

# Welcome to Structured Programming (CSE-141)

---



Dept. of Computer Science & Engineering

# Course Teachers

- Md. Billal Hossain  
Assistant Professor  
Dept. of CSE, CUET
- Hasan Murad  
Lecturer  
Dept. of CSE, CUET



# Course Description

---

- Title: Structured Programming
- Credits: 3.0 [3 class of 50 minutes per week]



# Course Content

- Introduction
  - Data types
  - Operators
  - Expressions
- Input and Output
  - Standard input and output
  - Formatted input and output
- Control Structures
  - Branching
  - Looping



# Course Content

- Basic Data Structure
  - 1-D array
  - Multidimensional array
  - Strings
- Functions and Program Structure
  - Parameter passing conventions
  - Scope rules and storage classes
  - Recursion
- User Defined Data Types
  - Structures
  - Unions
  - Enumerations



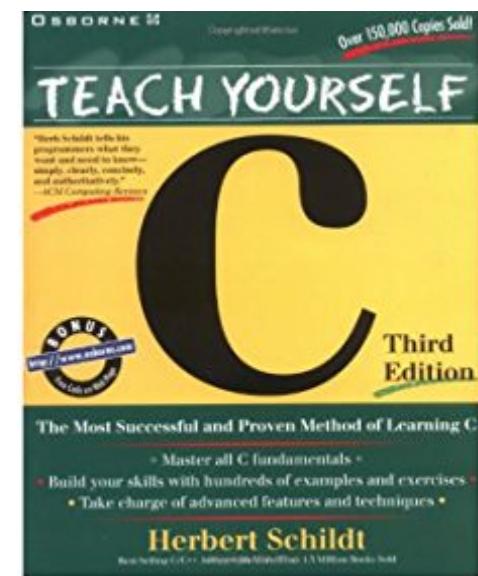
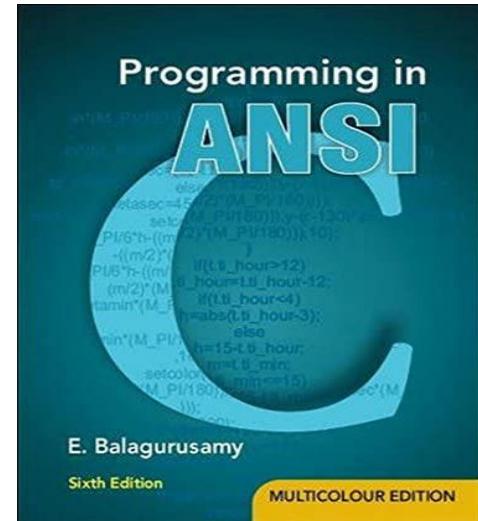
# Course Content

- Advance Topics
  - File Management
  - Error Handling
  - Variable Length Argument List
  - Command Line Parameters;
  - Header Files
  - Preprocessor
  - Linking
  - Library Functions.



# Reference Books

- Book 1
  - Title: Programming in ANSI C
  - Author: E. Balagurusamy
  - Edition: 6th
- Book 2
  - Title: Teach Yourself C
  - Author: Herbert Schildt
  - Edition: 3<sup>rd</sup>
- Note: Course teachers have the right to cover more reference books



# Marks Distribution

- Total: 300 marks
- Attendance: 10% [30 marks]
  - Less than 60% attendance results in backlog if you fail.
- Class Test: 20% [60 marks]
  - 4 class tests
  - Best 3 will be counted
- Final Exam 70% [210 marks]
  - 2 Sections [105\*2 marks]

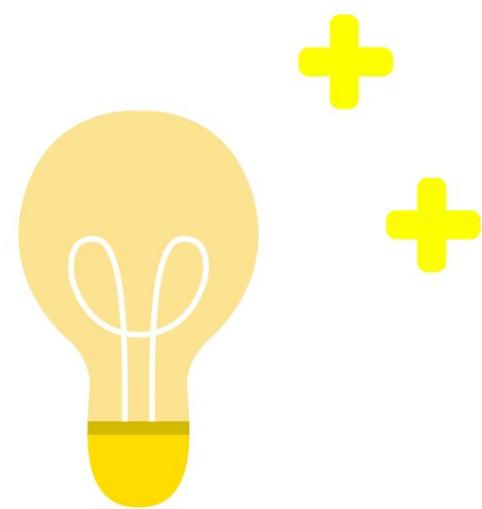


# Class Rules

- Must carry reference book
- Must join the class in time, late joining will be counted as absent
- Note down the class lecture properly
- Stay focused during the class
- Questions will be asked during the class, failing to answer will be counted as absent in the class
- **Mobile must be kept silent**
- All class rules will be available at classroom



# HAPPY C CODING



CSE-141  
Structured Programming  
(Sessional)

## Lecture: 1

# Introduction to Programming language



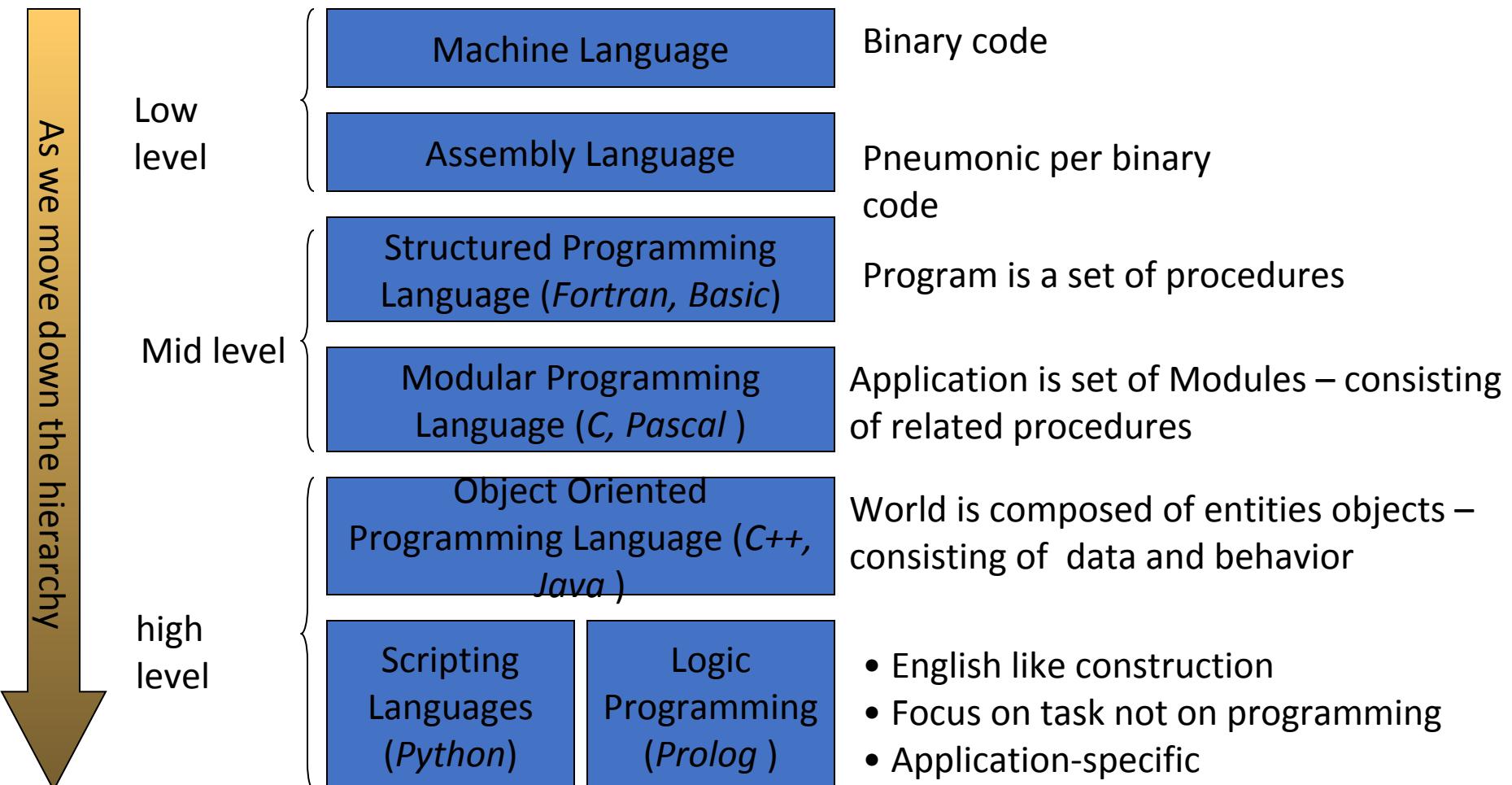
Department of CSE  
**CUET**

# Learning Resources

- Teach Yourself C - Herbert Schildt



# Programming level



# Programming level example

C code	Intel 80x80 Assembly code	Intel 80x80 Machine Code
int m, n, p;  m = n + p;	MOV AX, n  ADD AX, p  MOV m, AX	A1 4828  01 06 482A  A3 4826



# Generations of Programming languages

- **The first generation languages, or 1GL:** Low-level languages that are machine language
- **The second generation languages, or 2GL:** Also low-level languages that generally consist of assembly languages
- **The third generation languages, or 3GL:** High-level languages such as C, C++, JAVA
- **The fourth generation languages, or 4GL:** Designed to build specific programs, Commonly used in database programming and scripts. Python, PHP, SQL
- **The fifth generation languages, or 5GL:** Designed to make the computer solve a given problem with a set of rules or visual. Prolog



# Levels of Programming languages

- **Machine Language**

- Uses binary code
- Machine-dependent
- Not portable

- **Assembly Language**

- Uses mnemonics
- Machine-dependent
- Not usually portable

- **High-Level Language (HLL)**

- Uses English-like language
- Machine independent
- Portable (but must be compiled for different platforms)



# Machine language

- The representation of a computer program which is actually read and understood by the computer.
  - A program in machine code consists of a sequence of machine instructions.
- **Instructions:**
  - Machine instructions are in binary code.
  - Instructions specify operations and memory cells involved in the operation.
- **Example:**

## Operation Address

0010      0000 0000 0100

0100      0000 0000 0101

0011      0000 0000 0110



# Assembly language

- A symbolic representation of the machine language of a specific processor.
  - Is converted to machine code by an assembler.
  - Usually, each line of assembly code produces one machine instruction (One-to-one correspondence).
  - Programming in assembly language is slow and error-prone but is more efficient in terms of hardware performance.
  - Mnemonic representation of the instructions and data.
- **Example:** ADD AX,BX

SUB AX,CX



# High-Level language

- programming language which use statements consisting of English-like keywords such as “FOR”, “PRINT” or “IF”, ... etc.
  - Each statement corresponds to several machine language instructions (one-to-many correspondence).
  - Much easier to program than in assembly language.
  - Data are referenced using descriptive names.
  - Operations can be described using familiar symbols.
- **Example:** Cost = Price + Tax



# Importance of C

- Robust language with rich set of built-in functions and operators that can be used to write any complex program
- Programs written in C are efficient and fast
- C is highly portable
- C language is a structured programming language
- Another important feature of C is its ability to extend itself



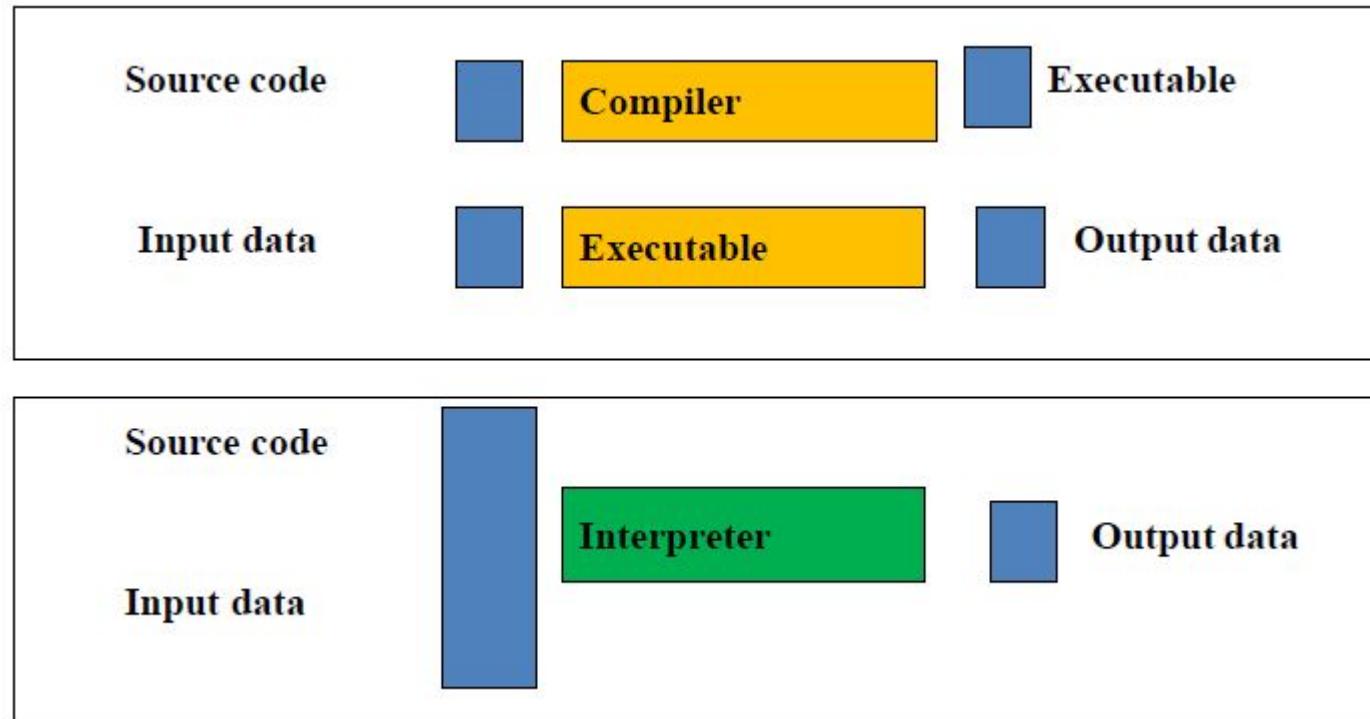
# Programs

- Source program
  - Form of procedure in some formal programming language, by the programmer.
  - Can be compiled directly into object/machine program by compiler or executed by interpreter
  - Extension - .c, .cpp, .java, etc.
- Object program
  - Output from the compiler
  - Contains equivalent machine program of the source program
  - Extension - .o
- Executable program
  - Machine language program linked with necessary libraries and other files
  - Can be executed directly
  - Extension - .exe



# Compiler vs interpreter

- Compilers – translates a source(human-writable) program to an executable(machine-readable) program
- Interpreter – translates a source program and executes it at the same time.



# Compile time vs run time

- Compile time – span of time while the program is being compiled by the compiler
  - Checks for syntactical error
- Run time – span of time while the .exe has been brought to RAM and is being executed.
  - Checks for semantic error and reveal it.



# Program structure

A sample C Program –

```
#include<stdio.h>
int main()
{
    --other statements
}
```



# Header files

- The files that are specified in the include section is called header file
- Precompiled files that has some functions defined in them
- We can call those functions in our program by supplying parameters
  - Example – printf().
- Header file is given an extension .h



# Example of Standard header file

- stdio.h – file and console (also a file) IO: *perror, printf, open, close, read, write, scanf, etc.*
- stdlib.h - common utility functions: *malloc, calloc, strtol, atoi, etc*
- string.h - string and byte manipulation: *strlen, strcpy, strcat, memcpy, memset, etc.*
- math.h – math functions: *ceil, exp, floor, sqrt, etc.*
- time.h – time related facility: *asctime, clock, time\_t, etc*



# Main Function

- Entry point of a program
- From main function the flow goes as per the programmers choice.
- There may or may not be other functions written by user in a program
- Main function is compulsory for any c program



# First program example

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



# Hello World in C

```
#include <stdio.h>  
  
void main()  
{  
    printf("Hello, world!\n");  
}
```



Preprocessor used to share information among source files  
+ Cheaply implemented  
+ Very flexible

- This program prints Hello, world on the screen when we execute it



# Hello World in C

```
#include <stdio.h>  
  
void main()  
{  
    printf("Hello, world!\n");  
}
```

I/O performed by a library function

Program mostly a collection of functions  
“main” function special: the entry point  
“void” qualifier indicates function does not return anything



# Running a C Program

- Type a program
- Save it
- Compile the program – This will generate an exe file (executable)
- Run the program (Actually the exe created out of compilation will run and not the .c file)



# The preprocessor

- The preprocessor takes your source code and – following certain directives that you give it – tweaks it in various ways before compilation.
- A directive is given as a line of source code starting with the # symbol
- The preprocessor works in a very crude, “word-processor” way, simply cutting and pasting – it doesn’t really know anything about C!



