



# PROBLEM SOLVING WITH C

## UE23CS151B

---

**Prof. Sindhu R Pai**  
Department of Computer Science  
and Engineering

# PROBLEM SOLVING WITH C

---

## Introduction

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# Introduction

---

1. Inventors of C Language
2. Programming Language(PL)
3. Why PLs? and why so many PLs ?
4. Brief introduction on Paradigms
5. First Programmer
6. Why C after Python?
7. Applications of C
8. History of PLs and Development of C
9. Salient Features of C

### What is Programming Language(PL)?

- Formal computer language or constructed language designed to communicate instructions to a machine, particularly a computer
- A way of telling a machine what operations to perform
- Can be used to create programs to control the behavior of a machine or to express algorithms

### Why PLs and why so many PLs?

- To choose the right language for a given problem
- Applications
  - Web Browsers, Social Networks, Image Viewer
  - Facebook.com, Bing, Google.com, Games
  - Various OS

# PROBLEM SOLVING WITH C

## Introduction

---

### Paradigm – Style of Programming

- Imperative
  - Systematic way of instructing (in order)
  - FORTRAN, Algol, COBOL, Pascal
- Structured programming
  - Single entry and single exit. Avoid GOTO
  - C
- Procedural
  - Subset of imperative
  - Use of subroutines
  - C
- Declarative
  - Declares a set of rules about what outputs should result from which inputs
  - Lisp, SQL

### Paradigm – Continued..

- Functional
  - Function should be the first class citizen. No side effects. Assign Function name to another, pass function name as an argument, return function itself
  - If and recursion. No loops
  - Scheme, Haskell, Miranda and JavaScript
- Logical
  - Uses predicates
  - Extraction of knowledge from basic facts and relations
  - ASP, Prolog and Datalog
- Object-Oriented
  - User perception is taken into account.
  - Data needs protection
  - Java, c++, Python

# PROBLEM SOLVING WITH C

## Introduction -Levels of Languages

High Level Languages	Middle Level Languages	Low Level Languages
<ul style="list-style-type: none"><li>§ Allows a higher abstraction</li><li>§ Extensive Error checking</li><li>§ Automatic memory management, garbage collection</li><li>§ Easy to interface with Kernel</li><li>§ Portable</li><li>• Ex: Java, Python.</li></ul>	<ul style="list-style-type: none"><li>§ Binds the gap between the machine language and high level language.<ul style="list-style-type: none"><li>• Provides sufficient abstraction</li></ul></li><li>§ Versatile and suitable for a wide range of applications.</li><li>§ Portable<ul style="list-style-type: none"><li>• Ex: C</li></ul></li></ul>	<ul style="list-style-type: none"><li>§ More efficient in execution speed and memory usage.</li><li>§ Small code size, which can be crucial in environments with limited storage.</li><li>§ Direct hardware interaction — programmers control hardware components, like registers, and memory.<ul style="list-style-type: none"><li>• They are not portable</li><li>• Ex: Assembly Language.</li></ul></li></ul>

### Why C after Python ?

- Paradigms - Procedural vs OOP
- Aged compared to Python
- Growth rate of PLs - TIOBE Index

### TIOBE Index

- An indicator of the popularity of programming languages. Updated once in a Month
- The ratings are based on the number of skilled engineers world-wide, courses and third party vendors. Popular search engines such as Google, Bing, Yahoo!, Wikipedia, Amazon, YouTube and Baidu are used.
- It is not about the best programming language or the language in which most lines of code have been written

# PROBLEM SOLVING WITH C

## Introduction

TIOBE Index: <https://www.tiobe.com/tiobe-index/>

Jan 2024	Jan 2023	Change	Programming Language	Ratings	Change
1	1		 Python	13.97%	-2.39%
2	2		 C	11.44%	-4.81%
3	3		 C++	9.96%	-2.95%
4	4		 Java	7.87%	-4.34%
5	5		 C#	7.16%	+1.43%
6	7	▲	 JavaScript	2.77%	-0.11%
7	10	▲	 PHP	1.79%	+0.40%
8	6	▼	 Visual Basic	1.60%	-3.04%
9	8	▼	 SQL	1.46%	-1.04%
10	20	▲	 Scratch	1.44%	+0.86%
11	12	▲	 Go	1.38%	+0.23%
12	27	▲	 Fortran	1.09%	+0.64%
13	17	▲	 Delphi/Object Pascal	1.09%	+0.36%

### Best Applications of C/C++

- Linux Kernel: It is written in C
- Adobe Systems: Includes Photoshop and Illustrator
- Mozilla: Internet Browser Firefox Uses C++.
- Bloomberg: Provides real time financial information to investors
- Callas Software: Supports PDF creation, optimization, updation tools and plugins
- Symbian OS: Used in cellular phones.

### History of PLs

- 1820-1850 England, Charles Babbage invented two mechanical Computational device i.e., Analytical Engine and Difference Engine
- In 1942, United States, ENIAC used electrical signals instead of physical motion
- In 1945, Von Newmann developed two concepts: Shared program technique and Conditional control transfer
- In 1949, Short code appeared
- In 1951, Grace Hopper wrote first compiler, A-0
- Fortran: 1957, designed by John Backus
- Lisp, Algol- 1958
- Cobol: 1959
- Pascal: 1968, Niklaus Wirth
- C: 1972, D Ritchie
- C++: 1983, Bjarne Stroustrup, Compile time type checking, templates are used.
- Java: 1995, J. Gosling, Rich set of APIs and portable across platform through the use of JVM

# PROBLEM SOLVING WITH C

## Introduction

---

### First Programmer – Ada Lovelace



# PROBLEM SOLVING WITH C

## Introduction

---

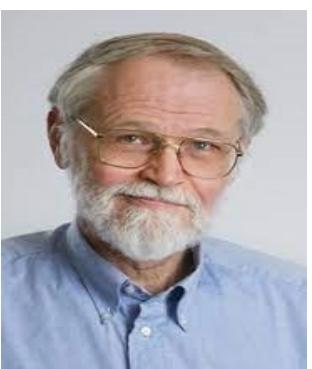
- Charles Babbage – Father of Computer



- Dennis Ritchie



- Brian Kernighan



# PROBLEM SOLVING WITH C

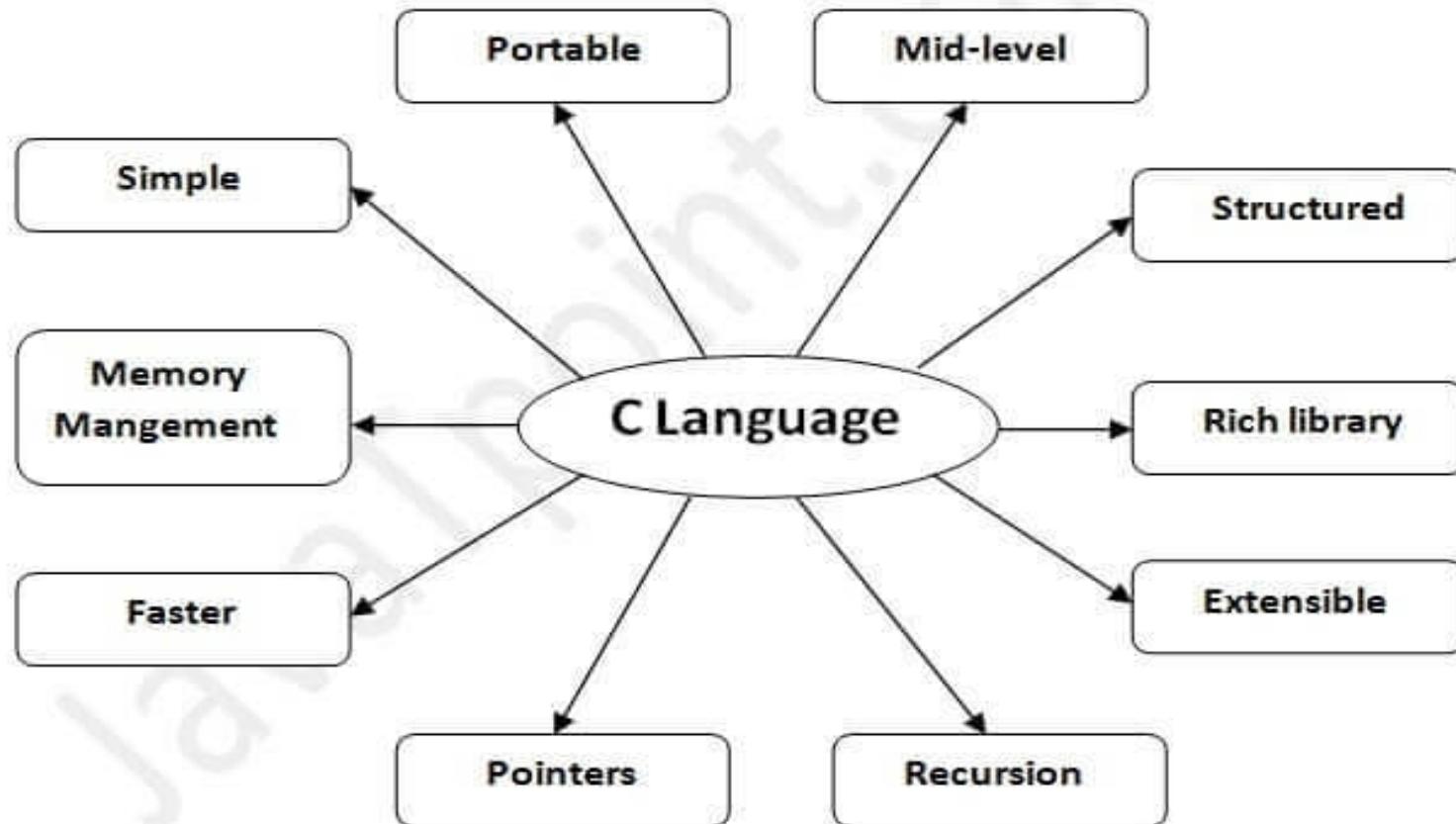
## Introduction

---

### Development of C

- Martin Richards, around 60's developed BCPL [Basic Combined Programming Language]
- Enhanced by Ken Thompson and Introduced B language.
- C is Originally developed between 1969 and 1973 at Bell Labs by Dennis Ritchie and Kernighan. Closely tied to the development of the Unix operating system
- Standardized by the ANSI [American National Standards Institute] since 1989 and subsequently by ISO[International Organization for Standardization].
- C89, C99, C11

## Features



### Some Salient Features of C Language

- 1. Simple and Efficient:** The basic syntax of implementing C language is very simple and easy to learn. This makes the language easily comprehensible and enables a programmer to redesign or create a new application.
- 2. Fast:** C is a compiled and statically typed language.
- 3. Portability:** C programs are machine-independent which means that you can run the code on various machines with little or no modifications

### Some Salient Features of C Language

**4. Dynamic Memory Management:** One of the most significant features of C language is its support for dynamic memory management (DMA). It means that you can utilize and manage the size of the data structure in C during runtime. C also provides several predefined functions to work with memory allocation.

**5. Mid-Level Programming Language:** C language supports low level programming such as working with memory as well as supports high level programming making it a mid level programming language.

**6. Pointers:** With the use of pointers in C, you can directly interact with memory. As the name suggests, pointers point to a specific location in the memory and interact directly with it.



**THANK YOU**

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering



# PROBLEM SOLVING WITH C

## UE23CS151B

---

**Prof. Sindhu R Pai**  
Department of Computer Science  
and Engineering

# PROBLEM SOLVING WITH C

---

## C Programming Environment

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

## C Programming Environment

---



1. Installation of gcc on different Operating systems
2. Program Development Life Cycle [PDLC]
3. First Program in C
4. Structure of C Program
5. Steps involved in execution of C Program
6. C Compiler standards
7. Errors during execution

# PROBLEM SOLVING WITH C

## C Programming Environment

---



### Installation of gcc on different Operating systems

- Windows OS – Installation of gcc using Mingw.
  - <https://www.youtube.com/watch?v=sXW2VLrQ3Bs>
- Linux OS – gcc available by default

# PROBLEM SOLVING WITH C

## C Programming Environment

## Program Development Life Cycle [PDLC]

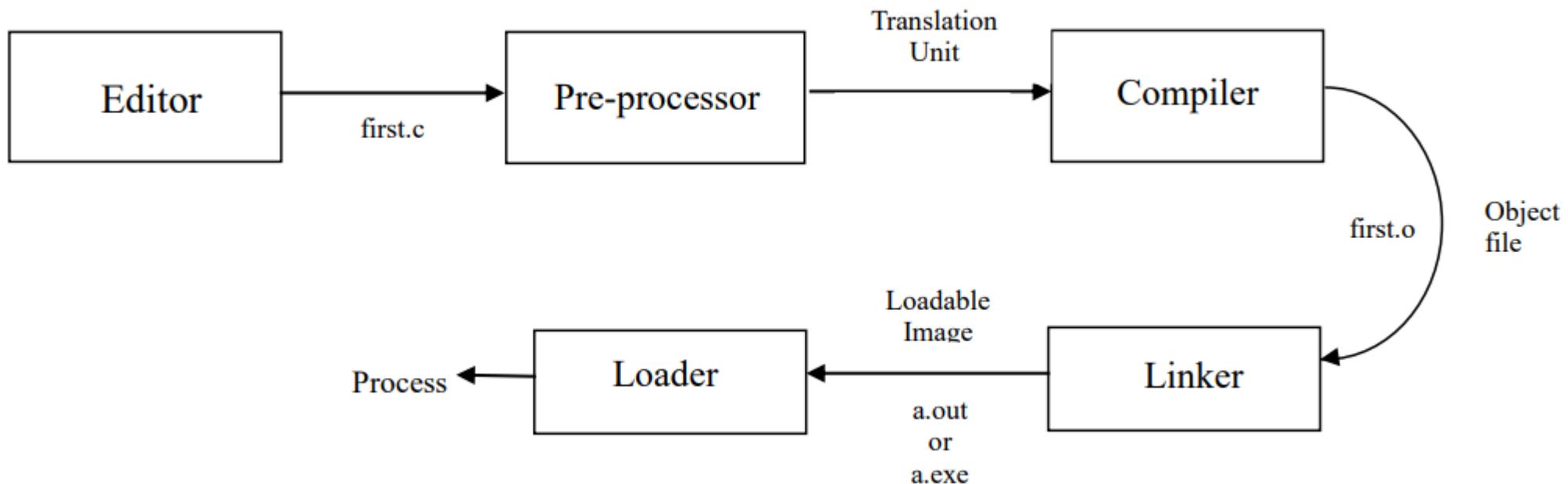


Fig 1: Phases involved in PDLC

# PROBLEM SOLVING WITH C

## C Programming Environment

---



### First Program in C

```
#include<stdio.h>
int main()
{
    printf("Hello PES\n");
    return 0;
}
```

Note: main is a function which returns int value. There will be a temporary location to hold the return value which caller can use.

# PROBLEM SOLVING WITH C

## C Programming Environment

---



### Program Structure

- Case sensitive
- Indentation is not a language requirement, but preferred
- The main() is the starting point of execution
- Comments in C
  - Using // for single line comment
  - Using /\* and \*/ for multiline comment

# PROBLEM SOLVING WITH C

## C Programming Environment

---

### Steps involved in execution of code

Method 1:

Step1: gcc <filename> // image a.out is the output  
Step2: ./a.out OR a.exe

Method 2: Creating the object files separately

Compile: gcc -c <filename> // filename.o is the output  
Link: gcc filename.o OR gcc <list of object files> -l <library>  
Execute: ./a.out OR a.exe

Method 3: Renaming the output files

Step1: gcc <filename> -o <imagename>  
Step2: <imagename>

# PROBLEM SOLVING WITH C

## C Programming Environment

---



### C Compiler standards

Standardized by ANSI and ISO:      **C89, C99 and C11**

Using c99:

```
gcc -std=c99 program.c
```

Using c11:

```
gcc -std=c11 program.c
```

Use **`_STDC_VERSION_`** we can get the standard C version

# PROBLEM SOLVING WITH C

## C Programming Environment

---



### Errors during Execution

- Compile time Error
- Link time Error
- Run time Error
- Logical Error



**THANK YOU**

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering



# PROBLEM SOLVING WITH C

## UE23CS151B

---

**Prof. Sindhu R Pai**  
Department of Computer Science  
and Engineering

# PROBLEM SOLVING WITH C

---

## Simple Input / Output

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

## Simple Input/Output

---



1. Formatted output function
2. Formatted input function
3. Unformatted functions

# PROBLEM SOLVING WITH C

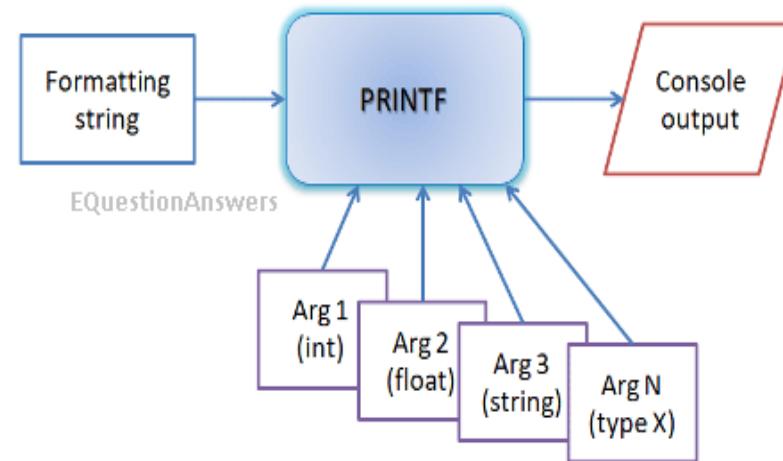
## Simple Input/Output

### Formatted output function – printf()

Syntax:

```
int printf (const char *format,arg1,arg2 ...)
```

- printf is a predefined function declared in “stdio.h” header file
- Sends formatted output to stdout by default



# PROBLEM SOLVING WITH C

## Simple Input/Output

---



### Formatted output function – printf()

- Output is controlled by the first argument
- Has the capability to evaluate an expression
- On success, it returns the number of characters successfully written on the output. On failure, a negative number is returned.
- Arguments to printf can be expressions

Ex1:

```
#include <stdio.h>
int main()
{
    printf("Let us C");
    return 0;
}
```

# PROBLEM SOLVING WITH C

## Simple Input/Output

---

### Example Code

```
#include <stdio.h>
int main()
{
    int num1;
    float num2;
    printf ("Input an integer value:");
    scanf ("%d", &num1);
    printf ("Input a float value:");
    scanf ("%f", &num2);
    printf ("\n return value is %d\n", printf ("%f",num2));
    printf ("\n return value is %d\n", printf ("%d", num1));
    return 0;
}
```



### Format string

- The format string contains two types of objects :ordinary characters, which are copied to the output stream and conversion specifications.
- The format string is of the form % [flags] [field\_width] [.precision] conversion\_character where components in brackets [] are optional. The minimum requirement is % and a conversion character (e.g. %d).
- **Flags** -optional argument which specifies output justification such as numerical sign, leading zeros or octal, decimal, or hexadecimal prefixes.

### Format string

- **field\_width** -optional minimum width of the output field
- **Precision** -optional argument which specifies the maximum number of characters to print.
- **Conversion character** indicates the type of the argument that is being formatted or parsed.
- **Ex-** %d(dec int),%x(hex int), o(octint),%f(float),%c(char),%p(address),%lf(double)

# PROBLEM SOLVING WITH C

## Simple Input/Output

### Escape Sequence

\a	Alarm or Beep	It is used to generate a bell sound
\b	Backspace	It is used to move the cursor one place backward.
\\\	Backslash	Use to insert backslash character.
\n	New Line	It moves the cursor to the start of the next line.
\r	Carriage Return	It moves the cursor to the start of the current line.
\t	Horizontal Tab	It inserts some whitespace to the left of the cursor and moves the cursor accordingly.
\'	Single Quote	It is used to display a single quotation mark.
\"	Double Quote	It is used to display double quotation marks.

# PROBLEM SOLVING WITH C

## Simple Input/Output

---

```
#include<stdio.h>
int main()
{
    int a,b;
    a = 16,b=78;
    float c = 2.5;
    printf("%d %d\n",sizeof(a),sizeof(short int));
    printf("%4.2f\n",c);           //      width.precision , f is float type
    printf("%2.1f\n",c);
    printf("%-5d %d\n",16,b);     //      - is for left justification
    printf("ravan\rram\n");
    printf("ram\nlakhan\n");
    printf("ram\tbheem\n");
    printf("\\"Mahabharath\\\"");
    //      To include quotes in the output
    return 0;
}
```

# PROBLEM SOLVING WITH C

## Simple Input/Output

---

Note:

`sizeof(short int) <= sizeof(int) <= sizeof(float) <= sizeof(double)`

size of a type depends on the system of implementation



### Formatted input function – scanf()

Syntax:

- int scanf(const char \*format, ...)
- predefined function declared in stdio.h
- & - address operator is compulsory in scanf for all primary types
- Reads formatted input using stdin. By default it is keyboard
- This function returns the following value
  - >0 — The number of items converted and assigned successfully
  - 0 — No item was assigned.
  - <0 — Read error encountered or end-of-file (EOF) reached before any assignment was made

# PROBLEM SOLVING WITH C

## Simple Input/Output

---

Example Code:

```
# include <stdlib.h>

int main()

{   int a,b,n;

    printf("Enter the value of a and b");

    scanf("%d,%d", &a, &b);

    printf("a : %d b : %d\n", a, b);

    printf("Enter the value of a");

    n = scanf("%d",&a);           // Return value of scanf

    printf("The return value of scanf is %d\n",n);

    printf("The value of a is %d",a); }
```





**PES**  
**UNIVERSITY**  
**ONLINE**

**THANK YOU**

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering



# PROBLEM SOLVING WITH C

## UE23CS151B

---

**Prof. Sindhu R Pai**

Department of Computer Science  
and Engineering



# PROBLEM SOLVING WITH C

---

## Operators and Expressions in C

Prof. Sindhu R Pai

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

## Operators in C

---



1. Operators and its Classification
2. Expression
3. Sequence point operation

# PROBLEM SOLVING WITH C

## Operators in C

---



### Operator and its Classification

- Operator is a symbol used for calculations or evaluations
- Has rank, precedence and Associativity

#### Classification:

##### 1. Based on the operation:

Arithmetic: +, -, \*, /, %, Increment(++) & Decrement(--)

Relational: >, <, <=, >, >=, ==, !=

Logical(short circuit evaluation): &&, ||, !

Bitwise: &, |, ^, ~, <<, >>

Address: &

Dereferencing Operator: \*

##### 2. Based on the number of operands: Unary, Binary, Ternary

# PROBLEM SOLVING WITH C

## Operators in C



### ARITHMETIC BINARY OPERATORS

Symbol	Description	Syntax
+	Adds two numeric values.	$a + b$
-	Subtracts two numeric values	$a - b$
*	Multiply two numeric values.	$a * b$
/	Divide two numeric values.	$a / b$
%	Returns the remainder after division .	$a \% b$

# PROBLEM SOLVING WITH C

## Operators in C



### ARITHMETIC UNARY OPERATORS

Symbol	Operator	Description	Syntax
+	Unary Plus	Used to specify the positive values.	+a
-	Unary Minus	Flips the sign of the value.	-a
++	Increment	Increases the value of the operand by 1.	a++ or ++a
--	Decrement	Decreases the value of the operand by 1.	a-- or --a

# PROBLEM SOLVING WITH C

## Operators in C



### RELATIONAL OPERATORS

Symbol	Operator	Description	Syntax
<	Less than	Returns true if the left operand is less than the right operand else returns false	$a < b$
>	Greater than	Returns true if the left operand is greater than the right operand else returns false	$a > b$
$\leq$	Less than or equal to	Returns true if the left operand is less than or equal to the right operand else returns false	$a \leq b$

# PROBLEM SOLVING WITH C

## Operators in C



### RELATIONAL OPERATORS

Symbol	Operator	Description	Syntax
<code>&gt;=</code>	Greater than or equal to	Returns true if the left operand is greater than or equal to right operand. else returns false	<code>a &gt;= b</code>
<code>==</code>	Equal to	Returns true if both the operands are equal.	<code>a == b</code>
<code>!=</code>	Not equal to	Returns true if both the operands are NOT equal.	<code>a != b</code>

# PROBLEM SOLVING WITH C

## Operators in C

---



### LOGICAL OPERATORS

Symbol	Operator	Description	Syntax
&&	Logical AND	Returns true if both the operands are true.	a && b
	Logical OR	Returns true if both or any of the operand is true.	a    b
!	Logical NOT	Returns true if the operand is false.	!a

# PROBLEM SOLVING WITH C

## Operators in C



### BITWISE OPERATORS

Symbol	Operator	Description	Syntax
&	Bitwise AND	Performs bit-by-bit AND operation and returns the result.	$a \& b$
	Bitwise OR	Performs bit-by-bit OR operation and returns the result.	$a   b$
^	Bitwise XOR	Performs bit-by-bit XOR operation and returns the result.	$a ^ b$

# PROBLEM SOLVING WITH C

## Operators in C



### BITWISE OPERATORS (Contd)

Note: Examples shown below have considered 16 bit representation of an integer instead of 32 bit

Symbol	Operator	Description	Syntax	Example
<code>~</code>	Bitwise one's Complement	Flips one to zero and zero to one	<code>~a</code>	<code>a=5</code> i.e. 0000 0000 0000 010 <code>~a = 1111 1111 1111 1010</code>
<code>&lt;&lt;</code>	Bitwise Left shift	Shifts the binary number to the left by one place and returns the result.	<code>a &lt;&lt; b</code>	<code>a=5</code> (0000 0000 0000 0101) <code>b=2</code> <code>a&lt;&lt;b</code> which means <code>5&lt;&lt;2</code> shift 5 left twice 0000 0000 0001 0100 The value is 20
<code>&gt;&gt;</code>	Bitwise Right shift	Shifts the binary number to the right by one place and returns the result	<code>a &gt;&gt; b</code>	<code>a=16</code> i.e 0000 0000 0001 0000 <code>b=3</code> <code>a&gt;&gt;b</code> which means <code>16&gt;&gt;3</code> right shift thrice 0000 0000 0000 0010 The value is 2

# PROBLEM SOLVING WITH C

## Operators in C

---



Few inputs on carrying out arithmetic operation using bit wise operators:

1.  $n \ll 1$  can be used to multiply variable n by 2 and  $n \gg 1$  for dividing n by 2

2.  $n \& 1$  can be used to check whether n is odd

3. To swap two variables a and b, the following code can be used

$a = a \wedge b$

$b = a \wedge b$

$a = a \wedge b$

4. To check whether  $i^{\text{th}}$  bit in a variable n is 1

$n \& (1 \ll i)$  : Result 0 implies not set and non 0 implies set

5. To set the  $i^{\text{th}}$  bit in n

$n = n | (1 \ll i)$

6. To clear the  $i^{\text{th}}$  bit in n

$n = n \& \sim(1 \ll i)$

# PROBLEM SOLVING WITH C

## Operators in C



### ASSIGNMENT OPERATORS

Symbol	Operator	Description	Syntax
=	Simple Assignment	Assign the value of the right operand to the left operand.	a = b
+=	Plus and assign	Add the right operand and left operand and assign this value to the left operand.	a += b
-=	Minus and assign	Subtract the right operand and left operand and assign this value to the left operand.	a -= b

# PROBLEM SOLVING WITH C

## Operators in C

---



## SHORT HAND ASSIGNMENT OPERATORS

Symbol	Operator	Description	Syntax
<code>*=</code>	Multiply and assign	Multiply the right operand and left operand and assign this value to the left operand.	<code>a *= b</code>
<code>/=</code>	Divide and assign	Divide the left operand with the right operand and assign this value to the left operand.	<code>a /= b</code>
<code>%=</code>	Modulus and assign	Assign the remainder in the division of left operand with the right operand to the left operand.	<code>a %= b</code>

# PROBLEM SOLVING WITH C

## Operators in C

---



## OTHER OPERATORS

Symbol	Operator	Description	Syntax
sizeof	sizeof	A compile-time unary operator which can be used to compute the size of its operand	sizeof(variable) sizeof(type)
,	Comma Operator	Used to string together several expression. The result of the rightmost expression becomes the value of the total comma-separated expression.	b=(a=3,a+1); First assigns 3 to a and assigns 4 to b.

