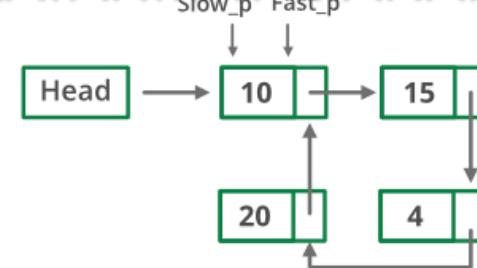
Step 3:



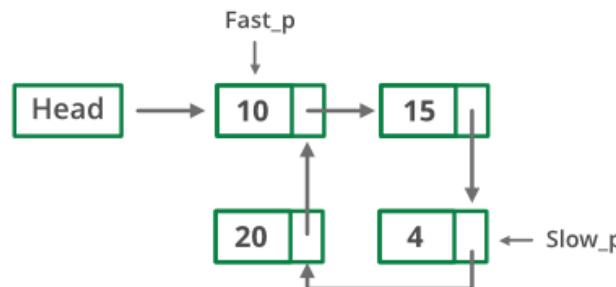
Step 1:



Step 4:



Step 2:

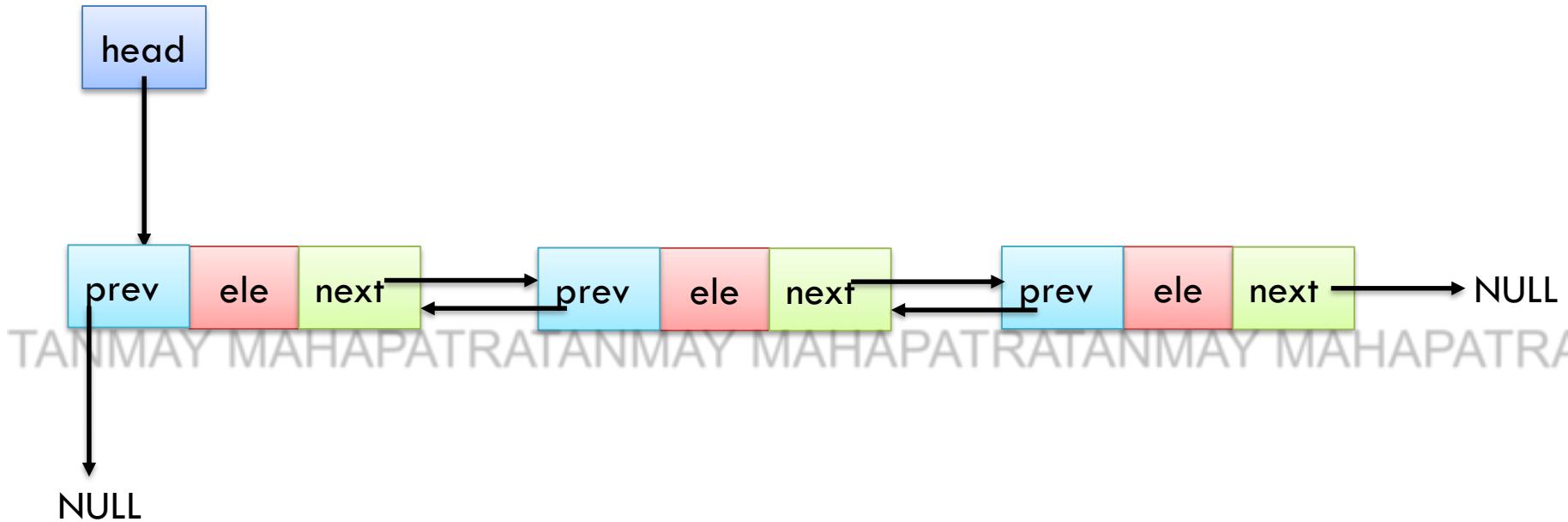


Loop Detected



Doubly Linked Lists

Doubly Linked Lists



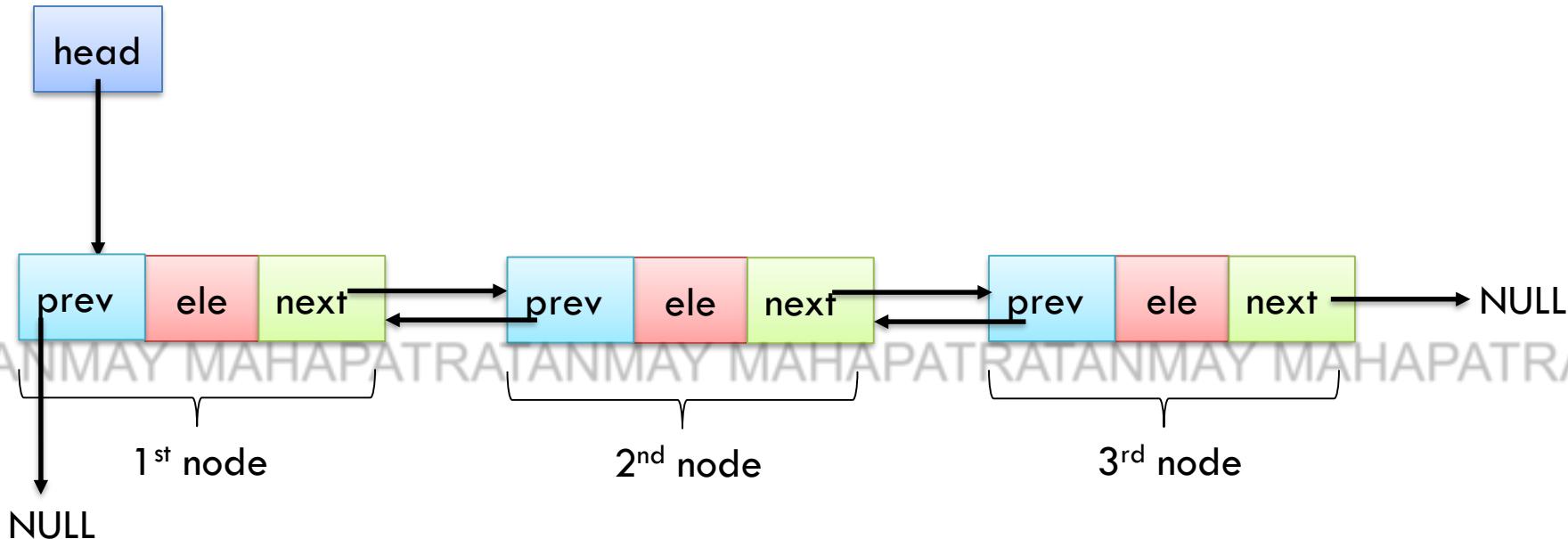
- Each node now stores:
 - value of the element stored at that node: **ele**
 - address of the next node: **next**
 - address of the previous node: **prev**
- The **head node** contains the **address of the first node**
- The **next of the last node points to NULL**, meaning end of the list
- The **prev of the first node points to NULL**, meaning beginning of the list

Supports two-way traversal of linked lists



Doubly Linked Lists – Implementation

Structure definitions for doubly linked lists



Consider that our doubly linked list stores integer elements.

```
struct dllnode{  
    int ele;  
    struct dllnode * next;  
    struct dllnode * prev;  
};
```

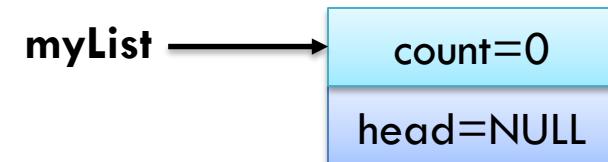
```