# Preprocessor directives (1/3)

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

- The #include directives “paste” the contents of the files stdio.h, stdlib.h and string.h into your source code, at the very place where the directives appear.
- These files contain information about some library functions used in the program:
  - stdio stands for “standard I/O”, stdlib stands for “standard library”, and string.h includes useful string manipulation functions.
- Want to see the files? Look in /usr/include



# Preprocessor Directives (2/3)

- `#include`
  - gives a program access to a library
- `<stdio.h>`
  - standard header file
  - include printf, scanf

⇒`#include <stdio.h>`

- notify the preprocessor that some names used in the program are found in `<stdio.h>`



# Preprocessor Directives (3/3)

- **#define**
  - using only data values that never change should be given names
- Constant macro
  - a name that is replaced by a particular constant value

⇒#define KMS\_PER\_MILE 1.609



# Comments in C

- Single line comment
  - // (double slash)
  - Termination of comment is by pressing enter key
- Multi line comment

```
/*....  
.....*/
```

This can span over to multiple lines



# Overview of a Simple C Program

```
#include "stdio.h"  
#include "stdlib.h"  
  
#define CONSTANT_NAME 4  
  
void main(void)  
{  
    float y=0.1;  
    int x[100];  
    char name[50];  
  
    printf("\nProgram start");  
    .....  
}
```

Header files to be included

Definition of any constants used in program

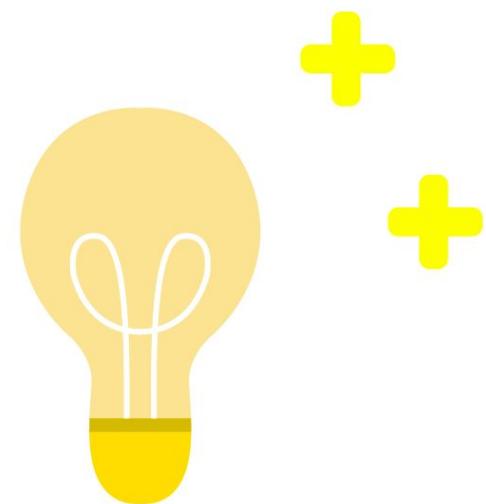
Main C Program Section

Variable declaration and (optional) initialisation

C Program Code



HAPPY  
CODING

A yellow lightbulb icon with two yellow plus signs floating above it, positioned to the right of the text.

# Constants, Variables and Data Types

---



Dept. of Computer Science & Engineering

# Lesson - 3

Topic	Lesson Learning Outcome (at the end of the lesson students will be able to...)	Teaching Learning Methodology	Assessment Method
Keywords, Constants and Identifiers	<ul style="list-style-type: none"><li>• Recognize different types of tokens</li><li>• Differentiate between keywords, constants and identifiers</li><li>• Know about the variable declaration rules</li></ul>	Multimedia Presentation , Question and Answer	Class Test, Exam, etc.



# Character Set

- The characters in C are grouped into following categories:
  1. Letters (A...Z, a...z)
  2. Digits (0...9)
  3. Special characters (, . ; : ? ! | “ \$ / \ ~ ^ { [ > % \_ & \* #)
  4. White spaces (e.g. Space, Tab, New line, Carriage return)
- Compiler ignores white spaces unless they are part of a string constant



# Trigraph Character

- Provides a way to enter certain characters that are not available on some keyboards
- Each trigraph sequence consists of three characters, 2 question marks followed by another character

Trigraph sequence	Equal character
??=	#
??(	[
??)	]
??/	\
??<	{
??>	}
??!	
??'	^
??-	~

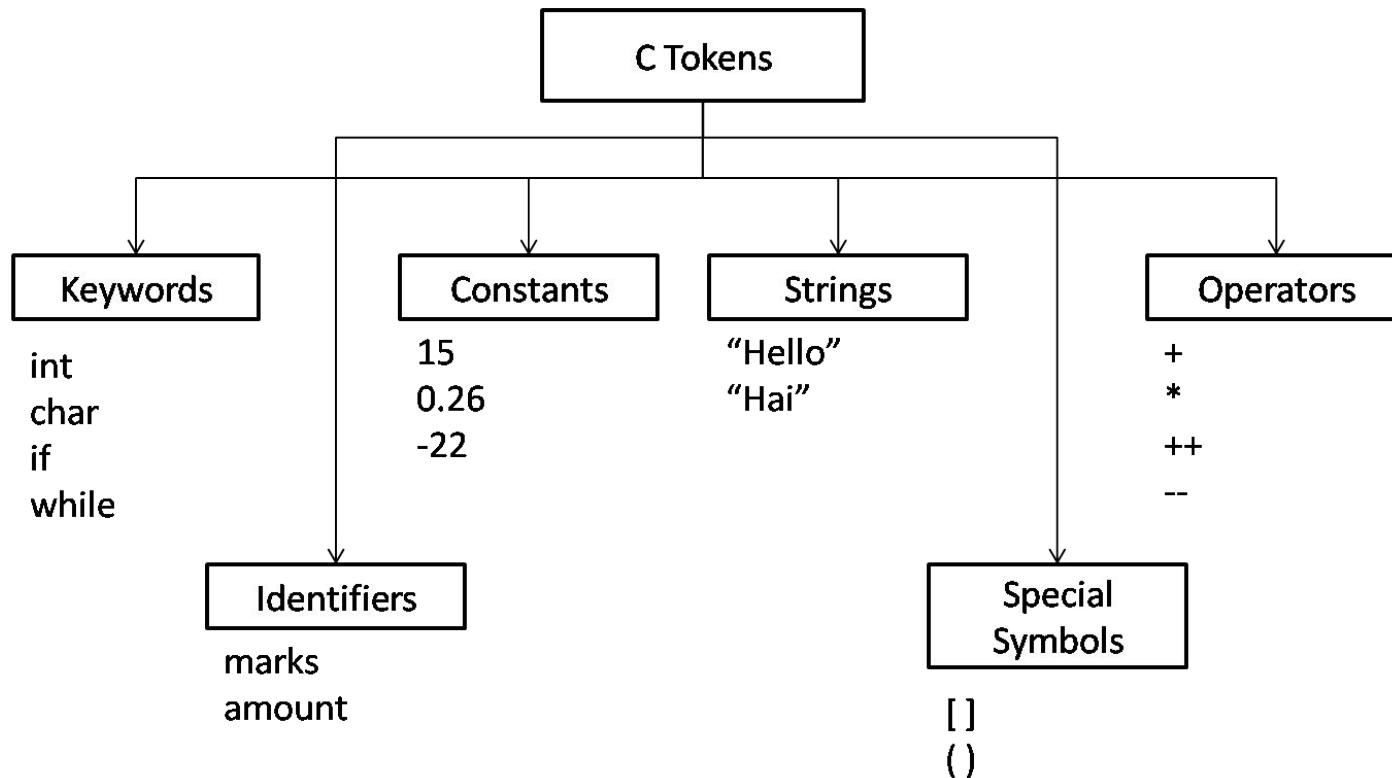
```
printf("??<");  
//Output: {
```

Try:  
`??=include<stdio.h>`



# C Tokens

- **Token:** Smallest individual unit
- C program is written using 6 different types of tokens



# Keywords

- Basic building blocks for program statements
- All keywords have fixed meanings and these meanings cannot be changed
- Keywords must be written in lower case
- Total 32 keywords in C

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



# Identifiers

- Refers to the names of variables, functions and arrays
- Rules for identifiers:

1. First character must be an alphabet or underscore
2. Must consist of only letters, digits or underscore
3. Only first 31 characters are significant
4. Cannot use keyword
5. Must not contain white space

Which are correct?

num_13	Correct
_x	Correct
_123	Correct
3_a	Wrong
first num	Wrong
num\$	Wrong

- E.g. main, num1, sum



# Constants

- Refers to fixed values that do not change during the execution of program
- **Integer Constants:** Sequence of digits preceded by an optional + or – sign. E.g. 123, -12, +82, 0x1A (hexa-decimal number)
- **Real Constants:** Represents quantities that may vary continuously. E.g. 12.42, .41, -0.9, 10e5, -1.2e-2
- **Single Character Constants:** Contains a single character enclosed in a pair of single quote marks. E.g. ‘a’, ‘Q’, ‘3’, ‘;’
- **String Constants:** Contains sequence of characters enclosed in double quotes. E.g. “Hello！”, “Well Done”, “3 number !”
- **Backslash Character Constants:** These are usually used in output functions. E.g: ‘\n’, ‘\t’, ‘\a’, ‘\b’, ‘\r’



# Some Rules for Constants (Contd.)

- Space is not allowed in constants except for string constants
- Character constants have integer values known as ASCII values.  
E.g. printf("%d", 'a'); // Output 97
- Character constant '5' is not same as number 5
- Character constant is not equivalent to single character string constant. E.g. 'B' is not same as "B"

25,000 ----- Valid? No

245 \$ ----- Valid? No

1231L ----- Valid? Yes

'num1' ----- Valid? No



# Variables

- Variable is a name that may be used to store a data value
- Variable may take different values at different times during execution
- A variable should be chosen by the programmer in a meaningful way
- Follows same rule as identifiers
- E.g. Average, num1, total, class\_no
- Variable names should not be same as a keyword.  
E.g. int (**invalid**)  
int\_x (**valid**)



# Constants, Variables and Data Types

---



Dept. of Computer Science & Engineering

# Lesson - 4

Topic	Lesson Learning Outcome (at the end of the lesson students will be able to...)	Teaching Learning Methodology	Assessment Method
Keywords, Constants and Identifiers	<ul style="list-style-type: none"><li>• Describe different data types</li><li>• Explain the size and range of data types</li><li>• Define storage class</li></ul>	Multimedia Presentation , Question and Answer	Class Test, Exam, etc.



# Data Types

- C supports 3 classes of data types:
  - 1) Primary data types  
(int, float, double)
  - 2) Derived data types  
(arrays, structure, functions, pointers)
  - 3) User-defined data types  
(typedef type identifier;)  
**(Fig 2.4)**



# Primary Data Types

- Basic data types in C: **int, float, double, char, and \_Bool**.
- **Data type int:** can be used to store integer numbers (values with no decimal places)
- **Data type float:** can be used for storing floating-point numbers (values containing decimal places).
- **Data type double:** the same as type float, only with roughly twice the precision.
- **Data type char:** can be used to store a single character, such as the letter a, the digit character 6, or a semicolon.
- **Data type \_Bool:** can be used to store just the values 0 or 1 (used for indicating a true/false situation).



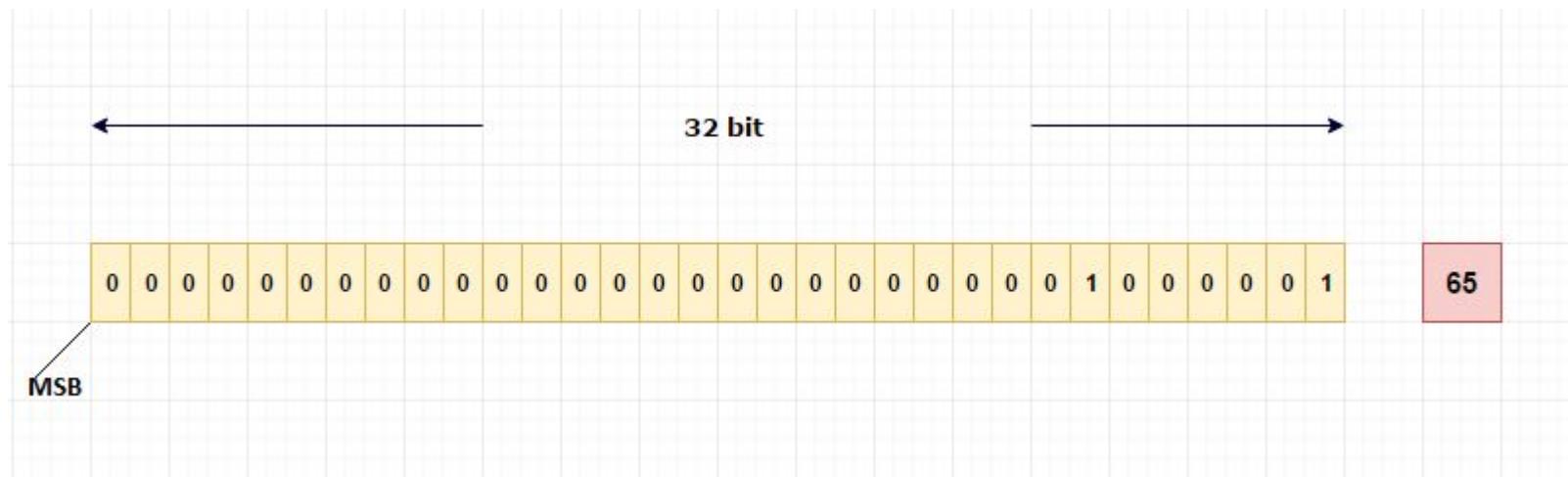
# Integer Types

---

- Integers are whole numbers with a range of values supported by particular machine.
- If we use 16-bit word length, the size of integer value is limited to range -32768 to 32767.
- If we use 32-bit word length can store an integer ranging from -2147483648 to 2147483647.
- In order to provide control over range of numbers and storage space C has 3 classes of integer storage namely: short int, int, long int in both signed and unsigned.



# Integer Types

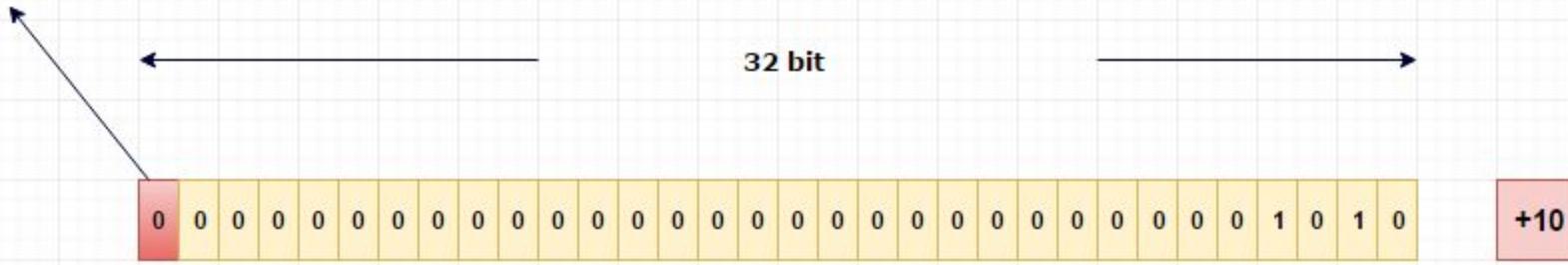


Source: <https://www.log2base2.com/storage/how-integers-are-stored-in-memory.html>

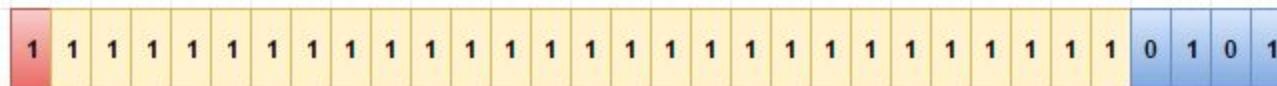


# Integer Types

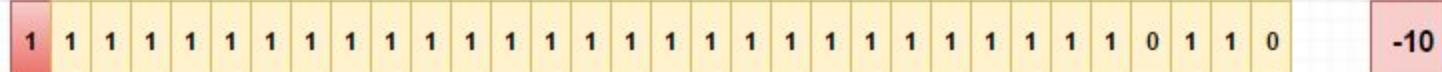
MSB 0 indicates positive number



1's complement of 10



Add 1 to make it 2's complement



MSB 1 indicates negative number

Source: <https://www.log2base2.com/storage/how-integers-are-stored-in-memory.html>

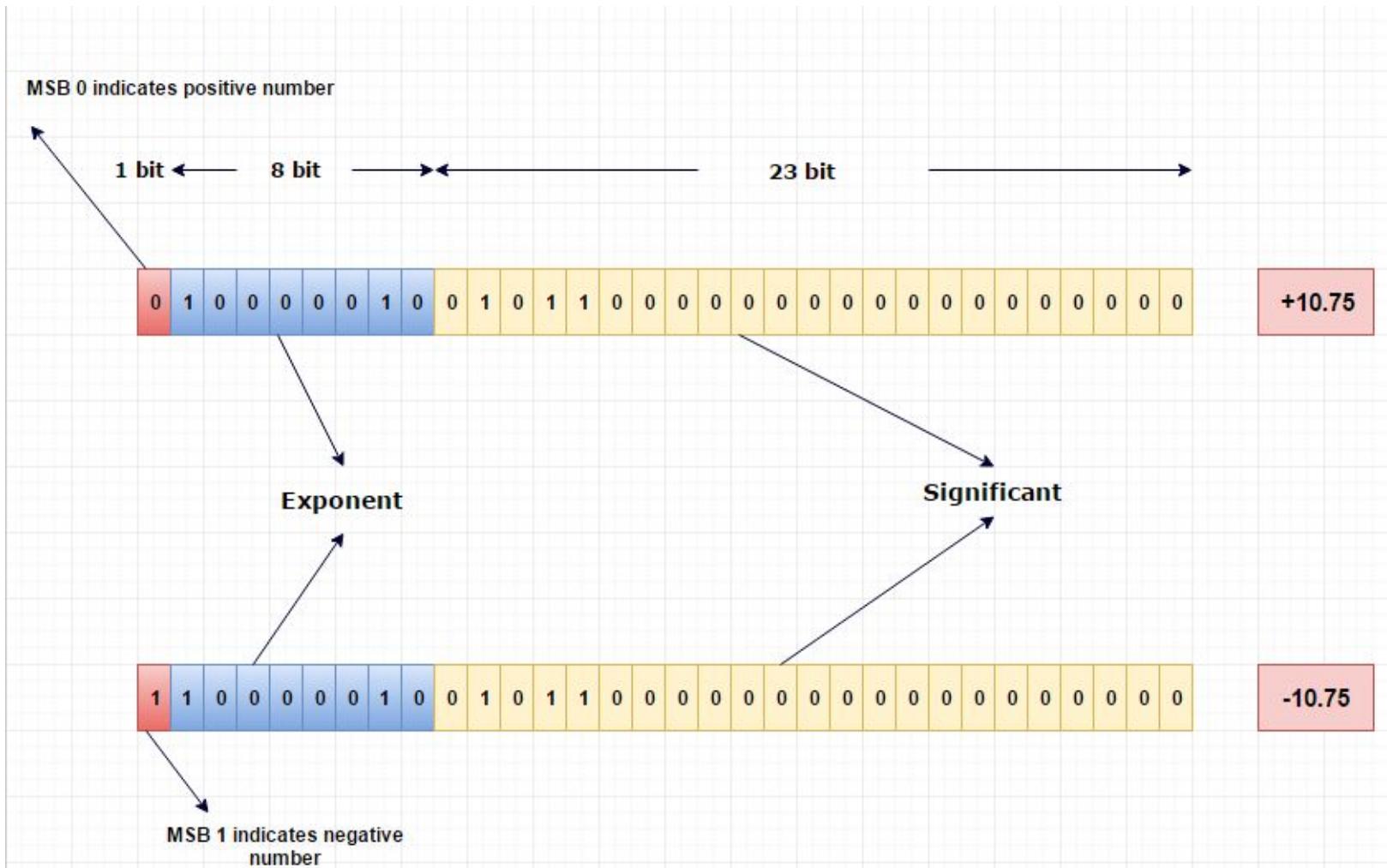


# Float Data Type

- Floating point numbers are stored in 32bits, with 6 digit precision.
- Floating point numbers are defined by keyword “float”.
- When accuracy is provided by a float number is not sufficient, double can be used to define number.