# PROBLEM SOLVING WITH C

## Operators in C



### OTHER OPERATORS

Symbol	Operator	Description	Example
?:	Ternary	<p><b>Syntax:</b> (Expression1) ? Expression2 : Expression3</p> <p>If Expression1 evaluates to true then Expression 2 is returned else Expression3 is returned</p>	<p>Example: (usage)</p> <pre>int a= 10; int b = 20; printf("%d\n",(a&gt;b)?a:b);</pre> <p>Outputs 20</p>

# PROBLEM SOLVING WITH C

## Operators in C

---



### Expression

- An expression consists of Operands and Operators Eg:  $6+2$ ,  $a+b$ ,  $a+b*c$
- Evaluation of Operands: Order is not defined
- Evaluation of Operators: Follows the rules of precedence and rules of Associativity

<http://web.cse.ohio-state.edu/~babic.1/COperatorPrecedenceTable.pdf>

# PROBLEM SOLVING WITH C

## Operators in C

---



### Expression

- The precedence of operators is the order in which the operators are evaluated in the expression.
- The operator precedence determines the priority to be given to a particular operator.
- Operator associativity is the order in which operators are evaluated when operators have same precedence.

# PROBLEM SOLVING WITH C

## Operators in C

### Associativity and Precedence



Precedence	Operator	Description	Associativity
1	()	Parentheses (function call)	Left-to-Right
	[]	Array Subscript (Square Brackets)	
	.	Dot Operator	
	->	Structure Pointer Operator	
	++ , --	Postfix increment, decrement	
	++ / --	Prefix increment, decrement	
	+ / -	Unary plus, minus	
2	!, ~	Logical NOT, Bitwise complement	Right-to-Left
	(type)	Cast Operator	
	*	Dereference Operator	
	&	Addressof Operator	

# PROBLEM SOLVING WITH C

## Operators in C

### Associativity and Precedence

	<b>sizeof</b>	Determine size in bytes	
3	<b>*, /, %</b>	Multiplication, division, modulus	Left-to-Right
4	<b>+/-</b>	Addition, subtraction	Left-to-Right
5	<b>&lt;&lt; , &gt;&gt;</b>	Bitwise shift left, Bitwise shift right	Left-to-Right
6	<b>&lt; , &lt;=</b>	Relational less than, less than or equal to	Left-to-Right
	<b>&gt; , &gt;=</b>	Relational greater than, greater than or equal to	
7	<b>== , !=</b>	Relational is equal to, is not equal to	Left-to-Right
8	<b>&amp;</b>	Bitwise AND	Left-to-Right
9	<b>^</b>	Bitwise exclusive OR	Left-to-Right
10	<b> </b>	Bitwise inclusive OR	Left-to-Right
11	<b>&amp;&amp;</b>	Logical AND	Left-to-Right
12	<b>  </b>	Logical OR	Left-to-Right



# PROBLEM SOLVING WITH C

## Operators in C

### Associativity and Precedence(contd)



13

?:

Ternary conditional

Right-to-Left

=

Assignment

+= , -=

Addition, subtraction assignment

\*= , /=

Multiplication, division assignment

14

%= , &amp;=

Modulus, bitwise AND assignment

Right-to-Left

^= , |=

Bitwise exclusive, inclusive OR assignment

&lt;&lt;=, &gt;&gt;=

Bitwise shift left, right assignment

15

,

comma (expression separator)

Left-to-Right

# PROBLEM SOLVING WITH C

## Operators in C

---



### Sequence point Operation

- It is any point in the execution of a program which guarantees or ensures that all the side effects of the previous evaluation of the program's code are successfully performed.
- The term “side effects” is nothing but the changes done by any sort of function or expression where the state of something gets changed.
- Sequence point ensures that none of the alterations or side effects of the subsequent evaluations is yet to be performed at all.

Some of the basic sequence points available in C are: Logical AND, Logical OR, conditional operator, comma operator etc.

# PROBLEM SOLVING WITH C

## Operators in C

---



Example Program (To demonstrate || as a sequence point)

```
#include<stdio.h>

int main()
{
    int a = 11;
    printf("%d",a++==12 || a==12);      //  outputs 1
    printf("\n%d",a);                  //  outputs 12
    return 0;
}
```



**THANK YOU**

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering



# PROBLEM SOLVING WITH C

## UE23CS151B

---

**Prof. Sindhu R Pai**

Department of Computer Science  
and Engineering



# PROBLEM SOLVING WITH C

---

## Control Structures

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

## Control Structures

---

1. Selection structures
2. Looping structures
3. Nested Control Structures
4. Practice Programs



# PROBLEM SOLVING WITH C

## Control Structures

---



### 1. Selection structures/ Decision Making/ Conditional structures

- Simple if Statement
- if – else Statement
- nested if-else statement
- else if Ladder
- switch statement

# PROBLEM SOLVING WITH C

## Control Structures

---



### 2. Looping structures:

- while statement
- for statement
- do ... while statement

### 3. Unconditional structures

- goto statement
- break Statement
- continue Statement
- return Statement

# PROBLEM SOLVING WITH C

## Control Structures

---



### 1. Selection Statements

Normally the statements are executed as written in the program.

But sometimes, the flow of control of the program has to be altered depending upon a condition. Selection statements allows us to do so.

It involves decision making to see whether a particular condition has been met or not.

Ex: if, else, switch

# PROBLEM SOLVING WITH C

## Control Structures



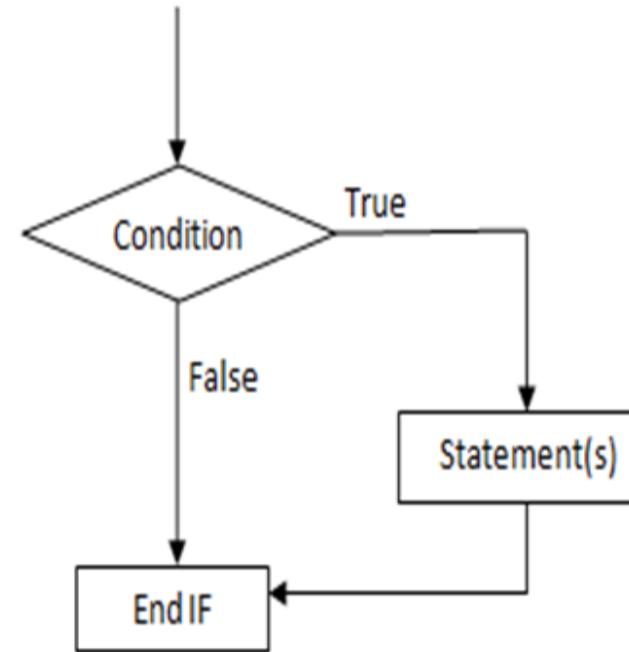
### Selection Structures

§ If

```
if (e1)
    <block> | <stmt>
```

Ex: #include <stdio.h>

```
int main()
{
    int x = 20;
    int y = 18;
    if (x > y)
        printf("x is greater than y");
    return 0;
}
```



# PROBLEM SOLVING WITH C

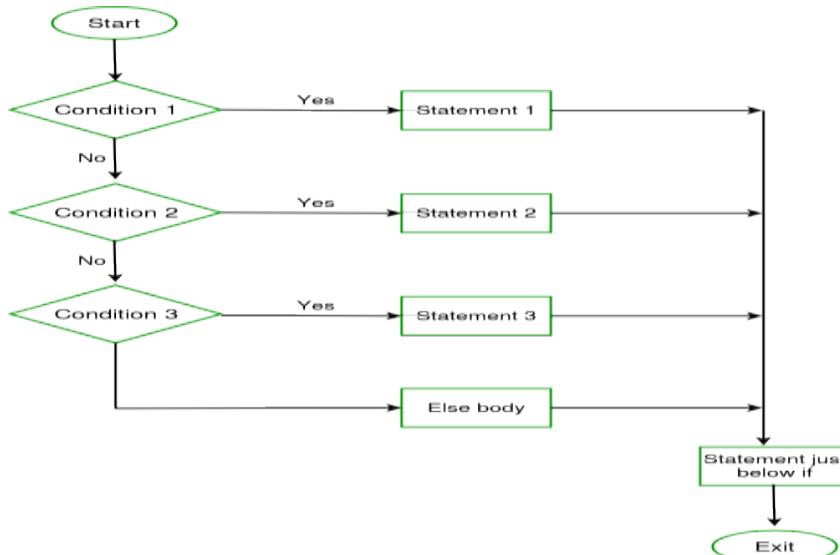
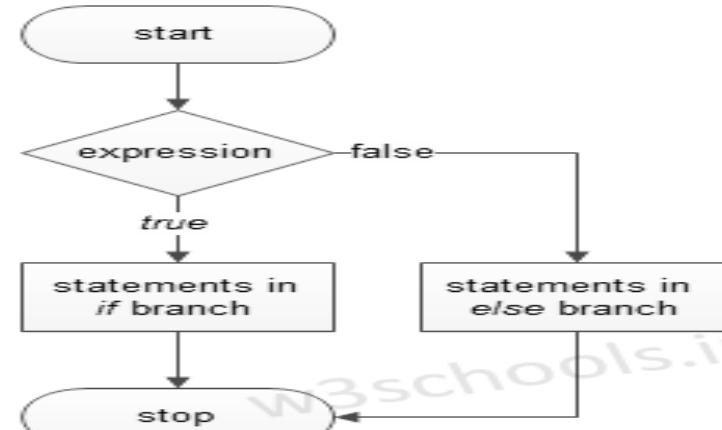
## Control Structures

- If – else

```
if (e1)
    <block>|<stmt>
else
    <block>|<stmt>
```

- If – else if – else if – else

```
if (e1)
    <block>|<stmt>
else if (e2)
    <block>|<stmt>
else
    <block>|<stmt>
```



# PROBLEM SOLVING WITH C

## Control Structures

---



### Example Program

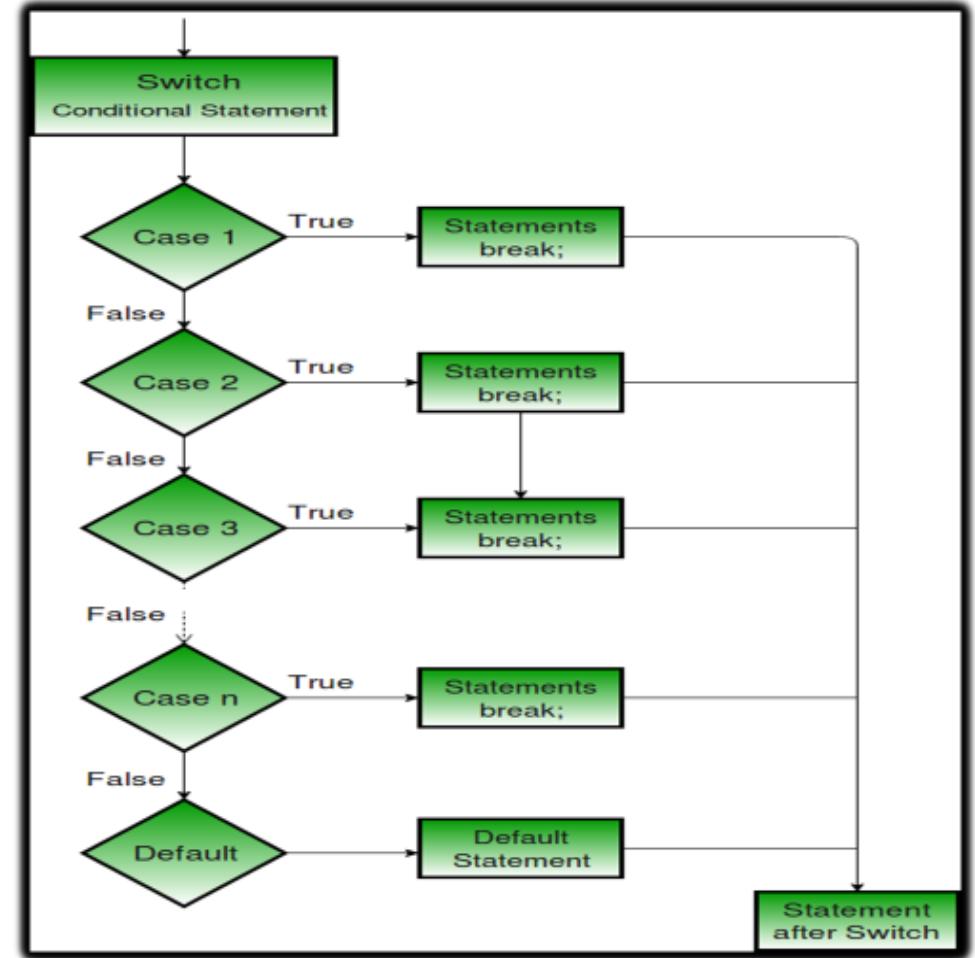
```
#include <stdio.h>

int main() {
    int time = 22;
    if (time < 10) {
        printf("Good morning.");
    } else if (time < 20) {
        printf("Good afternoon");
    } else {
        printf("Good evening.");
    }
    return 0;
}
```

# PROBLEM SOLVING WITH C

## Control Structures

```
switch (expression)
{
    case integral constant: <stmt> break;
    case integral constant: <stmt> break;
    default: <stmt>
}
```



# PROBLEM SOLVING WITH C

## Control Structures

### Example Program



```
#include <stdio.h>

int main() {
    int day = 4;

    switch (day) {
        case 1:
            printf("Monday");
            break;
        case 2:
            printf("Tuesday");
            break;
        case 3:
            printf("Wednesday");
            break;
        case 4:
            printf("Thursday");
            break;
        case 5:
            printf("Friday");
            break;
        case 6:
            printf("Saturday");
            break;
        case 7:
            printf("Sunday");
            break;
    }
    return 0;
}
```

# PROBLEM SOLVING WITH C

## Control Structures

---

### 2. Looping Structures

#### § for

```
for(e1; e2; e3)  
    <block>|<stmt>
```

#### § while

```
while(e2)  
    <block>|<stmt>
```

#### § do while

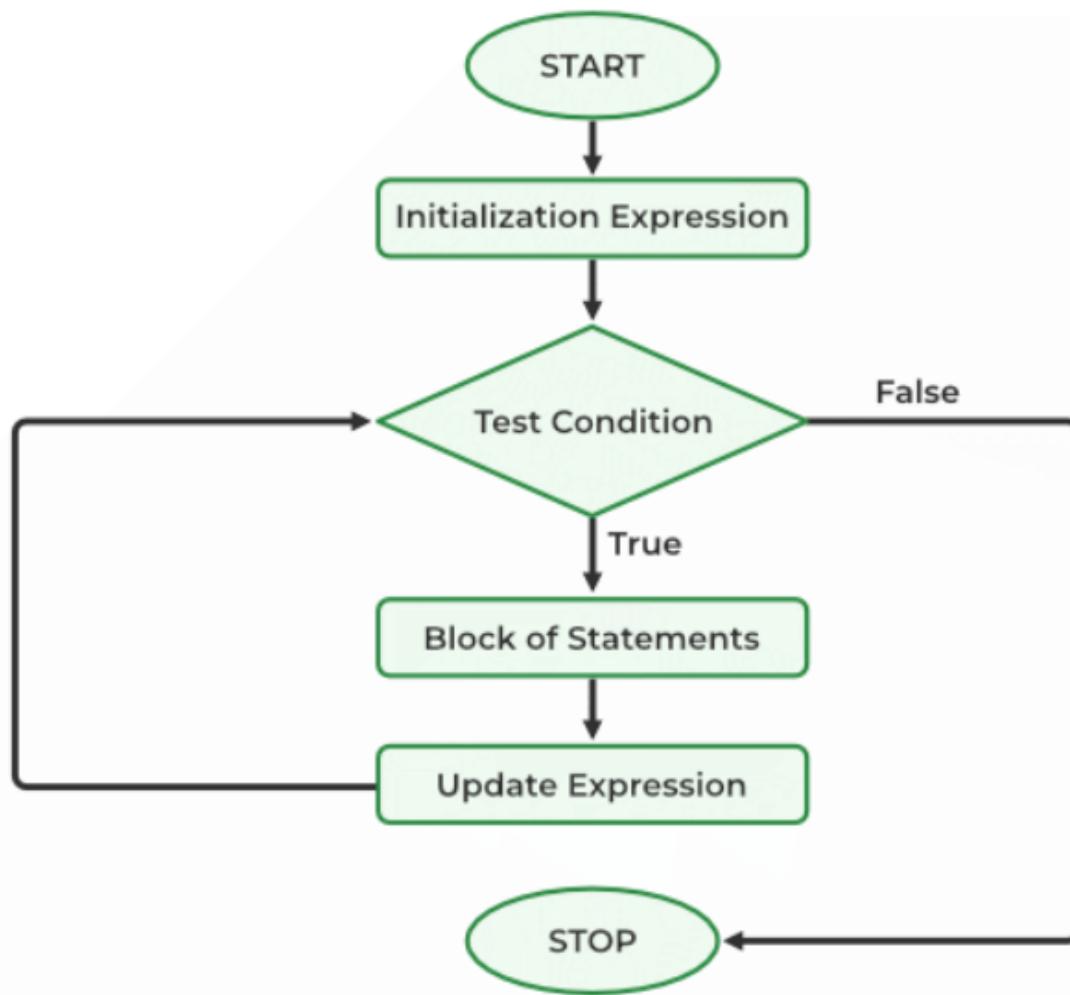
```
do  
{  <block>  
}while(e2);
```

e1,e2,e3 are expressions where e1: initialization, e2: condition, e3: modification



# PROBLEM SOLVING WITH C

## Control Structures



### Example Program:

```
#include <stdio.h>
int main()
{
    for(int i = 0; i < 5; i++)
    {
        printf("%d ", i);
    }
    return 0;
}
```

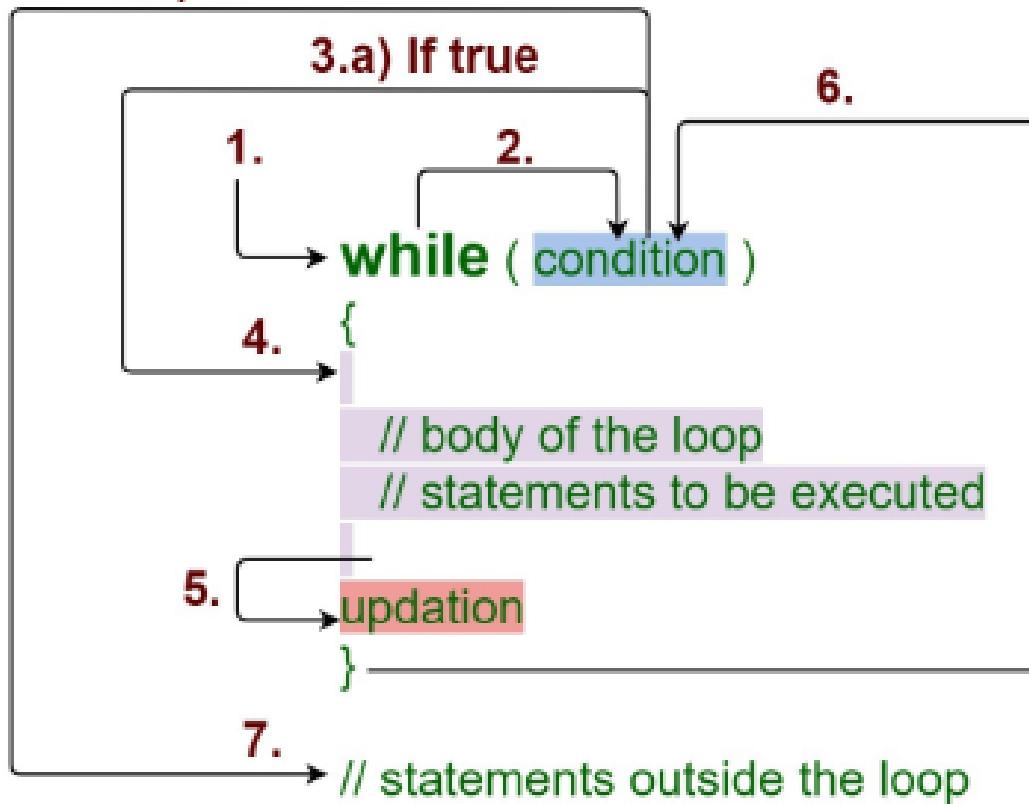
# PROBLEM SOLVING WITH C

## Control Structures



## While Loop

### 3.b) If false



### Example Program:

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

```
int main()
```

```
{
```

```
    int i = 5;  
    while(i-- > 0)
```

```
{
```

```
    printf("%d ", i);
```

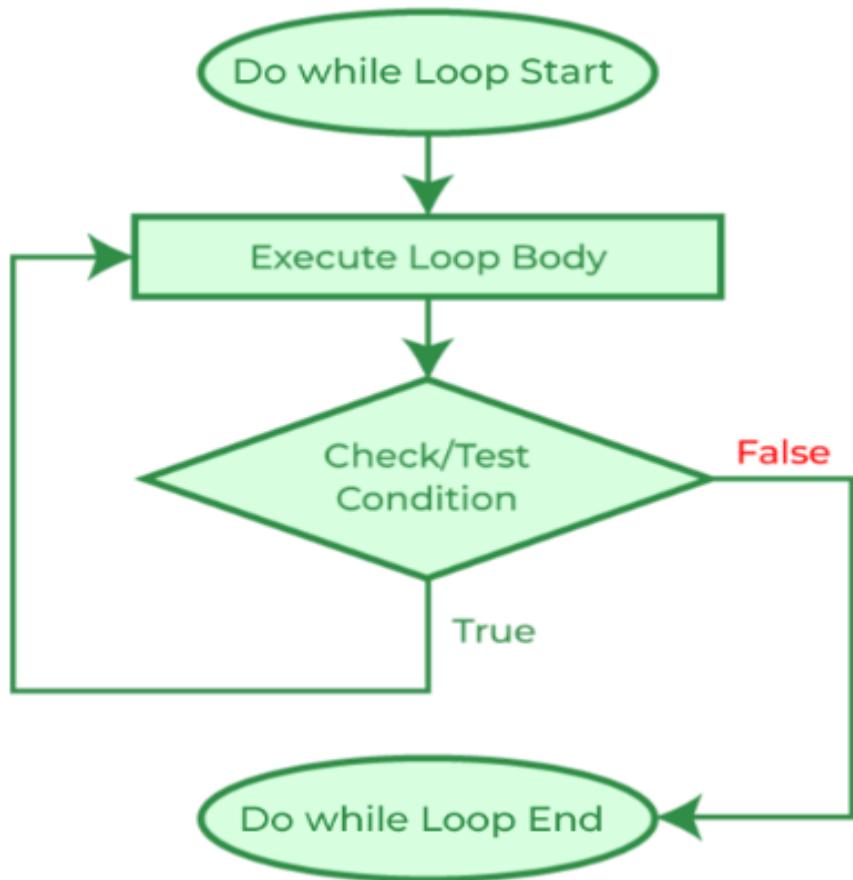
```
}
```

```
return 0;
```

```
}
```

# PROBLEM SOLVING WITH C

## Control Structures



### Example Program

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

```
int main()
```

```
{
```

```
    int i = 5;
```

```
    do
```

```
    {
```

```
        printf("%d ", i);
```

```
        i--;
```

```
    } while (i >= 0);
```

```
    return 0;
```

# PROBLEM SOLVING WITH C

## Control Structures

---



### Nested Control structures

- § One loop may be inside another
- § One if may be inside another: inner if is reached only if the Boolean condition of the outer if is true
- § Combination of above two
- § Coding Examples



**THANK YOU**

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering



# PROBLEM SOLVING WITH C

## UE23CS151B

---

**Prof. Sindhu R Pai**  
Department of Computer Science  
and Engineering

# PROBLEM SOLVING WITH C

---

## Variables and Types

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

## Variables and Data Types

---



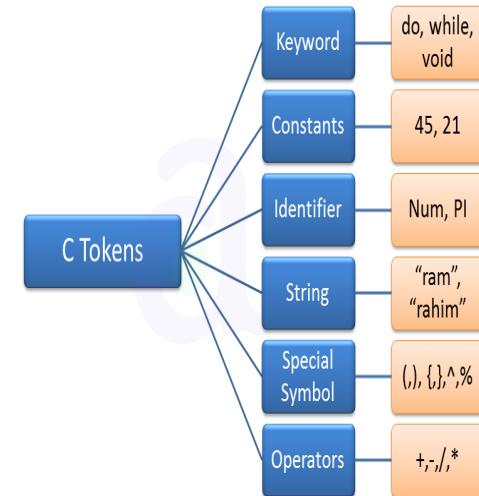
1. What are Tokens?
2. Identifiers
3. Variable declaration, definition and initialization
4. Keywords
5. Data types

# PROBLEM SOLVING WITH C

## Variables and Data Types

Tokens:

- A token or a lexical unit is the smallest individual unit of a program.
- Keywords: are nothing but reserved Words
- Identifiers: Names assigned to different entities of code.



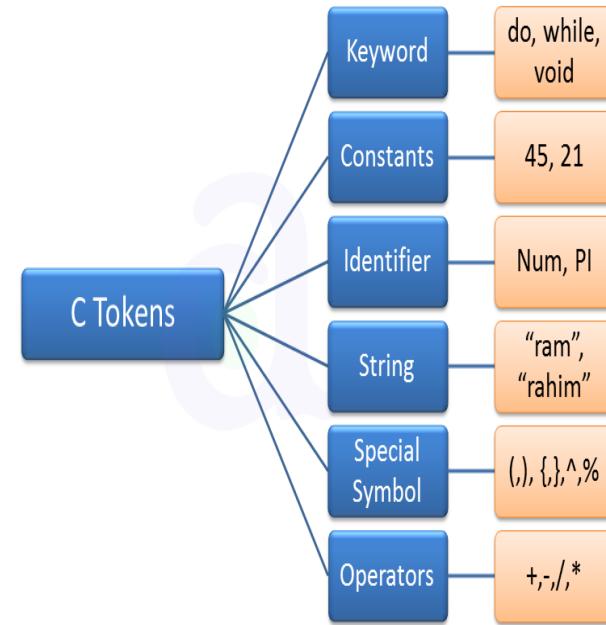
# PROBLEM SOLVING WITH C

## Variables and Data Types

§ Literals: Values assigned to Variables such as

numbers, strings, etc. Ex: 2.5

- Operators: Symbols that perform some operation on the operands.
- Punctuators: Symbols that are accepted by the compiler and is legal to be used in the program.
- Constants



# PROBLEM SOLVING WITH C

## Variables and Data Types

---



### Identifiers

- It is a name used to identify a variable, keyword, function, or any other user-defined item.
- Rules for naming a identifier
  - i. Starts with a letter A to Z, a to z, or an underscore ‘\_’ followed by zero or more letters, underscore, digits (0 to 9)
  - ii. An identifier cannot be keyword

# PROBLEM SOLVING WITH C

## Variables and Data Types

---

### Identifiers

- i. Except the first character identifier can contain digits
- ii. Characters other than underscore(\_) such as %,- cannot be part of identifier

Ex: of Legal Identifiers: Myname50, \_temp, j, a23b9, retVal

Ex: of Illegal Identifiers: 50ar, %name, auto, my name

### Variable declaration

- Variable is a name given to a storage area that a code can manipulate
- Has a name, location, type, life, scope and qualifiers
- An uninitialized variable will have some undefined value.
- Ex. : int a; int a=10, b=20;

# PROBLEM SOLVING WITH C

## Variables and Data Types

### Keywords

- Are identifiers which have special meaning in C.
- Cannot be used as constants or variables
- Ex: auto, else, long, switch, break, enum, case, extern, return char, float, for, void, sizeof, int auto = 10; // Error
- There are 32 keywords in C programming Language

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

# PROBLEM SOLVING WITH C

## Variables and Data Types

---



### Data Types

- Data types specify the type and range of values that can be stored in variable, operations that can be performed on it, amount of space taken by the variable.
- `Sizeof (short int)<= sizeof(int) <= sizeof(long int) <=sizeof(long long int) <= sizeof(float) <=sizeof(double) <=sizeof(long double)`
- There are mainly 3 types of Data Types in C
  - I. Primitive Data Types(`int,char,float,double`)
  - II. Derived Data Types(`Arrays`)
  - III. User Defined Data Types (`struct,union`)

# PROBLEM SOLVING WITH C

## Variables and Data Types

### FORMAT SPECIFIERS FOR DIFFERENT DATA TYPES

DATA TYPE	FORMAT SPECIFIER
int	%d
float	%f
char	%c
double	%lf
long double	%lf
string	%s
pointers	%p
Binary number	%b
Octal number	%o
Hexadecimal number	%x or %X

# PROBLEM SOLVING WITH C

## Variables and Data Types

### Example Program

```
#include<stdio.h>
int main()
{
    int a = 8;
    char p = 'c';
    double d = 89.7;
    int e = 0113;          // octal literal
    int h = 0xA;
    int i = 0b111;
    printf("%d\n",e);
    printf("%o\n",e);
    printf("%X\n",e);
    printf("%x\n",e);
    printf("%d",h);
    printf("%d",i);

    return 0;
}
```





**THANK YOU**

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering



# PROBLEM SOLVING WITH C

## UE23CS151B

---

**Prof. Sindhu R Pai**  
Department of Computer Science  
and Engineering



# PROBLEM SOLVING WITH C

---

## Single character Input and Output

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

## Single character input and output

---



- Single character I/O functions reads a single character from the keyboard and prints a character on the screen.
- A character represent characters in ASCII encoding scheme, including letters, numbers, symbols, etc.  
Ex: 'a', '6'
- **Range:** -128 to 127 or 0 to 255
- **Size:** 1 byte
- **Format Specifier:** %c

# PROBLEM SOLVING WITH C

## Single character Input Output

---



Approaches:

### I. Standard Input Output Functions:

The standard library provides several functions for reading or writing one character at a time.