struct doubly_linked_list{  
    int count;  
    struct dllnode * head;  
};
```

Creating a new doubly linked list

```
typedef struct dllnode * DLLNODE;
struct dllnode{
    int ele;
    DLLNODE next;
    DLLNODE prev;
};

typedef struct doubly_linked_list *
DLIST;
struct doubly_linked_list{
    int count;
    DLLNODE head;
};

DLIST createNewList(){
    DLIST myList;
    myList = (DLIST) malloc(sizeof(struct doubly_linked_list));
    // myList = (DLIST) malloc(sizeof(*myList));
    myList->count=0;
    myList->head=NULL;
    return myList;
}
```



Creating a new node

```

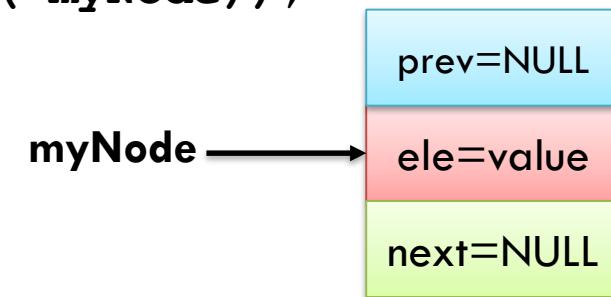
typedef struct dllnode * DLLNODE;           typedef struct doubly_linked_list *
struct dllnode{                                DLIST;
    int ele;
    DLLNODE next;
    DLLNODE prev;
};                                         struct doubly_linked_list{
                                                int count;
                                                DLLNODE head;
};