# Floating Point Types



Source: <https://www.log2base2.com/storage/how-float-values-are-stored-in-memory.html>



# Steps to Store a Float Constant

- Step-1: Floating number will be converted to binary number

$$(10.75)_{10} = (1010.11)_2$$

- Step-2 : Make the converted binary number to normalize form

$$(1010.11)_2 = 1.01011 * 2^3$$

- Step-3: Add bias to exponent (n=8)

$$\text{bias}_n = 2^{n-1} - 1 = 2^7 - 1 = 127$$

Nomalized exponent = Actual exponent + bias

$$= 3 + 127$$

$$= (130)_{10}$$

$$= (10000010)_2$$

**Normalized form**

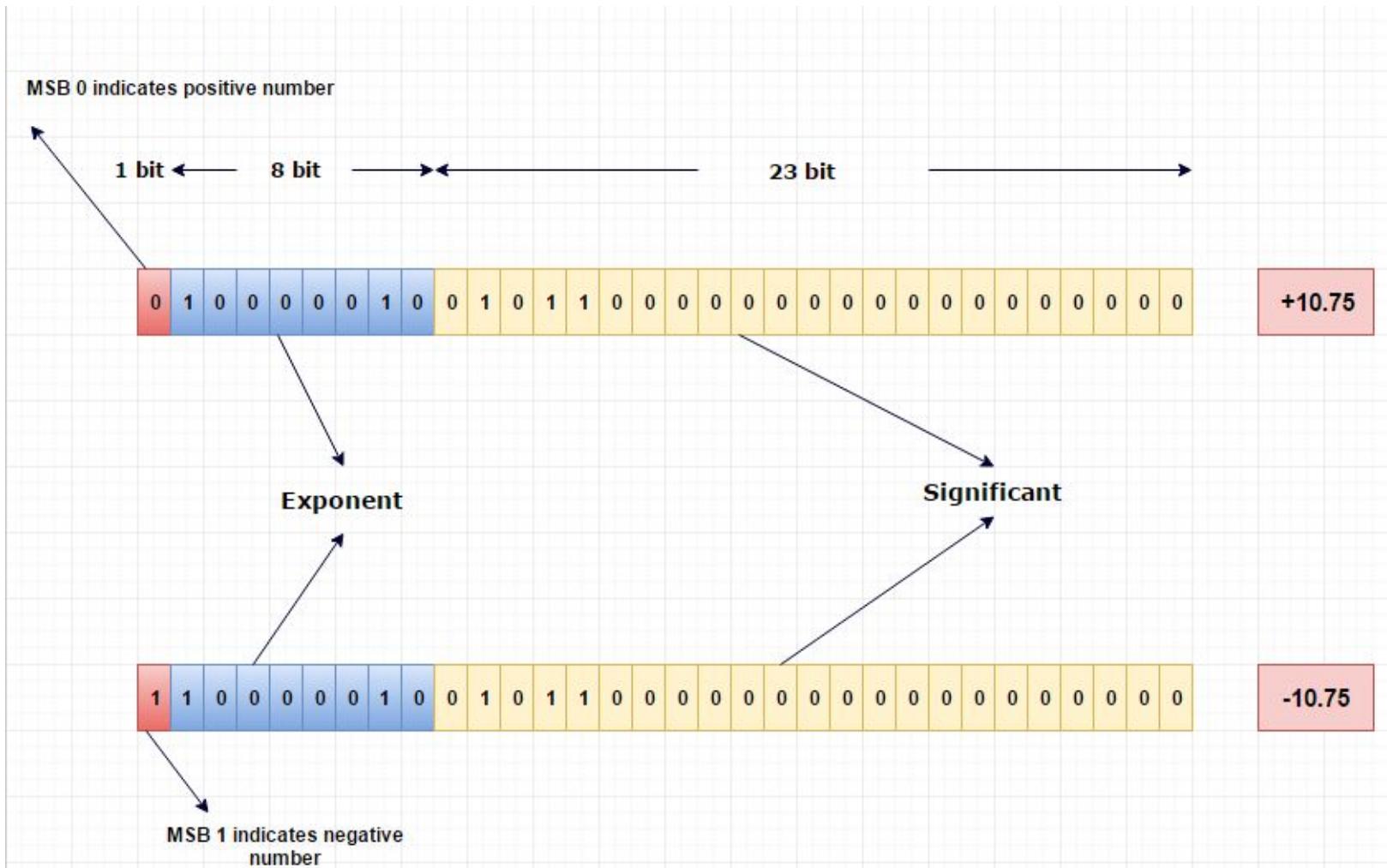
1.01011 \* 2<sup>3</sup>

**Exponent : 3**

**Significant : 1.01011**



# Floating Point Types



Source: <https://www.log2base2.com/storage/how-float-values-are-stored-in-memory.html>



# Double Data Type

- A double datatype number uses 64 bits giving a precision of 14 digits.
- These are known as double precision number.
- Double datatype represent the same datatype that float represents but with greater precision.



# Double Data Type

- To store double, computer will allocate 8 byte (64 bit)
  - 1 bit for sign
  - 11 bit for exponent
  - 52 bit for significant



# Steps to Store a Double Constant

- Step-1: Floating number will be converted to binary number

$$(10.75)_{10} = (1010.11)_2$$

- Step-2 : Make the converted binary number to normalize form

$$(1010.11)_2 = 1.01011 * 2^3$$

- Step-3: Add bias to exponent (n=11)

$$\text{bias}_n = 2^{n-1} - 1 = 2^{10} - 1 = 1023$$

Nomalized exponent = Actual exponent + bias

$$= 3 + 1023$$

$$= (1026)_{10}$$

$$= (1000000010)_2$$

**Normalized form**

**1.01011 \* 2<sup>3</sup>**

**Exponent : 3**

**Significant : 1.01011**



# Void Types

- Void type has no values. This is used to specify the type of functions.
- The type of function is said to be void when it doesn't return any value to the calling function.



# Character Types

- A single character can be defined as a character (char) type data.
- Characters are usually stored in one byte of internal storage.
- Qualifier signed or unsigned may be used in char explicitly. Unsigned characters have values between 0 and 255, signed characters have values from -128 to 127



# Character Types

## ASCII Table

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	'
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	\$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(	72	48	110	H	104	68	150	h
9	9	11		41	29	51	)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	:	91	5B	133	[	123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	
29	1D	35		61	3D	75	=	93	5D	135	]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	-	127	7F	177	

Source: <https://www.log2base2.com/storage/how-characters-are-stored-in-memory.html>



# Character Types

	<b>8 Bit memory</b>	<b>ASCII value</b>
<b>A</b>	0 1 0 0 0 0 0 1	<b>65</b>
<b>a</b>	0 1 1 0 0 0 0 1	<b>97</b>

Source: <https://www.log2base2.com/storage/how-characters-are-stored-in-memory.html>



# Declaration of Variables

```
#include <stdio.h>
int main (void)
{
    int integerVar = 100;
    float floatingVar = 331.79;
    double doubleVar = 8.44e+11;
    char charVar = 'W';
    printf ("integerVar = %d\n", integerVar);
    printf ("floatingVar = %f\n", floatingVar);
    printf ("doubleVar = %lf\n", doubleVar);
    printf ("charVar = %c\n", charVar);
    return 0;
}
```

(See Table 2.9 (sixth edition))



# Summary of Primary Data Type

Type	Size (bytes)	Format Specifier
int	at least 2, usually 4	%d , %i
char	1	%c
float	4	%f
double	8	%lf
short int	2 usually	%hd
unsigned int	at least 2, usually 4	%u
long int	at least 4, usually 8	%ld , %li
long long int	at least 8	%lld , %lli
unsigned long int	at least 4	%lu
unsigned long long int	at least 8	%llu
signed char	1	%c
unsigned char	1	%c
long double	at least 10, usually 12 or 16	%Lf



# Size and Range of Data Types

(vvi)

Type	Size (bits)	Range
Char	8	-128 to 127
Int	16	-32768 to 32767
Float	32	3.4E-38 to 3.4E+38
Double	64	1.7E-308 to 1.7E+308

See table 2.8 for detail list.



# User defined type declaration

- C supports type definition that allows users to define an identifier that represent existing data types.
- The user defined data type identifier can later be used to declare variables.

**typedef type identifier;**

- Here **type** represent existing data type and **identifier** refers to new name given to the data type.

**typedef int units;**

**typedef float marks;**

- Units symbolizes **int** and marks symbolized **float** which can be later used to declare variables.



# Declaration of Storage Class

- **Global variables:** can be used by all the functions in the program.
- **Local variables:** are visible and meaningful only inside the function in which they are declared.

Storage Class	Meaning
auto	Local variables known only to the function in which it is declared.
static	Local variable which exists and retains its value even after the control is transferred to the calling function
extern	Global variables known to all functions in the file.
register	Local variables stored in the register.



# Managing Input and Output

---



Dept. of Computer Science & Engineering

# Lesson - 4

Topic	Lesson Learning Outcome (at the end of the lesson students will be able to...)	Teaching Learning Methodology	Assessment Method
Managing Input and Output	<ul style="list-style-type: none"><li>• Read and write data in different format</li><li>• Use different format specifiers</li></ul>	Multimedia Presentation , Question and Answer	Class Test, Exam, etc.



# Reading a Character

- `scanf("%c", &variable_name);`
- `variable_name = getchar();`

```
#include<stdio.h>

int main()
{
    char ch;
    scanf("%c", &ch);
    printf("%c", ch);
}
```

```
#include<stdio.h>

int main()
{
    char ch;
    ch = getchar();
    printf("%c", ch);
}
```



# Character Test Functions

Function	Test
isalnum(c)	Is c an alphanumeric character?
isalpha(c)	Is c an alphabetic character?
isdigit(c)	Is c a digit?
islower(c)	Is c lower case letter?
isprint(c)	Is c a printable character?
ispunct(c)	Is c a punctuation mark?
isspace(c)	Is c a white space character?
isupper(c)	Is c an upper case letter?

```
#include<stdio.h>
#include <ctype.h>
int main()
{
    char ch;
    ch = getchar();
    printf("%c\n", ch);
    printf("%d", isalpha(ch));
}
```



# Writing a Character

- `putchar (variable_name);`

```
#include<stdio.h>

int main()
{
    char ch;
    ch = getchar();
    putchar(ch);
}
```



# Formatted Input

- `scanf ("control string", arg1, arg2, ..... argn);`

```
#include<stdio.h>

int main()
{
    int a;
    float b;
    double c;
    scanf("%d %f %lf", &a, &b, &c);
    printf("%d %f %lf", a, b, c);
}
```



# Formatted Input for Integer Numbers

Type	Format Specifier
int	%d , %i
char	%c
float	%f
double	%lf
short int	%hd
unsigned int	%u
long int	%ld , %li
long long int	%lld , %lli
unsigned long int	%lu
unsigned long long int	%llu
signed char	%c
unsigned char	%c
long double	%Lf



# Formatted Input for Integer Numbers

- `scanf ("%2d %5d", &num1, &num2);`

**Input:** 50 31426

**Assigned Value:** num1 = 50 and num2 = 31426

**Input:** 31426 50

**Assigned Value:** num1 = 31 and num2 = 426

- **Solution:** `scanf("%d %d", &num1, &num2);`



# Formatted Input for Integer Numbers

- `scanf("%d %*d %d", &a, &b);`

**Input:** 123 456 789

**Assigned Value:** a = 123 and b = 789



# Formatted Input for Integer Numbers

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

**Input:** 123-456

**Assigned Value:** a = 123 and b = 456



# Formatted Input for Real Numbers

- Decimal point notation: 475.89
- Exponential notation: 43.21E-1
- `scanf("%f %f %f", &x, &y, &z);`



# Formatted Input for Real Numbers

- Decimal point notation: 475.89
- Exponential notation: 43.21E-1
- `scanf("%f %f %f", &x, &y, &z);`//for float data type
- `scanf("%lf %lf %lf", &x, &y, &z);`//for double data type



# Formatted Input for Character String

```
#include<stdio.h>

int main()
{
    int no;
    char name1[15], name2[15], name3[15];
    printf("Enter serial number and name one\n");
    scanf("%d %15s", &no, name1);
    printf("%d %15s\n\n", no, name1);
    printf("Enter serial number and name two\n");
    scanf("%d %s", &no, name2);
    printf("%d %15s\n\n", no, name2);
    printf("Enter serial number and name three\n");
    scanf("%d %15s", &no, name3);
    printf("%d %15s\n\n", no, name3);
}
```



# Formatted Input for Character String

```
#include<stdio.h>

int main()
{
    char address[80];
    printf("Enter address\n");
    scanf("%[a-z]", address);
    printf("%-80s\n\n", address);
}
```



# Reading Mixed Data Types

- `scanf ("%d %c %f %s", &count, &code, &ratio, name);`



# Return Value of scanf Function

- `scanf("%d %f %c", &a, &b, &c) == 3`



# Detection of Errors in Input

- `scanf("%d %f %s, &a, &b, name);`



# Formatted Output

- `printf("control string", arg1, arg2, ...., argn);`

```
#include<stdio.h>

int main()
{
    int a;
    float b;
    double c;
    scanf("%d %f %lf", &a, &b, &c);
    printf("%d %f %lf", a, b, c);
}
```



# Formatted Output for Integer Numbers

```
printf("%d\n", 1234);  
printf("%6d\n", 1234);  
printf("%2d\n", 1234);  
printf("%-6d\n", 1234);  
printf("%06d\n", 1234);
```

*Output*

9	8	7	6			
		9	8	7	6	
9	8	7	6			
9	8	7	6			
0	0	9	8	7	6	



# Formatted Output for Integer Numbers

```
int m = 12345;  
long n = 987654;  
printf("%d\n",m);  
printf("%10d\n",m);  
printf("%010d\n",m);  
printf("%-10d\n",m);  
printf("%10ld\n",n);  
printf("%10ld\n",-n);
```

12345
12345
0000012345
12345
987654
- 987654



# Formatted Output for Real Numbers

```
float y = 98.7654;  
printf("%7.4f\n", y);  
printf("%7.2f\n", y);  
printf("%-7.2f\n", y);  
printf("%f\n", y);  
printf("%10.2e\n", y);  
printf("%11.4e\n", -y);  
printf("%-10.2e\n", y);  
printf("%e\n", y);
```

Output

9	8	.	7	6	5	4					
		9	8	.	7	7					
9	8	.	7	7							
9	8	.	7	6	5	4					
		9	.	8	8	e	+	0	1		
-	9	.	8	7	6	5	e	+	0	1	
9	.	8	8	e	+	0	1				
9	.	8	7	6	5	4	0	e	+	0	1

# Formatted Output for Real Numbers

float y = 98.7654;	98.7654
printf("%7.4f\n", y);	98.765404
printf("%f\n", y);	98.77
printf("%7.2f\n", y);	98.77
printf("%-7.2f\n", y);	98.77
printf("%07.2f\n", y);	0098.77
printf("%.*f", 7, 2, y);	98.77
printf("\n");	
printf("%10.2e\n", y);	9.88e+001
printf("%12.4e\n", -y);	-9.8765e+001
printf("%-10.2e\n", y);	9.88e+001
printf("%e\n", y);	9.876540e+001