1. `scanf()` and `printf ()`
2. `getchar()` and `putchar ()`
3. `getc()` and `putc ()`

### II. Non- standard Input Output Functions:

Functions are called as non-standard functions since they are not part of standard library. They are available in **conio.h** header file.

1. `getch()`
2. `getche()`
3. `putch()`

# PROBLEM SOLVING WITH C

## Single character Input Output

---



### I. Standard Input Output Functions:

#### 1. **scanf() and printf() :**

The %c conversion specification allows scanf and printf to read and write single characters:

**Ex :** `scanf("%c", &ch);`

- It reads a single character from the input and stores it in the variable ch.
- Returns the number of successfully read items.

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

- prints a single character from on the screen.
- Returns the number of characters written

# PROBLEM SOLVING WITH C

## Single character Input Output

---



Unformatted I/O functions :

These are the console I/O standard library functions defined inside the `<stdio.h>` header file.

**getchar ( ):**

Used to read a single character from the standard input  
getchar() function does not take any parameters.

**Syntax:**    `int getchar(void);`

On success: The input from the standard input is read as an unsigned char and then it is typecast and returned as an integer value(int).

On failure: EOF

EOF is returned in two cases:

- Ø When the file end is reached
- Ø When there is an error during the execution

# PROBLEM SOLVING WITH C

## Single character Input Output

---



### **putchar ( ):**

Used to write an unsigned char type, to stdout.(Screen)

#### •**Syntax:**

§ int putchar(int ch);

§ This method accepts a mandatory parameter **ch** which is the character to be written to stdout.

#### **Return Value:**

§ It returns the character written on the stdout as an unsigned char.

§ It also returns EOF when some error occurs.

# PROBLEM SOLVING WITH C

## Simple Input/Output

---

### Example program

```
include <stdio.h>
int main()
{
    char c;
    c=getchar();
    putchar(c);
    return 0;
}
```



# PROBLEM SOLVING WITH C

## Single character Input Output

---

### 3. Unformatted I/O functions - getc and putc

C treats all the devices as files. Devices such as the display are addressed in the same way as files and the following three files are automatically opened when a program executes to provide access to the keyboard and screen.

Standard File Device	File Pointer	Device
Standard input	stdin	Keyboard
Standard output	stdout	Screen
Standard error	stderr	Screen

Eg. `char ch = getc(stdin)`  
`putc(ch, stdout);`

`// Read input from keyboard`  
`// display on monitor`

# PROBLEM SOLVING WITH C

## Simple Input/Output

---

Example Program:

```
#include<stdio.h>

int main ()
{
    char c;
    printf("Enter character: ");
    c = getc(stdin);
    printf("Character entered: ");
    putc(c, stdout);
    return 0;
}
```



# PROBLEM SOLVING WITH C

## Single character Input Output

---



### II. Non- standard Input Output Functions:

getch() , getche() and putch() functions are called as non-standard functions since they are not part of standard library and ISO C(certified c programming language). It is available in **conio.h**.

#### 1. getch():

getch() also reads a single character from the keyboard.

**Syntax:** int getch();

**Return value:** This method returns the ASCII value of the key pressed.

- getch() method pauses the Output Console until a key is pressed.
- It does not use any buffer to store the input character.
- The entered character is immediately returned without waiting for the enter key.
- The entered character does not show up on the console.

# PROBLEM SOLVING WITH C

## Single character Input Output

### 2. **getche():**

getche() function reads a single character from the keyboard and displays immediately on the output screen without waiting for enter key.

**Syntax:** int getche();

**Return value:** Returns the character read as an unsigned char cast to an int or EOF (End-of-File) on an error.

### 3. **putch():**

putch function outputs a character stored in the memory using a variable on the output screen.

**Syntax:**

putch(ch);



THANK YOU

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering



# PROBLEM SOLVING WITH C

## UE23CS151B

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# **PROBLEM SOLVING WITH C**

---

## **Language Specifications/Behaviors**

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

## Language Specifications/Behaviors

---



1. Language Specification
2. Standards
3. Behaviors defined by C Standards
4. Undefined Behavior

# PROBLEM SOLVING WITH C

## Language Specifications/Behaviors

---

### Language Specification

A Programming language specification is a document which describes how a system is supposed to work.

It is a documentation that defines a programming language so that users and implementers can agree on what programs in that language mean.

In case of a programming language, it describes how the compiler or interpreter must react to various inputs - which are valid/invalid

Typically it is a detailed and formal document, and primarily used by implementers referring to them in case of ambiguity



# PROBLEM SOLVING WITH C

## Language Specifications/Behaviors

---



### Behaviors defined by C Standards

- Locale-specific behavior - Not discussed here
- Unspecified behavior - Order of evaluation of arguments in printf function
- Implementation-defined behavior – Size of each type
- Undefined behavior

# PROBLEM SOLVING WITH C

## Language Specifications/Behaviors

---



### Unspecified behaviour

Unspecified behaviour is the behaviour that is not defined by the language but it is left to the implementation. This means that the behaviour may vary between different compilers, platforms or versions of the language.

A common examples of such a behaviour is

**Ex. :** #include <stdio.h>

```
int main() {  
    int d = 10;  
    printf("%d %d %d\n", d++, d++ - --d,--d);  
    return 0;  
}
```

# PROBLEM SOLVING WITH C

## Language Specifications/Behaviors

---



### Implementation-defined behaviour

Implementation-defined behaviour is behaviour that is specified by the language and left to the implementation to define.

This means that the behaviour is consistent within a particular implementation, but may vary between different implementations.

A common example of implementation-defined behaviour is the **size of data types**.

For Ex.

```
printf("Size of int: %d\nSize of char:%d\n", sizeof(int), sizeof(char));
```

# PROBLEM SOLVING WITH C

## Language Specifications/Behaviors

---



### Undefined Behavior in detail

The result of executing a program whose behavior is unpredictable in the language specification to which the computer code adheres.

Standard imposes no requirements: May fail to compile, may crash, may generate incorrect results, may fortunately do what exactly the programmer intended

# PROBLEM SOLVING WITH C

## Language Specifications/Behaviors

---



Some of the undefined behaviour are :

1. Division by zero

Ex. : `printf("%d\n", 5 / 0);`

2. Signed integer overflow

Ex. : `printf("Size of int: %d\n", INT_MAX + 1);`

3. Null pointer dereference

Ex. : `int *ptr = NULL;  
printf("%d", *ptr);`

4. Memory access outside of array bounds

Ex.: `int arr[5] = {0, 1, 2, 3, 4};  
printf("%d", arr[7])`

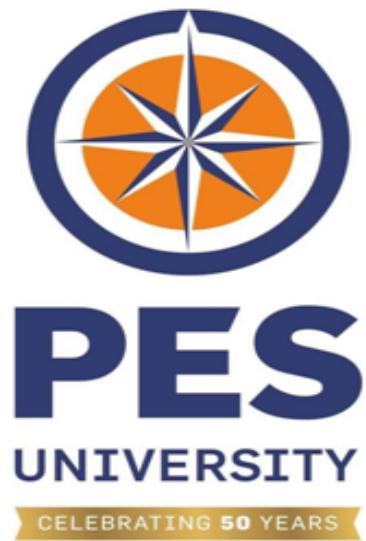


**THANK YOU**

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering



# PROBLEM SOLVING WITH C

## UE23CS151B

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering



# PROBLEM SOLVING WITH C

---

## Counting Problem

Prof. Sindhu R Pai

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

## Counting Problem

---



1. Statement of the Problem
2. Requirements to solve the problem
3. Demonstration of the C solution

# PROBLEM SOLVING WITH C

## Counting Problem

---



### 1. Example Problem

#### a. Statement of the problem

- Write a program to find the frequency of occurrence of a particular character and display the count in a proper output format.

#### b. Requirements

- user needs to give a line of text
- user needs to give a particular character as input (to search)
- Iterating till EOF
- counting (if matching ) and display the count

#### Expected output

```
Output
/tmp/xaaY01TFVv.o
Enter the string : Pes University
Enter character to be searched: i
character 'i' occurs 2 times
|
```

# PROBLEM SOLVING WITH C

## Counting Problem

---

```
#include <stdio.h>
int main()
{
    char ch, char_to_search;
    int ctr=0;
    printf("Enter the character to be searched");
    char_to_search=getchar();
    printf("Enter the line of text : ");
    while((ch=getchar() )!=EOF)
    {
        if (ch==char_to_search)
            ctr++;
    }
    printf("%c occurs %d times in the given text", char_to_search,ctr);
    return 0; }
```



# PROBLEM SOLVING WITH C

## Counting Problem

---



### 2. Example Problem

#### a. Statement of the problem

- Write a program to find the number of characters , words and lines present in a given lines if text. Display the count of each in a new line with proper message .

#### b. Requirements

To solve this problem stated, we should know

How to read and display a character?-Discussed in Character I/O

How to read a line?–Discussed in Character I/O

How to make out when we reach the end of file?

How to break a given sequence of characters into words and lines?

# PROBLEM SOLVING WITH C

## Counting Problem

---

### c. Solution to the problem

```
#include<stdio.h>
int main()
{
    int nl,nw,nc,inword = 0;
    char ch;
    while((ch = getchar()) != EOF)
    {
        //putchar(ch);
        nc++;
        if(ch == '\n')
            nl++;
    }
}
```



# PROBLEM SOLVING WITH C

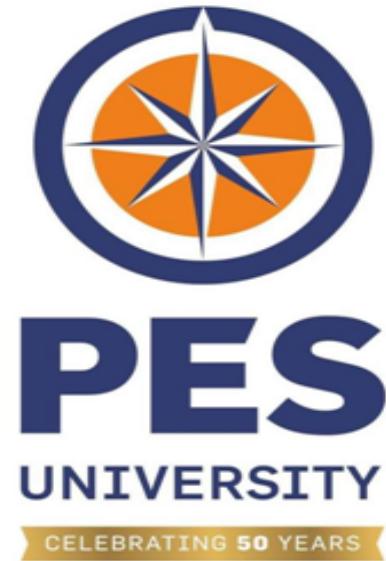
## Counting Problem

---

### c. Solution to the problem(contd..)

```
if(inword && (ch == ' ' || ch == '\n' || ch == '\t'))  
    {  
        nw++;  
        inword = 0;  
    }  
else if(!(ch == ' ' || ch == '\n' || ch == '\t'))  
    {  
        inword = 1;  
    }  
printf("words = %d\nlines = %d\ncharacters=%d\n",nw,nl,nc);  
return 0;  
}
```





**THANK YOU**

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering



# PROBLEM SOLVING WITH C

## UE23CS151B

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

---

## Functions in C

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

## Functions

---



1. Introduction to functions
2. Types of Functions
3. Function Definition, Call and Declaration

# PROBLEM SOLVING WITH C

## Functions

---



### Introduction

- A sub-program to carry out a specific task
- Functions break large computing tasks into smaller ones.
- Enable people to build on what others have done instead of starting from scratch
- **Benefits:**
  - Reduced Coding Time – Code Reusability
  - Divide and Conquer - Manageable Program development
  - Reduced Debugging time
  - Treated as a black box - The inner details of operation are invisible to rest of the program

# PROBLEM SOLVING WITH C

## Functions

---



### Types of Functions

- **Standard Library Functions**

Must Include appropriate header files to use these functions.  
Already declared and defined in C libraries  
`printf`, `scanf`, etc..

- **User Defined functions**

Defined by the developer at the time of writing program  
Developer can make changes in the implementation

# PROBLEM SOLVING WITH C

## Functions

---



### Function Definition, Call and Declaration

#### Function Definition

- Provides actual body of the function
- Function definition format

```
return-type function-name( parameter-list )
{
    declarations and statements
}
```
- Variables can be declared inside blocks
- Coding examples

# PROBLEM SOLVING WITH C

## Functions

---



## Function Definition, Call and Declaration

### Function Call

- Function can be called by using function name followed by list of arguments (if any) enclosed in parentheses
- Function call format  
**function-name(list of arguments);**
- The arguments must match the parameters in the function definition in its type, order and number.
- Multiple arguments must be separated by comma. Arguments can be any expression in C
- Coding examples

# PROBLEM SOLVING WITH C

## Functions

---



### Function Definition, Call and Declaration

#### Function Declaration/ Prototype

- All functions must be declared before they are invoked or called.
- Function can be declared by using function name followed by list of parameters (if any) enclosed in parentheses
- Function declaration format  
**return\_type function\_name (parameters list);**
- Use of identifiers in the declaration is optional.
- The parameter names do not need to be the same in declaration and the function definition.
- The types must match the type of parameters in the function definition in number and order.
- Coding examples



# THANK YOU

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

[sindhurpai@pes.edu](mailto:sindhurpai@pes.edu)



# PROBLEM SOLVING WITH C

## UE23CS151B

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering



# PROBLEM SOLVING WITH C

---

## Arrays - Initialization and Traversal Pointers

Prof. Sindhu R Pai

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

## Arrays: Initialization and Traversal, Pointers:

---



- Array
  - 1. What is an Array?
  - 2. Properties of Arrays.
  - 3. Classification of Arrays
  - 4. Declaration and Initialization
  - 5. Representation of the Array
  - 6. Array Traversal
- Pointer
  - 1. What is a Pointer?
  - 2. Declaration and Initialization
  - 3. Arithmetic operations on Pointer
  - 4. Array Traversal using pointers
  - 5. Array and Pointer

# PROBLEM SOLVING WITH C

## Arrays, Initialization and Traversal

---



### What is an Array?

- A linear data structure, which is a Finite collection of similar data items stored in successive or consecutive memory locations
- **Only homogenous types of data** is allowed in any array. May contain all integer or all character elements, but not both together.
- `char c_array [10]; short s_array [20];`

# PROBLEM SOLVING WITH C

## Arrays, Initialization and Traversal

---



### Properties of Arrays

- Non-primary data type or secondary data type
- Memory allocation is contiguous in nature
- Elements need not be unique.
- Demands same /homogenous types of elements
- Random access of elements in array is possible
- Elements are accessed using index/subscript which starts from 0
- Memory is allocated at compile time.
- Size of the array is fixed at compile time and cannot be changed at runtime.  
Returns the number of bytes occupied by the array.
- Arrays are assignment incompatible.
- Accessing elements of the array outside the bound can have undefined behavior at runtime

# PROBLEM SOLVING WITH C

## Arrays, Initialization and Traversal

---



### Classification of Arrays

#### Category 1:

- Fixed Length Array
  - Size of the array is fixed at compile time
- Variable Length Array – Not discussed here

#### Category 2:

- One Dimensional (1-D) Array
  - Stores the data elements in a single row or column.
- Multi Dimensional Array
  - More than one row and column is used to store the data elements

# PROBLEM SOLVING WITH C

## Arrays, Initialization and Traversal

---



### Declaration and Initialization

**Declaration:** `Data_type Array_name[Size];`

`Data_type`: Specifies the type of the element that will be contained in the array

`Array_name`: Identifier to identify a variable as an array

`Size`: Indicates the max no. of elements that can be stored inside the array

**Example:** `double x[15]; // Can contain 15 elements of type double, 0 to 14 are valid array indices or subscript`

- Subscripts in array can be integer constant or integer variable or expression that yields integer
- C performs no bound checking. Care should be taken to ensure that the array indices are within the declared limits

# PROBLEM SOLVING WITH C

## Arrays, Initialization and Traversal

---



### Declaration and Initialization

- After an array is declared, it must be initialized.
- An array can be initialized at either **compile time** or at **runtime**.

#### Compile time Initialization

- data-type array-name[size] = { list of values };
- float area[5]={ 23.4, 6.8, 5.5 }; // Partial initialization
- int a[15] = {[2] = 29, [9] = 7, [14] = 48}; //C99's designated initializers
- int a[15] = {0,0,29,0,0,0,0,0,0,7,0,0,0,0,48}; //C99's designated initializers
- int arr[] = {2, 3, 4}; // sizeof arr is decided
- int marks[4]={ 67, 87, 56, 77, 59 }; // undefined behavior
- Coding Examples

#### Runtime Initialization

- Using a loop and input function in C
- Coding examples

# PROBLEM SOLVING WITH C

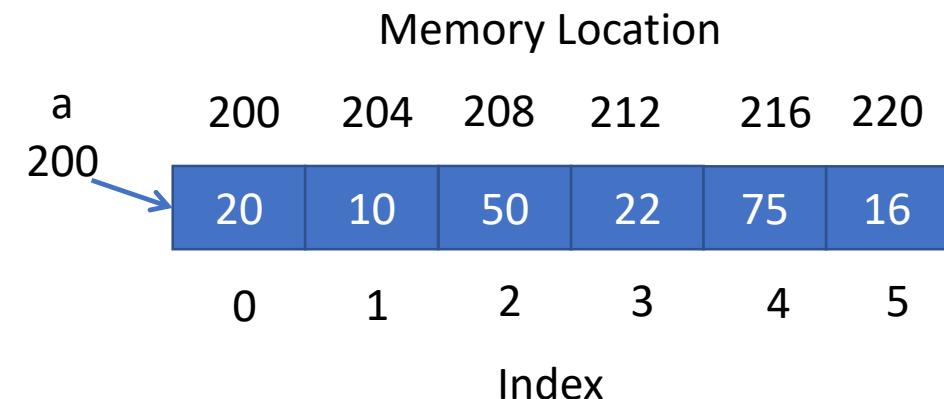
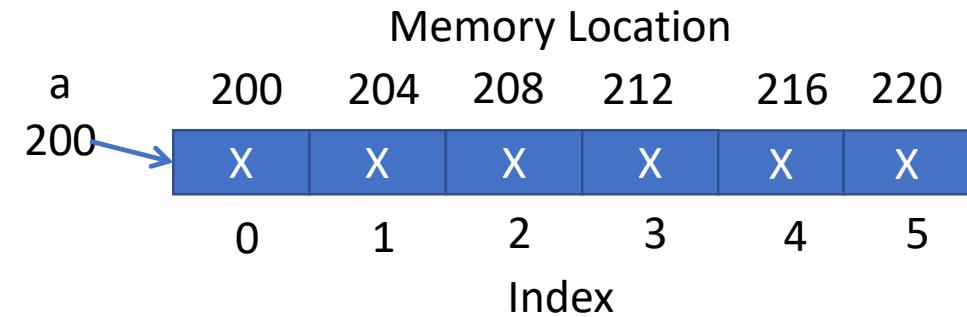
## Arrays, Initialization and Traversal



### Representation of the Array

- int a[6];
- int a[6] = {20, 10, 50, 22, 75, 16};
- Address of the first element is called the Base address of the array.
- Address of ith element of the array can be found using formula:

**Address of ith element = Base address + (size of each element \* i)**



# PROBLEM SOLVING WITH C

## Arrays, Initialization and Traversal

---



### Traversal

- Nothing but accessing each element of the array
- int a[10];
  - How do you access the 5th element of the array? // a[4]
  - How do you display each element of the array? // Using Loop
  - How much memory allocated for this?  
**Number of bytes allocated = size specified\*size of integer**
- Anytime accessing elements outside the array bound is an undefined behavior
- Coding examples

# PROBLEM SOLVING WITH C

## Pointers

---



### What is a Pointer?

- A variable which contains the address. This address is the location of another object in the memory
- Used to access and manipulate data stored in memory.
- Pointer of particular type can point to address of any value in that particular type.
- Size of pointer of any type is same/constant in that system
- Not all pointers actually contain an address  
Example: NULL pointer // Value of NULL pointer is 0.
- Pointer can have three kinds of contents in it
  - The address of an object, which can be dereferenced.
  - A NULL pointer
  - Undefined value // If p is a pointer to integer, then – int \*p;

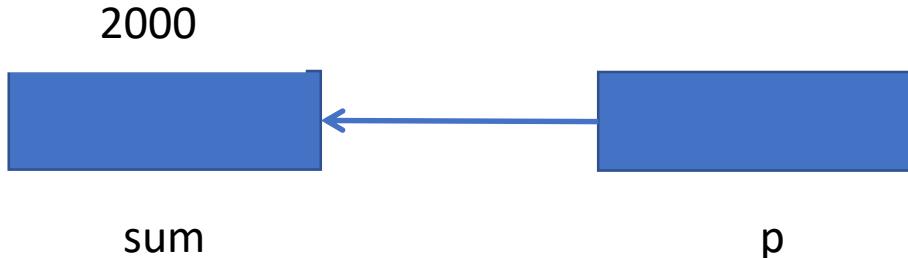
### Declaration and Initialization

**Declaration :** Data-type \*name;

- ```
int *p;
```

  - Compiler assumes that any address that it holds points to an integer type.
- ```
p= &sum;
```

  - Memory address of sum variable is stored into p.



**Example code:**

```
int *p; // p can point to anything where integer is stored. int* is the type. Not just int.
```

```
int a = 100;
```

```
p=&a;
```

```
printf("a is %d and *p is %d", a,*p);
```



# PROBLEM SOLVING WITH C

## Pointers

---



### Pointer Arithmetic Operations

1. Add an int to a pointer
2. Subtract an int from a pointer
3. Difference of two pointers when they point to the same array.

**Note: Integer is not same as pointer.**

- Coding examples

# PROBLEM SOLVING WITH C

## Pointers



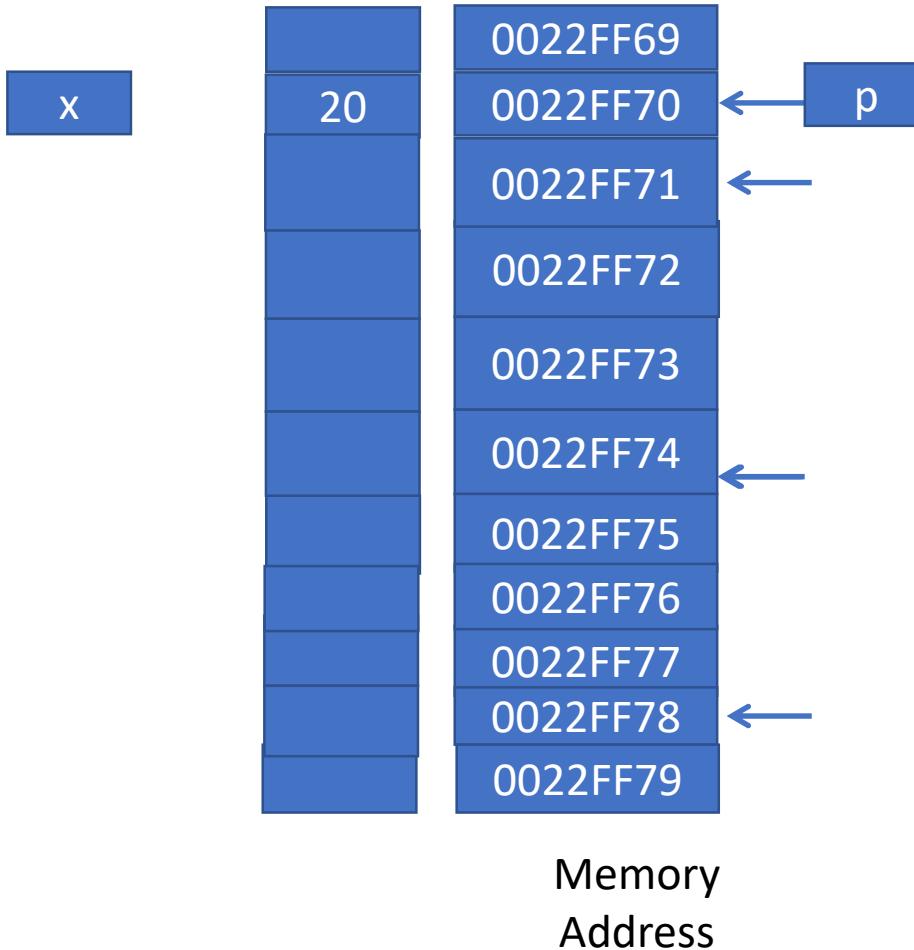
### Pointer Arithmetic Operations continued..

#### Example Code:

```
int *p, x = 20;  
p = &x;  
printf("p      = %p\n", p);  
printf("p+1 = %p\n", (int*)p+1);  
printf("p+1 = %p\n", (char*)p+1);  
printf("p+1 = %p\n", (float*)p+1);  
printf("p+1 = %p\n", (double*)p+1);
```

#### Sample output:

```
p      = 0022FF70  
p+1 = 0022FF74  
p+1 = 0022FF71  
p+1 = 0022FF74  
p+1 = 0022FF78
```



# PROBLEM SOLVING WITH C

## Pointers

---



### Array Traversal using Pointers

Consider int arr[] ={12,44,22,33,55};    int \*p = arr;    int i;

Coding examples to demo below points

- Array notation. Index operator can be applied on pointer.
- Pointer notation
- Using `*p++`
- Using `*++p`, Undefined behavior if you try to access outside bound
- Using `(*p)++`
- Using `*p` and then `p++`

### Array and Pointer

- An array during compile time is an actual array but degenerates to a constant pointer during run time.
- Size of the array returns the number of bytes occupied by the array. But the size of pointer is always constant in that particular system.

```
int *p1; float *f1 ; char *c1;
printf("%d %d %d ",sizeof(p1),sizeof(f1),sizeof(c1)); // Same value for all
```
- An array is a constant pointer. It cannot point to anything in the world



# PROBLEM SOLVING WITH C

## Pointers

---

### Array and Pointer continued..

#### Example code:

- int a[] = {22,11,44,5};
- int \*p = a;
- a++; // Error : a is constant pointer
- p++; // Fine
- p[1] = 222;
- a[1] = 222 ; // Fine
  
- If variable i is used in loop for the traversal, a[i], \*(a+i), p[i], \*(p+i), i[a], i[p] are all same.