```

```

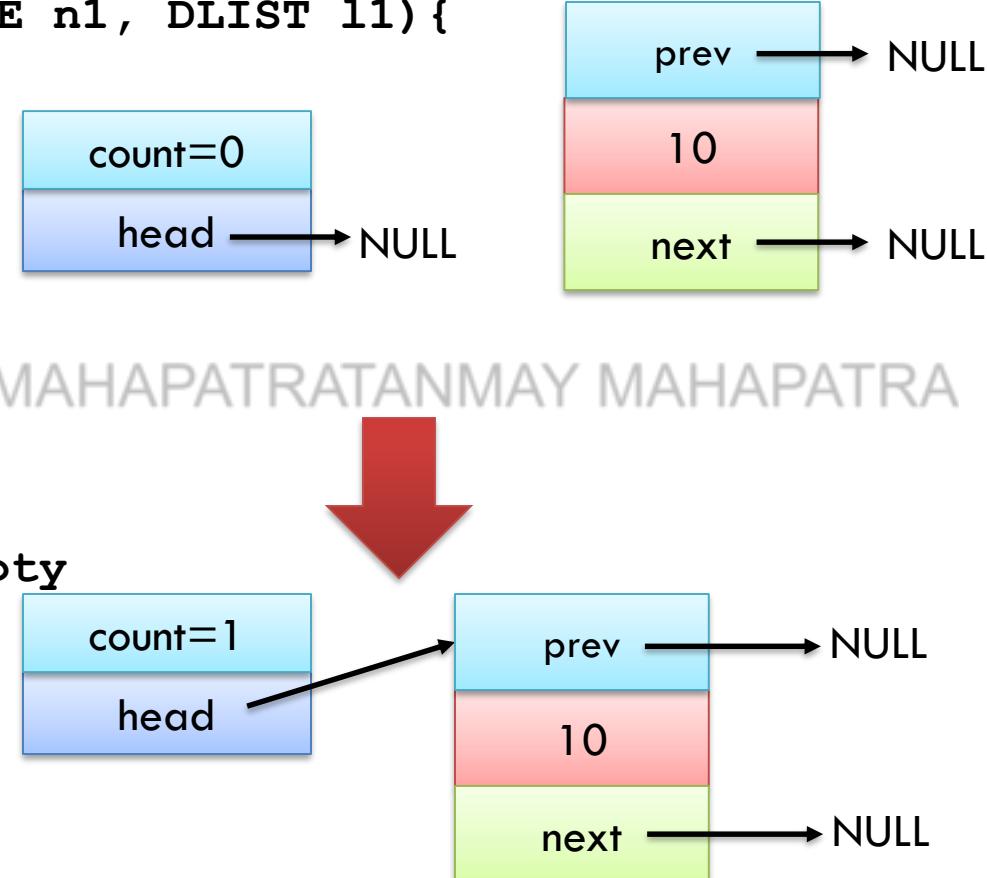
DLLNODE createNewNode(int value) {
    DLLNODE myNode;
    myNode = (DLLNODE) malloc(sizeof(struct dllnode));
    // myList = (DLLNODE) malloc(sizeof(*myNode));
    myNode->ele=value;
    myNode->next=NULL;
    myNode->prev=NULL;
    return myNode;
}

```



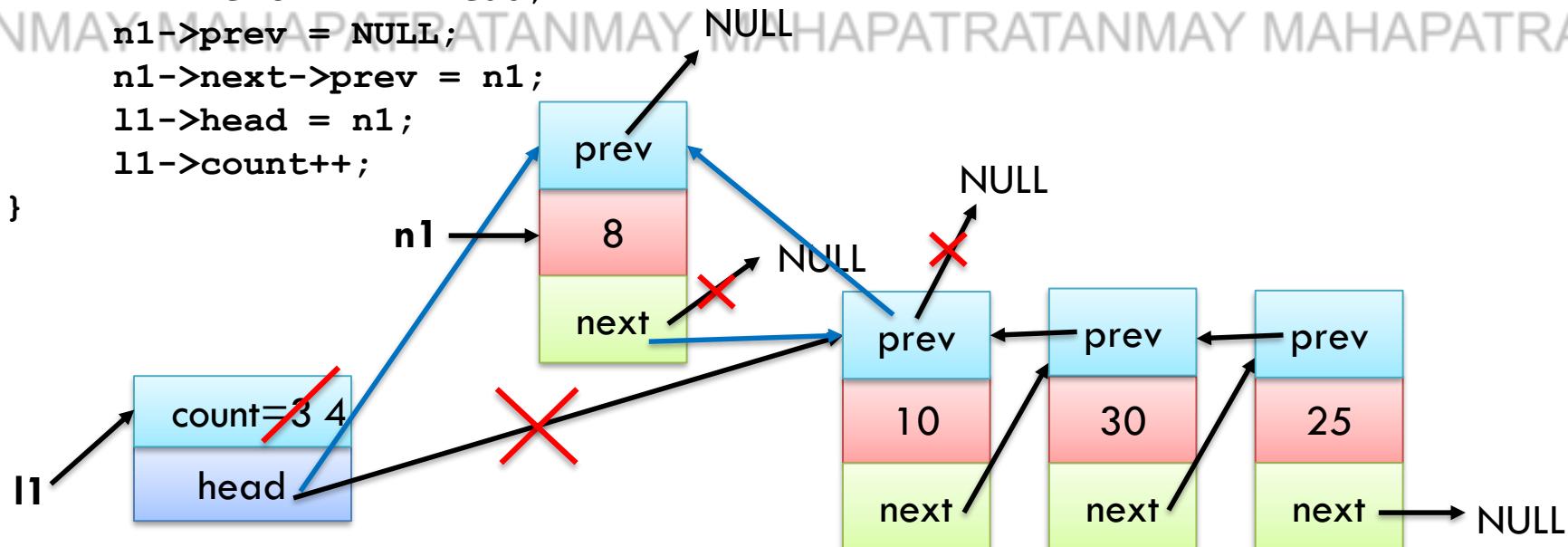
Inserting a node into the list

```
void insertNodeIntoList(DLLNODE n1, DLIST l1) {
    // case when list is empty
    if(l1->count == 0) {
        l1->head = n1;
        n1->next = NULL;
        n1->prev = NULL;
        l1->count++;
    }
    // case when list is non empty
    else {
        . . .
    }
}
```