# User Defined Formatted Output for Real Numbers

- `printf("%.*f", width, precision, number);`
- `printf("%.*f",7,2,number);`
- `printf("%7.2f",number);`



# Formatted Output for Character/String

- `char ch = 'a'`
- `printf("%c", ch);`
- `printf("%3c", ch);`



# Formatted Output for Character/String

- char str[20] = “NEW DELHI 110001”

	<i>Specification</i>										<i>Output</i>									
%s	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0
	N	E	W		D	E	L	H	I		1	1	0	0	0	1				
%20s					N	E	W		D	E	L	H	I		1	1	0	0	0	1
%20.10s											N	E	W		D	E	L	H	I	
.5s	N	E	W		D															
%-20.10s	N	E	W		D	E	L	H	I											
%5s	N	E	W		D	E	L	H	I		1	1	0	0	0	1				



# Formatted Output for Character/String

```
char x = 'A';
char name[20] = "ANIL KUMAR GUPTA";
printf("OUTPUT OF CHARACTERS\n\n");
printf("%c\n%3c\n%5c\n", x,x,x);
printf("%3c\n%c\n", x,x);
printf("\n");
printf("OUTPUT OF STRINGS\n\n");
printf("%s\n", name);
printf("%20s\n", name);
printf("%20.10s\n", name);
printf("%.5s\n", name);
printf("%-20.10s\n", name);
printf("%5s\n", name);
```

```
A
      A
          A
      A
A
OUTPUT OF STRINGS
ANIL KUMAR GUPTA
ANIL KUMAR GUPTA
      ANIL KUMAR
ANIL
ANIL KUMAR
ANIL KUMAR GUPTA
```

# Mixed Data Output

- `printf(“%d %f %s %c”, a, b, c, d);`

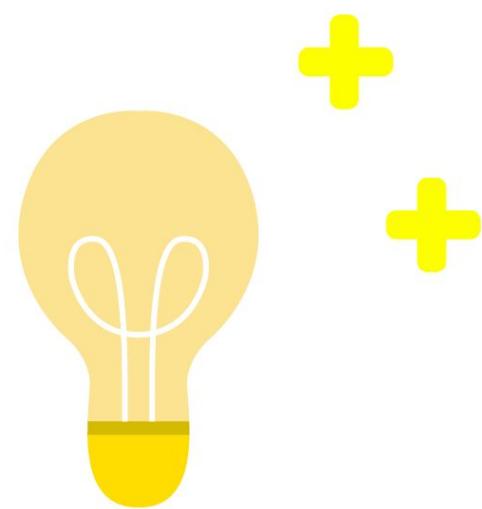


# Formatted Output

<i>Code</i>	<i>Meaning</i>
%c	print a single character
%d	print a decimal integer
%e	print a floating point value in exponent form
%f	print a floating point value without exponent
%g	print a floating point value either e-type or f-type depending on
%i	print a signed decimal integer
%o	print an octal integer, without leading zero
%s	print a string
%u	print an unsigned decimal integer
%x	print a hexadecimal integer, without leading Ox



# HAPPY CODING



# Operators and Expressions

---



Dept. of Computer Science & Engineering

# Lesson – (8, 9)

Topic	Lesson Learning Outcome (at the end of the lesson students will be able to...)	Teaching Learning Methodology	Assessment Method
Operators (B1: Ch-3, B2-Ch-4)	<ul style="list-style-type: none"><li>• Know about different arithmetic, relational, logical and assignment operators</li><li>• Use short hand of operators</li><li>• Know about increment-decrement and conditional operator</li></ul>	Multimedia Presentation , Question and Answer	Class Test, Exam, etc.
Expressions (B1: Ch-3, B2: Ch-4)	<ul style="list-style-type: none"><li>• Evaluate an arithmetic expression</li><li>• Know about precedence and associativity of operators in a program</li><li>• Apply type conversion when required</li></ul>	Multimedia Presentation , Question and Answer	Class Test, Exam, etc



# Introduction

- An **operator** is a symbol that tells the computer to perform certain manipulations.
- An **expression** is a sequence of operands and operators that reduces to a single value.
- C operators can be classified into a number of categories.
  - Arithmetic operators
  - Relational operators
  - Logical operators
  - Assignment operators
  - Increment and decrement operators
  - Conditional operators
  - Bitwise operators
  - Special operators



# Arithmetic Operators

- The arithmetic operators used for numeric calculations.  
They are two types:
- Unary Arithmetic Operators
- $+x$   $-y$ ,  $++$ ,  $--$ ,  $!$ ,  $\sim$
- Binary Arithmetic Operators

Operator	meaning
$+$	Addition or unary plus
$-$	Subtraction or unary minus
$*$	Multiplication
$/$	Division
$\%$	modulo division



# Arithmetic Operators

- ❖ Note:,
  - ❖ Integer division truncates remainder
  - ❖ The % operator cannot be applied to a float or double.
  - ❖ The precedence of arithmetic operators
    - ❖ Unary + or -
    - ❖ \* / %
    - ❖ + -
- ❖ Real Arithmetic, Mixed-mode Arithmetic



# Example

```
#include<stdio.h>
main()
{
    int months , days ;
    printf ( "Enter days \n " );
    scanf ( "%d" , &days ) ;
    months = days / 30 ;
    days = days % 30 ;
    printf ( "Months=%d Days= %d \n", months, days);
}
```



# Relational Operators

- The relational operators in C are :

**Used to compare values of two expressions depending on their relations.**

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



# Relational Operators

- A relational expression yields a value of 1 or 0.

□ 5 < 6	TRUE	1
□ 4.5 < 10	TRUE	1
□ -34 + 8 > 23 - 5	FALSE	0
□ 4.5 < -10	FALSE	0
□ 5 == 0	FALSE	0
□ if a=3, b=2, c =1; then	a > b > c	is ?

- Arithmetic operators have higher priority over relational operators.
- The associativity of relational operators is left □ right



# Relational Operators

- ❖ The relational operators  $>$ ,  $\geq$ ,  $<$  and  $\leq$  have the same precedence. Just below them in precedence are the equality operators:  $= =$  and  $!=$ .
- ❖ Relational operators have lower precedence than arithmetic operators , so an expression like  $i < \text{lim-1}$  is taken as  $i < (\text{lim}-1)$ .
- ❖ **Relational Operator Complements (From Book)**



# Logical operators

- ❖ C has the following three logical operators
  - ❖ `&&` meaning logical **AND** ( binary operator )
  - ❖ `||` meaning logical **OR** ( binary operator )
  - ❖ `!` meaning logical **NOT**( unary operator )
- ❖ Expressions connected by `&&` or `||` are evaluated left to right, and evaluation stops as soon as the truth or falsehood of the result is known.



# Logical operators

- ❖ The logical operators `&&` and `||` are used when we want to test more than one condition and make decisions. An example is ;

`a>b && x==10`

- An expression of this kind, which combines two or more relations expressions, is termed as a logical expression or compound relational expression.
- The logical expression given above is **TRUE** only if `a>b` is true and `x==10` is true. If either (or both) of them are false, the expression is false.



# Logical operators

op-1	op2	op-1&&op-2	op-1  op-2	!op-1
Non-zero	Non-zero	1	1	0
Non-zero	0	0	1	0
0	Non-zero	0	1	1
0	0	0	0	1



# Logical operators

- The precedence of `&&` is higher than that of `||`, and both are lower than relational operators, so
- `3 < 5 && -3 < -5 || 3 > 2`
- `char ch;` to decide whether `ch` is an uppercase,
  - `ch >= 'A' && ch <= 'Z'`
- to decide whether a year is leap year,
  - `year % 4 == 0 && year % 100 != 0 || year % 400 == 0`



# Assignment operators

- Assignment operators are used to assign the result of an expression to a variable.
- C has a set of ‘shorthand’ assignment operators of the form  
 $v \text{ op=} \text{ exp};$
- where **v** is a variable, **exp** is an expression and **op** is a C binary arithmetic operator. The operator **op=** is known as the shorthand assignment operator.
- The assignment statement  $v \text{ op=} \text{ exp};$  is equivalent to  $v = v \text{ op} (\text{exp});$
- For example,  $x += y + 1;$  This is same as the statement  $x = x + (y + 1);$



# Assignment operators

Statement with simple assignment operator	Statement with shorthand operator
$a = a + 1$	$a += 1$
$a = a - 1$	$a -= 1$
$a = a * (n+1)$	$a *= n+1$
$a = a / (n+1)$	$a /= n+1$
$a = a \% b$	$a \%= b$



# Assignment operators

The use of shorthand assignment operators has three advantages:

1. What appears on the left-hand side need not be repeated and therefore it becomes easier to write.
2. The statement is more concise and easier to read.
3. The statement is more efficient.



# Increment and decrement operators

- C provides two unusual operators for incrementing and decrementing variables.
- The increment operator `++` adds 1 to its operand, while the decrement operator `--` subtracts 1.
- The unusual aspect is that `++` and `--` may be used either as prefix operators (before the variable, as in `++n`), or postfix operators (after the variable: `n++`).
- In both cases, the effect is to increment `n`. But the expression `++n` increments `n` *before* its value is used, while `n++` increments `n` *after* its value has been used.



# Increment and decrement operators

## □ Prefix Increment/Decrement:

The statement  $y=++x$  means first increment the value of  $x$  by 1, then assign the value to  $y$ . this statement belongs to the following two statement:  $x=x+1;$        $x=x-1;$

$y=x;$                    $y=x;$

#include<stdio.h> main() { int x=8; printf("x=%d\n",x); printf("x=%d\n",++x); printf("x=%d\n",x); printf("x=%d\n",-x); printf("x=%d",x); }	Output
	$x=8$
	$x=9$
	$x=9$
	$x=8$
	$x=8$



# Increment and decrement operators

## □ Postfix Increment/Decrement:

The statement  $y=x++$  means first the value of  $x$  is assigned to  $y$  and then  $x$  is incremented. This statement belongs to the following two statements:  $y=x;$        $y=x;$

$x=x+1;$

$x=x-1;$

#include<stdio.h> main() { int x=8; printf("x=%d\n",x); printf("x=%d\n",x++); printf("x=%d\n",x); printf("x=%d\n",x--); printf("x=%d",x); }	Output
	x=8
	x=8
	x=9
	x=9
	x=8



# Increment and decrement operators

- for example, if n is 5, then
  - $x = n++;$
- is equivalent
  - $x = n; n++;$
- but
  - $x = ++n;$
- is equivalent
  - $n++; x = n;$
- The increment and decrement operators can only be applied to variables; an expression like  $(i+j)++$  is illegal.



# Increment and decrement operators

- The increment and decrement operators can be used in complex statements. Example:
  - $m=n++ -j +10;$
- Consider the expression
  - $m = - n++ ;$
- The precedence of `++` and `--` operators are the same as those of unary `+` and `-`.
- The associativity of them is **right to left**.
- $m = - n++;$  is equivalent to  $m = - (n++)$



# Increment and decrement operators

- suppose,
- int a, b, c ; a = b = c = 1;
- After execution the following statements, what are the values of the expression and variables. (**Homework**)
- (1) a>b && b>c++;
- (2) a-- || c++;
- (3) !a && b++;
- (4) ++a && ++b && ++c;
- (5) ++a && --b && ++c;



# Conditional operator

- ❖ a ternary operator pair “? : ” is available in C to construct conditional expressions of the form

*expr1 ? expr2 : expr3*

- ❖ the expression *expr1* is evaluated first. If it is non-zero (true), then the expression *expr2* is evaluated, and that is the value of the conditional expression. Otherwise *expr3* is evaluated, and that is the value. Only one of *expr2* and *expr3* is evaluated.



# Conditional operator

- $x = (a > b) ? a : b;$
- Consider an example two find the largest number between two numbers.

```
#include<stdio.h>
main()
{
    int a,b,MAX;
    scanf("%d %d ",&a, &b);
    MAX=a>b?a:b;
    printf("The large value is : %d",MAX);
}
```

## Output;

12 7

The large value is:12

Or

8 14

The large value is:14



# Special operators

- **1. The Comma Operator**
- The comma operator can be used to link the related expressions together. A comma-linked list of expressions is evaluated left to right and the value of right-most expression is the value of the combined expression. For example, the statement
- `value = (x=10, y=5, x+y);`
- first assigns the value 10 to x, then assigns 5 to y, and finally assigns 15 to value. Since comma operator has the **lowest precedence** of all operators, the parentheses are necessary.



# Special operators

- **2. The sizeof Operator**
- The sizeof is a compile time operator and, when used with an operand, it returns the number of bytes the operand occupies. The operand may be variable, a constant or a data type qualifier. Examples:
  - `m = sizeof( sum );`
  - `n = sizeof( long int );`
  - `k = sizeof( 235L );`



# Program employs different kinds of operators

```
#include<stdio.h> (Code this in home)
main()
{
    int a, b, c, d;
    a = 15;
    b = 10;
    c = ++a - b;
    printf ("a=%d b=%d c=%d\n", a, b, c );
    d = b++ +a;
    printf("a = %d b = %d d = %d\n", a, b, d );
    printf("a/b = %d\n", a / b);
    printf("a%b = %d\n", a%b );
    printf("a *= b = %d\n", a *= b );
    printf("%d\n", ( c>d ) ? 1 : 0 );
    printf("%d\n", ( c<d ) ? 1 : 0 );
}
```



# Arithmetic expressions

- ❖ An arithmetic expression is a combination of variables, constants, and operators.
- ❖ For example,
- ❖  $a*b-c$          $a*b-c$
- ❖  $(m+n)(x+y)$          $(m+n)*(x+y)$
- ❖  $ax^2+bx+c$          $a*x*x+b*x+c$

# Arithmetic expressions

- **Integer Arithmetic:** When both operands are integers  
for example if  $a=17$  and  $b=4$

Expression	Result
$a+b$	21
$a-b$	13
$a*b$	68
$a/b$	4
$a\%b$	1

# Arithmetic expressions

- **Real Arithmetic:** When both operands are float types  
for example if  $a=12.4$  and  $b=3.1$

Expression	Result
$a+b$	15.5
$a-b$	9.3
$a*b$	38.44
$a/b$	4.0
$a\%b$	NA



# Arithmetic expressions

- **Mixed-mode Arithmetic:** When one operand is integers and another is real .for example if  $a=12$  and  $b=2.5$

Expression	Result
$a+b$	14.5
$a-b$	9.5
$a*b$	30.0
$a/b$	4.8

# Variables in expressions and their evaluation

**(Variable = expression;)**

```
#include<stdio.h>
main()
{
    float a, b, c, x, y, z;
    a = 9;
    b = 12;
    c = 3;
    x = a - b / 3 + c * 2 - 1;
    y = a - b / ( 3+ c ) * ( 2 - 1 );
    z = a - ( b / ( 3 + c ) * 2 ) - 1;
    printf( "x = %f \n", x );
    printf(" y = %f \n", y );
    printf( "z = %f \n", z );
}
```



# Some Computational Problems

- When expressions include real values, then it is important to take necessary precautions to guard against certain computational errors. For example, consider the following statements:
  - $a = 1.0 / 3.0;$
  - $b = a * 3.0;$
- There is no guarantee that the value of  $b$  will equal 1.
- Another problem is division by zero.
- The third problem is to avoid overflow and underflow errors.



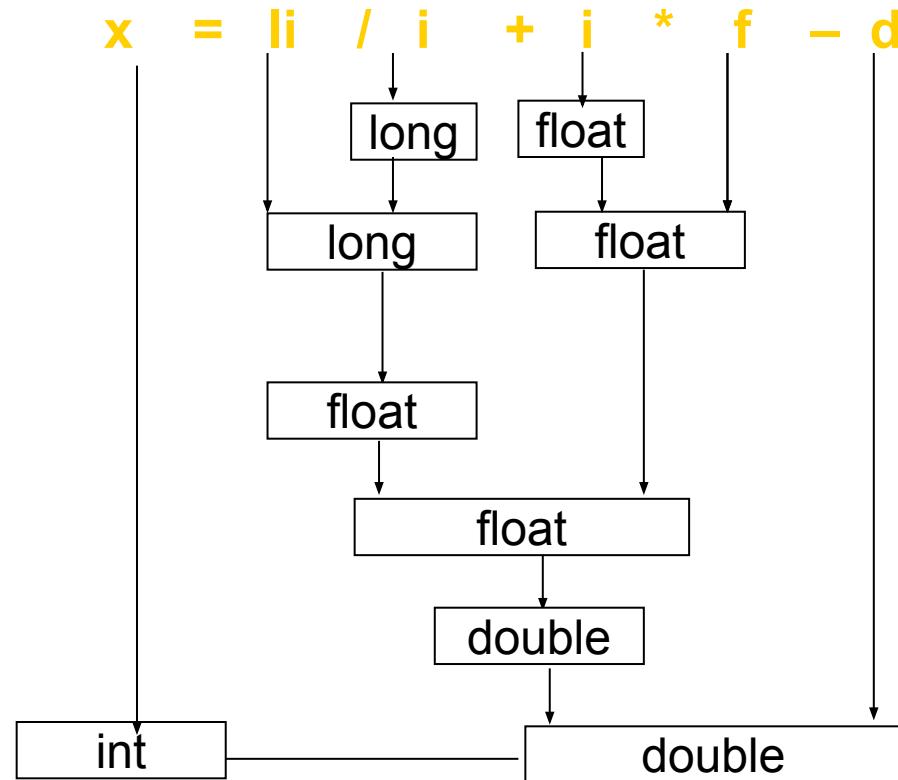
# Type conversions in expressions

- **1. Implicit Type Conversion**
- C permits mixing of constants and variables of different types in an expression. C automatically converts any intermediate values to the proper type so that the expression can be evaluated without losing any significance. This automatic conversion is known as **implicit type conversion**.
- The rule of type conversion: the **lower** type is automatically converted to the **higher** type.