# PROBLEM SOLVING WITH C

## Pointers

### Array and Pointer continued..

---

#### Differences

1. the sizeof operator
  - a. sizeof(array) returns the amount of memory used by all elements in array
  - b. sizeof(pointer) only returns the amount of memory used by the pointer variable itself
2. the & operator
  - a. &array is an alias for &array[0] and returns the address of the first element in array
  - b. &pointer returns the address of pointer
3. string literal initialization of a character array
  - a. char array[] = "abc" sets the first four elements in array to 'a', 'b', 'c', and '\0'
  - b. char \*pointer = "abc" sets pointer to the address of the "abc" string (which may be stored in read-only memory and thus unchangeable)
4. Pointer variable can be assigned a value whereas array variable cannot be.
5. Arithmetic operations on pointer variable is allowed. On array, not allowed.





# THANK YOU

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

[sindhurpai@pes.edu](mailto:sindhurpai@pes.edu)



# PROBLEM SOLVING WITH C

## UE23CS151B

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering



# PROBLEM SOLVING WITH C

---

## Arrays and Functions

**Prof. Sindhu R pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

## Arrays and Functions

---



1. Passing an Array to a function
2. Array as a Formal parameter and actual parameter
3. Pointer as a Formal parameter and Array as an Actual parameter
4. Passing individual array element to a function

# PROBLEM SOLVING WITH C

## Arrays and Functions

---



### Passing an Array to a function

- When array is passed as an argument to a function, arguments are copied to parameters of the function and parameters are always pointers.
- Array degenerates to a pointer at runtime.
- All the operations that are valid for pointer will be applicable for array too in the body of the function.
- Function call happens always at run time.

# PROBLEM SOLVING WITH C

## Arrays and Functions

---



### Array as a formal parameter and Actual parameter

- **Array being formal parameter** - Indicated using empty brackets in the parameter list.

```
void myfun(int a[],int size);
```

- **Array being actual parameter** – Indicated using the name of the array

Array a is declared as int a[5];

Then myfun is called as myfun(a,n);

- Coding example to read and display the array elements

# PROBLEM SOLVING WITH C

## Arrays and Functions

---

### Pointer as a formal parameter and Array as an Actual parameter



- **Pointer being formal parameter** - Indicated using empty brackets in the parameter list.

```
void myfun(int *a,int size);
```

- **Array being actual parameter** – Indicated using the name of the array

Array a is declared as int a[5];

Then myfun is called as myfun(a,n);

- Coding example to read and display the array elements

# PROBLEM SOLVING WITH C

## Arrays and Functions

---



### Passing individual Array elements to a function

- Indexed variables can be arguments to functions
- Program contains these declarations: int a[10]; int i; void myfunc(int n);
- Variables a[0] through a[9] are of type int, making below calls is legal
  - myfunc(a[0]);
  - myfunc(a[3]);
  - myfunc(a[i]); // i is between 0 and 9



# THANK YOU

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

[sindhurpai@pes.edu](mailto:sindhurpai@pes.edu)



# PROBLEM SOLVING WITH C

## UE23CS151B

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering



# PROBLEM SOLVING WITH C

## Multi-Dimensional Arrays

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

## Agenda

---

1. Introduction
2. Two-Dimensional Array: Declaration
3. Two Dimensional Array: Initialization
4. Internal Representation of a 2D Array
5. Pointer and 2D Array
6. Passing 2D array to a function
7. Three Dimensional (3D) Array



# PROBLEM SOLVING WITH C

## Multi-Dimensional Arrays

---



### Introduction

- An array with more than one level or dimension.
- 2-Dimensional and 3-Dimensional and so on.

General form of declaring N-dimensional arrays:

**Data\_type Array\_name[size1][size2][size3]..[sizeN];**

# PROBLEM SOLVING WITH C

## Multi-Dimensional Arrays



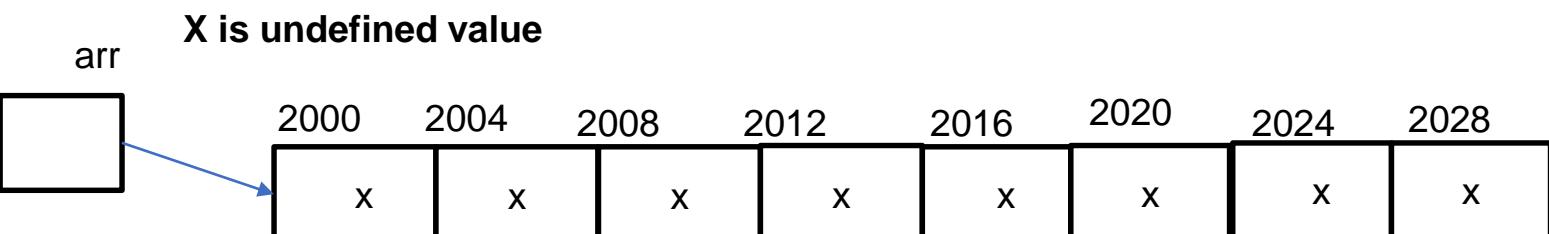
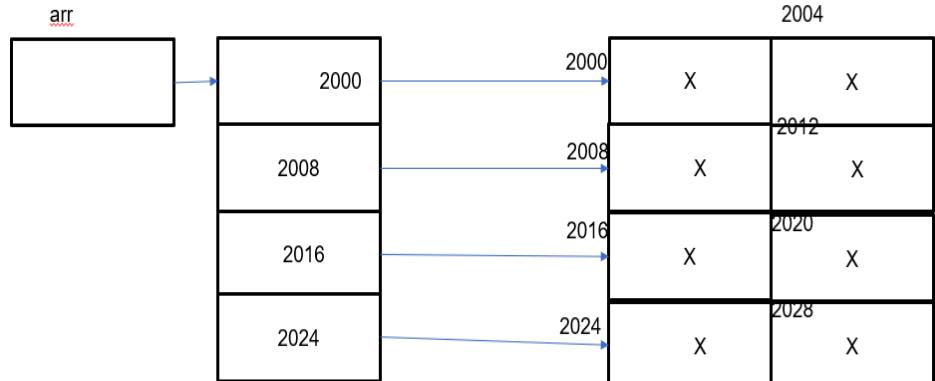
### Two – Dimensional Array

- Treated as an array of arrays.
- Every element of the array must be of same type as arrays are homogeneous

### Declaration

**Syntax:** `data_type array_name[size_1][size_2]; // size_1 and size_2 compulsory`

```
int arr[4][2];
```



# PROBLEM SOLVING WITH C

## Multi-Dimensional Arrays

---



### Two – Dimensional Array

#### Initialization

- **data\_type array\_name[size\_1][size\_2] = {elements separated by comma};**
- int arr[][] = {11,22,33,44,55,66}; // Error. Column size is compulsory
- int abc[3][2] ={{11,22},{33,44},{55,66}};//valid
- int arr[][2] = {11, 22, 33 ,44,55,66 } ;//valid. Allocates 6 contiguous memory locations and assign the values
- int arr[][3] = {{11,22,33},{44,55},{66,77}}; // partial initialization

# PROBLEM SOLVING WITH C

# Multi-Dimensional Arrays

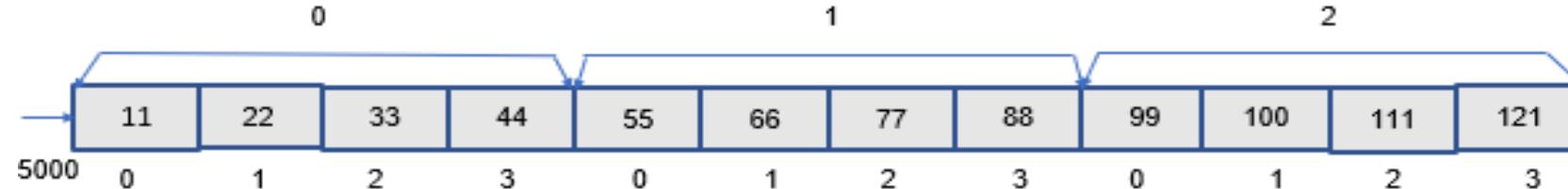


# Internal Representation of a 2D Array

- 2D Array itself is an array, elements are stored in contiguous memory locations.

Consider, int arr[3][4] = {{11, 22, 33, 44}, {55, 66, 77, 88}, {99, 100, 111, 121}};

- **Row major Ordering:** All elements of one row are followed by all elements of the next row and so on.



- **Column Major Ordering:** All elements of one column are followed by all elements of the next column and so on.



- Generally, systems support Row Major Ordering.

# PROBLEM SOLVING WITH C

## Multi-Dimensional Arrays



### Address of an Element in a 2D Array

- **Address of  $A[i][j] = \text{Base\_Address} + ( (i * \text{No. of columns in every row}) + j) * \text{size of every element};$**
- Consider, int matrix[2][3]={1,2,3,4,5,6}; // base address is 100 and size of integer is 4 bytes

matrix[0][0]	100	1
matrix[0][1]	104	2
matrix[0][2]	108	3
matrix[1][0]	112	4
matrix[1][1]	116	5
matrix[1][2]	120	6

		Column
		0      1      2
Row	0	1      2      3
	1	4      5      6

- Address of matrix[1][0]=100+((1\*3)+0)\*4=100+3\*4=112

# PROBLEM SOLVING WITH C

## Multi-Dimensional Arrays

---

### Demo of C Code

- To read and display a 2D Array



# PROBLEM SOLVING WITH C

## Multi-Dimensional Arrays

---



### Pointer and 2D Array

- **Pointer expression for  $a[i][j]$  is  $\ast(\ast(a + i) + j)$**
- Array name is a pointer to a row.
- $(a + i)$  points to ith 1-D array.
- $\ast(a + i)$  points to the first element of ith 1D array
- Coding examples

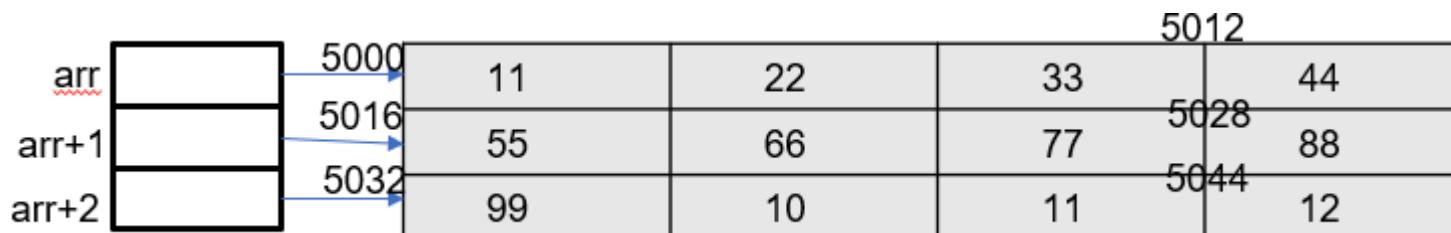
# PROBLEM SOLVING WITH C

## Multi-Dimensional Arrays



### Pointer and 2D Array continued..

- Consider, `int arr[3][4] = {11, 22, 33, 44, 55, 66, 77, 88, 99, 100, 111, 121};`  
arr – points to 0th elements of arr- Points to 0th 1-D array-5000  
arr+1-Points to 1st element of arr-Points to 1st 1-D array-5016  
arr+2-Points to 2nd element of arr-Points to 2nd 1-D array  
arr+ i Points to ith element of arr ->Points to ith 1-D array



# PROBLEM SOLVING WITH C

## Multi-Dimensional Arrays

---



### Pointer and 2D Array continued..

- `int *p = arr;` // assigning the 2D array to a pointer results in warning
- Using `p[5]` results in 66. But `p[1][1]` results in error. `p` doesn't know the size of the column.
- Solution is to create a **pointer to an array of integers**.

`int (*p)[4] = arr;` //subscript([ ]) have higher precedence than indirection(\*)

- Think about the **size of p and size of \*p!!**

# PROBLEM SOLVING WITH C

## Multi-Dimensional Arrays

---



### Passing 2D array to a function

- Read and display 2D array using functions
- Write a function to add, subtract and multiply two matrices. Display appropriate message when these two matrices are not compatible for these operations.
- Write a program to take n names from the user and print it. Each name can have maximum of 20 characters.

# PROBLEM SOLVING WITH C

## Multi-Dimensional Arrays

---



### Three Dimensional (3D) Array

- Accessing each element by using three subscripts
- First dimension represents table ,2nd dimension represents number of rows and 3rd dimension represents the number of columns
- `int arr[2][3][2] = { {{5, 10}, {6, 11}, {7, 12}}, {{20, 30}, {21, 31}, {22, 32}} };` //2 table 3 rows 2 coloumns.

# PROBLEM SOLVING WITH C

## Multidimensional Arrays

---



### Practice programs

1. Given two matrices, write a function to find whether these two are identical.
2. Program to find the transpose of a given matrix.
3. Program to find the inverse of a given matrix.
4. Write a function to check whether the given matrix is identity matrix or not.
5. Write a program in C to find sum of right diagonals of a matrix.
6. Write a program in C to find sum of rows and columns of a matrix



# THANK YOU

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

[sindhurpai@pes.edu](mailto:sindhurpai@pes.edu)



# PROBLEM SOLVING WITH C

## UE23CS151B

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering



# PROBLEM SOLVING WITH C

---

## Recursion

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

## Recursion

---



1. Introduction
2. Why Recursion?
3. Implementing Recursion – The Stack
4. Arguments and Return Values
5. Practice Programs

### Introduction

### Recursive Function

- A function that calls itself Directly or indirectly
- Each recursive call is made with a new, independent set of arguments
  - Previous calls are suspended
- Allows building simple solutions for complex problems

# PROBLEM SOLVING WITH C

## Recursion

---



### Why Recursion?

- Used to solve various problems by dividing it into smaller problems
- Some problems are *too hard* to solve without recursion
  - Most notably, the compiler!
  - Most problems involving linked lists and trees

# PROBLEM SOLVING WITH C

## Recursion

---



### Points to note while using Recursion

- The problem is broken down into smaller tasks
- Programmers need to be careful to define an exit condition from the function, otherwise results in an infinite recursion
- The exit condition is defined by the ***base case*** and the solution to the base case is provided
- The solution of the bigger problem is expressed in terms of smaller problems called as ***recursive relationship***

# PROBLEM SOLVING WITH C

## Recursion

---



### How is a particular problem solved using recursion?

- **Problem to be solved:** To compute factorial of n.
  - knowledge of factorial of  $(n-1)$  is required, which would form the recursive relationship
  - The base case for factorial would be  $n = 0$
  - return 1 when  $n == 0$  or  $n == 1$  // base case
  - return  $n * (n-1)!$  when  $n != 0$  // recursive relationship
  - Demo of C code

### Implementing Recursion - The Stack

- **Definition – *The Stack***
  - A **last-in, first-out** data structure provided by the operating system for running each program
  - For temporary storage of automatic variables, arguments, function results, and other information
  - The storage for each function call.
  - Every single time a function is called, an area of the stack is reserved for that particular call.
- Known as ***activation record***, similar to that in python

# PROBLEM SOLVING WITH C

## Recursion

---



### Implementing Recursion - The Stack continued..

- Parameters, results, and automatic variables allocated on the stack.
- Allocated when function or compound statement is entered
- Released when function or compound statement is exited
- Values are not retained from one call to next (or among recursions)

# PROBLEM SOLVING WITH C

## Recursion

---



### Arguments and Return values

1. Space for storing result is allocated by caller
  - On The Stack
  - Assigned by **return** statement of function
  - For use by caller
2. Arguments are values calculated by caller of function
  - Placed on The Stack by caller in locations set aside for the corresponding parameters
  - Function may assign new value to parameter
  - caller never looks at parameter/argument values again!
3. Arguments are removed when callee returns
  - Leaving only the result value for the caller

# PROBLEM SOLVING WITH C

## Recursion

---



### Practice Programs based on Recursion

1. Separate Recursive functions to reverse a given number and reverse a given string
2. Recursive function to print from 1 to n in reverse order
3. Find the addition, subtraction and multiplication of two numbers using recursion. Write separate recursive functions to perform these.
4. Find all combinations of words formed from Mobile Keypad.
5. Find the given string is palindrome or not using Recursion.
6. Find all permutations of a given string using Recursion



# THANK YOU

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

[sindhurpai@pes.edu](mailto:sindhurpai@pes.edu)



# PROBLEM SOLVING WITH C

## UE23CS151B

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering



# PROBLEM SOLVING WITH C

---

## Storage Classes in C

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

## Storage classes in C

---



- Introduction
- Automatic Variables
- External Variables
- Static Variables
- Register Variables
- Global Variables

### Introduction

- To describe the features of a variable/function . Features include **scope(visibility)** and **life-time** to trace the existence of a particular variable/function during the runtime
- List of storage classes
  - **Automatic variables (auto)**
  - **External variables (extern)**
  - **Static variables(static)**
  - **Register variables (register)**
  - **Global Variables**

### Automatic Variables

- A variable declared inside a function without any storage class specification is by default an automatic variable
- Created when a function is called and are destroyed automatically when the function execution is completed
- Also called as called local variables because they are local to a function. By default, assigned to undefined values
- Can be accessed outside their scope. But how ?
  - **By using Pointers**
- Coding Examples

### External Variables

- To inform the compiler that the variable is declared somewhere else and make it available during linking
- Does not allocate storage for variables
- The default initial value of external integral type is 0 otherwise null.
- All functions are of type extern
- Coding Examples

### Static Variables

- Tells the compiler to persist the variable until the end of program.
- Initialized only once and remains into existence till the end of program
- Can either be **local or global depending upon the place of declaration**
  - Scope of local static variable remains inside the function in which it is defined but the life time of is throughout that program file
  - Global static variables remain restricted to scope of file in each they are declared and life time is also restricted to that file only
- All static variables are assigned 0 (zero) as default value
- Coding Examples

### Register Variables

- Registers are faster than memory to access. So, the variables which are most frequently used in a program can be put in registers using **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.
- Compilers themselves do optimizations and put the variables in register. If a free register is not available, these are then stored in the memory only
- If & operator is used with a register variable, then compiler may give an error or warning
- Coding Examples

# PROBLEM SOLVING WITH C

## Storage classes in C

---



### Global variables- Additional Storage class

- The variables declared outside all function are called global variables. They are not limited to any function.
- Any function can access and modify global variables
- Automatically initialized to 0 at the time of declaration
- Coding Examples



**THANK YOU**

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

[sindhurpai@pes.edu](mailto:sindhurpai@pes.edu)



# PROBLEM SOLVING WITH C

## UE23CS151B

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

---

## Enums(Enumerations) in C

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

## Enumerations

---



- Introduction
- Enum creation
- Points wrt Enums
- Demo of Enums in C

# PROBLEM SOLVING WITH C

## Enumerations

---



### Introduction

- A way of creating user defined data type to assign names to integral constants. Easy to remember names rather than numbers
- Provides a symbolic name to represent one state out of a list of states
- The names are symbols for integer constants, which won't be stored anywhere in program's memory
- Used to replace **#define chains**

# PROBLEM SOLVING WITH C

## Enumerations

---



### Enum creation

- **Syntax:**

```
enum identifier { enumerator-list };      // semicolon compulsory  
                                         // identifier optional
```

- Example:

```
enum Error_list { SUCCESS, ERROR, RUN_TIME_ERROR, BIG_ERROR };
```

- Coding Examples

# PROBLEM SOLVING WITH C

## Enumerations

---



### Points wrt Enums

- Enum names are automatically assigned values if no value specified
- We can assign values to some of the symbol names in any order. All unassigned names get value as value of previous name plus one.
- Only integer constants are allowed. Arithmetic operations allowed-> + , - , \* , / and %
- Enumerated Types are Not Strings. Two enum symbols/names can have same value
- All enum constants must be unique in their scope. It is not possible to change the constants
- Storing the symbol of one enum type in another enum variable is allowed
- One of the short comings of Enumerated Types is that they don't print nicely

# PROBLEM SOLVING WITH C

## Enumerations



### Demo of Enum in C

- Demo of Enum points discussed in the previous slide



# THANK YOU

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

[sindhurpai@pes.edu](mailto:sindhurpai@pes.edu)



# PROBLEM SOLVING WITH C

## UE23CS151B

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

---

## Interface and Implementation

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

## Interface and Implementation

---



1. Introduction
2. Header file creation
3. Source file creation
4. Demo of C Code
5. Compilation steps
6. Intro to make utility
7. Creation of make file
8. Usage of make command

# PROBLEM SOLVING WITH C

## Interface and Implementation

---



### Introduction

- Interface is *declaration* and Implementation is *Definition*
- Good C code organizes
  - Interfaces in header file
  - Implementations in source files
- Benefits
  - Modularity
  - Recompilation time is reduced
  - Readability
  - Debugging is easier

# PROBLEM SOLVING WITH C

## Interface and Implementation

---



### Header file creation

- Use '.h' extension
  
- Typically contains
  - Function declaration (except statics)
  - Variable declaration (typically global)
  - User defined type declaration (read struct, union etc.)
  - Macro definition

# PROBLEM SOLVING WITH C

## Interface and Implementation

---



### Source file creation

- Use '.c' extension
- Typically contains
  - Function/variable definition
  - Static function declaration and definition (you don't want to expose these to your clients)

# PROBLEM SOLVING WITH C

## Interface and Implementation

---



### Demo of C Code

Problem to be solved: **Find whether the given number is a palindrome or not**

# PROBLEM SOLVING WITH C

## Interface and Implementation

---



### Compilation steps

- Source file compilation: **gcc -c 1\_palin.c**
- Client file compilation: **gcc -c 1\_palin\_main.c**
- Linking object files: **gcc 1\_palin.o 1\_palin\_main.o**
- Execution:
  - **a.exe // windows**
  - **./a.out // ubuntu**

# PROBLEM SOLVING WITH C

## Interface and Implementation

---



### Introduction to make utility

- What if you have many implementation files and modifications are done to few of these?
  - Developer must remember the files for which modifications done and recompile only them.
  - OR it is waste of time to recompile all the files and link objects of all again.
- Make is a Unix utility designed to start execution of a makefile.
- A makefile is a special file containing shell commands
- Use '.mk' extension preferably.
- A makefile that works well in one shell may not execute properly in another shell.

# PROBLEM SOLVING WITH C

## Interface and Implementation

---



### Creation of make file

- Contains a list of **rules to compile and link a series of files**. Rules are specified in two lines.
  - **Dependency line** – Made up of two parts. Target file(s) : source file(s)
  - **Action line** – Must be intended with a tab.
- Make command reads a make file and creates a **Dependency Tree**.
- **Target files are rebuilt using Action line if they are missing or older than the source files**.
- Create a sample make file to execute the palindrome problem.

# PROBLEM SOLVING WITH C

## Interface and Implementation

---



### Usage of make command

- Command to execute on **Ubuntu**:
  - **make -f filename.mk** // -f to read file as a make file
- Command to execute on **Windows using mingw**
  - **mingw32-make -f filename.mk** // -f to read file as a make file

**Note:** In windows, utility must be downloaded. If installed gcc using mingw, go to mingw/bin in cmd prompt and type **mingw-get install mingw32-make** and press enter



# THANK YOU

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

[sindhurpai@pes.edu](mailto:sindhurpai@pes.edu)



# PROBLEM SOLVING WITH C

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering



# PROBLEM SOLVING WITH C

---

## Strings, String manipulation & Errors

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

## Strings, String manipulation & Errors

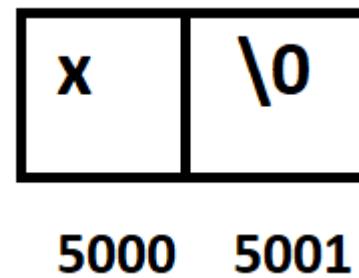
---



1. Introduction
2. Declaration
3. Initialization
4. Demo of C Code
5. String v/s Pointer
6. Builtin String manipulation functions
7. User defined string functions
8. Error demonstration

### Introduction

- An array of characters and terminated with a special character '\0' or NULL. ASCII value of NULL character is 0.
- String constants are always enclosed in double quotes. It occupies one byte more to store the null character.
- Example: “X” is a String constant



# PROBLEM SOLVING WITH C

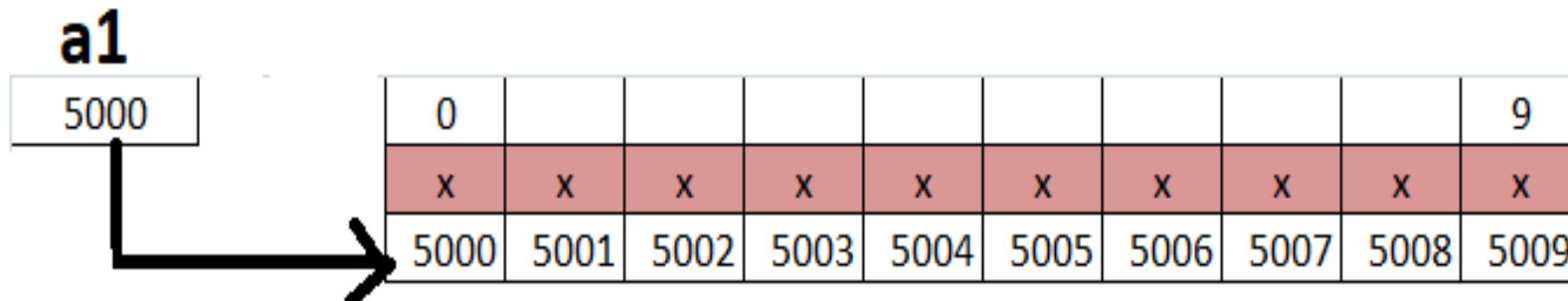
## Strings, String manipulation & Errors



### Declaration

Syntax: **char variable\_name[size];** //Size is compulsory

```
char a1[10];           // valid declaration  
char a1[];            // invalid declaration
```



# PROBLEM SOLVING WITH C

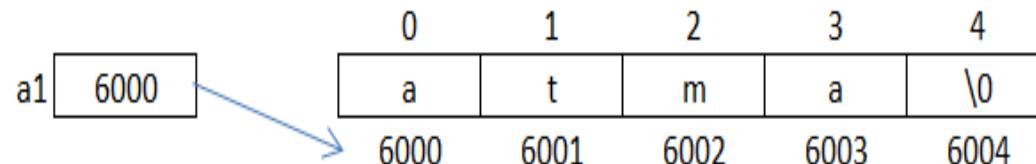
## Strings, String manipulation & Errors

### Initialization

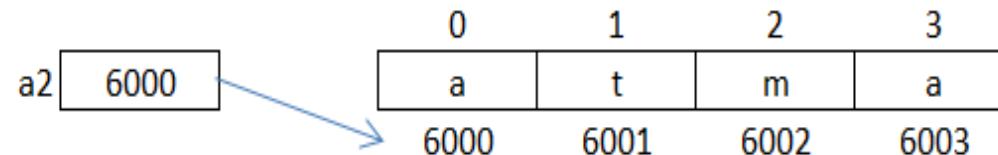
Syntax: **char variable\_name[size] = {Elements separated by comma};**

#### Version 1:

- `char a1[] = {'a', 't', 'm', 'a', '\0' };`
- Shorthand notation: `char a1[] = "atma";`

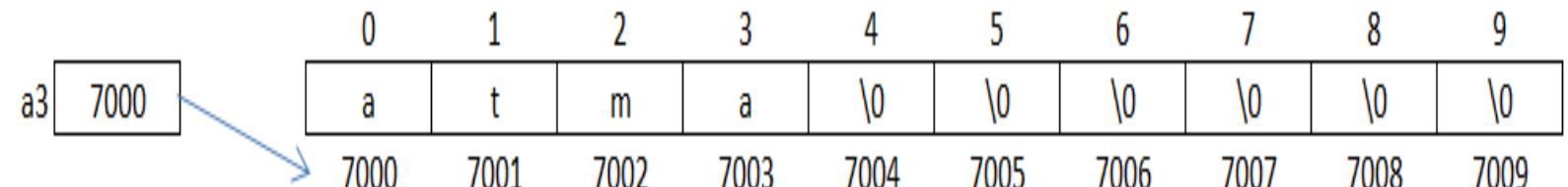


#### Version 2: `char a2[] = {'a', 't', 'm', 'a' };`



#### Version 3: Partial initialization

- `char a3[10] = {'a','t','m','a'};`



# PROBLEM SOLVING WITH C

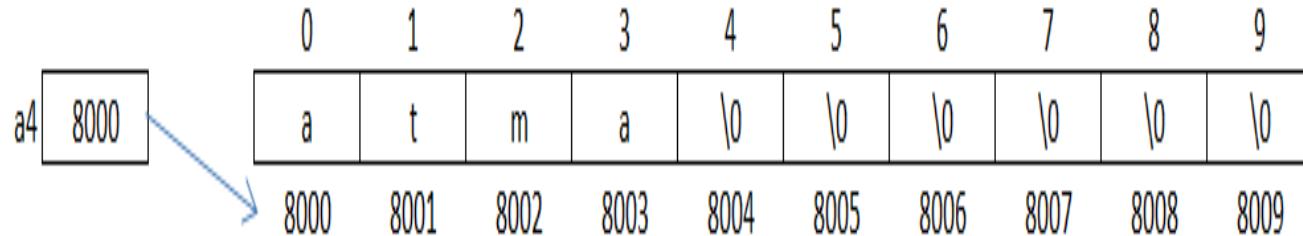
## Strings, String manipulation & Errors



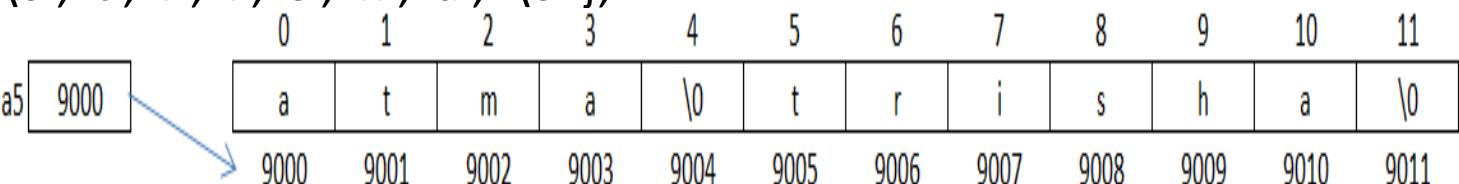
### Initialization continued..