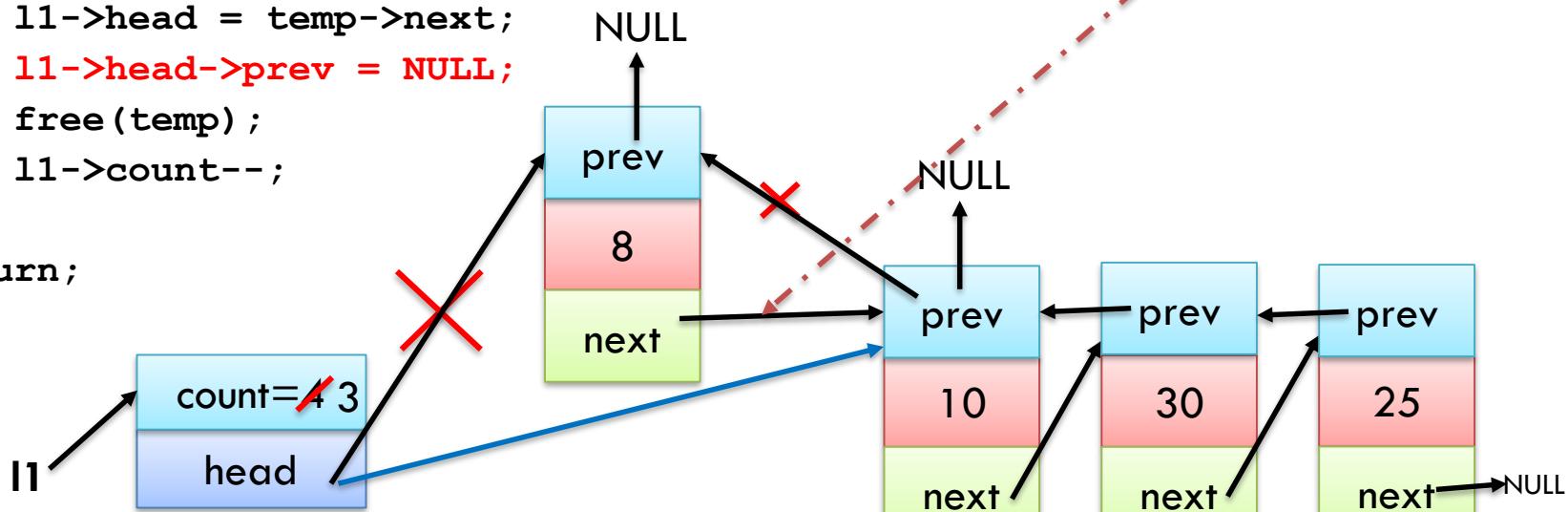
Inserting a node into the list (contd.)

```
void insertNodeIntoList(DLLNODE n1, DLIST l1) {  
    // case when list is empty  
    if(l1->count == 0) {  
        ...  
    }  
    // case when list is non empty  
    else {  
        n1->next = l1->head;  
        n1->prev = NULL;  
        n1->next->prev = n1;  
        l1->head = n1;  
        l1->count++;  
    }  
}
```



Removing a node from the beginning of the list

```
void removeFirstNode(LIST l1)
{
    if (l1->count == 0)
    {
        printf("List is empty. Nothing to remove\n");
    }
    else
    {
        NODE temp = l1->head;
        l1->head = temp->next;
        l1->head->prev = NULL;
        free(temp);
        l1->count--;
    }
    return;
}
```



Other functions (exercise)

Exercise: Implement the following functions for a linked list:

- **insertNodeAtEnd(DLIST mylist, DLLNODE n1)** : inserts n1 at the end of mylist
- **insertAfter(DLIST mylist, DLLNODE n1, int v)** : inserts n1 into mylist after a node containing a value v
- **removeLastNode(DLIST mylist)** : removes the last node from mylist
- **search(int data, DLIST mylist)** : returns the node that contains its ele=data
- **printList_forward(DLIST mylist)** : prints the elements present in the entire list in a sequential fashion in forward direction starting from the first element
- **printList_backward(DLIST mylist)** : prints the elements present in the entire list in a sequential fashion in backward direction starting from the last element
- **removeElement(int data, DLIST mylist)** : removes the node that has its ele=data
- **isEmpty(DLIST mylist)** : checks if the list is empty or not
- Modify the insert/delete functions to first check whether the list is empty using **isEmpty()** function.