# Type conversions in expressions

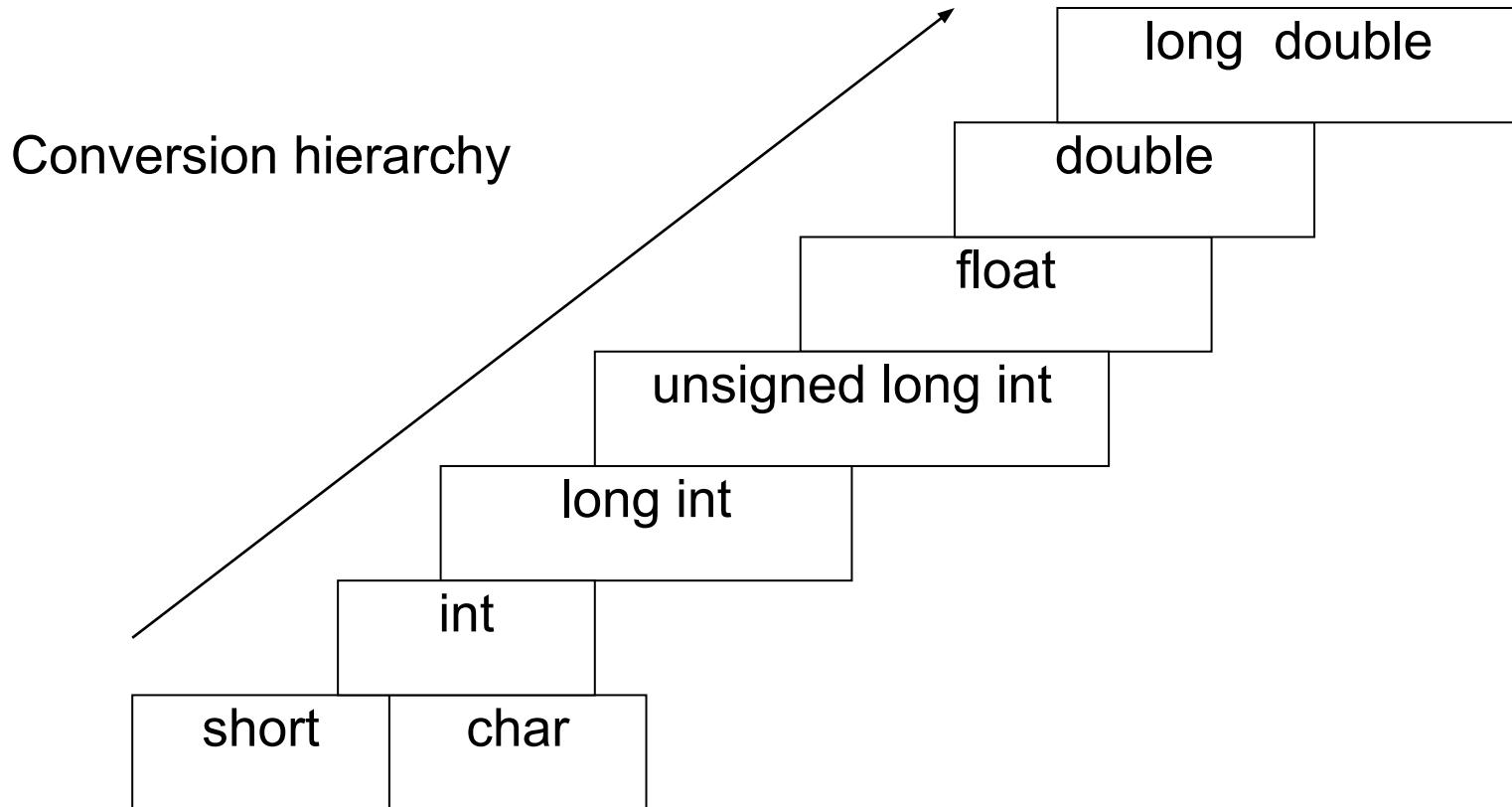
- for example,
  - int i, x;
  - float f;
  - double d;
  - long int li ;



- The final result of an expression is converted to the type of the variable on the left of the assignment.

# Type conversions in expressions

Conversion Hierarchy is given below



# Type conversions in expressions

- **2. Explicit conversion**
- We can force a type conversion in a way .
- The general form of explicit conversion is
  - ( type-name ) expression
- for example
  - `x = ( int ) 7.5 ;`
  - `a = ( int ) 21.3 / (int) 4.5;`
  - `a = ( float ) 3 / 2 ;`
  - `a = float ( 3 / 2 ) ;`



# Operator precedence and Associativity

- Rules of Precedence and Associativity
  - (1)Precedence rules decides the order in which different operators are applied.
  - (2)Associativity rule decide the order in which multiple occurrences of the same level operator are applied.
- **Table3.8 on page71 shows the summary of C Operators.**
- **a = i +1== j || k and 3 != x**



# Mathematical functions

- Mathematical functions such as cos, sqrt, log, etc. are frequently used in analysis of real-life problems.
- Table 3.9 on page 72 lists some standard math functions.
- All mathematical functions implement double type parameters and return double type values.
- To use any of mathematical functions, we should include the line:

```
#include <math.h>
```

Please read the text book to do well in the examination.  
Write the exercise and example programs of the book.



# Thank You



**Yes!  
Finally Over!**

www.funnyimage.us/images



# Decision Making and Branching

---



Dept. of Computer Science & Engineering

# Lesson – 10

Topic	Lesson Learning Outcome <b>(at the end of the lesson students will be able to...)</b>	Teaching Learning Methodology	Assessment Method
If-else and else-if ladder (B1: Ch-5)	<ul style="list-style-type: none"><li>• Make decisions and implement logic using if else statements</li><li>• Explain flow-chart constructed from else-if ladder</li></ul>	Multimedia Presentation , Question and Answer	Class Test, Exam, etc.



# Introduction

- C language possesses following decision making statements.
  - ✓ **If** statement
  - ✓ **Switch** statement
  - ✓ **Conditional operator** statement
  - ✓ **Goto** statement



# Decision Making with If Statement

- The if statement may be implemented in different forms depending on the complexity of conditions to be tested.
  
- Simple **if** statement
- **If... else** statement
- Nested **if... else** statement
- **Else if** ladder



# Simple If Statement

---

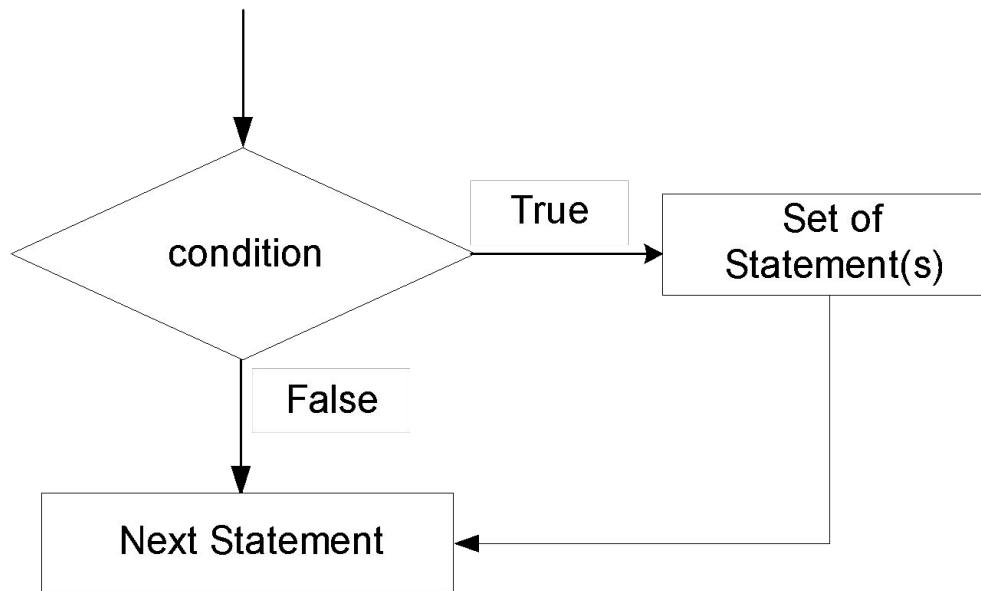
- In a simple ‘if’ statement, a condition is tested
- If the condition is true, a set of statements are executed
- If the condition is false, the statements are not executed and the program control goes to the next statement that immediately follows if block



# Simple If Statement

**Syntax:**

```
if (condition)
{
    Statement(s);
}
Next Statement;
```



# If...else Statement

- In simple ‘if’ statement, when the condition is true, a set of statements are executed. But when it is false there is no alternate set of statements
  
- If the test expression is true then the true block of statements are executed otherwise the false block of statements are executed.
  
- Either true block or false block will be executed not both



# If...else Statement

## Syntax:

```
if (condition)
```

```
{
```

```
    Statement set-1;
```

```
}
```

```
else
```

```
{
```

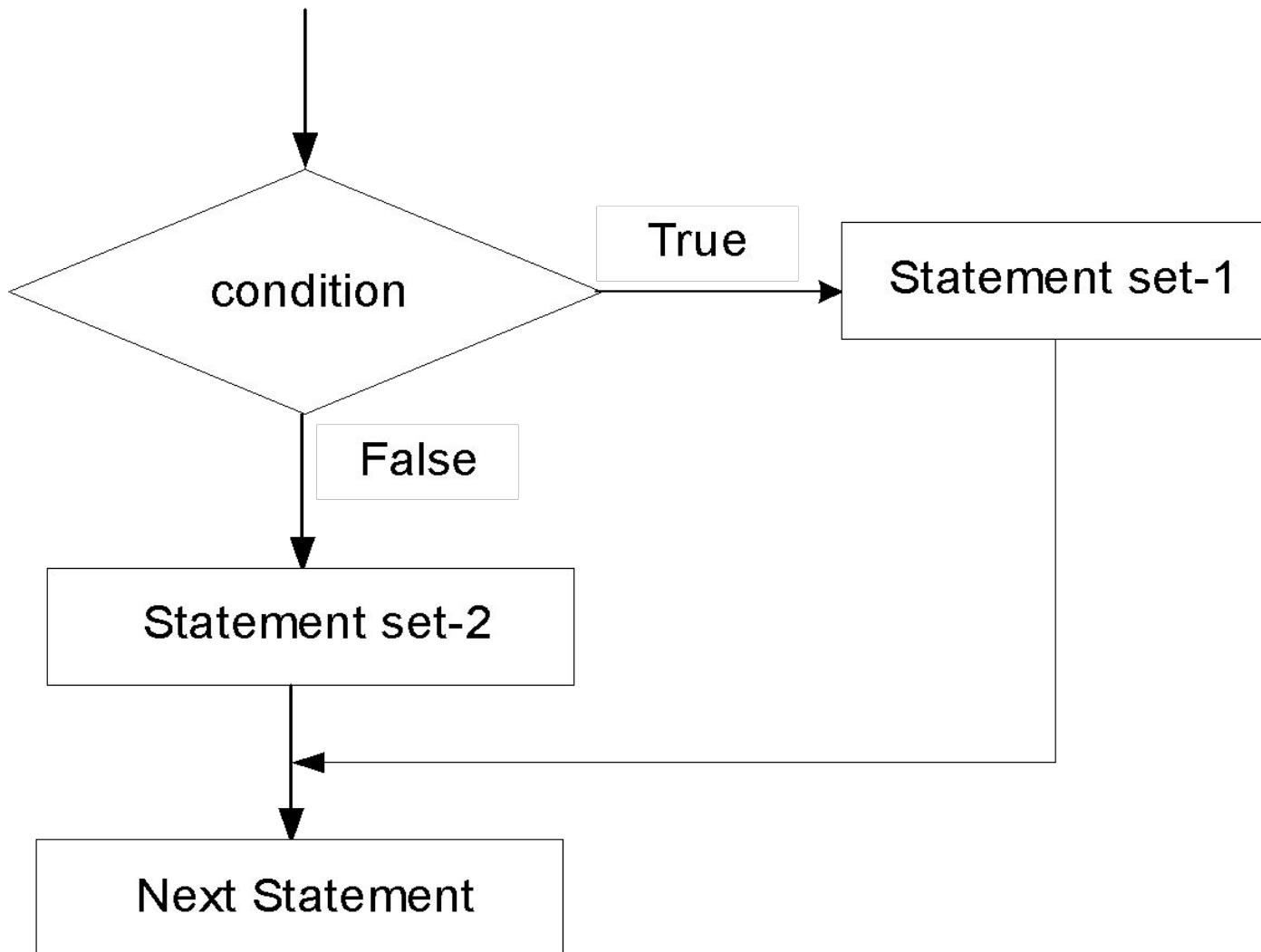
```
    Statement set-2;
```

```
}
```

```
Next Statement;
```



# If...else Statement



# If...else Statement

## Example:

```
if (Duration >= 3) {  
    /* If duration is equal to or more than 3 years,  
    Interest rate is 6.0 */  
    RateOfInterest = 6.0;  
}  
else {  
    /* else, Interest rate is 5.5 */  
    RateOfInterest = 5.5;  
}
```



# Else if Ladder

- The ‘else if’ statement is to check for a sequence of conditions
- When one condition is false, it checks for the next condition and so on.
- When all the conditions are false the ‘else’ block is executed.
- When a true conditional block is executed, the other ‘if’ statements are skipped



# Else if Ladder

Syntax:

```
if (condition-1)
{
    Statement set-1;
}
else if (condition-2)
{
    Statement set-2;
}
.....
else if (condition-n)
{
    Statement set-n;
}
else
{
    Statement set-x;
}
```

See the flow chart  
of else...if ladder  
from book.



# Else if Ladder

```
#include<stdio.h>
main()
{
    int test1,test2, result;
    printf("Enter your mark for Test 1:");
    scanf("%d",&test1);
    printf("Enter your mark for Test 2:");
    scanf("%d",&test2);

    result=(test1+test2)/2;
    if(result>=80) {
        printf("Passed: Grade A\n");
    }
    else if (result>=70) {
        printf("Passed: Grade B\n");
    }
    else if (result >=55) {
        printf("Passed: Grade C\n");
    }
    else {
        printf("Failed\n");
    }
}
```





---

# Thank You

# **Nested if-else, Switch Statement, Conditional Operator and Goto Statement**

---



Dept. of Computer Science & Engineering

# Lesson – 11, 12

Topic	Lesson Learning Outcome (at the end of the lesson students will be able to...)	Teaching Learning Methodology	Assessment Method
Nested if-else, Switch statement and Goto statement	<ul style="list-style-type: none"><li>• Construct nested if-else for solving complex decision problems</li><li>• Describe the rules for switch statements</li><li>• Apply conditional operator to shorten if-else statement</li><li>• Apply goto statements for controlling flow of a program</li></ul>	Multimedia Presentation , Question and Answer	Class Test, Exam, etc.



# Nested if-else

```
if (test condition - 1)
{
    if (test condition - 2)
    {
        statement 1;
    }
    else
    {
        statement 2;
    }
    else
    {
        statement 3;
    }
    statement x;
```

The diagram illustrates the control flow of a nested if-else structure. It starts with an outer if-statement. If the condition is true, it executes 'statement 1'. If the condition is false, it executes 'statement 2'. If the condition is true again, it executes 'statement 3'. Finally, it executes 'statement x'. Arrows indicate the flow from each if-block to its corresponding statements and then back to the end of the outer if-block.