#### Version 4: Partial initialization

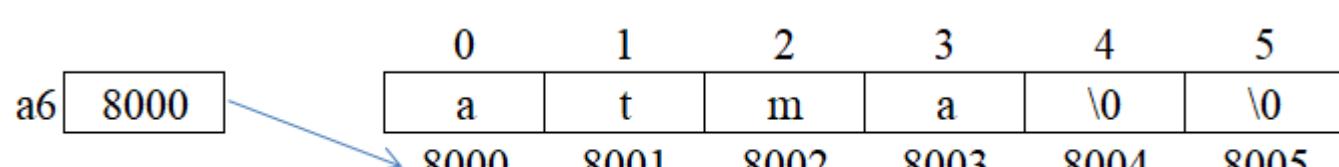
- `char a4[10] = {'a','t','m','a', '\0'};`
- `char a4[10] = "atma";`



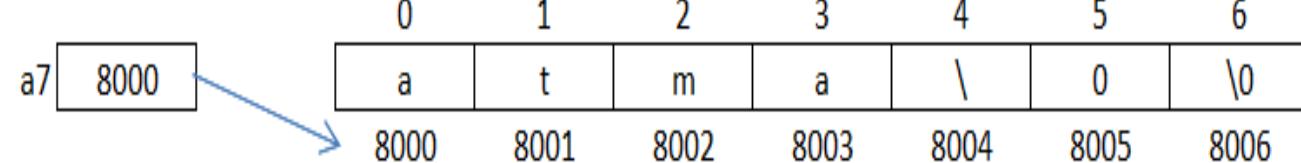
#### Version 5: `char a5[ ] = {'a', 't', 'm', 'a', '\0', 't', 'r', 'i', 's', 'h', 'a', '\0' };`



#### Version 6: `char a6[ ] = "atma\0" ;`



#### Version 7: `char a7[ ] = "atma\\0" ;`



#### Version 8: `char a8[ ] = "at\0ma" ;`

# PROBLEM SOLVING WITH C

## Strings, String manipulation & Errors

---



### Demo of C Code

- To read and display a string in C
- Points to note:
  - If the string is hard coded, it is programmer's responsibility to end the string with '\0' character.
  - scanf terminates when white space is given in the user input.
  - scanf with %s will introduce '\0' character at the end of the string. printf with %s requires the address and will display the characters until '\0' character is encountered
  - If you want to store the entire input from the user until user presses new line in one character array variable, use [^\n] with %s in scanf

# PROBLEM SOLVING WITH C

## Strings, String manipulation & Errors

---



### String v/s Pointer

- `char x[] = "pes";` // x is an array of 4 characters with 'p', 'e', 's', '\0  
**Stored in the Stack segment of memory**
- Can change the elements of x. `x[0] = 'P';`
- Can not increment x as it is an array name. x is a constant pointer.
  
- `char *y = "pes";`  
**y is stored at stack. “pes” is stored at code segment of memory. It is read only.**
- `y[0] = 'P' ;` // undefined behaviour
- Can increment y . y is a pointer variable.

### Built-in String manipulation Functions

- Expect '\0' and available in **string.h**
- In character array, if '\0' is not available at the end, result is undefined when passed to built-in string functions
- **strlen(a)** – Expects string as an argument and returns the length of the string, excluding the NULL character
- **strcpy(a,b)** – Expects two strings and copies the value of b to a.
- **strcat(a,b)** – Expects two strings and concatenated result is copied back to a.
- **strchr(a,ch)** – Expects string and a character to be found in string. Returns the address of the matched character in a given string if found. Else, NULL is returned.
- **strcmp(a,b)** – Compares whether content of array a is same as array b. If a==b, returns 0. Returns 1, if array a has lexicographically higher value than b. Else, -1.

### User defined String functions

- Start with iterative solution
- Usage of pointer arithmetic operations
- Usage of recursion to get the solution for few of the below functions
  1. my\_strlen()
  2. my\_strcpy()
  3. my\_strcmp()
  4. my\_strchr()
  5. my\_strcat()

### Error Demonstration

1. 

```
char * name[10] ; // declaring an array of 10 pointer pointing each one to a char
char name[10];
```
2. 

```
name = "choco"; // Cannot assign a string by using the operator =
char name[10] = "choco" OR use strcpy()
```
3. 

```
printf("%s\n", *name);
```

Provide only the address of the first element of the string
4. Coding examples to demo the error caused while applying built-in string functions on strings without specifying '\0' at the end of the string



# THANK YOU

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

[sindhurpai@pes.edu](mailto:sindhurpai@pes.edu)



# PROBLEM SOLVING WITH C

## UE23CS151B

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering



# PROBLEM SOLVING WITH C

## Dynamic Memory Management

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

## Dynamic Memory Management

---



1. Problem with the Arrays
2. Memory Allocation
3. Dynamic allocation
4. Use of malloc(), calloc(), realloc(), free()

# PROBLEM SOLVING WITH C

## Dynamic Memory Management

---



### Problem with the Arrays

- Few situations while coding:
  - Amount of data cannot be predicted beforehand
  - Number of data items keeps changing during program execution
- In such cases, use of fixed size array might create problems:
  - Wastage of memory space (under utilization)
  - Insufficient memory space (over utilization)
- **Example:** A[1000] can be used but what if the user wants to run the code for only 50 elements  
//memory wasted

**Solution:** Can be avoided by using the concept of Dynamic memory management

# PROBLEM SOLVING WITH C

## Dynamic Memory Management

---



### Memory Allocation

#### 1. Static allocation

- decided by the compiler
- allocation at load time [before the execution or run time]
- example: variable declaration (int a, float b, a[20];)

#### 2. Automatic allocation

- decided by the compiler
- allocation at run time
- allocation on entry to the block and deallocation on exit
- example: function call (stack space is used and released as soon as callee function returns back to the calling function)

#### 3. Dynamic allocation

- code generated by the compiler
- allocation and deallocation on call to memory allocation and deallocation functions

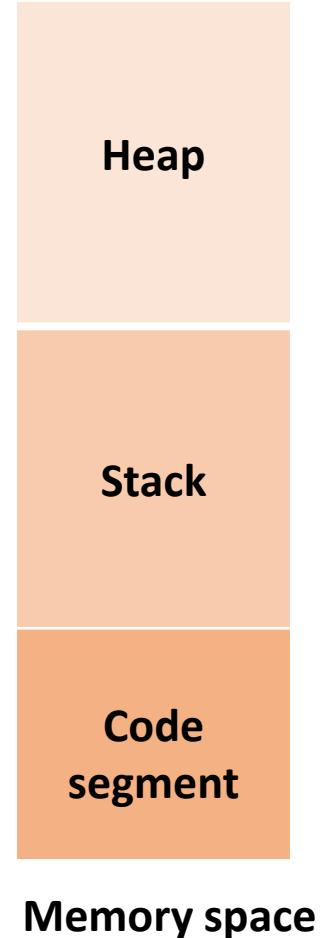
# PROBLEM SOLVING WITH C

## Dynamic Memory Management



### Dynamic Allocation

- Process of allocating memory at runtime/execution
- Uses the **Heap region of Memory segment**
- No operator in C to support dynamic memory management
- Library functions are used to dynamically allocate/release memory
  - malloc()
  - calloc()
  - realloc()
  - free()
- Available in stdlib.h

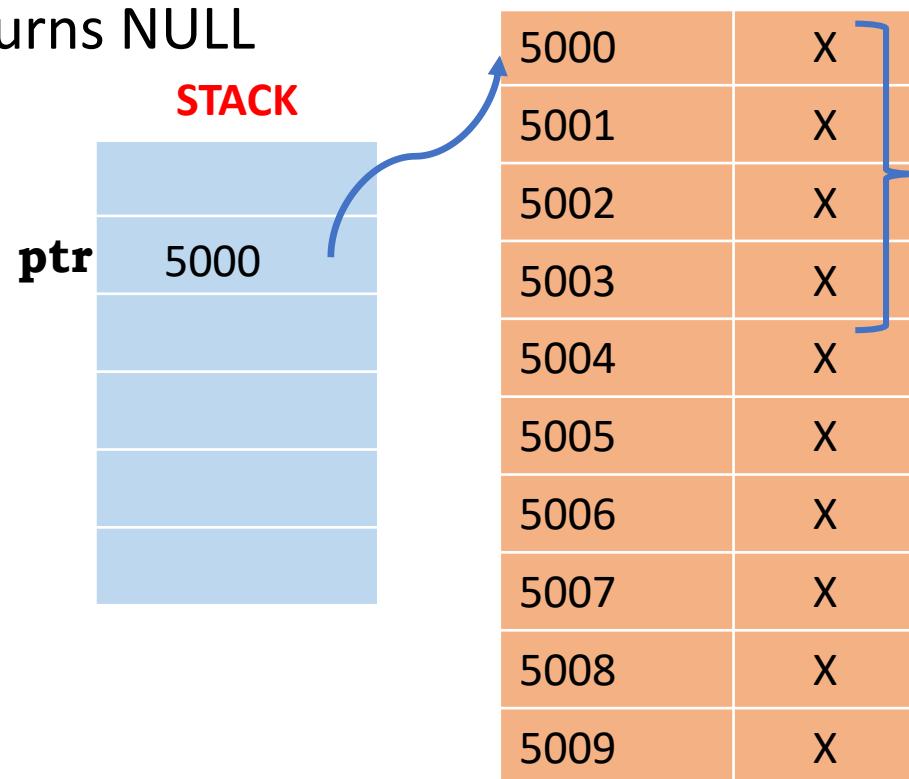


# PROBLEM SOLVING WITH C

# Dynamic Memory Management

## malloc() - memory allocation

- Allocates requested size of bytes and returns a void pointer pointing to the first byte of the allocated space on success. Else returns NULL
  - The return pointer can be type-casted to any pointer type
  - Memory is not initialized
  - **Syntax:**  
`void *malloc(size_t N); // Allocates N bytes of memory`
  - **Example:**  
`int* ptr = (int*) malloc(sizeof (int)); // Allocate memory for an int`
  - Coding example



# PROBLEM SOLVING WITH C

## Dynamic Memory Management



### calloc() - contiguous allocation

- Allocates space for elements, initialize them to zero and then returns a void pointer to the memory. Else returns NULL

- The return pointer can be type-casted to any pointer type **ptr**

- Syntax:**

```
void *calloc(size_t nmemb, size_t size);
```

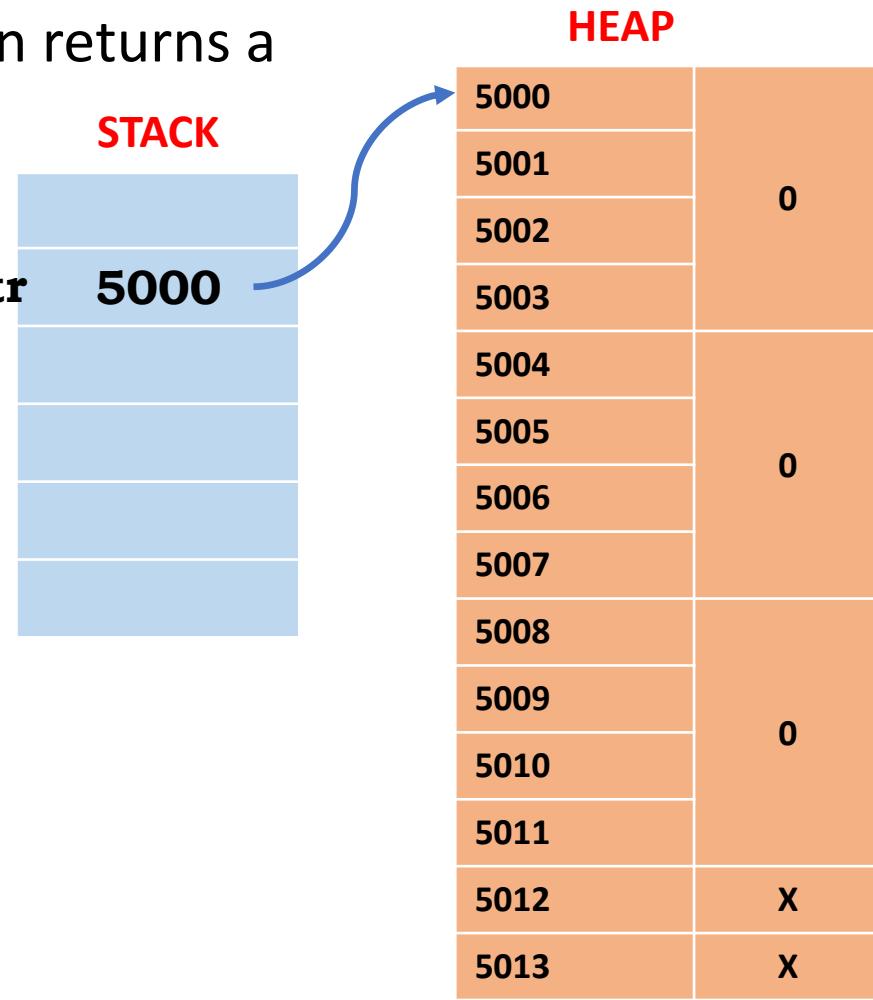
//allocates memory for an array of nmemb elements of size bytes each

- Example:**

```
int* ptr = (int*) calloc (3,sizeof (int));
```

//Allocating memory for an array of 3 elements of integer type

- Coding example**



# PROBLEM SOLVING WITH C

## Dynamic Memory Management

### realloc() - reallocation of memory



- Modifies the size of previously allocated memory using malloc or calloc functions.
- Returns a pointer to the newly allocated memory which has the new specified size. Returns NULL for an unsuccessful operation
- If realloc() fails, the original block is left untouched
- **Syntax:** `void *realloc(void *ptr, size_t size);`
- If ptr is NULL, then the call is equivalent to malloc(size), for all values of size
- If size is equal to zero, and ptr is not NULL, then the call is equivalent to free(ptr)
- This function can be used only for dynamically allocated memory, else behavior is undefined

# PROBLEM SOLVING WITH C

## Dynamic Memory Management



### realloc() continued..

- The content of the memory block is preserved up to the lesser of the new and old sizes, even if the block is moved to a new location
- If the new size is larger than the old size, then it checks if there is an option to expand or not.
  - If the existing allocation of memory can be extended, it extends it but the added memory will not be initialized.
  - If memory cannot be extended, a new sized memory is allocated, initialized with the same older elements and pointer to this new address is returned. Here also added memory is uninitialized.
- If the new size is lesser than the old size, content of the memory block is preserved.
- Coding examples

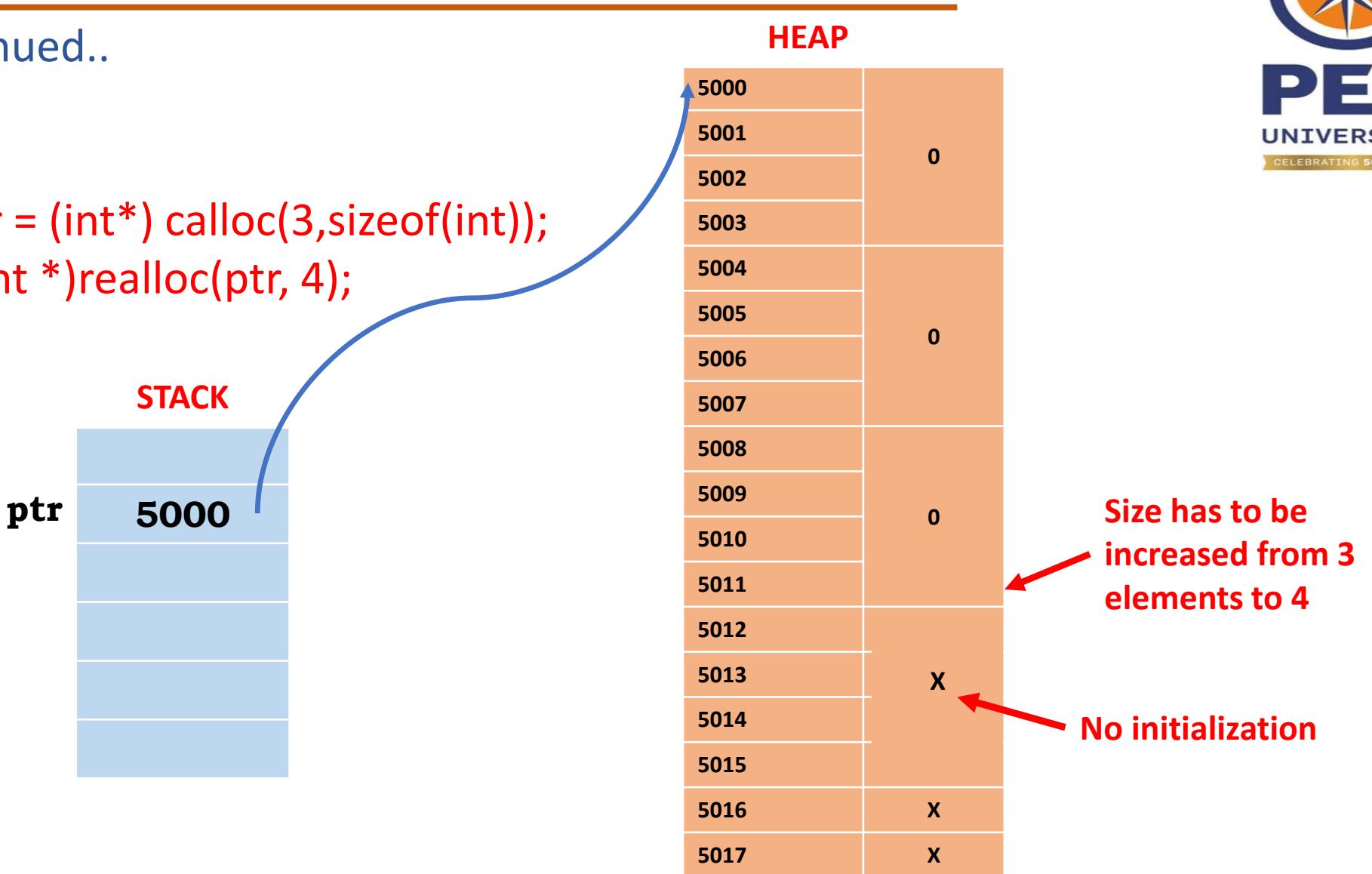
# PROBLEM SOLVING WITH C

## Dynamic Memory Management

realloc() continued..

Example:

```
int* ptr = (int*) calloc(3,sizeof(int));  
ptr = (int *)realloc(ptr, 4);
```



# PROBLEM SOLVING WITH C

## Dynamic Memory Management



### free()

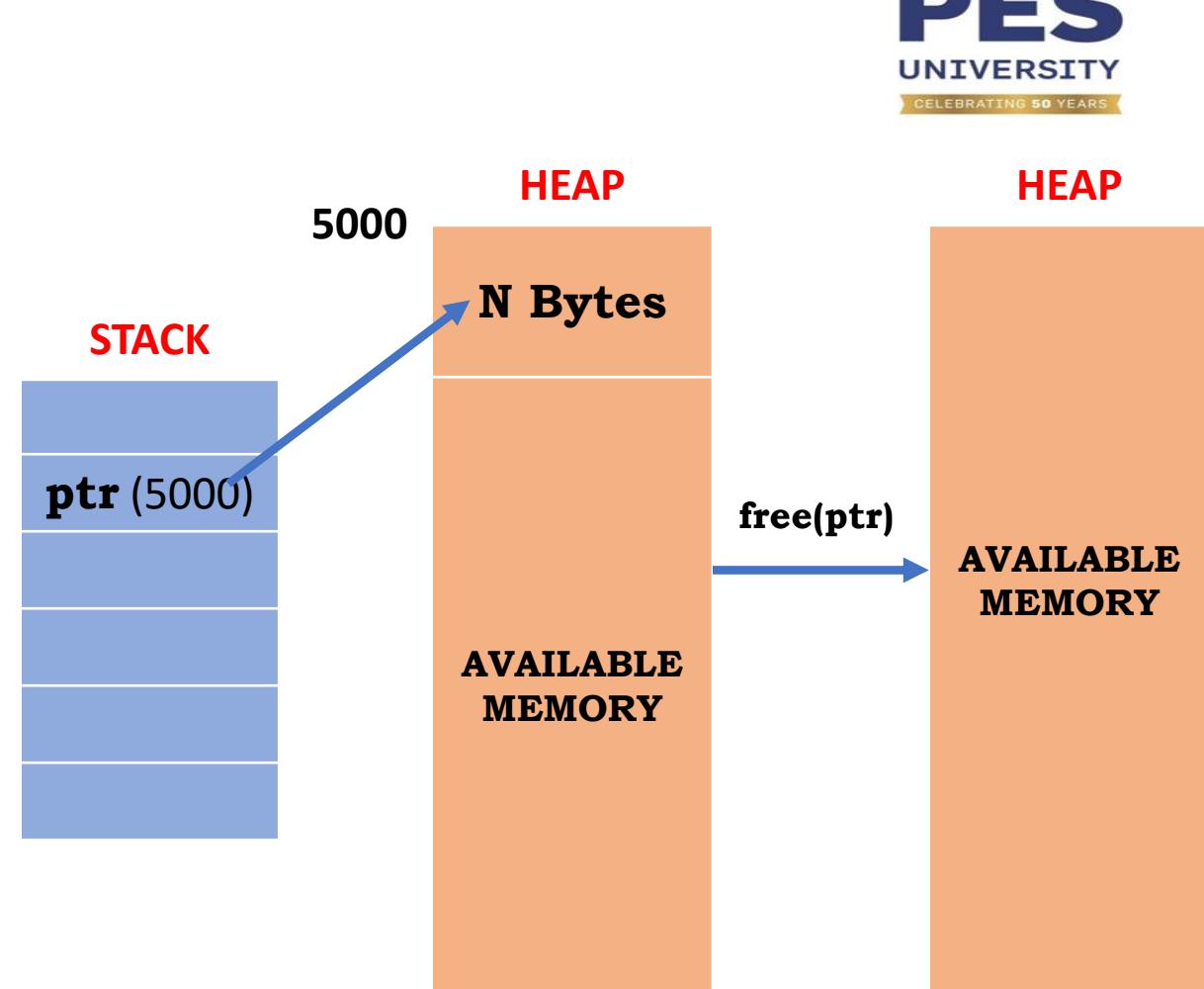
- Releases the allocated memory and returns it back to heap

- **Syntax:**

`free (ptr);` //ptr is a pointer to a memory block which has been previously created using malloc/calloc

- No size needs to be mentioned in the free().

- On allocation of memory, the number of bytes allocated is stored somewhere in the memory. This is known as **book keeping information**





# THANK YOU

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

[sindhurpai@pes.edu](mailto:sindhurpai@pes.edu)



# PROBLEM SOLVING WITH C

## UE23CS151B

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering



# PROBLEM SOLVING WITH C

## Dynamic Memory Management continued..

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

## Dynamic Memory Management

---



1. Common Programming Errors
2. Demonstration of Errors

# PROBLEM SOLVING WITH C

## Dynamic Memory Management

---



### Common Programming Errors

- **Dangling Pointer:**
  - Points to a location which doesn't exist
  - Freeing the memory results in dangling pointer
  - Using new pointer variable to store return address in realloc results in dangling pointer
  - Dereferencing the dangling pointer results in undefined behavior
  - Can happen anywhere in the memory segment
  - Solution is assigning the pointer to NULL
- **NULL Pointer:**
  - Dereferencing the NULL pointer results in Guaranteed crash

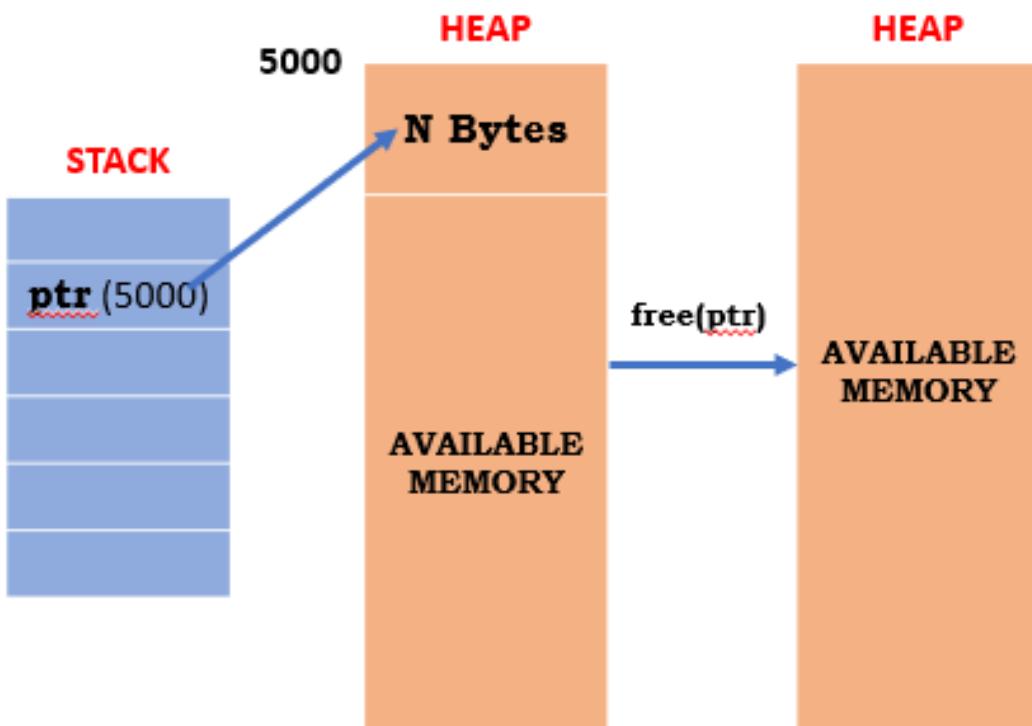
# PROBLEM SOLVING WITH C

## Dynamic Memory Management



### Pictorial Representation of Dangling pointer

- Pointer that points to the memory, which is de-allocated



Using free() to freed the memory space



Memory got freed but the pointer **ptr** is still pointing to the same address 5000.