Thank you
Q & A

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA



BITS Pilani
Pilani Campus

Module 13 – *File Handling*

Department of Computer Science & Information Systems

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA

Module Overview

- **File and File Handling**
- **File Handling with Command Line Arguments**



File and File Handling

File

- Applications require information to be read from or written to memory device (**DISK**) which can be accomplished using **files**.
- All files are **administered by the operating system** which allocates and deallocates disk blocks.
- The operating system also controls the transfer of data between programs and the files they access.

File handling Basics

- A file has to be **opened** before data can be read from or written to it.
- When a file is opened, the operating system associates a **stream** with the file.
- A **common buffer** and a **file position indicator** are maintained in the memory for a function to know how much of the file has already been read.
- The stream is disconnected when the file is closed.

fopen: Opening a File

- **FILE *fp;** *Defines file pointer*
- **fp = fopen ("foo.txt", "r");** *Only reading permitted*
- If the call is successful, **fopen** returns a pointer to a structure **typedef'd** to **FILE**.
- The pointer variable, **fp**, assigned by **fopen** acts as a file handle which will be used subsequently by all functions that access the file.

File Opening Modes

- **r** – Reading a file
- **w** – Writing a file
- **a** – Appending to the end of an existing file

When used with “**w**” or “**a**”, **fopen** creates a file if it does not find one.

fopen will fail if the file does not have the necessary permissions (r, w and a) or if the file does not exist in case of “r”.

File Opening Modes: Extended

- Database applications often need both read and write access for the same file, in which case, you must consider using the “r+”, “w+” and “a+” modes.

Mode	File opened for
r	Reading only
r+	Both reading and writing
w	Writing only
w+	Both reading and writing
a	Appending only
a+	Reading entire file but only appending permitted

File Read/Write Functions

The standard library offers a number of functions for performing read/write operations on files.

1. Character-oriented functions (**fgetc** and **fputc**)
2. Line-oriented functions (**fgets** and **fputs**)
3. Formatted functions (**fscanf** and **fprintf**)

All of these functions are found in the standard library **stdio.h**.

fclose: Closing a file

- Operating systems have a limit on the number of files that can be opened by a program.
 - Files must be closed with the **fclose** function:
- **fclose(fp);**
- Closing a file frees the file pointer and associated buffers.

Example 1

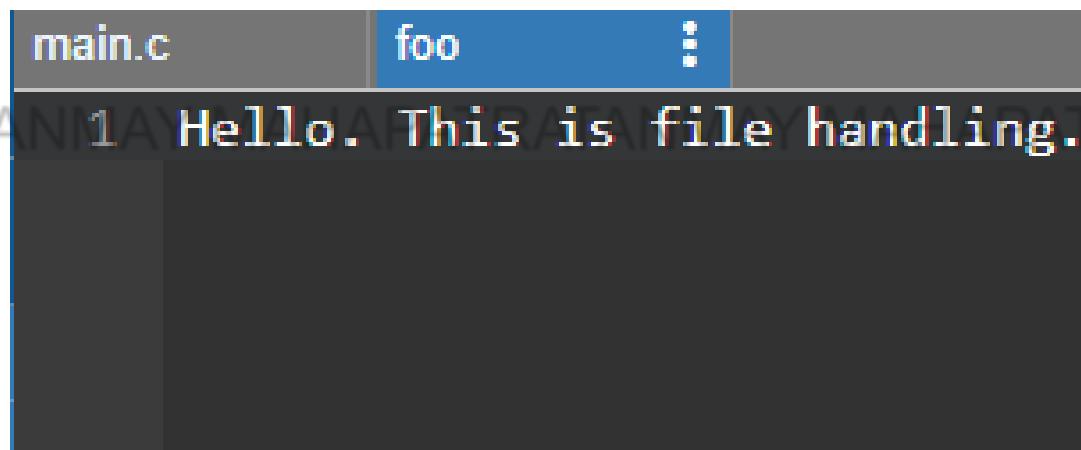
```
int main() {
    FILE *fp; char buf1[80], buf2[80];
    fputs("Enter a line of text: \n", stdout);
    fgets(buf1, sizeof(buf1), stdin);
    fp = fopen("foo", "w");
    fputs(buf1, fp);
    fclose(fp);
    fp = fopen("foo", "r");
    fgets(buf2, sizeof(buf2), fp);
    fputs(buf2, stdout);
    fclose(fp);
    return 0;
}
```

```
if (fp == NULL){
    fputs("Error", stdout);
    return 0;
}
```

OUTPUT

Enter a line of text:

Hello. This is file handling.



A screenshot of a terminal window. The title bar shows "main.c" and "foo :". The main area of the terminal displays the text "Hello. This is file handling." in white on a black background. The cursor is visible at the end of the text.

Hello. This is file handling.