# Nested if-else (Contd.)

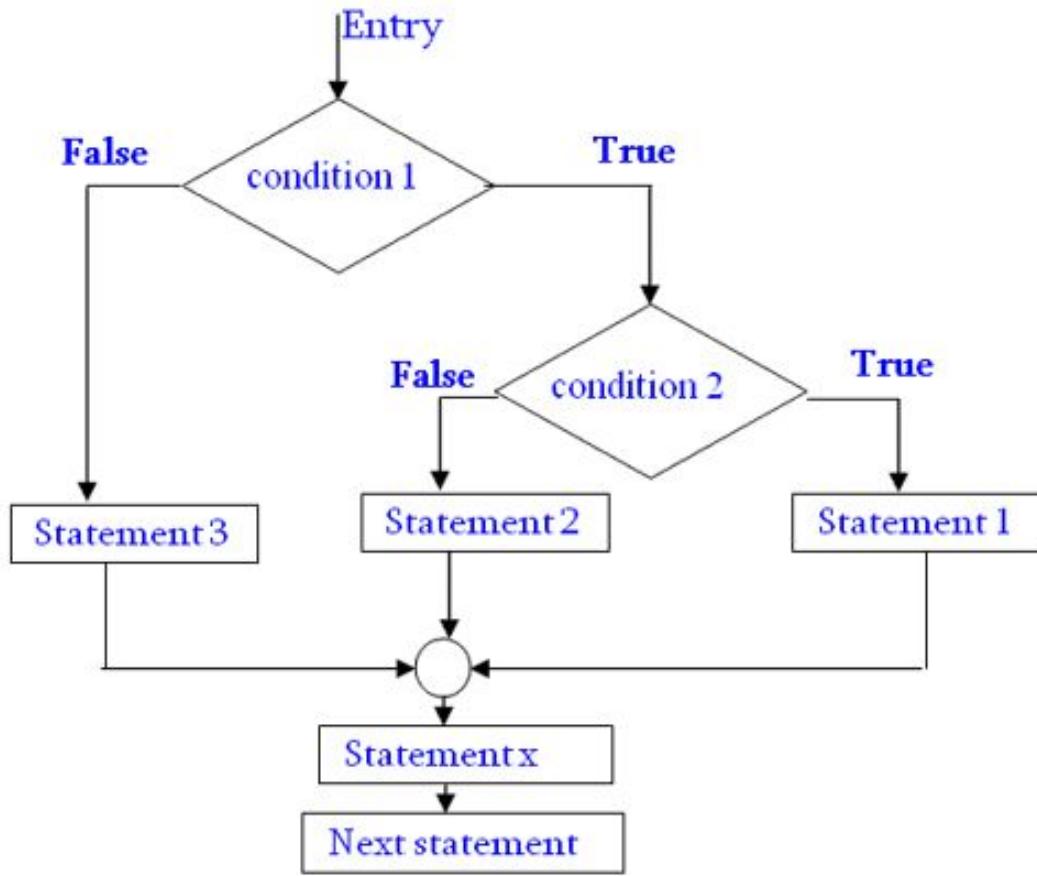


Fig. Flowchart of nested if...else statements

# Nested if-else (Contd.)

□ Example: Finding Largest among three numbers.

```
int main(){
    int a, b, c;
    scanf("%d %d %d", &a,&b, &c);
    if(a>b){
        if(a>c){
            printf("Largest = %d\n", a);
        }
        else{
            printf("Largest = %d\n", c);
        }
    }
    else{
        if(b>c){
            printf("Largest = %d\n", b);
        }
        else{
            printf("Largest = %d\n", c);
        }
    }
    return 0;
}
```



# Switch statement

```
switch( expression )
{
    case value-1:
        Block-1;
        Break;
    case value-2:
        Block-2;
        Break;
    case value-n:
        Block-n;
        Break;
    default:
        Block-1;
        Break;
}
Statement-x;
```

Fig. Syntax of switch-case statements



# Switch statement (Contd.)

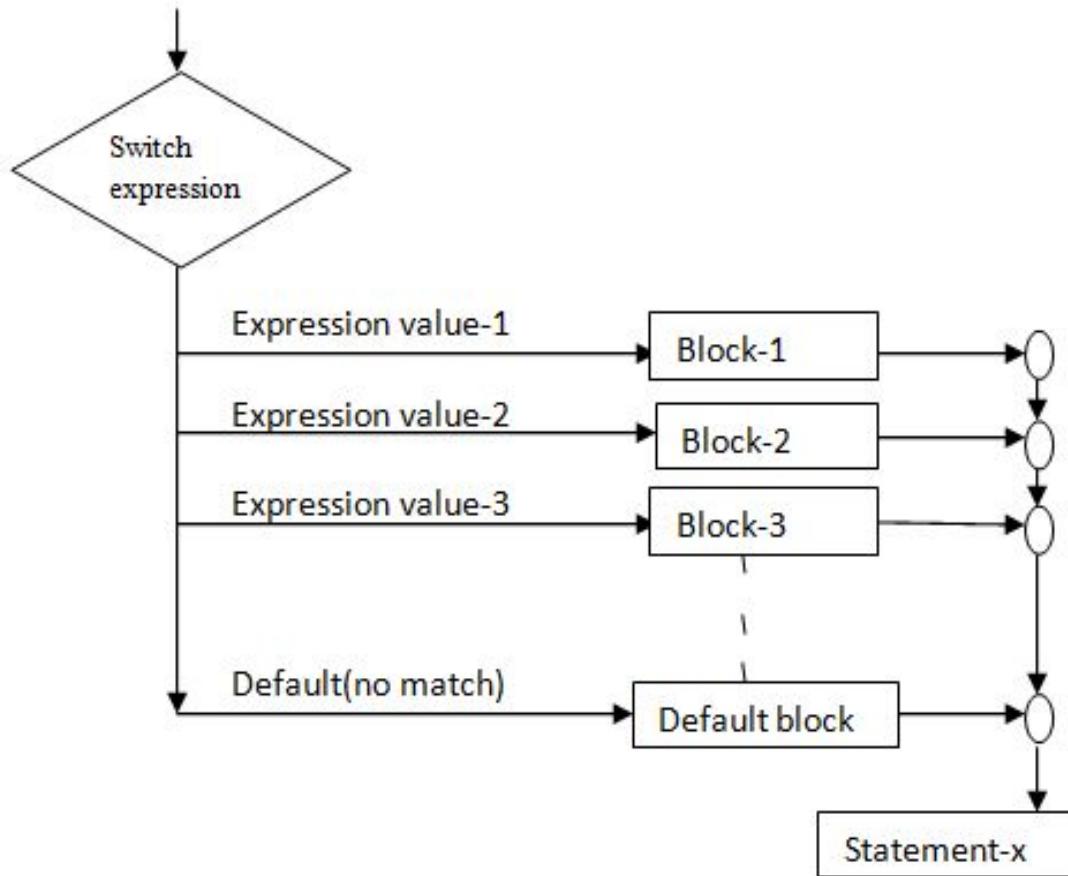


Fig. Selection process of switch statement

# Switch statement (Contd.)

- Example: Finding grade of a student.

```
int marks;
scanf("%d", &marks);
marks = marks / 10;
switch(marks){
    case 10:
    case 9:
    case 8:
        printf("A+");
        break;
    case 7:
        printf("A");
        break;
    case 6:
        printf("A-");
        break;
    default:
        printf("Fail");
        break;
}
```



# Switch statement (Contd.)

## □ Rules for switch statements:

- Switch expression must be an integral type (integer/character)
- Case label must be constants or constant expression
- No two case labels should be same
- Break will cause exit from switch block
- Default block executes if expression does not find a matching case label
- There can be at most one default label
- Nesting of switch statement is allowed



# Conditional Operator

If-else statement:

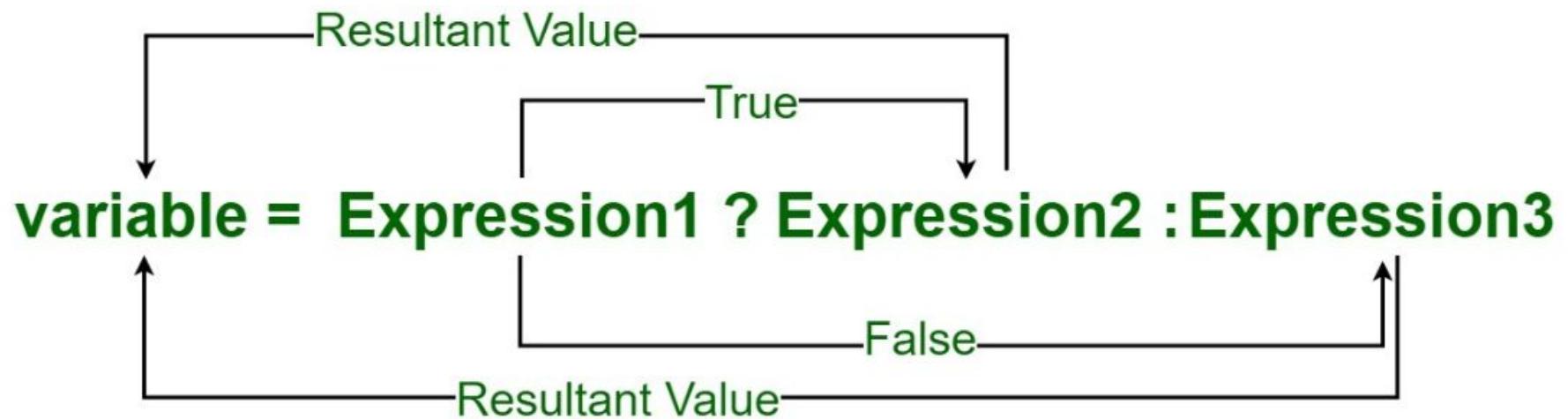
```
if(Expression1){  
    variable = Expression2;  
}  
else {  
    variable = Expression3;  
}
```

Equivalent conditional operator:

```
variable = Expression1 ? Expression2 : Expression3
```



# Conditional Operator (Contd.)



# Conditional Operator (Contd.)

Example-1:

$$y = 1.5x + 3, \quad \text{for } x \leq 2$$

$$y = 2x + 5, \quad \text{for } x > 2$$

Ans:  $y = (x > 2) ? (2x + 5) : (1.5x + 3)$

Example-2:

$$y = 4x + 100, \quad \text{for } x < 40$$

$$y = 300, \quad \text{for } x = 40$$

$$y = 4.5x + 150, \quad \text{for } x > 40$$

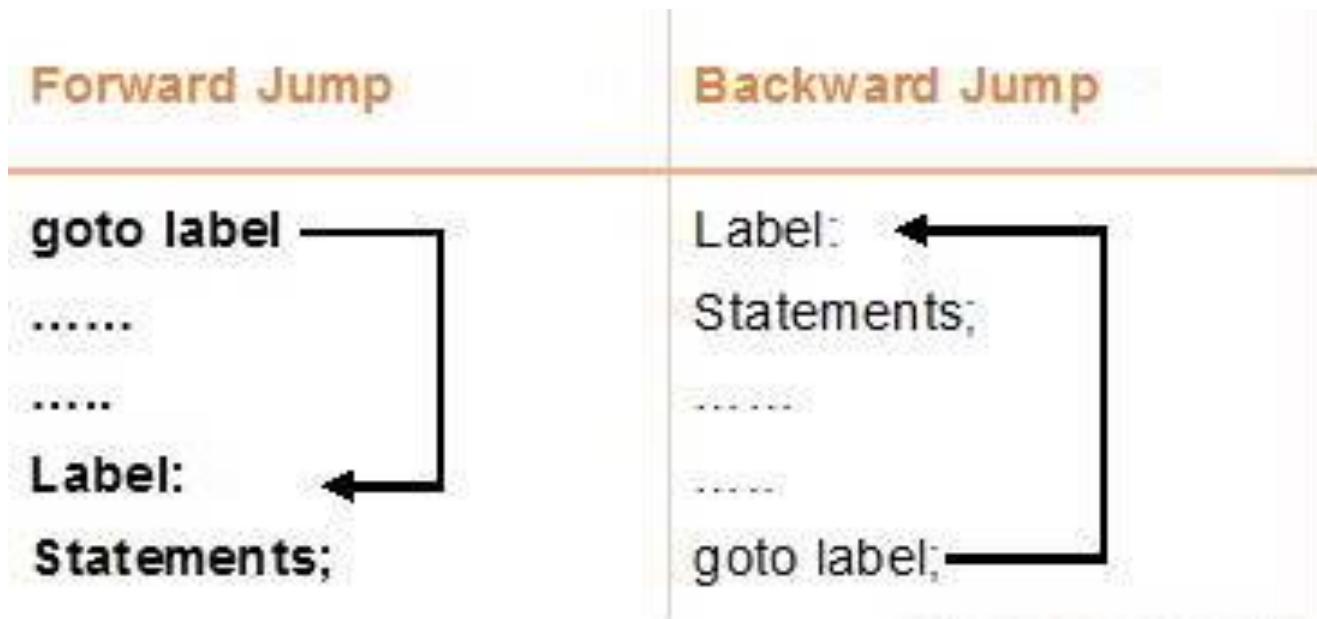
Ans:  $y = (x != 40) ? (x < 40 ? 4x+100 : 4.5x+150) : 300$

Short Code!  
But less readable!



# GOTO statement

- Branch unconditionally from one point to another
- Requires label to locate branching location
- label can be either before or after goto label



# GOTO statement (Contd.)

Example:

```
#include<stdio.h>
int main(){
    int n;
    start:
    scanf("%d", &n);
    if(n==0) goto finish;
    printf("Here we go again..!!\n");
    goto start;
    finish:
    printf("Bye bye..!!\n");
    return 0;
}
```



# Recap

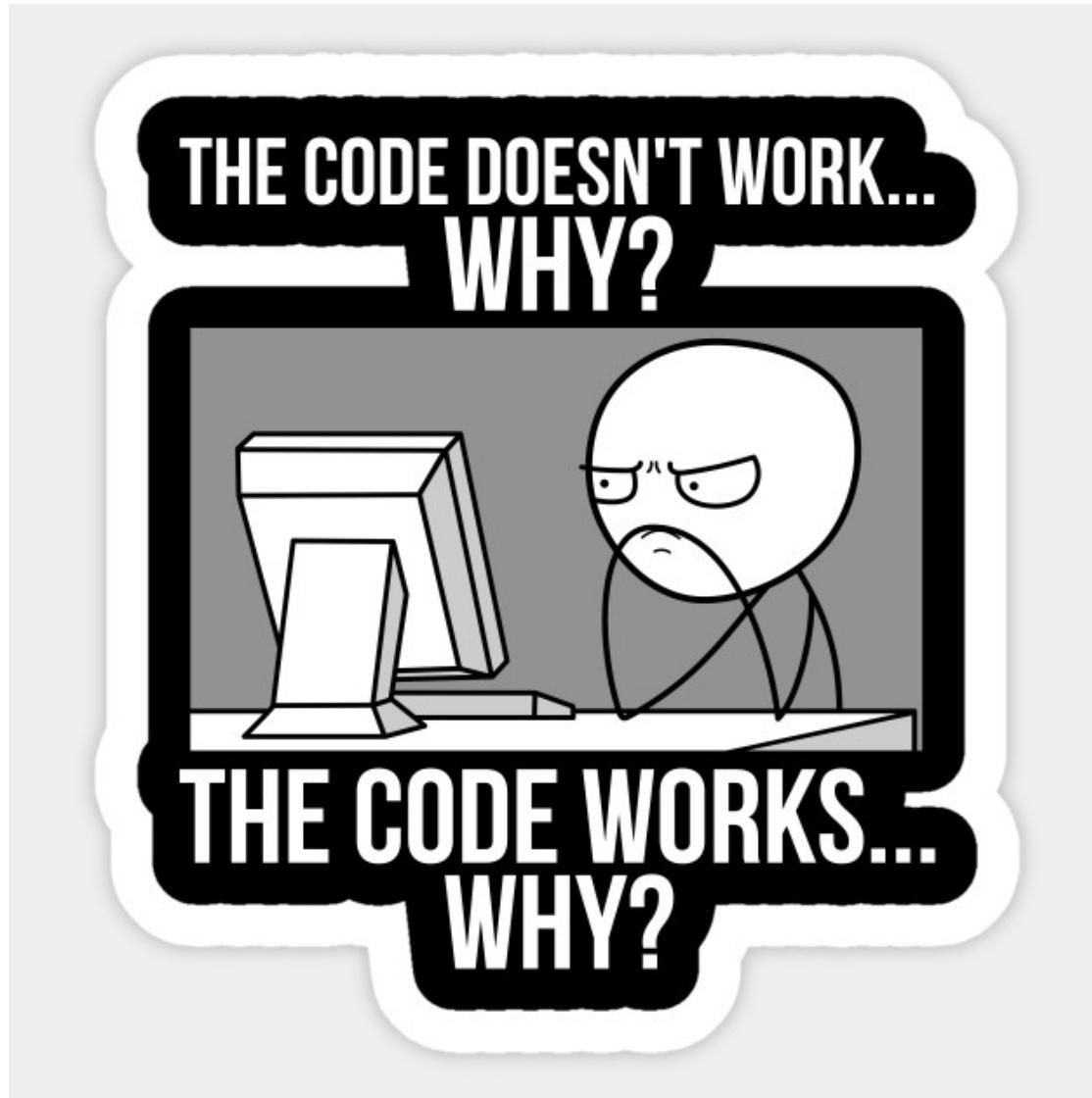
---

- Constants, Variables and Data Types
- Operators and Expressions
- Managing Input and Output Operations
- Decision Making and Branching

Next class: Decision Making and Looping!!



# Thank you



# Decision Making and Looping

---



Dept. of Computer Science & Engineering

# Lesson – 13, 14

Topic	Lesson Learning Outcome (at the end of the lesson students will be able to...)	Teaching Learning Methodology	Assessment Method
While, do-while, for loop	<ul style="list-style-type: none"><li>• Explain the structure of while, do-while and for loop</li><li>• Apply looping techniques to implement different programs</li><li>• Differentiate between counter-controlled and sentinel-controlled loop</li><li>• Compare between while, do-while and for loop</li></ul>	Multimedia Presentation, Question and Answer	Class Test, Exam, etc.