# PROBLEM SOLVING WITH C

## Dynamic Memory Management

---



### Common Programming Errors continued..

- **Garbage:**
  - Garbage is a location which has no name and hence no access
  - If the same pointer is allocated memory more than once using the DMA functions, initially allocated spaces becomes a garbage.
  - **Garbage in turn results in memory leak.**
  - **Memory leak can happen only in Heap region**
- **Double free error: DO NOT TRY THIS**
  - If free() is used on a memory that is already freed before
  - Leads to **undefined behavior**.
  - Might corrupt the state of the memory manager that can cause existing blocks of memory to get corrupted or future allocations to fail.
  - Can cause the program to crash or alter the execution flow.

# PROBLEM SOLVING WITH C

## Dynamic Memory Management

---



### Demonstration of Errors

- C code demo resulting in Errors



# THANK YOU

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

[sindhurpai@pes.edu](mailto:sindhurpai@pes.edu)



**PES**  
UNIVERSITY

CELEBRATING 50 YEARS

# PROBLEM SOLVING WITH C

## UE23CS151B

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering



# PROBLEM SOLVING WITH C

## Text Processing

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

## Text Processing

---



1. List of Practice programs
2. Solution to the problem using C Code

# PROBLEM SOLVING WITH C

## Text Processing

---

### List of Practice Programs



- Write a function to find the position of First occurrence of the input character in a given string.
- Program to Find the count of given character in a given string.
- Given a string, Create a text with only alphabets.
- Given a string, display all the digits in that string.

# PROBLEM SOLVING WITH C

## Text Processing

---

### Solution using C Code

- Demonstrations of C Code





# THANK YOU

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

[sindhurpai@pes.edu](mailto:sindhurpai@pes.edu)

**+91 8277606459**



# PROBLEM SOLVING WITH C

## UE23CS151B

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering



# PROBLEM SOLVING WITH C

---

## String Matching

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

## String Matching

---



1. Statement of the Problem
2. Requirements to solve the problem
3. Solution to the problem

# PROBLEM SOLVING WITH C

## String Matching

---



### Statement of the Problem

- Given a text  $\text{txt}[0..n-1]$  and a pattern  $\text{pat}[0..m-1]$ , write a function `pattern_match(char txt[], int n, char pattern[], int m)` that returns the position of the character in the text string when the first occurrence of pattern is matched with the text string

# PROBLEM SOLVING WITH C

## String Matching

---



### Requirements to solve the problem

- How to read and display a string? - Discussed in Strings
- Read a Text string and Pattern string
- Usage of loop variables to traverse across two strings
- Action to be taken when characters do not match in text and pattern
- Action to be taken whenever characters in both strings match

# PROBLEM SOLVING WITH C

## String Matching

---



### Solution to the Problem

- Demonstration of C code



# THANK YOU

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering



# PROBLEM SOLVING WITH C

## UE23CS151B

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering



# PROBLEM SOLVING WITH C

## Structures in C

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

## Structures in C

---

- Introduction
- Characteristics
- Declaration
- Accessing members
- Initialization
- Memory allocation
- Comparison



### Introduction

- A user-defined data type that allows us to combine data of different types together.
- Helps to construct a complex data type which is more meaningful.
- Provides a single name to refer to a collection of related items of different types.
- Provides a way of storing many different values in variables of potentially different types under the same name.
- Generally useful whenever a lot of data needs to be grouped together.
- Creating a new type decides the binary layout of the type

### Characteristics/Properties

- Contains one or more components(homogeneous or heterogeneous) – Generally known as data members. These are named ones.
- **Order of fields and the total size of a variable** of that type is decided when the new type is created
- Size of a structure depends on implementation. Memory allocation would be **at least equal to the sum of the sizes of all the data members** in a structure. Offset is decided at compile time.
- Compatible structures may be assigned to each other.

# PROBLEM SOLVING WITH C

## Introduction



### Syntax :

- Keyword **struct** is used for creating a structure.
- The format for declaring a structure is as below:

```
struct <structure_name>
{
    data_type member1;
    data_type member2;
    ....
    data_type memebn;
};           // semicolon compulsory
```

**Example:** User defined type Student entity is created.

```
struct Student
{
    int roll_no;
    char name[20];
    int marks;
};
```

**Note: No memory allocation for declaration/description of the structure.**

# PROBLEM SOLVING WITH C

## Structures in C



s1

roll_no	X
name	X
marks	X

Fig. 1. After declaration, only undefined entries (X)

### Declaration

- Members of a structure can be accessed only when instance variables are created
- If struct Student is the type, the instance variable can be created as:

```
struct student s1; // s1 is the instance variable of type struct
```

Student

```
struct student* s2; // s2 is the instance variable of type struct
```

student\*.

```
// s2 is pointer to structure
```

- Declaration (global) can also be done just after structure body but before semicolon.

# PROBLEM SOLVING WITH C

## Structures in C

### Initialization

- Structure members can be initialized using curly braces '{}' and separated by comma.
- Data provided during initialization is mapped to its corresponding members by the compiler automatically.
- Further extension of initializations can be:
  1. **Partial initialization:** Few values are provided.  
Remaining are mapped to zero. For strings, '\0'.
  2. **Designated initialization:**
    - Allows structure members to be initialized in any order.
    - This feature has been added in C99 standard.
    - Specify the name of a field to initialize with '.member\_name =' OR 'member\_name:' before the element value. Others are initialized to default value.

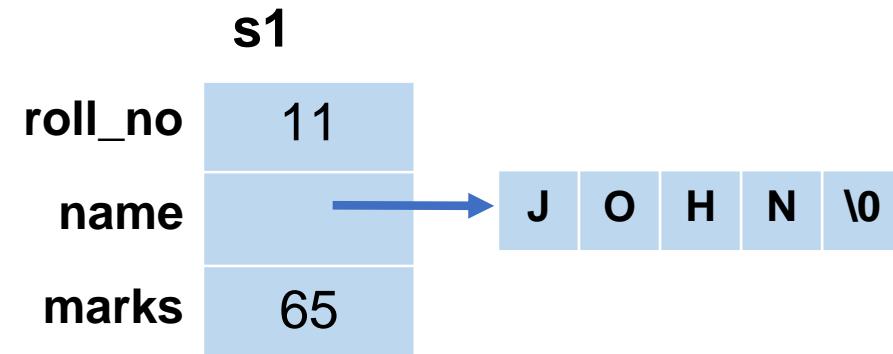


Fig. 2. After initialization, entries are mapped

# PROBLEM SOLVING WITH C

## Structures in C

---



### Accessing data members

- Operators used for accessing the members of a structure.
  1. Dot operator (.)
  2. Arrow operator (->)
- Any member of a structure can be accessed using the structure variable as:

**structure\_variable\_name.member\_name**

Example:

`s1.roll_no`

//where s1 is the structure variable name and roll\_no member is data member of s1.

- Any member of a structure can be accessed using the pointer to a structure as:

**pointer\_variable->member\_name**

`s2->roll_no`

Example:

// where s2 is the pointer to structure variable and we want to access roll\_no member of s2.

# PROBLEM SOLVING WITH C

## Structures in C

---

### Memory allocation

- At least equal to the sum of the sizes of all the data members.
- Size of data members is implementation specific.
- Coding Examples



### Comparison of structures



- Comparing structures in C is not permitted to check or compare directly with logical and relational operators.
- Only structure members can be compared with relational operator.
- Coding examples



# THANK YOU

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering



# PROBLEM SOLVING WITH C

## UE23CS151B

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering



# PROBLEM SOLVING WITH C

## Array of Structures

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

## Array of Structures

---



1. Introduction
2. Structure Variable Declaration
3. Structure Variable Initialization
4. Pointer to an Array of Structures

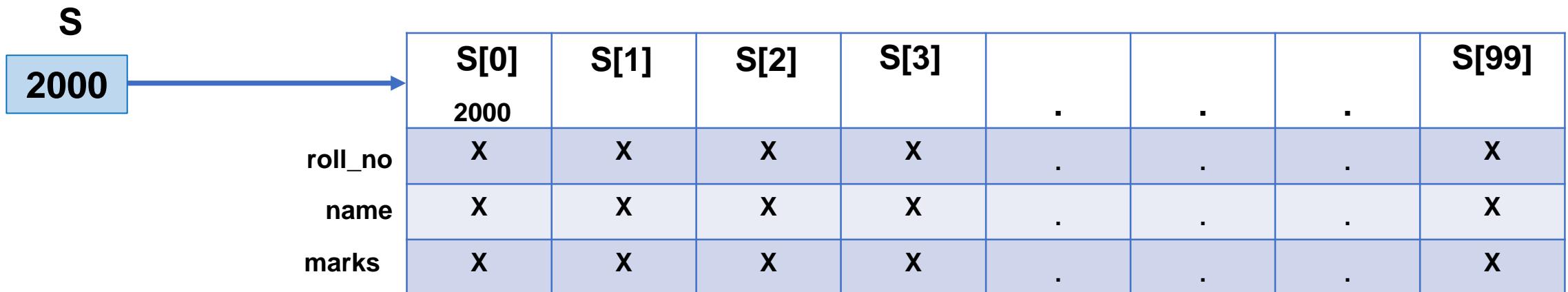
# PROBLEM SOLVING WITH C

## Array of Structures



### Introduction

- Think about storing the roll number, name and marks of one student  
**We need structures to store different types of related data.**  
**struct student s;**
- Think about storing the roll number, name and marks of 100 students  
**We need an Array of structures**  
**struct student S[100];**
- S[0] stores the information of first student, S[1] stores the information of second student and so on.



# PROBLEM SOLVING WITH C

## Array of Structures

---



### Declaration of Structure variable

- Can be done in two ways
  - Along with structure declaration after closing } before semicolon (;) of structure body.

```
struct student {  
    int roll_num; char name[100]; int marks;  
}s[100];
```
  - After structure declaration (either inside main or globally using struct keyword)

```
struct student s[100];
```
- Coding examples

# PROBLEM SOLVING WITH C

## Array of Structures

---



### Initialization of structure variable

- **Compile time initialization:** Using a brace-enclosed initializer list

```
struct student S[] = { {1, "John", 60}, {2,"Jack", 40}, {3, "Jill", 77} };  
struct student S[3] = { {11, "Joseph", 60}}; //partial initialization  
struct student S[2] = {1, "John", 60, 2,"Jack", 40};
```

- **Run time initialization (using loops preferably)**
- Coding examples

# PROBLEM SOLVING WITH C

## Array of Structures

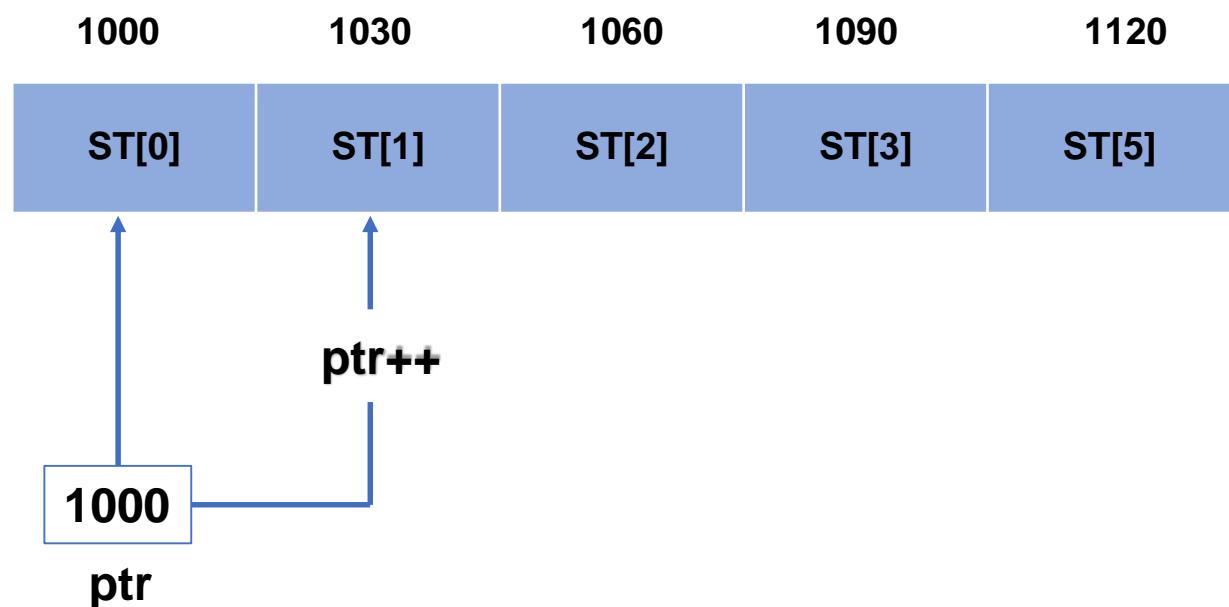


### Pointer to an Array of Structures

- Used to access the array of structure variables efficiently

```
struct student
{
    int roll_no;
    char name[22];
    int marks;
}ST[5];
```

```
struct student *ptr = ST;
```



- Coding examples to print the array of structures using pointer



**THANK YOU**

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering



# PROBLEM SOLVING WITH C

## UE23CS151B

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering



# PROBLEM SOLVING WITH C

---

## Array of Pointers to Structures

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

## Array of Pointers to Structures

---



- Array of Pointers
- Pointers to Structures
- Array of Pointers to Structures
- Demo of C Code

# PROBLEM SOLVING WITH C

## Array of Pointers to Structures



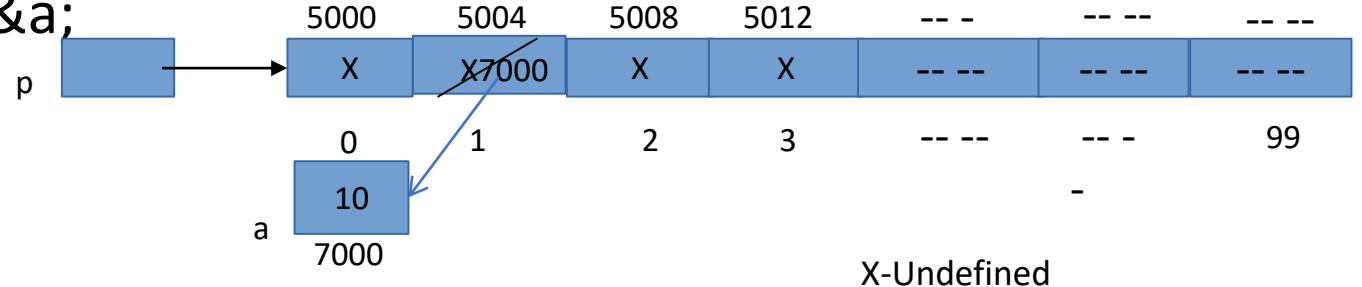
### Array of Pointers

- Is an indexed set of variables in which the variables are pointers

**Syntax:** `int *var_name[array_size];`

**Example:** `int *p[100];`

- The element of an array of a pointer can be initialized by assigning the address of other element. Example : `int a = 10; p[1] = &a;`



- Used to create complex data structures such as linked lists, trees, graphs

# PROBLEM SOLVING WITH C

## Array of Pointers to Structures

---



### Pointers to Structures

- Holds the address of structure variable
- Declaring a pointer to structure is same as pointer to variable

**Syntax:** `struct tagname *ptr;`

- Members of the structure are accessed using arrow (`->`) operator.

# PROBLEM SOLVING WITH C

## Array of Pointers to Structures

---



### Array of pointers to Structures

- Creating an array of structure variable

**Syntax:** `struct tagname array-variable[size];`

- Creating a pointer variable to hold the address of structure variable

**Syntax:** `struct tagname *pointer-variable;`

- Creating an array of pointers with size specified to hold the addresses of structures in the array of structure variable.

**Syntax:** `struct tagname *pointer_variable[size];`

- Coding Examples

# PROBLEM SOLVING WITH C

## Array of Pointers to Structures

---

### Demo of C Code



- Program to swap first and last elements of the array of structures using array of pointers and display the array of structures using array of pointers.



**THANK YOU**

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering



# PROBLEM SOLVING WITH C

## UE23CS151B

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering



# PROBLEM SOLVING WITH C

---

## Sorting

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

## Sorting

---



1. Why Sorting?
2. Sorting Algorithms
3. Bubble Sort Algorithm
4. Demonstration of C Code

### Why Sorting ?

- To access the data in a very quick time
- Think about searching for something in a sorted drawer and unsorted drawer
- If the large data set is sorted based on any of the fields, then it becomes easy to search for a particular data in that set.

### Sorting Algorithms

- **Bubble Sort**
- **Insertion Sort**
- **Quick Sort**
- **Merge Sort**
- **Radix Sort**
- **Selection Sort**
- **Heap Sort**



### Bubble Sort Algorithm

- An array is traversed from left and adjacent elements are compared and the higher one is placed at right side.
- In this way, the largest element is moved to the rightmost end at first.
- This process is continued to find the second largest number and this number is placed in the second place from rightmost end and so on until the data is sorted.

# PROBLEM SOLVING WITH C

## Sorting

---

### Demonstration of C Code

- Demo of Bubble sort on structures





# THANK YOU

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering



# PROBLEM SOLVING WITH C

## UE23CS151B

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering



# PROBLEM SOLVING WITH C

---

## Linked List

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

## Linked List

---



1. Introduction
2. Self Referential Structures
3. Characteristics
4. Operations on linked list
5. Pictorial Representation
6. Different Types
7. Applications

# PROBLEM SOLVING WITH C

## Linked List

---



### Introduction

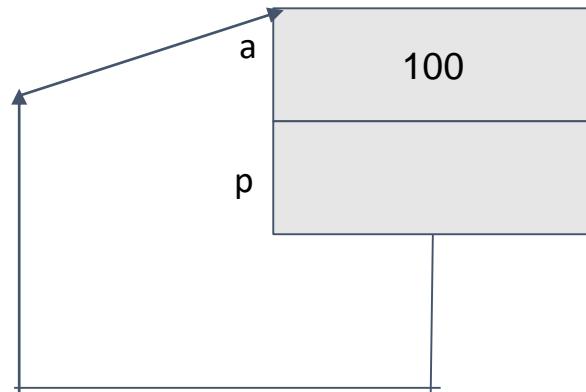
- Collection of nodes connected via links.
- The node and link has a special meaning in C.
- Few points to think!!
  - Can we have pointer data member inside a structure? - Yes
  - Can we have structure variable inside another structure? - Yes
  - Can we have structure variable inside the same structure ? – No

**Solution is: Have a pointer of same type inside the structure**

### Self Referential Structures

- A structure which has a pointer to itself as a data member

```
struct node
{
    int a;
    struct node *p;
} ; // defines a type struct node
// memory not allocated for type declaration
```



- Variable declaration to allocate memory and assigning values to data members

```
struct node s;  s.a = 100;      s.p = &s;
```

### Characteristics

- A data structure which consists of zero or more nodes.
- Every node is composed of two fields: data/component field and a pointer field
- The pointer field of every node points to the next node in the sequence
- Accessing the the nodes is always one after the other. There is no way to access the node directly as random access is not possible in linked list. Lists have sequential access
- Insertion and deletion in a list at a given position requires no shifting of element

# PROBLEM SOLVING WITH C

## Linked List

---



### Operations on Linked List

- Insertion
- Deletion
- Search
- Display
- Merge
- Concatenate

# PROBLEM SOLVING WITH C

## Linked List

### Pictorial Representation

- Structure definition of a node

```
struct node {  
    int info; // component field  
    struct node *link; // pointer field  
};  
typedef struct node NODE_T;
```

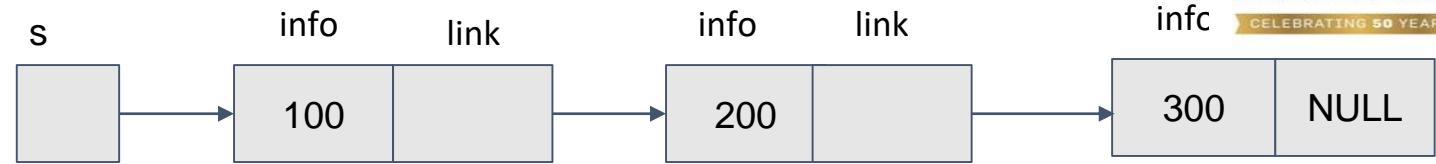


Fig1: Linked list Representation

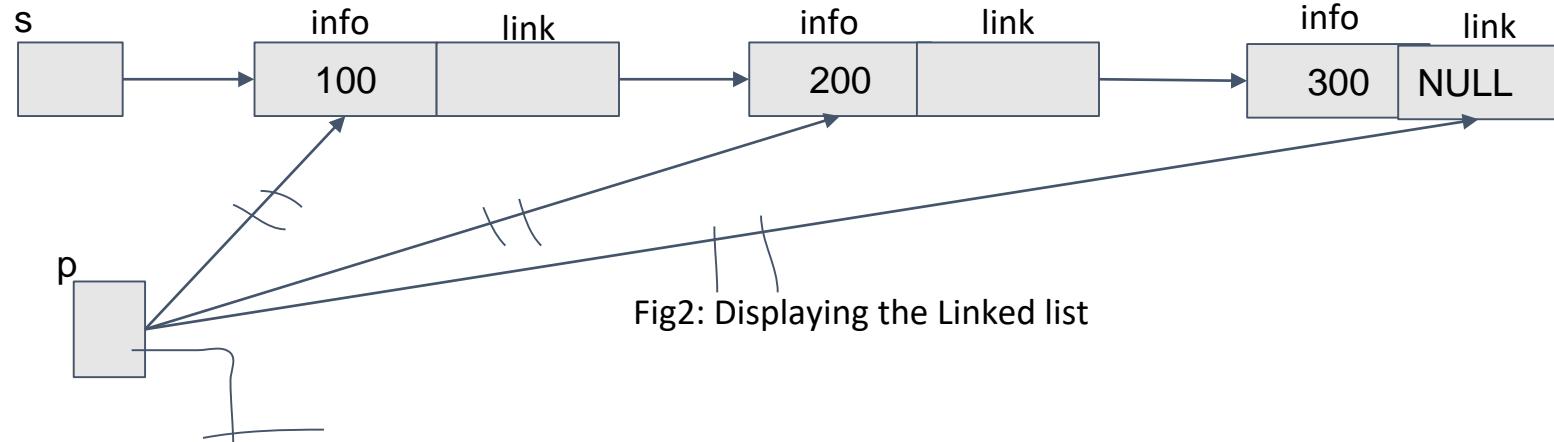


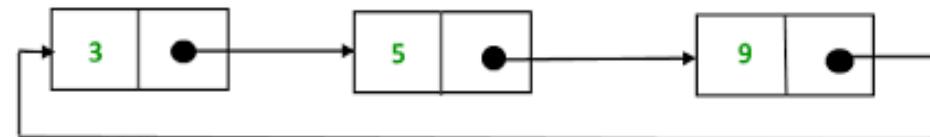
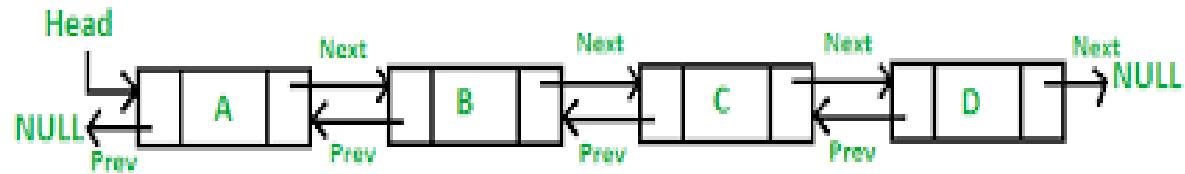
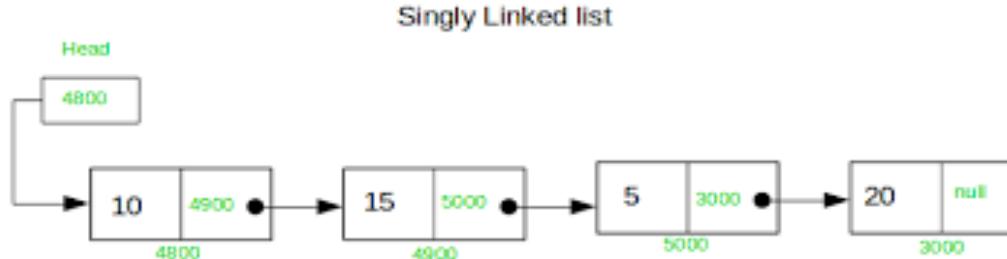
Fig2: Displaying the Linked list

# PROBLEM SOLVING WITH C

## Linked List

### Different Types

- Singly Linked List
- Doubly Linked List
- Circular Linked List



### Applications

- Implementation of Stacks & Queues
- Implementation of Graphs
- Maintaining dictionary
- Gaming
- Evaluation of Arithmetic Expression





# THANK YOU

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

[sindhurpai@pes.edu](mailto:sindhurpai@pes.edu)



# PROBLEM SOLVING WITH C

## UE23CS151B

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# **PROBLEM SOLVING WITH C**

---

## **Priority Queue**

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

## Priority Queue

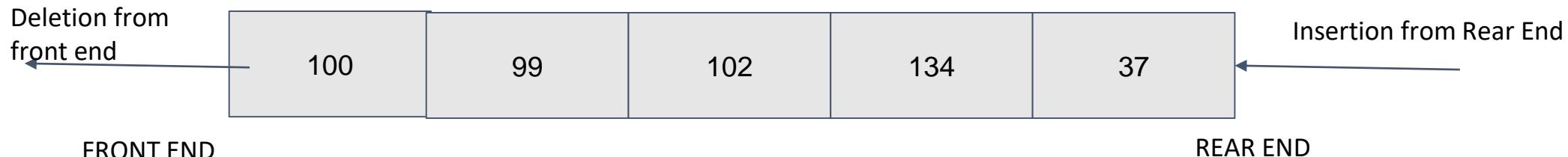
---



1. Introduction to Queue
2. Operations on Queue
3. Types of Queues
4. Priority Queue
5. Applications of Priority Queue

### Introduction to Queue

- A line or a sequence of people or vehicles awaiting for their turn to be attended or to proceed.
- In computer Science, a list of data items, commands, etc., stored so as to be retrievable in a definite order
- A Data structure which has 2 ends – **Rear end and a Front end**. Open ended at both ends
- Data elements are inserted into the queue from the Rear end and deleted from the front end.
- Follows the Principle of **First In First Out (FIFO)**



# PROBLEM SOLVING WITH C

## Priority Queue

---



### Operations on Queue

- **Enqueue** – Add (store) an item to the queue from the Rear end.
- **Dequeue** – Remove (access) an item from the queue from the Front end.

### Types of Queues

- **Ordinary Queue** - Insertion takes place at the Rear end and deletion takes place at the Front end
- **Priority Queue** - Special type of queue in which each element is associated with a priority and is served according to its priority. If elements with the same priority occur, they are served according to their order in the queue
- **Circular Queue** - Last element points to the first element of queue making circular link.
- **Double ended Queue** - Insertion and Removal of elements can be performed from both front and rear ends

# PROBLEM SOLVING WITH C

## Priority Queue

---



### Priority Queue

- Type of Queue where each element has a "**Priority**" associated with it.
- Priority decides about the Deque operation.
- The Enque operation stores the item and the “Priority” information
- Types of Priority Queue:

**Ascending Priority Queue:** Smallest Number - Highest Priority

**Descending Priority Queue:** Highest Number - Highest Priority

# PROBLEM SOLVING WITH C

## Priority Queue

---



### Applications of Priority Queue

1. Implementation of Heap Data structure.
2. Dijkstra's Shortest Path Algorithm
3. Prim's Algorithm
4. Data Compression
5. OS - Load Balance Algorithm.