Example 2:

Read the numbers in the “data.txt”, compute their sum and write the sum to “output_file.txt”.

data.txt:

45

34

67

46

99

103

183

59

30

56

fprintf and fscanf

int fprintf (FILE * stream, const char * format, ...)

- The function call of the fprintf in C accepts a stream that is a pointer to a file object in which we need to write data, and the format that consists of text to be written to the stream. If it is a success, the function returns an integer value, the count of characters written to the file.

int fscanf(FILE *stream, const char *format, ...)

- reads formatted input from a stream

Example 2: Code

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    char toRead[100];
    FILE * fp = fopen("data.txt", "r");
    int sum = 0;
    while(fgets(toRead,100,fp) != NULL) {
        printf( "%s\n" , toRead) ;
        sum += atoi(toRead);
    }
    fclose(fp);
    FILE * fp2 = fopen("output_file.txt", "w");
    fprintf(fp2, "%d", sum);
    fclose(fp2);
    return 0;
}
```

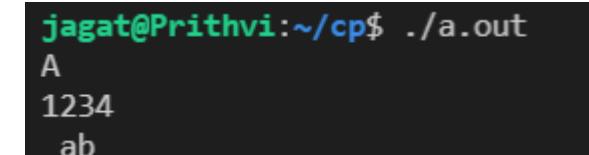
Program to manipulate the file offset/position pointer

```
int main(){
    FILE *fp; int c, x;
    char buf1[80] = "A1234 abcd"; char buf2[80];
    fp = fopen("foo", "w+");
    fputs(buf1, fp);
    rewind(fp); ←
    c = fgetc(fp);
    fputc(c, stdout);
    printf("\n");
    fscanf(fp, "%d", &x);
    fprintf(stdout, "%d", x);
    printf("\n");
    fgets(buf2, 4, fp);
    fputs(buf2, stdout);
    printf("\n");
    return 0;
```

Rewind function sets the file position to the beginning of the file of the given stream.



main.c | foo :
1 A1234 abcd |



jagat@Prithvi:~/cp\$./a.out
A
1234
ab

Program to read text from terminal, save it in a file and then reading again

```
int main() {  
    FILE *fp; char buf[80];  
    fp = fopen("foo", "w+");  
    fputs("Enter a few lines, [Ctrl-d] to exit\n", stdout);  
    while(fgets(buf, 80, stdin) != NULL)  
        fputs(buf, fp);  
    rewind(fp);  
    fputs("Reading from foo... \n", stdout);  
    while(fgets(buf, 80, fp) != NULL)  
        fputs(buf, stdout);  
    return 0;  
}
```

OUTPUT

```
Enter a few lines, [Ctrl-d] to exit
```

```
Line 1 of file hadling.
```

```
Line 2 of file handling.
```

```
main.c
```

```
foo
```

```
:
```

```
1 Line 1 of file hadling.
```

```
2 Line 2 of file handling.
```

```
Reading from foo...
```

```
Line 1 of file hadling.
```

```
Line 2 of file handling.
```

Practice problems

- Write a C program which prints itself! [Hint: use file handling]
- Extend the above to program such that comments are not printed. For simplicity, consider single line comments beginning with //
- Given a file f1, write a program to create a new file f2 which contains the contents of f1 double-spaced (i.e., each space replaced by two spaces, each new line by two new lines, and each tab (\t) by two tabs. Rest of the content remains unchanged).



Example of File Handling with Command Line Arguments

Command Line Arguments

- Values can be passed from command line when the C programs are executed.
- To handle command line arguments, `main()` function is modified as:
 - `int main(int argc, char *argv[])`
 - `argc` – number of arguments passed
 - `argv[]` – is a pointer array to the arguments passed

Example 1:

```
int main(int argc, char *argv[])
{
    printf("Number of arguments: %d \n", argc);
    int i = 0;
    while(i < argc){
        printf("Argument %d: %s \n", i, argv[i]);
        i++;
    }
    return 0;
}
```

```
amitesh@Prithvi:~$ gcc test1.c
amitesh@Prithvi:~$ ./a.out hello world
```

```
Number of arguments is 3
```

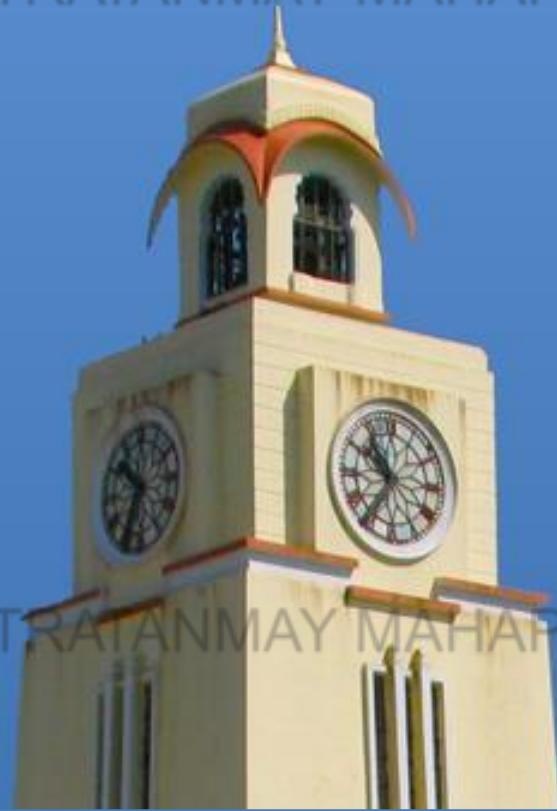
```
Argument 0: ./a.out
Argument 1: hello
Argument 2: world
```

Example 2: To display the content of one file