# Introduction (Loop)

- Statements in a program is executed one after another.

Ex. int main(){

```
    int n1, n2, sum;  
    scanf("%d%d", &n1, &n2);  
    sum=n1+n2; printf("Sum =  
    %d", sum);  
}
```

The diagram shows a code snippet within a rounded rectangle. A green arrow points from the text "Control statement" to the line "for(i=1; i<=5; ++i){". Another green arrow points from the text "Loop body" to the line "printf("Hello");". The code itself is:

```
for(i=1; i<=5; ++i){  
    printf("Hello");  
}
```

- Sometimes user wants to execute a set of statements repeatedly.
- Loop statements are used to repeat the execution of a set of statements until a certain condition is met.
- Consist of two parts:
  - Control statement: tests certain condition
  - Body of the loop: gets executed at each iteration of the loop



# Loop control structure

- Depending on the position of the control statement Loop can be classified as two types:
  1. Entry controlled: tests condition before executing the body
  2. Exit controlled: tests condition after executing the body

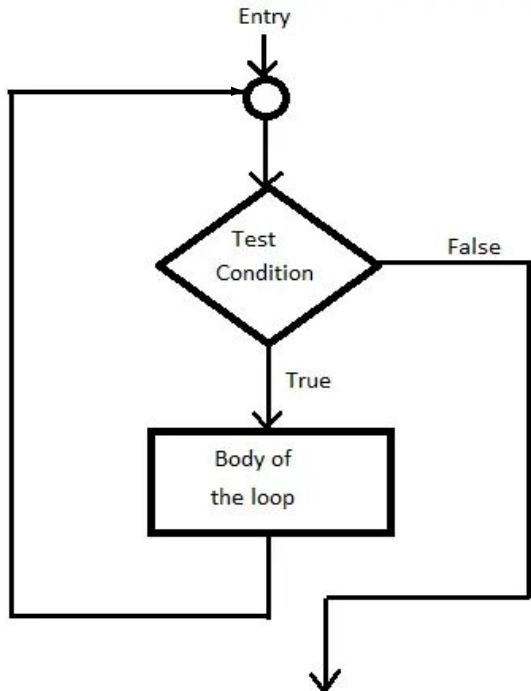


Fig. Entry Controlled Loop (Pre-test)

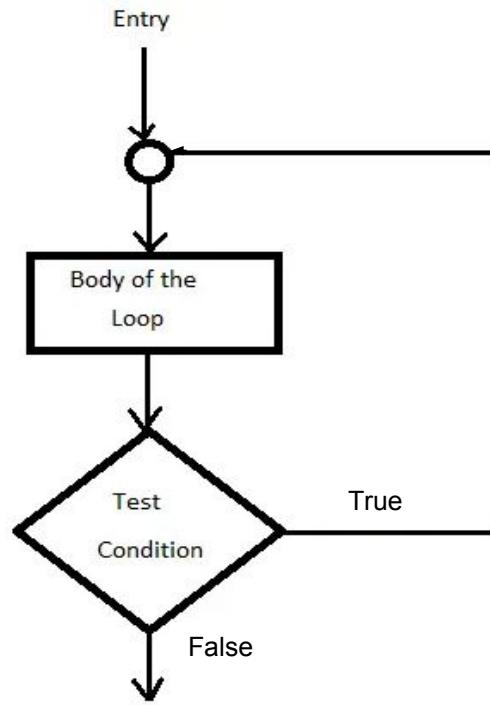


Fig. Exit Controlled Loop (Post-test)

# Types of Loop

- Three types of looping statements:
  1. while loop
  2. do-while loop
  3. for loop



# While Loop

- Simplest of all the looping structures in C
- Loop condition is tested before each iteration
- Its an entry controlled loop
- The basic format of the while statement is:

```
Control statement  
While(test-condition) {  
    ...  
    body of the loop  
    ...  
}
```

Loop body



# While Loop (Example)

- Printing “Hello!” 5 times:

```
int main(){  
    int i=1;  
    while(i<=5){  
        printf("Hello!\n");  
        i++;  
    }  
}
```

Output:  
Hello!  
Hello!  
Hello!  
Hello!  
Hello!

- Calculating  $y=x^n$ :

```
int main(){  
    int y=1, x=5, n=3;  
    while(n>0){  
        y*=x;  
        n--;  
    }  
    printf("%d\n", y);  
}
```

Output:  
125



# Do-While Loop

- Executes the body of the loop before the test is performed
- Loop body is executed at least once
- Its an exit controlled loop
- The basic format of the do-while statement is:

```
do {  
    ...  
    body of the loop   
    ...  
} While(test-condition)  

```



# Do-While Loop (Example)

- Taking input until a non-negative number is given:

```
int main(){  
    int n;  
    do{  
        printf("Enter a positive number: ");  
        scanf("%d", &n);  
    } while(n>0);  
}
```

## Output:

```
Enter a positive number: 4  
Enter a positive number: 10  
Enter a positive number: -5
```



# For Loop

- Another entry-controlled loop with more concise loop structure
- The basic format of the ‘for loop’ statement is:

```
for(initialization; test-condition; increment) {  
    ...  
    body of the loop    } Loop body  
    ...  
}
```

Control statement

Loop body

- Initialization of the control statement is done at the start of the loop
- Loop condition is tested before executing loop body
- Increment is performed after executing loop body



# For Loop (Example)

- Calculating sum of squares of integers upto n:
- i.e:  $1^2 + 2^2 + 3^2 + \dots + n^2 = ?$

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

Output:  
sum=14

**Note:** for loop allows all three actions (initialization, testing and increment) to be placed within the for statement itself, thus making them visible to the programmers and users, in one place.



# For Loop (Example)

- Checking if a number is prime or not:
- i.e: 2, 3, 5, 7... are prime but 4, 6, 8, 9, ... are not

```
int main(){
    int i, n=23, flag=0;
    for(i=2; i<n; ++i){
        if(n%i==0){
            flag=1;
        }
    }
    if(flag==0){
        printf("%d is prime\n", n);
    }
    else{
        printf("%d is not prime\n", n);
    }
}
```

Output:

23 is prime



# Additional Features of For Loop

- More than one variable can be initialized at the beginning

```
for(i=2, flag=0; i<n; i++)
```

- The test-condition may contain multiple operators

```
for(i=2, flag=0; i<n && flag==0; i++)
```

- Can use expression in initialization and increment section

```
for(i=2, flag=0; i<n && flag==0; i=i+1)
```

- One or more sections can be omitted. If the test-condition section is empty then the loop will run infinitely

```
i=2;  
flag=0  
for(; i<n && flag==0; i=i+1)
```



# Comparison of Three Loops

Topic	while	do-while	for
1) Initialization of control variable	Before the beginning of the loop	Before or within the body of the loop	At the initialization stage of the loop
2) Test condition	Before executing loop body	After executing loop body	Before executing loop body
3) Type	Entry controlled loop	Exit controlled loop	Entry controlled loop
4) Body of the loop	May or may not execute	Executes at least once	May or may not execute
5) Structure of the loop (Iterating 10 times)	<pre>n=1; while(n&lt;=10){     ...     ...     n++; }</pre>	<pre>n=1; do{     ...     ...     n++; } while(n&lt;=10);</pre>	<pre>for(n=1; n&lt;=10; n++){     ...     ... }</pre>



# Counter-control and Sentinel-controlled Loops

- Example: Count the number of iterations of the following programs:

Counter-controlled:

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

Will run 3 times

Sentinel-controlled:

```
int main(){  
    int n;  
    do{  
        printf("Enter a positive number: ");  
        scanf("%d", &n);  
    } while(n>0);  
}
```

How many times?!!



# Counter-control and Sentinel-controlled Loops (Contd.)

- Based on the nature of control variable, the loops may be classified into two categories:
  1. Counter-controlled loops
  2. Sentinel-controlled loops
- Counter-controlled: Number of iterations of the loop is known in advance. Also known as definite repetition loop.
- Sentinel-controlled: Number of iteration of the loop is not known in advance. Also known as indefinite repetition loop.



# Selecting a Loop

- Find out if the problem requires a pre-test or post-test loop
- For post-test loop or if we want to execute the body of the loop at least once, simply use “do while”
- For pre-test loop, we can choose either “while” or “for”
- For counter-based control use “for” loop
- For sentinel-based control use “while” loop

Note: Any kind of loops can be implemented by all the three control structures



# Exercise

- Given ‘n’, Compute factorial of ‘n’.

$$n! = 1 * 2 * 3 * \dots * n$$

- Find out the n’th Fibonacci number.

Fibonacci series: 1, 1, 2, 3, 5, 8, ...

- Given a number, write a program to reverse the number using loop.

E.g: input = 12045, output = 54021



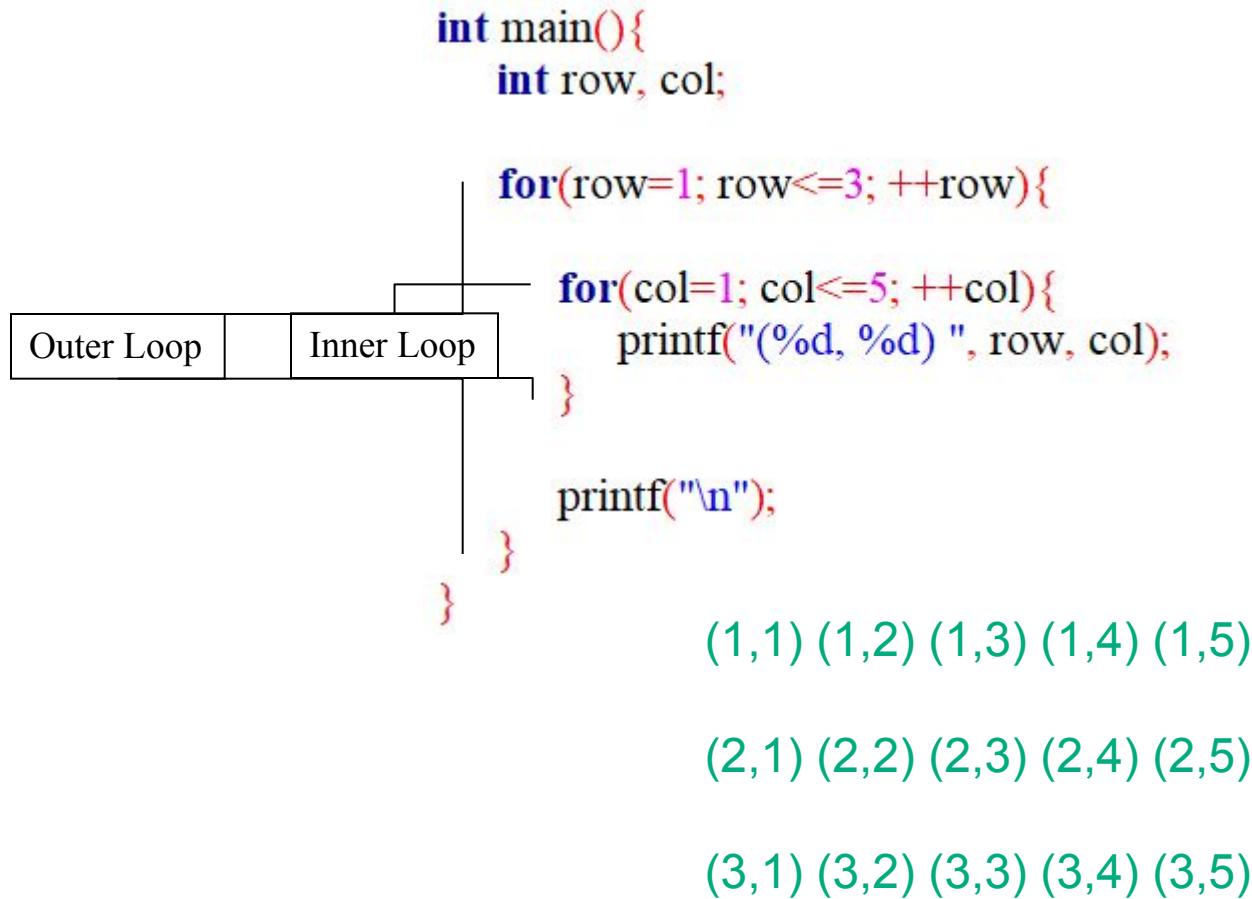
# Lesson – 15, 16

Topic	Lesson Learning Outcome (at the end of the lesson students will be able to...)	Teaching Learning Methodology	Assessment Method
Nested Loop	<ul style="list-style-type: none"><li>• Construct different nested loop structure such as for-while, while-for, for-for etc.</li><li>• Apply nested-loop for solving complex decision problems</li></ul>	Multimedia Presentation, Question and Answer	Class Test, Exam, etc.



# Nested-loop

- One loop statement within another loop statement.



# Nested Loop (Example)

- Print the following pattern of length ‘n’ using loop

n = 5  
\*  
\* \*  
\* \* \*  
\* \* \* \*  
\* \* \* \* \*

```
int main(){  
    int n, row, col;  
    scanf("%d", &n);  
  
    for(row=1; row<=n; ++row){  
        for(col=1; col<=row; ++col){  
            printf("* ");  
        }  
        printf("\n");  
    }  
}
```



A 5x5 grid of asterisks (\*). The first column has 1 asterisk. The second column has 2 asterisks. The third column has 3 asterisks. The fourth column has 4 asterisks. The fifth column has 5 asterisks. This represents the pattern for n=5.

# Nested Loop (Example)

- Print the following pattern of length ‘n’ using loop

n = 5  
5  
4 4  
3 3 3  
2 2 2 2  
1 1 1 1 1

```
int main(){
    int n, row, col;
    scanf("%d", &n);

    for(row=n; row>=1; --row){
        for(col=1; col<=n-row+1; ++col){
            printf("%d ", row);
        }
        printf("\n");
    }
}
```

5	5	5	5	5	5
4	4	4	4	4	4
3	3	3	3	3	3
2	2	2	2	2	2
1	1	1	1	1	1



# Nested Loop (Example)

- Print the following pattern of length ‘n’ using loop

n = 5  
\*  
\*\*\*  
\*\*\*\*\*  
\*\*\*\*\*  
\*\*\*\*\*

```
int main(){  
    int n, i, j;  
    scanf("%d", &n);  
  
    for(i=1; i<=n; ++i){  
        for(j=1; j<=n-i; ++j){  
            printf(" ");  
        }  
        for(j=1; j<=2*i-1; ++j){  
            printf("*");  
        }  
        printf("\n");  
    }  
}
```



5  
\*  
\*\*\*  
\*\*\*\*\*  
\*\*\*\*\*  
\*\*\*\*\*

# Nested Loop (Example)

- Print the following pattern of length ‘n’ using loop

n = 5  
1  
121  
12321  
1234321  
123454321

```
int main(){
    int n, i, j;
    scanf("%d", &n);

    for(i=1; i<=n; ++i){
        for(j=1; j<=n-i; ++j){
            printf(" ");
        }
        j=1;
        while(j<=2*i-1){
            if(j>i) printf("%d", 2*i-j);
            else printf("%d", j);
            j++;
        }
        printf("\n");
    }
}
```

5  
1  
121  
12321  
1234321  
123454321

# Lesson – 17

Topic	Lesson Learning Outcome (at the end of the lesson students will be able to...)	Teaching Learning Methodology	Assessment Method
Jump in loops: Break, Continue	<ul style="list-style-type: none"><li>• Use break and continue to skip part of a loop</li><li>• Write programs to implement the concept of break and continue</li></ul>	Multimedia Presentation, Question and Answer	Class Test, Exam, etc.



# Skipping a Part of a Loop

```
int i, n=23, flag=0;  
for(i=2; i<n; ++i){  
    if(n%i==0){  
        flag=1; } }  
if(flag==0) printf("%d is prime\n", n);  
else printf("%d is not prime\n", n);
```

Here, 'i' divides 'n'

```
int i;  
for(i=1; i<=5; ++i){  
    float x, y, div;  
    scanf("%f %f", &x, &y);  
    div = x/y; }  
printf("%0.2f\n", div);
```

- How can we skip a part of loop under certain conditions?
- use break/continue



# Skipping a Part of a Loop (Break)

```
int i, n=23, flag=0;  
for(i=2; i<n; ++i){  
    if(n%i==0){  
        flag=1;  
        break;    ↪ 'break' will cause immediate  
        }          exit from the loop  
    }  
if(flag==0) printf("%d is prime\n", n);  
else printf("%d is not prime\n", n);
```

- When a ‘**break**’ statement is encountered inside a loop:
  - the loop is immediately terminated



# Skipping a Part of a Loop (Break)

```
while (test expression) {  
    statement/s  
    if (test expression) {  
        — break;  
    }  
    statement/s  
}  
→
```

```
do {  
    statement/s  
    if (test expression) {  
        — break;  
    }  
    statement/s  
}  
while (test expression);  
→
```

```
for (initial expression; test expression; update expression) {  
    statement/s  
    if (test expression) {  
        — break;  
    }  
    statements/  
}  
→
```

NOTE: The break statement may also be used inside body of else statement.