**THANK YOU**

---

**Prof Sindhu R Pai**

Department of Computer Science and Engineering

[sindhurpai@pes.edu](mailto:sindhurpai@pes.edu)



# PROBLEM SOLVING WITH C

## UE23CS151B

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# **PROBLEM SOLVING WITH C**

---

## **Unions in C**

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

## Unions in C

---



- What is Union?
- Accessing Union Members
- Union vs Structure

# PROBLEM SOLVING WITH C

## Unions in C

---



### What is Union?

- A user defined data type which may hold members of different sizes and types
- Allow data members which are **mutually exclusive to share the same memory**
- Unions provide an efficient way of using the same memory location for multiple-purpose
  - At a given point in time, only one can exist
- The memory occupied will be large enough to hold the largest member of the union
  - **The size of a union is at least the size of the biggest component**
- All the **fields overlap** and they have the **same offset : 0.**
- Used while coding embedded devices

# PROBLEM SOLVING WITH C

## Unions in C

---



### Accessing the Union members

- **Syntax:**

```
union Tag      // union keyword is used
{
    data_type member1;   data_type member2;   ... data_type member n;
};      // ; is compulsory
```

- To access any member using the variable of union type,
  - use the **Member Access operator (.)**
- To access any member using the variable of pointer to union type,
  - use the **Arrow operator (->)**
- Coding Examples

# PROBLEM SOLVING WITH C

## Unions in C

### Union Vs Structure

	<b>STRUCTURE</b>	<b>UNION</b>
<b>Keyword</b>	The keyword <b>struct</b> is used to define a structure	The keyword <b>union</b> is used to define a union.
<b>Size</b>	When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is greater than or equal to the sum of sizes of its members.	when a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of union is equal to the size of largest member.
<b>Memory</b>	Each member within a structure is assigned unique storage area of location.	Memory allocated is shared by individual members of union.
<b>Value Altering</b>	Altering the value of a member will not affect other members of the structure.	Altering the value of any of the member will alter other member values.
<b>Accessing members</b>	Individual member can be accessed at a time.	Only one member can be accessed at a time.
<b>Initialization of Members</b>	Several members of a structure can initialize at once.	Only the first member of a union can be initialized.



# THANK YOU

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

[sindhurpai@pes.edu](mailto:sindhurpai@pes.edu)



# PROBLEM SOLVING WITH C

## UE23CS151B

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# **PROBLEM SOLVING WITH C**

---

## **Bit fields in C**

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

## Bit fields in C

---

- What is a Bit field?
- Bit field creation
- Few points on Bit fields



### What is a Bit field ?

- Data structure that allows the programmer to allocate memory to structures and unions in bits in order to utilize computer memory in an efficient manner.
- The variables defined with a predefined width and can hold more than a single bit
- Consists of a number of adjacent computer memory locations which have been allocated to hold a sequence of bits.
- **Great significance in C programming** because of the following reasons
  - Used to reduce memory consumption
  - Easy to implement
  - Provides flexibility to the code

# PROBLEM SOLVING WITH C

## Bit fields in C

### Bit field Creation

- **Syntax:** struct [tag] {        type [member\_name] : width ;        };

**type** - Determines how a bit-field's value is interpreted. May be int, signed int, or unsigned int

**member\_name** - The name of the bit-field

**width** - Number of bits in the bit-field. The width must be less than or equal to the bit width of the specified type. The largest value that can be stored for an unsigned int bit field is  $2^n - 1$ , where n is the bit-length

- **Example:** struct Status {

```
    unsigned int bin1:1; // 1 bit is allocated for bin1. only two digits can be stored 0 and 1
```

```
    unsigned int bin2:1; // if it is signed int bin1:1 or int bin1:1, one bit is used to represent the sign
```

```
};
```

- Coding examples



# PROBLEM SOLVING WITH C

## Bit fields in C

---



### Few points on Bit fields

- The first field always starts with the first bit of the word.
- Cannot extract the address of bit field
- Should be assigned values that are within the range of their size. It is implementation defined to assign an out-of-range value to a bit field member
- Cannot have pointers to bit field members as they may not start at a byte boundary
- Array of bit fields not allowed
- Storage class cannot be used on bit fields
- Can use bit fields in union
- Unnamed bit fields results in forceful alignment of boundary



# THANK YOU

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

[sindhurpai@pes.edu](mailto:sindhurpai@pes.edu)



# DATA STRUCTURES AND ITS APPLICATIONS

---

**Shylaja S S & Kusuma K V**  
Department of Computer Science  
& Engineering

# DATA STRUCTURES AND ITS APPLICATIONS

---

## ArrayList, Linked List Introduction

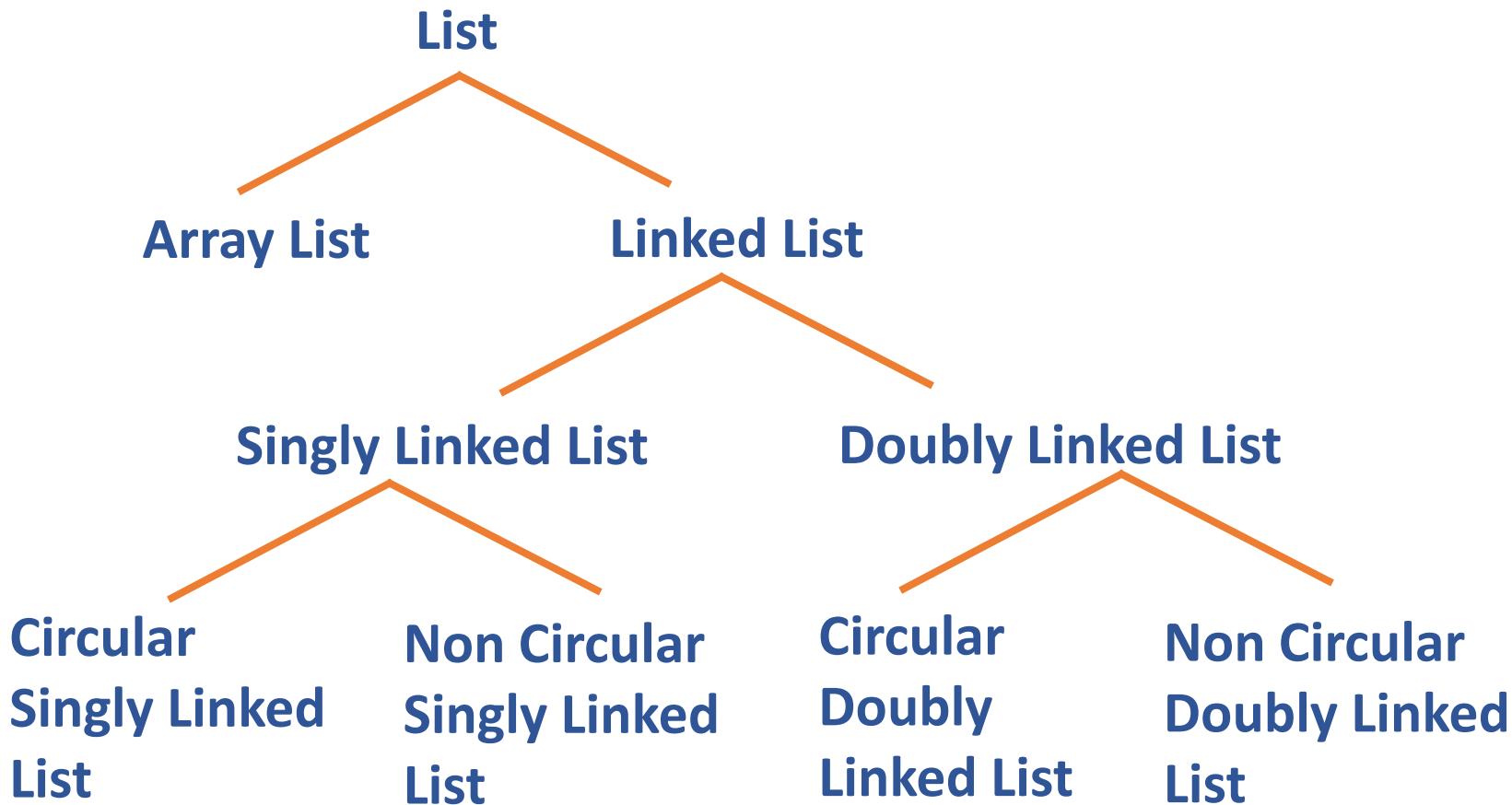
**Shylaja S S & Kusuma K V**

Department of Computer Science & Engineering

## List as a Data Structure

---

### Possible Implementations of List

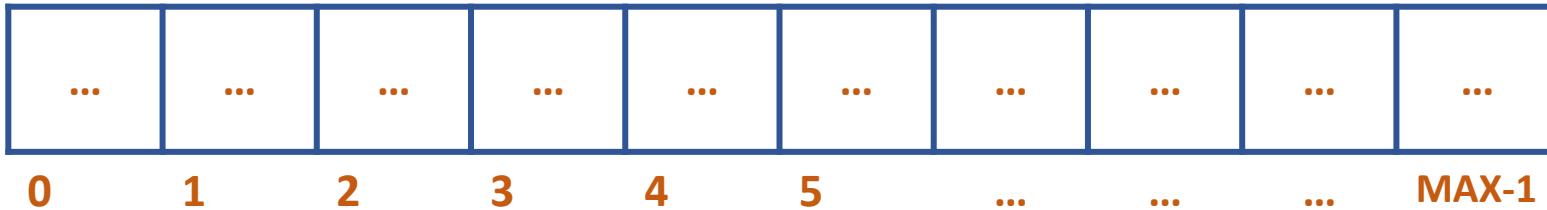


## List: ArrayList Implementation

---

Let us consider a List of Integers

Container: Array



- Append
- DeleteLast
- Insert at Position
- Delete at Position

# DATA STRUCTURES AND ITS APPLICATIONS

## List: ArrayList Implementation

---

```
typedef struct ArList
```

```
{
```

```
    int a[MAX];
```

```
    int last;
```

```
}ARLIST;
```

where MAX is a constant

last = -1 indicates list is empty



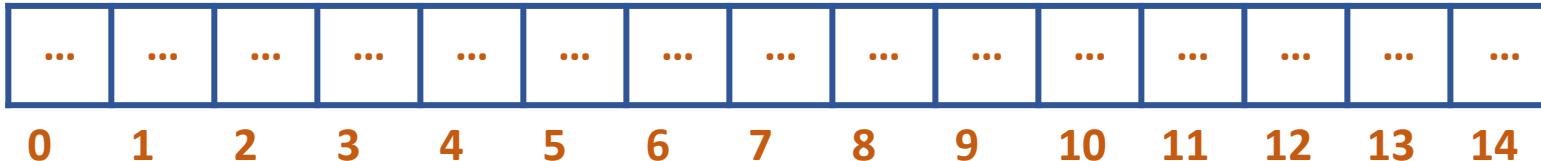
# DATA STRUCTURES AND ITS APPLICATIONS

## List: ArrayList Implementation

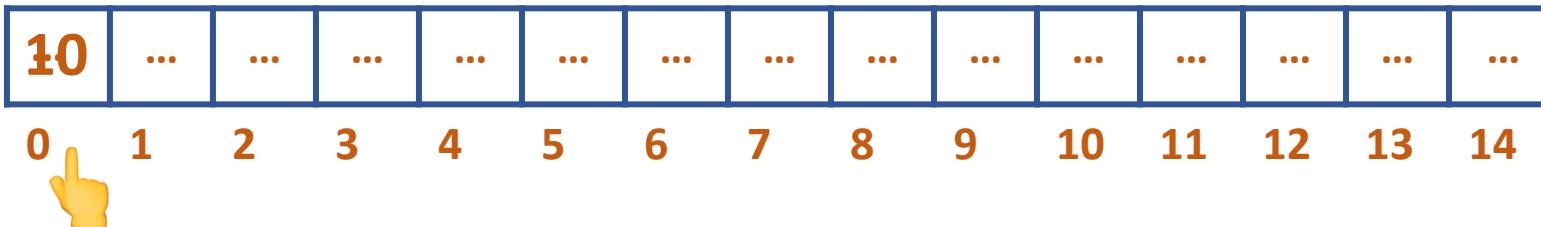
### Append

int a[MAX]; Let MAX = 15

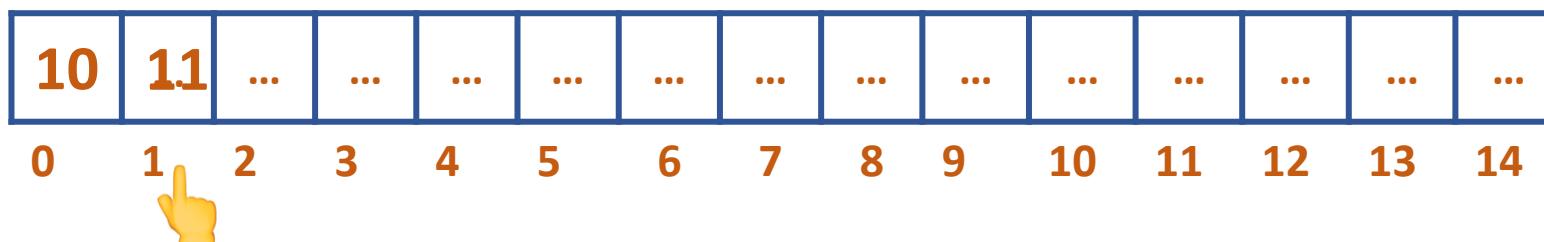
Initially: last = -1



Append 10: last = 0



Append 11: last = 1



# DATA STRUCTURES AND ITS APPLICATIONS

## List: ArrayList Implementation

### DeleteLast

int a[MAX]; Let MAX = 15

Now: last = 4

10	11	12	13	14	...	...	...	...	...	...	...	...	...	...	...
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	



DeleteLast : \*pe = 14 last = 3

10	11	12	13	14	...	...	...	...	...	...	...	...	...	...	...
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	



DeleteLast : \*pe = 13 last = 2

10	11	12	13	...	...	...	...	...	...	...	...	...	...	...	...
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	



# DATA STRUCTURES AND ITS APPLICATIONS

## List: ArrayList Implementation

---

### Insert at Position

int a[MAX]; Let MAX = 15

10	11	12	13	14	16	17	18	19	20	...	...	...	...	...
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

If I want to insert 15 at position 5 in the array

10	11	12	13	14	15	17	18	19	20	...	...	...	...	...
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

# DATA STRUCTURES AND ITS APPLICATIONS

## List: ArrayList Implementation

---

### Delete at Position

int a[MAX]; Let MAX = 15

10	11	12	13	14	15	16	17	18	19	20	...	...	...	...
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

If I want to delete an element at position 5 in the array

10	11	12	13	14	15	16	17	18	19	20	...	...	...	...
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

## Linked List: Introduction

---

Why Linked List??

ArrayList: Random Insertion/Deletion

int a[1000]; //Let us consider array is filled till position 998

10	11	12	13	14	15	16	...	...	...	...	...	...	...	...	650	700	...
0	1	2	3	4	5	6	...	...	...	...	...	...	...	997	998	999	



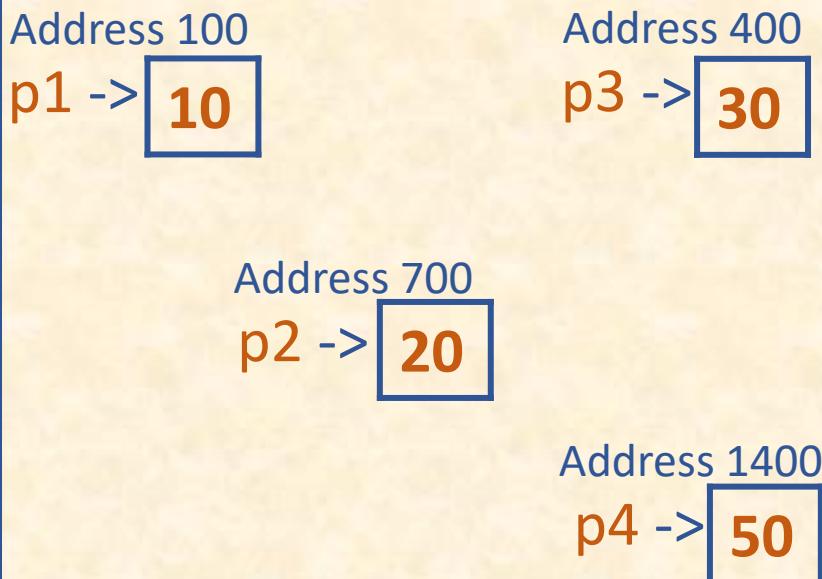
If I want to insert/delete an element at position 5 in the array ???

**Insertion:** Shift all the elements from position 998 till position 5

by one place to the right and Insert element at position 5

**Deletion:** Delete (say Copy) element at position 5 and shift all the elements from position 6 till position 998 by one place to the left

Alternative Approach: Dynamic allocation of memory to store only one integer as and when required



```
int *p[50];
```

Again size is fixed!!!

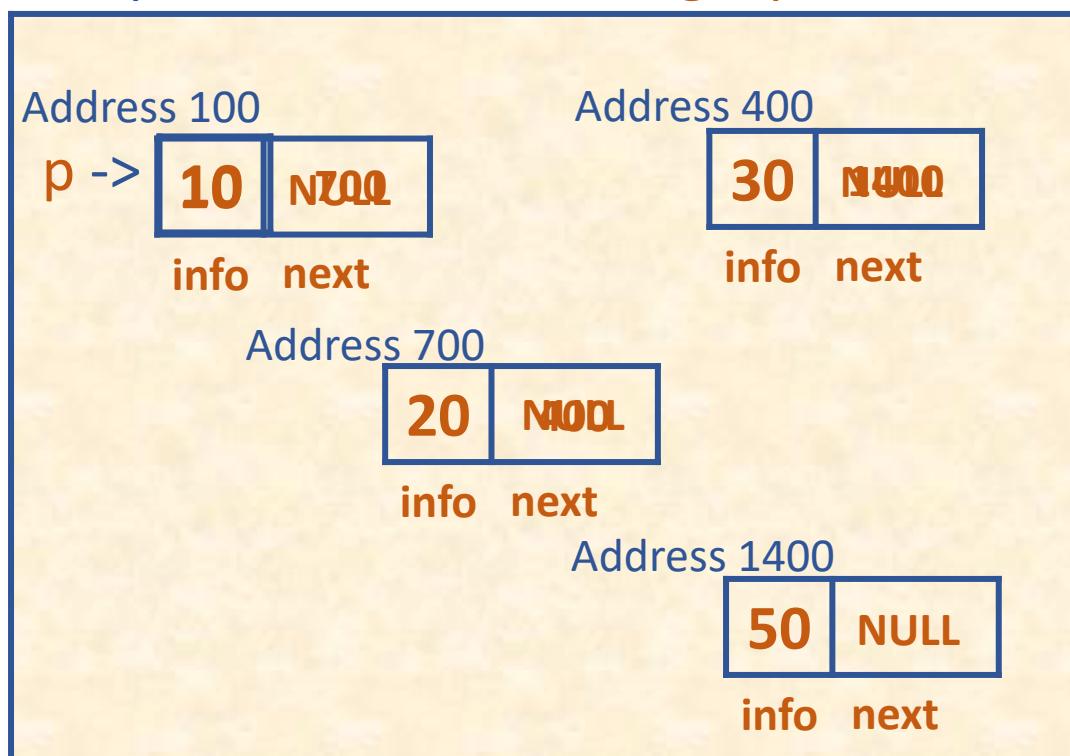
But we need a place holder for explicit storage of the address of these elements !!

# DATA STRUCTURES AND ITS APPLICATIONS

## Linked List: Introduction

Solution: Along with the data, allocate memory to pointer also dynamically

```
int *p; //start with a single pointer
```



Now we have a  
Linked List with  
4 nodes !!!

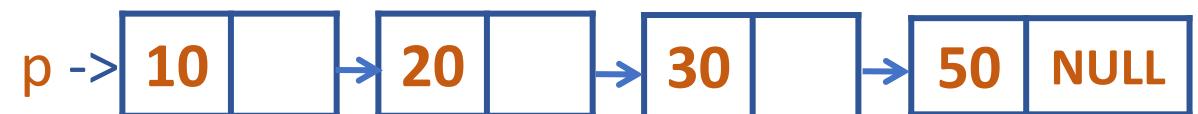
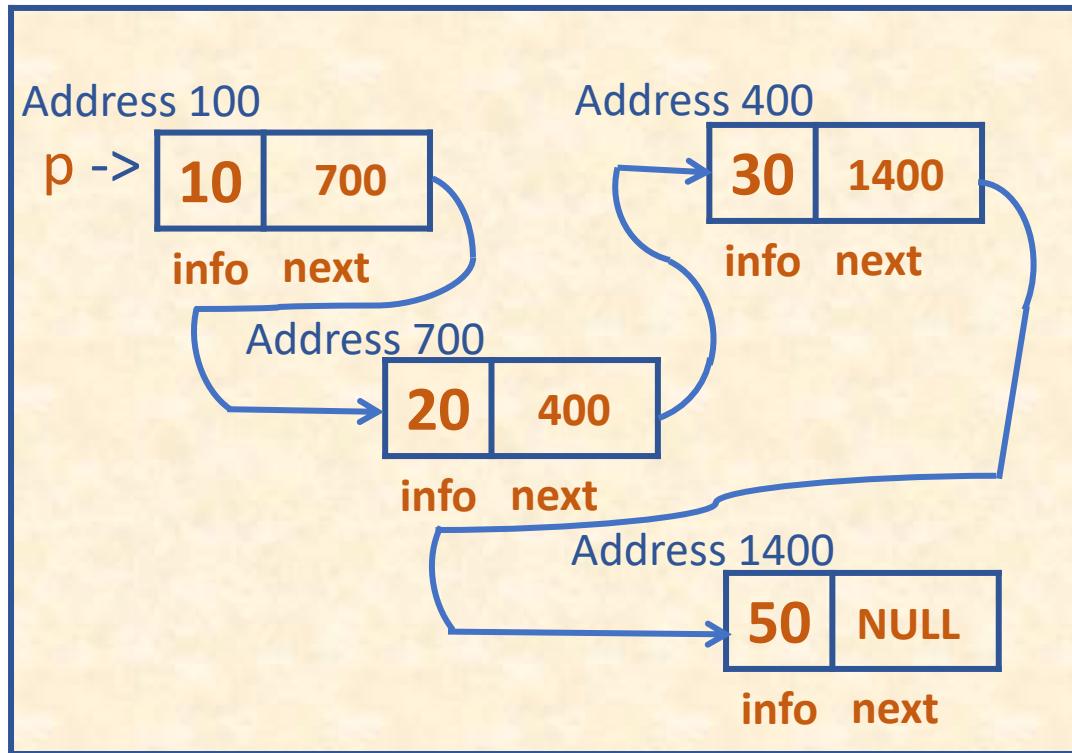
Note: The data type of p is now NODE \*p;  
Refer to Slide 26 to see what is NODE !!

# DATA STRUCTURES AND ITS APPLICATIONS

## Linked List: Introduction

Solution: Along with the data, allocate memory to pointer also dynamically

```
int *p; //start with a single pointer
```



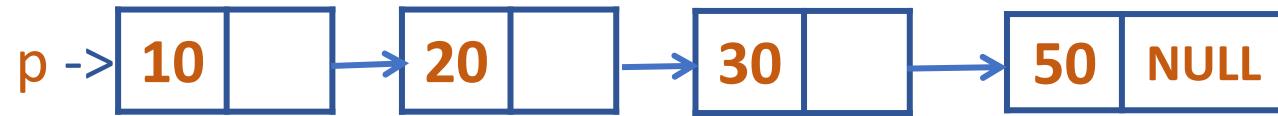
Now we have a  
Linked List with  
4 nodes !!!

Note: The data type of p is now NODE \*p;  
Refer to Slide 26 to see what is NODE !!

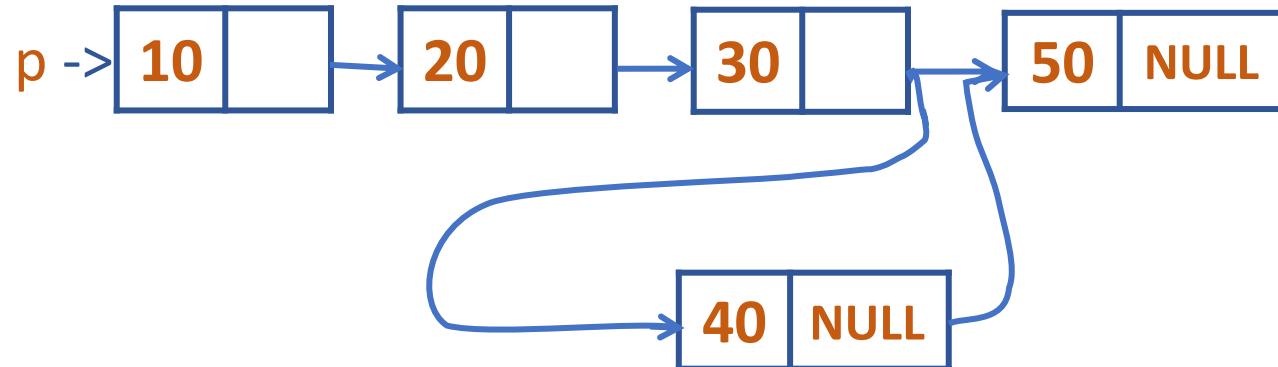
# DATA STRUCTURES AND ITS APPLICATIONS

## List as a Data Structure: Linked List

### Linked List: Random Insertion



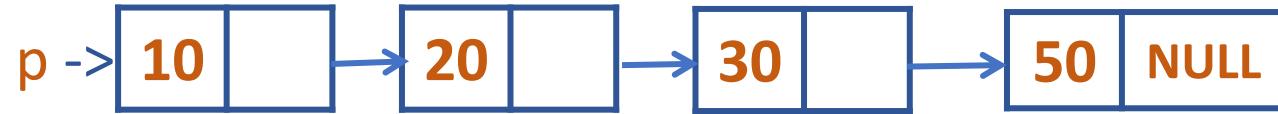
Insert 40 between 30 and 50



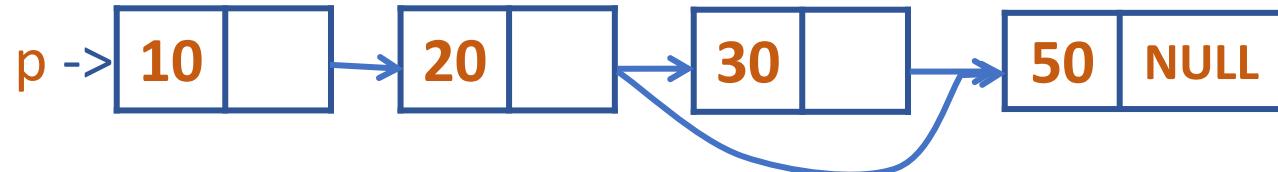
# DATA STRUCTURES AND ITS APPLICATIONS

## List as a Data Structure: Linked List

### Linked List: Random Deletion



Delete 30



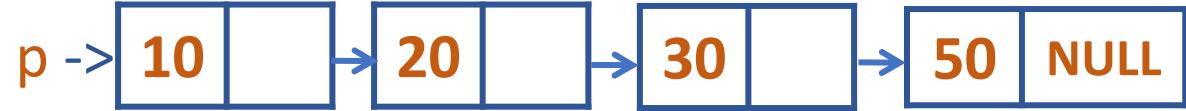
## Array List and Linked List

---

- Random Access
  - ArrayList is better than LinkedList
- Random Insertion/Deletion
  - LinkedList is better than ArrayList
- Size
  - Fixed in ArrayList and Dynamic in LinkedList

# DATA STRUCTURES AND ITS APPLICATIONS

## Node Structure



Structure of the Node object //self referential structure

typedef struct node

{

    int info;

    struct node \*next;

}NODE;



# **PROBLEM SOLVING WITH C**

## **UE22CS151B**

---

**Prof. Sindhu R Pai and Prof. Kusuma K V**  
Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

---

## File Handling in C

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

## File Handling

---



1. Points to Discuss
2. Introduction
3. Need of Files
4. File Classification
5. Operations on Files
6. Read/Write Operations on Files

# PROBLEM SOLVING WITH C

## File Handling

---



### Points to Discuss!!!

- Can we use the data entered by the user in one program execution, in another program execution without asking the user to enter again?
- How to generate same input several times?
- How to store the output produced for future references?
- Program stores the result what if one wants to store other data too?
- How to categorize and manage forms of data produced?