```
#include <stdio.h>
int main(int argc, char *argv[]){
FILE *fp; char ch;
fp = fopen(argv[1], "r");
if (fp == NULL) {
    printf("Error");
    return(0);
}
ch = fgetc(fp);
while (ch != EOF) {
    printf("%c", ch);
    ch = fgetc(fp);
}
fclose(fp);
return 0;
}
```

Example 3: To display the content of more than one file

```
#include <stdio.h>
int main(int argc, char *argv[]){
    int nof = argc-1;
    while (nof > 0) {
        FILE *fp; char ch;
        fp = fopen(argv[nof], "r");
        ch = fgetc(fp);
        while (ch != EOF) {
            printf ("%c", ch);
            ch = fgetc(fp);
        }
        fclose (argv[nof]);
    }
    nof--;
    return 0;
}
```



Thank you
Q & A

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA



Module 14: The C Preprocessor

BITS Pilani

Pilani Campus

Department of Computer Science & Information Systems

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA

Module Overview

- C Preprocessor
 - Preprocessor Directives
 - Macro expansion
 - File inclusion
 - Conditional Compilation

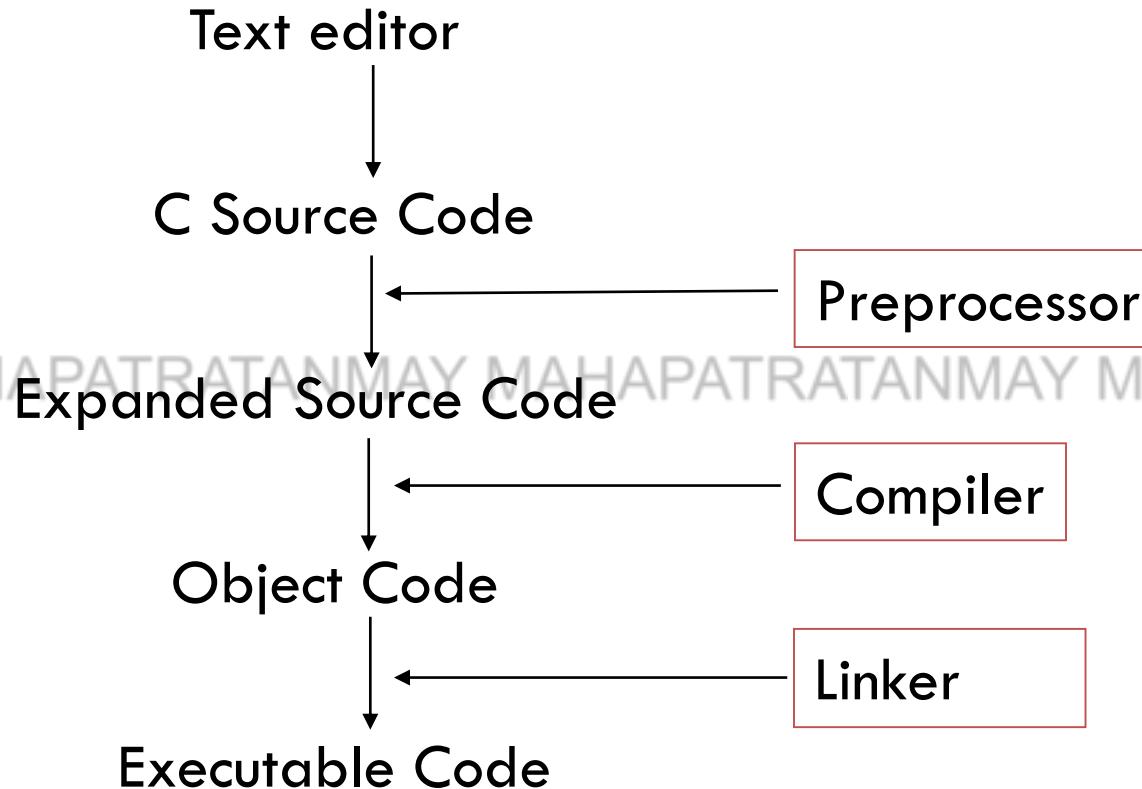


The C Preprocessor

The C Preprocessor

- The **C Preprocessor** is a “Program that processes source program before it is passed to the compiler”
- Preprocessor commands are called as **Directives**
- Each directive begins with a **# symbol**

C Program execution





Macro Expansion Directives

Macro Expansion Directives: Example 1 (Defining constants)