# PROBLEM SOLVING WITH C

## File Handling

---



### Introduction

- Variables used in program will die at the end of execution
- To persist data even after the program execution is complete, use Files
- File represents a sequence of bytes and it is a source of storing information in sequence of bytes on a disk
- The data in a file can be structured or unstructured
- A program in C can itself be the data file for another program
- The keyboard and the output screen are also considered as files

# PROBLEM SOLVING WITH C

## File Handling

---



### Need of Files

- When a program is terminated, the entire data is lost. Storing in a data file will preserve the data even if the program terminates
- If the data is too large, a lot of time spent in entering them to the program every time the code is run
- If stored in a data file, easier to access the contents of the data file using few functions in C

# PROBLEM SOLVING WITH C

## File Handling



### File Classification

- **Text File**

- Contains textual information in the form of alphabets, digits and special characters or symbols
- Created using a text editor

- **Binary File**

- Contain bytes or a compiled version of a text file i.e. data in the form of 0's and 1's
- Can store larger amount of data that are not readable but secured.

# PROBLEM SOLVING WITH C

## File Handling



### Operations on Files

- Reading the contents of the file
- Writing the contents to the file

**Note:** To perform any operation on Files, **The physical filename, the logical filename and the mode** must be connected using **fopen()**

- **Physical Name:** A file is maintained by the OS. The OS decides the naming convention of a file
- **Logical Name:** In a C Program, identifier is used to refer to a file. Also called as **File Handle**
- **Mode:** Can be read only, write only, append or a combination of these.

# PROBLEM SOLVING WITH C

## File Handling

### C Functions to perform Operations

- Creation of new file or open an existing file using **fopen()**
- Read operation on a file using **fgetc()**, **getc()**, **fscanf()**, **fgets()** and **fread()**
- Write operation on a file using **fputc()**, **putc()**, **fprintf()**, **fputs()** and **fwrite()**
- Moving to a specific location in a file using **fseek()** and **rewind()**
- Knowing the location of a file pointer using **ftell()**
- Closing a file using **fclose()**

# PROBLEM SOLVING WITH C

## File Handling



# PROBLEM SOLVING WITH C

## File Handling

---



### fclose()

- Closes the stream. All buffers are flushed
- Returns zero if the stream is successfully closed. On failure, EOF is returned
- All links to the file are broken
- Misuse of files is prevented

**Syntax:** `int fclose(file_pointer);`

- File pointer can be reused
- Coding Examples

# PROBLEM SOLVING WITH C

## File Handling

### Read /Write operations on File

---

- Categories
  - Character read/write
    - fputc() , fgetc(), getc() and putc().
  - String read/write
    - fgets() and fputs().
  - Formatted read/write
    - fscanf() and fprintf().
  - Block read/write
    - fread() and fwrite()

# PROBLEM SOLVING WITH C

## File Handling

---

### Character I/O operations on File

- Reads a character from the file and increments the file pointer position.
  - **Syntax:** `int fgetc(FILE *fp);`
  - **Return Value:** Next byte from the input stream on success, EOF on error.
- Write operation at current file position and increments the file pointer position.
  - **Syntax:** `int fputc(int c,FILE *fp);`
  - **Return Value:** Character that is written on success, EOF on error.
- `getc()` and `putc()` are equivalent to `fgetc()` and `fputc()` respectively, except that they may be implemented as macros.
- Coding Examples

# PROBLEM SOLVING WITH C

## File Handling



### String I/O Operations on File(fgets() and fputs())

- Reads a line of characters from file and stores it into the string pointed to by char\_array variable. It stops when either (n-1) characters are read, the newline character is read, or the end-of-file is reached, whichever comes first
  - **Syntax:** `char* fgets(char *char_array, int n, FILE *stream);`
  - **Return Value:** Pointer to the string buffer on success, NULL on EOF or Error.
- Write a line of characters to a file.
  - **Syntax:** `int fputs(const char *s, FILE *stream);`
  - **Return Value:** A non-negative number on success, EOF on error.
- Coding examples

# PROBLEM SOLVING WITH C

## File Handling

### Formatted Read/Write Operations on File(fscanf(),fprintf())

- Reads the formatted data from the file instead of standard input.
  - **Syntax:** `int fscanf(FILE *fp,const char *format[,address,.....]);`
  - **Return Value:** The number of **values read** on success , **EOF** on failure.
- Writes the formatted data to a file instead of standard output.
  - **Syntax:** `int fprintf(FILE *fp, const char *format[,argument,....]);`
  - **Return value:** The number of **characters written** on success , **EOF** on failure.
- Coding Examples

# PROBLEM SOLVING WITH C

## File Handling

### Block Read/Write Operations on a File

- Reads an entire block from a given file.
  - **Syntax:** `size_t fread(void *p, size_t size, size_t n, FILE *fp);`
  - **Return Value:** On success, it reads n items from the file and returns n. On error or end of the file, it returns a number less than 'n'.
- Writes an entire block to the file. `fwrite()` function writes the data to the file stream in the form of binary data block.
  - **Syntax:** `size_t fwrite(const void *p, size_t size, size_t n, FILE *fp);`
  - **Return Value:** On success, it returns the count of the number of items successfully written to the file, on error, it returns a number less than 'n'.

# PROBLEM SOLVING WITH C

## File Handling

---

### Random access to a file

- The **fseek()** function sets the file position indicator for the stream pointed to by *stream*.  
The new position, measured in bytes, is obtained by adding *offset* bytes to the position specified by *whence*.
- If *whence* is set to **SEEK\_SET**, **SEEK\_CUR**, or **SEEK\_END**, the offset is relative to the start of the file, the current position indicator, or end-of-file, respectively.
- **SEEK\_SET**, **SEEK\_CUR**, or **SEEK\_END** can be replaced with the values 0, 1 and 2 respectively.
- **Syntax:** `int fseek( FILE* pointer, long offset, int whence);`
- **Return Value:** zero if successful, or else it returns a non-zero value.

# PROBLEM SOLVING WITH C

## File Handling

---



### Random access to a file

- The **fseek()** function obtains the current value of the file position indicator for the stream pointed to by *pointer*.
  - **Syntax:** `int ftell(FILE* pointer);`
  - **Return Value:** 0 or a positive integer on success and -1 on error.
- The **rewind()** function sets the file position indicator for the stream pointed to by *pointer* to the beginning of the file. It is equivalent to: `(void) fseek(pointer, 0L, SEEK_SET)` except that the error indicator for the stream is also cleared.
- **Syntax:** `void rewind(FILE* pointer);` // Function does not return anything.



**THANK YOU**

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

**sindhurpai@pes.edu**



# PROBLEM SOLVING WITH C

## UE21CS141B

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

---

## File Handling in C continued..

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

## File Handling

---



1. Common Programming Errors
2. Error Handling
3. Demo of Errors
4. Best Practices

# PROBLEM SOLVING WITH C

## File Handling

---



### Common Programming Errors that occur during File I/O

- Open with invalid filename
- Operation on a file that has not been opened
- Operation not permitted by ‘fopen’
- Write to a write-protected file

# PROBLEM SOLVING WITH C

## File Handling

---

### Error Handling

- Not supported by C directly and is accomplished by using **errno.h**
- When a function is called, a global variable named as **errno** is automatically assigned a value used to identify the type of error that has been encountered

errno value	Type of Error
1	Operation not permitted
2	No such file or directory
5	I/O error
7	Arg list too long
9	Bad file descriptor
11	Try again
12	Out of memory
13	Permission denied

- Coding Examples

# PROBLEM SOLVING WITH C

## File Handling

---

### Error Handling continued..

- **Two library functions** are used to prevent performing any operation beyond EOF
- **feof()**: Used to test for an end of file condition
  - **Syntax:** `int feof(FILE *file_pointer);`
  - **Return Value:** Non-zero if EOF, Else zero
- **ferror()**: Used to check for the error in the stream
  - **Syntax:** `int ferror (FILE *file_pointer);`
  - **Return Value:** Non-zero if error occurs, Else zero
- The error indication will last until the file is closed(`fclose()`) or cleared by the `clearerr()` function.

# PROBLEM SOLVING WITH C

## File Handling



### Demo of Error Handling

- C code demonstration of Error handling in File I/O

# PROBLEM SOLVING WITH C

## File Handling



### Best Practices

- Given a file pointer check whether it is NULL before proceeding with further operations
- Use errno.h and global variable errno to know the type of error that occurred. Usage of strerror() and perror() helps in providing textual representation of the current errno value
- Good to check whether EOF is reached or not before performing any operation on the file



**THANK YOU**

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

**sindhurpai@pes.edu**



# PROBLEM SOLVING WITH C

## UE23CS151B

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

---

## Problem Solving: File Handling

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

## Problem Solving: File Handling

---



1. Data file Description
2. Problem Statements
3. Requirements to solve the Problem
4. Demo of C Solution

# PROBLEM SOLVING WITH C

## Problem Solving: File Handling

---



### Data File Description

- Data file to be used: **matches.csv**
- Contains the information of cricket matches held between 2008 and 2016
- First row represents the column headers such as id, season, city, date, team1, team2, toss\_winner, toss\_decision, result, dl\_applied, winner, win\_by\_runs, win\_by\_wickets, player of match, venue, umpire1, umpire2 and umpire3
- Contains around 577 rows. Every data is stored with a comma in between

# PROBLEM SOLVING WITH C

## Problem Solving: File Handling

---



### Problem Statements

- **Count the number of matches played in the year 2008 – Solution: 58**
- Count the number of times the Toss winner is same as the Winner of the match
- Display the count of matches played between KKR and RCB
- Display the Winner of each match played in 2016
- Display the list of Player of the match when there was a match between RCB and CSK in the year 2010
- The output of all the above can be written to a file to store the record of outputs in one file

# PROBLEM SOLVING WITH C

## Problem Solving: File Handling

### Requirements to solve the problem



- Reading the csv file – Two ways

Read character by character, count the number of commas, then extract the field after the second comma till third comma.

Read the whole line using fgets and split the line based on comma using a function '**strtok**'

- **Strtok:** Function returns a pointer to the first token found in the string. A NULL pointer is returned if there are no tokens left to retrieve.

Two arguments – **A source string or NULL and the Delimiter string.**

First time strtok is called, the string to be split is passed as the first argument. In subsequent calls, NULL is passed to indicate that strtok should keep splitting the same string for the next token.

# PROBLEM SOLVING WITH C

## Problem Solving: File Handling

---

### Demo of C Solution

- Demonstration of C Code





**THANK YOU**

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering



# PROBLEM SOLVING WITH C

## UE23CS151B

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

---

## Searching

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

## Searching

---



1. Points for Discussion
2. Introduction
3. Searching Algorithms
4. Linear Search
5. Binary Search - Pictorial Representation and Implementation

# PROBLEM SOLVING WITH C

## Searching

---



### Points for Discussion!

- Have you spent a day without searching for something?
- Finding a particular item among many hundreds, thousands, millions or more.
  - Scenario: Finding someone's phone number in our phone
- What if one wants to save the time consumed in looking to each item in a collection of items?

# PROBLEM SOLVING WITH C

## Searching

---



### Introduction

- Identifying or finding a particular record/item/element in a collection of records/items/elements and knowing the place of it
- Collection/Group where searching must be done may be sorted or unsorted
- Search may be Successful search or Unsuccessful based on the availability of the record/item/element in a collection

# PROBLEM SOLVING WITH C

## Searching

---



### Searching Algorithms

- Random Search
- Sequential or Linear search
- Non - Sequential or Binary Search.

### Linear Search

- Performs search on any kind of data
- Starts from 0<sup>th</sup> item till the end of the collection



One-Dimensional Array having 7 Elements

- Coding Example

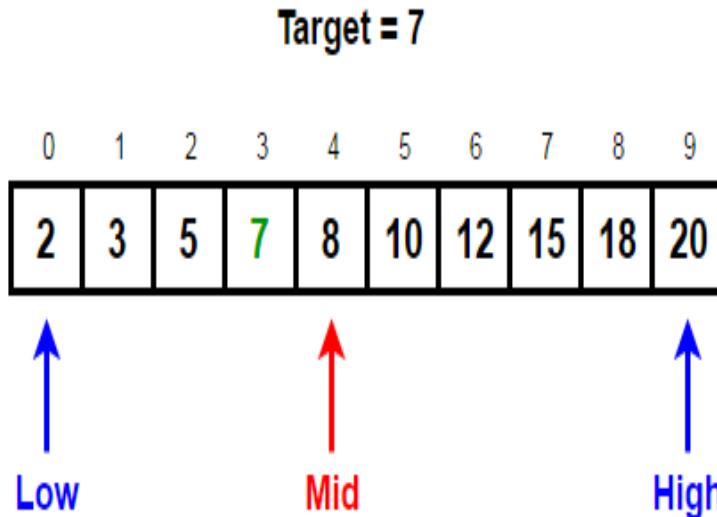
### Binary Search

- Necessary condition: **Collection of data should be sorted**
- **Begins comparison at the middle** of the collection
  - If matched, return the index of the middle element
  - If not matched, check whether the element to be searched is lesser or greater than the middle element
- If the element to be searched is greater than the middle element, pick the elements on the right side of the middle element and repeat from the start
- If the element to be searched is lesser than the middle element, pick the elements on the left side of the middle element and repeat from the start

# PROBLEM SOLVING WITH C

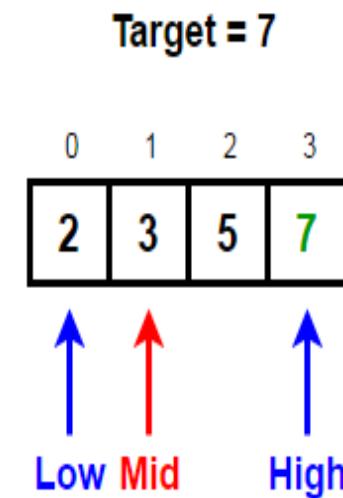
## Searching algorithms

### Pictorial Representation of Binary Search



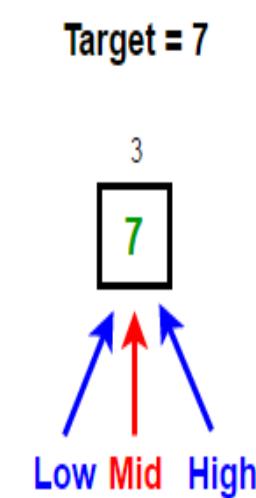
Since 8 (Mid) > 7 (target),  
we discard the right half and go LEFT

New High = Mid - 1



Since 3 (Mid) < 7 (target),  
We discard the left half and go RIGHT

New low = mid + 1



Now our search space consists of only one element 7. Since 7 (Mid) = 7 (target), we return index of 7 i.e. 3 and terminate our search

# PROBLEM SOLVING WITH C

## Searching algorithms



### Binary Search Implementation

Given a sorted Array A of n elements and the target value is T

#### Iterative Algorithm

1. Set L: 0 and R: n-1
2. If(L>R), Unsuccessful Search
3. Else Set m:  $(L+R)/2$  // m: position of middle element
4. If  $A_m < T$ , set L to  $m+1$  and go to step 2
5. If  $A_m > T$ , set R to  $m-1$  and go to step 2
6. If  $A_m$  is T, search done, return m

#### Recursive Algorithm

- BinarySearch(T, A)
1. Set L: 0 and R: n-1
2. If(L>R), return -1 // Unsuccessful Search
3. Else Set m:  $(L+R)/2$  // m: position of middle element
4. If  $A_m$  is T, search done, return m
5. If  $A_m < T$ , return BinarySearch( $T, A_0$  to  $A_{m-1}$ )
6. Else return BinarySearch( $T, A_{m+1}$  to  $A_{n-1}$ )

# PROBLEM SOLVING WITH C

## Searching

---



**Demo of Binary search on array of integers present in a file**



**THANK YOU**

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering



# PROBLEM SOLVING WITH C

## UE23CS151B

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering



# PROBLEM SOLVING WITH C

---

## Problem Solving: Sorting using Array of Pointers

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

## Problem Solving: Sorting using Array of Pointers

---



1. Data file Description
2. Problem Statement
3. Requirements to solve the Problem
4. Demo of C Solution

# PROBLEM SOLVING WITH C

## Problem Solving: Sorting using Array of Pointers

---



### Data File Description

- Data file to be used: **student.csv**
  - Contains the information of students in two columns only
  - First row represents the column headers such as roll\_no and name
  - Contains around 55 rows. Every data is stored with a comma in between

# PROBLEM SOLVING WITH C

## Problem Solving: Sorting using Array of Pointers

---



### Problem Statement

- Create a menu driven program to perform bubble sort based on roll\_number and name. Use student.csv to extract the dataset and write the sorted record to a new file.

# PROBLEM SOLVING WITH C

## Problem Solving: Sorting using Array of Pointers

---



### Requirements to solve the problem

- Creating the student type with two data members – roll\_no and name
- Reading the csv file line by line
- Splitting the line based on the delimiter using strtok function
- Copying the data from the data file to the data members of the student structure
- Client code requires menu driven code - Provide the option to the user to sort based on roll\_no or name
- Perform bubble sort based on the option from the user
- Create a new file using fopen and sorted record write to a new file(roll\_no+name)

# PROBLEM SOLVING WITH C

## Problem Solving: Sorting using Array of Pointers

---

### Demo of C Solution

- Demonstration of C Code





**THANK YOU**

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering



# PROBLEM SOLVING WITH C

## UE23CS151B

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

---

## Callback in C

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

## Callback in C

---



1. Points to Discuss
2. Introduction
3. Function Pointer/ Pointer to function
4. Demo of C Callback

# PROBLEM SOLVING WITH C

## Callback in C

---



### Points to Discuss!!!

- How to extend the features of one function using another one? - What is the method used if one function communicates with the other through parameter?
- How to have one common method to develop libraries and event handlers for many programming languages?
- How redirect page action is performed for the user based on one click action?

### Introduction

- Any executable code that is passed as an argument to other code, which is expected to call (execute) the argument at a given time.
- In simple language, if a function name is passed to another function as an argument to call it, then it will be called as a Callback function.
- A callback function has a specific action which is bound to a specific circumstance.
- A callback function is an important element of GUI in C programming
- In C, a callback function is a function that is called through a function pointer/pointer to a function.

# PROBLEM SOLVING WITH C

## Callback in C

---



### Function Pointer/ pointer to function.

- Points to a code, not data and it stores the start of the executable code
- Used in reducing the redundancy
- Think about the difference between below statements

```
int *a1(int, int, int); // a1 is a function which takes three int arguments and returns a pointer to int.
```

```
int (*p)(int, int, int); // p is a pointer to a function which takes three int as parameters and returns an int
```

- Coding Examples

# PROBLEM SOLVING WITH C

## Callback in C

---

### Demo of C Callback



- Simple Coding example to demo Callback



# THANK YOU

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering



# PROBLEM SOLVING WITH C

## UE23CS151B

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

---

## Problem Solving: Callback

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

## Problem Solving: Callback

---



1. Problem Statement
2. Requirements to solve the problem
3. Demo of C Solution

# PROBLEM SOLVING WITH C

## Problem Solving: Callback

---



### Problem Statements

- Perform Binary search on a collection of elements. Client is adding constraints every now and then . Initial constraints to perform search are as follows.
  - i) Display only if the number is even
  - ii) Display only the number is less than 22
- Create a menu driven code to perform Bubble sort on a collection of students using an array of pointers. Use student.csv as the data file to process. Right now there are 2 headers in this csv file: Roll\_no and Name.
  - Perform sort based on Roll\_no
  - Perform sort based on Name

Anytime client might add more information about students to the data file and requirement might change to sort based on that added info.

# PROBLEM SOLVING WITH C

## Problem Solving: Callback

---



### Requirements to solve the problem

- Create a type called **student** with two data members: **roll\_no** and **name**
- **Read csv file and copy the data to array of structures of student type**
- **Initialize the Array of pointers with the address of each structure in the array**
- Bubble sort function must be modified to have **function pointer parameter**. This will be called inside the bubble sort function based on the choice from the client/user.
- Create two function definitions separately to compare two **roll\_numbers** and **names**.
- Pass these functions to the bubble sort as an argument based on the choice from the client/user.

# PROBLEM SOLVING WITH C

## Problem Solving: Callback

---

### Demo of C Solution



- Demo of Bubble sort using callback



**THANK YOU**

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering



# PROBLEM SOLVING WITH C

## UE23CS151B

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

---

## Qualifiers in C

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

## Qualifiers in C

---



1. Introduction
2. Size Qualifiers
3. Sign Qualifiers
4. Type Qualifiers
5. Applicability of Qualifiers to Basic Types

# PROBLEM SOLVING WITH C

## Qualifiers in C

---

### Introduction

- Keywords which are applied to the data types resulting in Qualified type
- Applied to basic data types to alter or modify its sign or size
- Types of Qualifiers
  - **Size Qualifiers**
  - **Sign Qualifiers**
  - **Type qualifiers**



### Size Qualifiers

- Prefixed to **modify the size of a data type** allocated to a variable
- Supports two size qualifiers, **short** and **long**
- Rules Regarding size qualifier as per ANSI C standard
  - **short int <= int <=long int //** short int may also be abbreviated as short and long int as long. But, there is no abbreviation for long double.
- Coding Examples

### Sign Qualifiers

- Used to specify the signed nature of integer types
- It specifies whether a variable can hold a negative value or not
- Sign qualifiers are used with int and char types
- There are two types of Sign Qualifiers in C. **Signed** and **Unsigned**
- A **s signed qualifier** specifies a variable which can hold both positive and negative integers
- An **unsigned qualifier** specifies a variable with only positive integers
- Coding Examples

# PROBLEM SOLVING WITH C

## Qualifiers in C

---



### Type Qualifiers

- A way of expressing additional information about a value through the type system and ensuring correctness in the use of the data
- Type Qualifiers consists of two keywords
  - **const**
  - **volatile.**

# PROBLEM SOLVING WITH C

## Qualifiers in C

### Type Qualifiers continued..



- **const:**
  - Once defined, their values cannot be changed.
  - Called as literals and their values are fixed.
  - **Syntax: const data\_type variable\_name**
- **volatile:**
  - Intended to prevent the compiler from applying any optimizations
  - Their values can be changed by code outside the scope of current code at any time
  - Also can be changed by any external device or hardware
  - **Syntax: volatile data\_type variable\_name**
- Coding Examples

# PROBLEM SOLVING WITH C

## Qualifiers in C

### Applicability of Qualifiers to Basic Types



No.	Data Type	Qualifier
1.	char	signed, unsigned
2.	int	short, long, signed, unsigned
3.	float	No qualifier
4.	double	long
5.	void	No qualifier



**THANK YOU**

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering



# PROBLEM SOLVING WITH C

## UE23CS151B

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

---

## Preprocessor Directives

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

## Preprocessor Directives

---



- Introduction
- Types of Pre-processor Directives
- Macros
- Points to know about Macros
- Predefined Macros
- Macro vs Enum
- File Inclusion Directives
- Conditional compilation Directives
- Other Directives

### Introduction

- **The lines of code** in a C program which are **executed by the ‘C’ pre-processor**
- A **text substitution tool** and instructs the compiler to do required pre-processing before the actual compilation
- All pre-processor commands begin with a hash symbol (#)
- It must be the first nonblank character and for readability, pre-processor directive should begin in the first column conventionally

# PROBLEM SOLVING WITH C

## Preprocessor Directives

---



### Types of Preprocessor Directives

- Types include:
  - Macros
  - File Inclusion
  - Conditional Compilation
  - Other directives
- Possible to see the effect of the pre-processor on source files directly by using the **-E option of gcc**

### Macros

- A piece of code in a program which has been given a name
- During preprocessing, it substitutes the name with the piece of code
- **#define** directive is used to define a macro.
- Example: `#define PI 3.14`
- Coding Examples

### Points to know about Macros

- Macro **does not** judge anything
- **No memory Allocation** for Macros
- Can define string using macros
- Can define macro with expression
- Can define macro with parameter
- Macro can be used in another macro
- Constants defined using #define cannot be changed using the assignment operator
- Redefining the macro with #define is allowed. But not advisable

# PROBLEM SOLVING WITH C

## Preprocessor Directives

---

### Predefined Macros



S.no	Macro & Description
1	<b><u>DATE</u></b> The current date as a character literal in "MMM DD YYYY" format.
2	<b><u>TIME</u></b> The current time as a character literal in "HH:MM:SS" format.
3	<b><u>FILE</u></b> This contains the current filename as a string literal.
4	<b><u>LINE</u></b> This contains the current line number as a decimal constant.
5	<b><u>STDC</u></b> Defined as 1 when the compiler complies with the ANSI standard.

### Macro vs Enum

- Macro doesn't have a type and enum constants have a type int.
- Macro is substituted at pre-processing stage and enum constants are not.
- Macro can be redefined using #define but enum constants cannot be redefined. However assignment operator on a macro results in error

### File Inclusion Directives

- Instructs the pre-processor to include a file in the program using **#include** directive
- Two types of files
  - **Header File or Standard files: Included between < and >**
    - Contains the definition of pre-defined functions like printf(), scanf() etc.
    - To work with these functions, header files must be included
  - **User defined files: Included using “ and “**
    - When a program becomes very large, it is good practice to divide it into smaller files and include whenever needed.
    - Adding user defined header files in the program

### Conditional Compilation Directives

- A few blocks of code will be compiled in a particular program based on the result of some condition
- Conditions can be mentioned using - **#ifdef, #ifndef, #if, #else, #elif, #else, #endif**
- Coding Examples

### Other Directives

- **#undef Directive:** Used to undefine an existing macro
  - Example: #undef LIMIT
  - After this statement every “#ifdef LIMIT” statement will evaluate to false
- **#pragma startup and #pragma exit:** Helps to specify the functions that are needed to run before program startup and just before program exit
- **#pragma warn:** This directive is used to hide the warning messages which are displayed during compilation using **-rwl, -par and -rch**



**THANK YOU**

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering



# PROBLEM SOLVING WITH C

## UE23CS151B

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering



# PROBLEM SOLVING WITH C

---

## Portable Program Development

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering

# PROBLEM SOLVING WITH C

## Portable Program Development

---



- Problem Statement
- Requirements
- Demo of C Solution

# PROBLEM SOLVING WITH C

## Portable Program Development

---



### Problem Statement

- Demonstrate the real use of conditional compilation and pre-defined macros. Make the code portable across different platforms. Based on the current platform the code is run, particular part of the code must be compiled and executed

# PROBLEM SOLVING WITH C

## Portable Program Development

---



### Requirements

- Part of the code to be compiled and executed depends on the macro set or not
    - If **mingw in windows system**, the value of `_MINGW32_` will be 1
    - If **unix/Linux system** is used, the value of `_unix_` is 1
    - If **mac system** is used, the value of `_APPLE` is 1
- Note that there is no {} to specify the block
- No usage of ( ) to specify the macro

# PROBLEM SOLVING WITH C

## Portable Program Development

---



### Demo of C Solution

- See the output of preprocessing on the terminal using gcc –E and then execute the code



**THANK YOU**

---

**Prof. Sindhu R Pai**

Department of Computer Science and Engineering