```
#define MAX 100
#define SIZE 10
void main()
{
    int k, a[SIZE];
    for(k=0; k<MAX; k+=10)
        printf("%d", k);
}
```

During preprocessing each occurrence of MAX is replaced by 100 and each occurrence of SIZE is replaced by 10.

And then the program is compiled!

Example 2 (Define Operators)

```
#define AND &&
#define OR ||
#define EQUALS ==
#define MOD %
#define NOTEQUAL !=
```

```
TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA
void Leap_year(unsigned int yr)
{
    if((yr MOD 4 EQUALS 0) AND (yr MOD 100 NOTEQUAL 0) OR (yr
        MOD 400 EQUALS 0))
        printf("Given Year is Leap Year\n");
    else
        printf("Given Year is Not a Leap Year\n");
}
```

Example 3 (Define Conditions)

```
#define AND &&
#define OR ||
#define RANGE ((ch>='a' AND ch<='z') OR (ch>='A' AND
    ch<='Z'))
void Alpha_Check(char ch)
{
    if(RANGE)
        printf("Entered character is an Alphabet\n");
    else
        printf("Entered character not an Alphabet\n");
}
```

Macros with Arguments

```
#define CUBE (x) (x*x*x)
```

If statements appears in the program like

```
vol = CUBE(side);
```

It will expand to:

```
vol = (side*side*side);
```

If statements appears in the program like

```
vol = CUBE(x+y);
```

It will expand to:

```
vol = (x+y*x+y*x+y);
```

If statements appears in the program like

```
printf("Volume = CUBE(a)"); Expansion???
```

Nesting of Macros

One Macro definition can be used in another Macro

Example:

Consider the Macro Definitions

```
#define SQUARE (p) ((p)*(p))  
#define CUBE (p)      (SQUARE (p) * (p))  
#define EIGHTH (p)   (CUBE (p) *CUBE (p) *SQUARE (p))
```

How will it expand???

Nesting of Macros (Contd.)

Macros calls can be nested as function calls

Consider a Macro definition:

```
#define MIN(P,Q)  (( (P<Q) ) ? (P) : (Q))
```

```
TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA
int MinThree(int a, int b, int c)
{
    int min;
    min = MIN(a, MIN(b,c)); /* Macro Call */
    return (min);
}
```

Exercise:

Write a macro call for getting min of five values.

Advantages of Macros

- One place replacement is required for any changes
- Easy to handle
- Less error prone
- Easy to debug

Macro vs Function()

- Functions make the program compact and smaller but they slow down the program
 - ***Each call results in a push on the activation stack.***
 - ***The corresponding return is a pop from the stack.***
- Macros make the program run faster but increase the program size
 - ***Macros reduce the function call overhead by performing code replacement at compile time.***
 - ***Code replacement happens once (per compilation), whereas function calls can be repetitive causing performance bottleneck.***

Important Questions:

When is it best to use macro?

And when is it best to use function?

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA

Can we replace all Macros with functions?



- Macros do not have types (for parameters)

- What will happen if we declare

```
#define fact(n) (n==0) ? 1 : n*fact(n-1)
```

- Try this out! Recursive substitutions are not possible

- Why not?

- Because the compiler does not “execute” code.
- Only code replacement is done.
- Code replacement is a one-time job.
- Code replacement is not recursive.



File Inclusion Directives

File Inclusion Directives

How to include a file in your program?

```
#include “file_name” OR  
#include<file_name>
```

What is the difference between above?

1. If a large program is divided into several files, each containing a set of related functions. These files should be included at the beginning of the main program file.
2. Inclusion of header files for using some predefined functions in C library.

When a header file is included, the preprocessor includes the file in the source code program (.c file) before its compilation.



Compiler Control Directives

Compiler Control Directives: #ifdef



We can skip over the compilation process of the part of the source program by inserting the preprocessing commands:

#ifdef and **#endif**

Syntax:

```
#ifdef macroname
    statement 1;
    statement 2;
#endif
```

If macro has been defined, the block of code will be processed as usual; otherwise not.

Compiler Control Directives: #if

```
#define CHECK 1
void main()
{ #if CHECK==1
    statement1;
    statement2;
#elif CHECK==0
    statement3;
    statement4;
#else
    statement5;
    statement6;
#endif}
```

CHECK is Macro Name

What we need?

1. If we don't want to compile some part of the program so we can skip those statements.

2. To make a program portable so it can work on two or more than two types of different architectures.

“`#ifndef`” guards for header files

```
#ifndef HEADER1
```

“if not defined”

```
#define HEADER1
```

then define

```
#include <stdio.h>
```

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA

...

...

```
#endif
```

While linking multiple .o files in a C project, if compiler sees same inclusion in multiple file that are being linked, it throws an error. This guard helps in including the header file only once.

TANMAY MAHAPATRATANMAY MAHAPATRATANMAY MAHAPATRA



Thank you
Q & A