# Skipping a Part of a Loop (Continue)

```
int i;
for(i=1; i<=5; ++i){
    float x, y, div;
    scanf("%f %f", &x, &y);
    if(y==0){
        printf("Invalid input");
        continue; ← 'continue' will cause immediate jump
    }                                to the beginning of the loop
    div = x/y;
    printf("%0.2f\n", div);
}
```

- When a ‘**continue**’ statement is encountered inside a loop:
  - the control skips the rest of the body of the loop, and
  - jumps to the beginning of the loop for next iteration



# Skipping a Part of a Loop (Continue)

```
→ while (test expression) {  
    statement/s  
    if (test expression) {  
        → continue;  
    }  
    statement/s  
}
```

```
do {  
    statement/s  
    if (test expression) {  
        → continue;  
    }  
    statement/s  
}  
→ while (test expression);
```

```
→ for (initial expression; test expression; update expression) {  
    statement/s  
    if (test expression) {  
        → continue;  
    }  
    statements/  
}
```

NOTE: The continue statement may also be used inside body of else statement.



# Break/Continue in nested loop

```
int main(){
    int i, j, k;
    for(i=1; i<=3; ++i){
        for(j=1; j<=5; ++j){
            if(j%2){
                continue;
            }
            printf("(%d %d) ", i, j);
        }
        printf("\n");
    }
}
```

Output: ??

```
(1 2) (1 4)
(2 2) (2 4)
(3 2) (3 4)
```

```
int main(){
    int i, j, k;
    for(i=1; i<=3; ++i){
        for(j=1; j<=5; ++j){
            if(j>i){
                break;
            }
            printf("(%d %d) ", i, j);
        }
        printf("\n");
    }
}
```

Output: ??

```
(1 1)
(2 1) (2 2)
(3 1) (3 2) (3 3)
```



# Break/Continue in nested loop

```
int main(){
    int i, j, n;
    for(i=1; i<=5; ++i){
        scanf("%d", &n);
        if(n<=0){
            printf("Invalid input\n");
            continue;
        }
        for(j=2; j<n; ++j){
            if(n%j==0){
                break;
            }
        }
        if(j==n) printf("Prime\n");
        else printf("Not Prime\n");
    }
}
```

Input:

-5  
7  
0  
15  
1

What will be the output?

```
-5
Invalid input
7
Prime
0
Invalid input
15
Not Prime
1
Not Prime
```



# Thank you

- End of Decision making and Looping
- See the **examples** and **exercises** of your book
- Make sure you have clear understanding of Looping



# Arrays

---



Dept. of Computer Science & Engineering

# Lesson – 18

Topic	Lesson Learning Outcome <b>(at the end of the lesson students will be able to...)</b>	Teaching Learning Methodology	Assessment Method
1-D Array (B1: Ch-7, B2: Ch-5)	<ul style="list-style-type: none"><li>• Explain the concept of array</li><li>• Declare and initialize one dimensional array</li><li>• Differentiate between runtime and compile time initialization of arrays</li></ul>	Multimedia Presentation , Question and Answer	Class Test, Exam, etc.



# Introduction

- An array is a **fixed-size sequenced collection of related data** items that share a common name.
- For example: salaries of a group of employees in an organization.
  - `int salary[10];` //declaration of salary of 10 employee
  - `salary[0]` //represents the salary of 1st employee
- Complete set of values- **array**
- Individual values- **elements**



# Types of Array

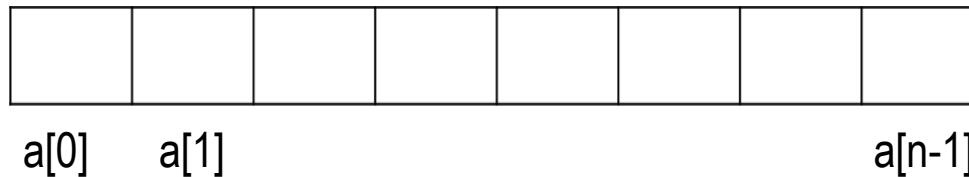
---

- One dimensional arrays
  - A list of items can be given one variable name using only one subscript and such a variable is called a one dimensional array.
- Two dimensional arrays
  - Situations where a table of values will have to be stored.
- Multidimensional arrays
  - Situations where more than one table is required.



# One-Dimensional Arrays

- The elements, the values of the items, of a one-dimensional array are conceptually arranged one after another in a single row.
- `int a[n];`



- Array elements are always numbered starting from 0, so the elements of an array of length  $n$  are indexed from **0 to  $n-1$**

# Declaration of Arrays

**type variable-name [size];**

- type specifies the type of element that will be contained in the array such as **int**, **float**, **char**
- Variable name must follow the rules of declaring variable that we have learned before
- Size indicates the maximum number of elements that can be stored in the array.

**float height [50];**

**int group [10];**

**char name [20];**



# Initialization of Arrays

---

- After declaration the elements of an array must be initialized.
- If an array is not properly initialized then it will contain **garbage** values.
- An array can be initialized at,
  - Compile time
  - Run time



# Compile Time Initialization

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

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

```
int a[10] = {1, 2, 3, 4, 5, 6};  
/* {1, 2, 3, 4, 5, 0, 0, 0, 0, 0} */
```

```
int a[10] = {0};  
/* {0, 0, 0, 0, 0, 0, 0, 0, 0, 0} */
```

```
int a[ ] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```



# Run Time Initialization

Arrays can also be explicitly initialized at runtime.

```
#include<stdio.h>

int main()
{
    int ara[10], sum[100], i;

    for(i=0;i<5;i++){
        scanf("%d", &ara[i]);
    }

    for(i=0;i<50;i++){
        if(i<25)
            sum[i]=0;
        else sum[i]=1;
    }

    return 0;
}
```



# Indexing

- C doesn't require that subscript bounds be checked. If a subscript goes out of range, the program's behavior is **undefined**.

```
int a[10]={0};  
print("%d", a[10])
```

- Array subscript may be any integer expression,

```
a[i+j*10];  
a[i] = b[i++];
```



# Lesson – 19

Topic	Lesson Learning Outcome <b>(at the end of the lesson students will be able to...)</b>	Teaching Learning Methodology	Assessment Method
Searching and sorting of array (B1: Ch-7, B2: Ch-5)	<ul style="list-style-type: none"><li>• Apply searching and sorting techniques in array</li><li>• Know about the complexity of searching and sorting</li></ul>	Multimedia Presentation , Question and Answer	Class Test, Exam, etc.



# Searching

---

- Searching is the process of finding the location of the specific element in an array.
- The specified element is called the search key
- Commonly used search techniques,
  - Sequential/linear search
  - Binary search



# Linear Search

Write a program to search an element in array,

```
| #include<stdio.h>

int main()
{
    int ara[100], key, flag=0,i;

    for(i=0; i<10;i++){
        scanf("%d", &ara[i]);
    }
    scanf("%d", &key);

    for(i=0;i<10;i++){
        if(ara[i]==key){
            printf("%d is found at location %d\n",key, i+1);
            flag=1;
        }
    }
    if(flag==0) printf("%d is not present in the array\n", key);
}
```

Activating Windows



# Complexity

- Best case complexity:  $O(1)$ 
  - searched element is the first element of the array
- Average case complexity:  $O(n)$
- Worst case complexity:  $O(n)$ 
  - searched element is the last element of the array



# Related problems

---

- Write a C program to find maximum and minimum element in an array.
- Write a C program to find second largest element in an array.
- Write a C program to count total number of even and odd elements in an array.
- Write a C program to count total number of negative elements in an array.



# Sorting

- Sorting is the process of arranging elements in the array according to their values.
- An array can be sorted either in ascending or descending order.
- Commonly used sorting techniques,
  - Bubble sort
  - Selection sort
  - Insertion sort

# Bubble Sort

Bubble sort can be implemented by following steps,

1. Start at index zero,
  - a. compare the element with the next one ( $a[0]$  &  $a[1]$ ), and **swap if  $a[0] > a[1]$ .**
  - b. Now compare  $a[1]$  &  $a[2]$  and **swap if  $a[1] > a[2]$ .**
  - c. Repeat this process until the end of the array. After doing this, the **largest element is present at the end**.
  - d. This whole thing is known as a pass. In the first pass, we process array elements from  $[0, n-1]$ .
2. Repeat step one but process array elements  **$[0, n-2]$**  because the last one, i.e.,  $a[n-1]$ , is present at its correct position. After this step, the **largest two elements are present at the end**.
3. Repeat this process  **$n-1$**  times.



# Bubble Sort

```
int main()
{
    int ara[100], i, j, temp,n;
    scanf("%d", &n);

    for(i=0; i<n;i++){
        scanf("%d", &ara[i]);
    }

    for(i=0;i<n-1;i++){
        for(j=0;j<n-i-1;j++){
            if(ara[j]>ara[j+1]){
                temp = ara[j];
                ara[j] = ara[j+1];
                ara[j+1] = temp;
            }
        }
    }
    printf("Sorted list in ascending order: ");
    for(i=0;i<n;i++)
        printf("%d ", ara[i]);
}
```



# Bubble Sort

- Write a C program to sort array elements in descending order.
  
- Complexity:  $O(n^2)$



# Lesson – 20

Topic	Lesson Learning Outcome <b>(at the end of the lesson students will be able to...)</b>	Teaching Learning Methodology	Assessment Method
Multidimensional array (B1: Ch-7, B2: Ch-5)	<ul style="list-style-type: none"><li>• Declare and initialize 2-D arrays</li><li>• Perform matrix multiplication</li><li>• Solve problems using multidimensional arrays</li></ul>	Multimedia Presentation , Question and Answer	Class Test, Exam, etc.



# Two-dimensional Arrays

- A two-dimensional array consists of both rows and columns of elements. It is essentially a matrix.
- To declare a two-dimensional array, we merely use two sets of square brackets. The first contains the **number of rows**. The second contains the **number of columns**.

```
//Creates a 2D array with 3 rows and 4 columns  
int vals[3][4];
```



# Two-dimensional Arrays

- ☐ Tables with rows and columns (m by n array)
- ☐ Like matrices: specify row, then column

	Col 0	Col 1	Col 2	Col 3
Row 0	ara[0][0]	ara[0][1]	ara[0][2]	ara[0][3]
Row 1	ara[1][0]	ara[1][1]	ara[1][2]	ara[1][3]
Row 2	ara[2][0]	ara[2][1]	ara[2][2]	ara[2][3]



# Initializing 2D arrays

- You can use additional braces to indicate when rows start and end, but you don't have to do that.
- ```
int val[3][4] = { {8,16,9,52},  
                   {3,15,27,6},  
                   {14,25,2,10}  
};
```
- ```
int val[3][4] = {8,16,9,52,3,15,27,6,14,25,2,10};
```

8	16	9	52
3	15	27	6
14	25	2	10

# Initializing 2D arrays

## Initialization

- `int b[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };`
- Initializers grouped by row in braces
- If not enough, unspecified elements set to zero
- `int b[ 2 ][ 2 ] = { { 1 }, { 3, 4 } };`

1	2
3	4

1	0
3	4

## Referencing elements

- Specify row, then column
- `printf( "%d", b[ 0 ][ 1 ] );`

# Using 2D arrays

---

- Just like 1D arrays, once you have specified the index, you are just working with a single variable of the given data type.
- 2D arrays work well with (for) loops like 1D arrays. However, to access all elements, typically you will need nested loops for 2D arrays.
- **Can you see why?**



# Example (1)

```
...
int main()
{
    int NUM_ROW=3;
    int NUM_COL=4;
    int row,col;
    int vals[NUM_ROW] [NUM_COL] = { {11, 12, 13, 14},
                                    {21, 22, 23, 24},
                                    {31, 32, 33, 34} };

    // output the transpose of the array
    for (row = 0; row < NUM_ROW; row++)
    {
        for (col = 0; col < NUM_COL; col++)
        {
            printf("%d",vals[row] [col]);
        }
        printf("\n");
    }
}
```



## Example (2)

```
...
int main()
{
    int NUM_ROW=3;
    int NUM_COL=4;
    int row,col;
    int vals[NUM_ROW] [NUM_COL] = { {11, 12, 13, 14},
                                    {21, 22, 23, 24},
                                    {31, 32, 33, 34} };

    // output the transpose of the array
    for (col = 0; col < NUM_COL; col++)
    {
        for (row = 0; row < NUM_ROW; row++)
        {
            printf("%d",vals[row] [col]);
        }
        printf("\n");
    }
}
```



# Matrix Addition Algorithm

---

- Add two  $n \times m$  matrices A, B;
  - for each row  $i$  of A do
    - for each column  $j$  of A do
      - $C[i][j] = A[i][j] + B[i][j];$
- Output matrices A, B and C.

# matrixAdd.cpp

```
...
int main()
{
    int NUM_ROWS=3; // number of matrix rows
    int NUM_COLS=2; // number of matrix columns
    int i,j;

    // Note: A, B and C have the same dimensions
    int A[NUM_ROWS][NUM_COLS] = { {3, 3}, {1, 1}, {2, 0} };
    int B[NUM_ROWS][NUM_COLS] = { {3, 0}, {3, 0}, {0, 1} };
    int C[NUM_ROWS][NUM_COLS];

    // C = A + B
    for (i=0; i<NUM_ROWS; i++)
    {
        for (j=0; j<NUM_COLS; j++)
        {
            C[i][j] = A[i][j] + B[i][j];
        }
    }
    ...
}
```



# matrixAdd.cpp (2)

```
...  
    // display C  
    for (i=0; i<NUM_ROWS; i++)  
    {  
        for (j=0; j<NUM_COLS; j++)  
        {  
            printf("%d",C[i][j]);  
        }  
        printf("\n");  
    }  
    printf("\n");  
...  
...
```



# Dynamic Arrays

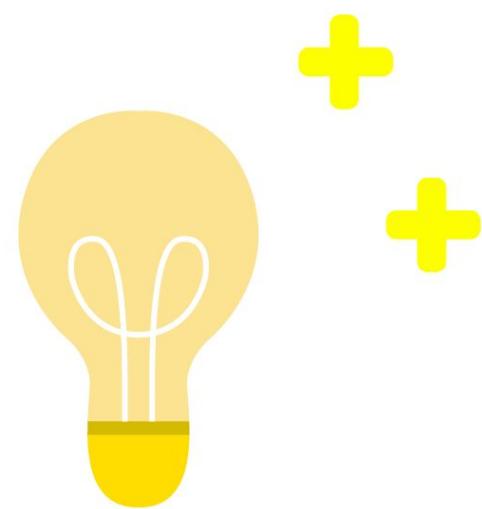
- The process of allocating memory at compile time is known as Static Memory Allocation and the arrays that receive static memory allocation are called Static Arrays.
- In C it is possible to allocate memory to arrays at run time. This feature is known as Dynamic Memory Allocation and the arrays created at run time are called Dynamic Arrays.



# Common Programming Errors

- Addressing indices that are out of bounds of the array range. This will run-time crash or at the very least a logical error. Be careful especially when using expressions to compute the indices.
  - **Remember, indexing starts with 0, not 1!!!**
- Forgetting to declare the array (either altogether or forgetting the [])
- **Assuming the array is initialized (to zero.)**

# HAPPY CODING



# Character Arrays and Strings

---



Dept. of Computer Science & Engineering

# Lesson – 21,22

Topic	Lesson Learning Outcome <b>(at the end of the lesson students will be able to...)</b>	Teaching Learning Methodology	Assessment Method
Character array and string (B1: Ch-8, B2: Ch-5)	<ul style="list-style-type: none"><li>• Declare and initialize string variables</li><li>• Read and write strings from and to terminals</li><li>• Use getchar, putchar, gets and puts functions</li><li>• know about the concept of field width in printf function</li></ul>	Multimedia Presentation , Question and Answer	Class Test, Exam, etc.



# Introduction

- C does not support strings as a data type. However it allows us to represent strings as character arrays.
- Strings are array of characters. Thus, a character array is called string.
- Each character within the string is stored within one element of the array successively.
- A string is always terminated by a null character (i.e. slash zero \0).



# Operations

□ Operations performed on a character array are,

- Reading and writing strings
- Combining strings together
- Copying one string to another
- Comparing strings for equality
- Extracting a portion of string



# Declaration and Initialization

- The general form of declaration of string is,

char string\_name[size];

char name2 [20];

- When the compiler assigns a character strings to a character array. It automatically supplies a null character ('\0') at the end of the string. Therefore, the size should be equal to the maximum number of characters in the string plus one.

char name [ 20 ] = “NEW YORK”

char name [10] = { ‘g’, ‘o’, ‘o’, ‘d’, ’\0’ };



# Declaration and Initialization

- Strings are initialized in either of the following two forms,

```
char name[4]={‘R’,‘A’,‘M’,‘\0’};
```

```
char name[]={‘R’,‘A’,‘M’,‘\0’};
```

```
char name[4]=“RAM”;
```

```
char name[]={“RAM”};
```

name[0]	name[1]	name[2]	name[3]
R	A	M	\0

- When we initialize a character array by listing its elements, the **null terminator or the size of the array must be provided explicitly.**



# Errors

- Following declarations are illegal in C,

char name[2] = “Bobby”;

char name[2];

name = “Bobby”; // is not allowed

char name1[2] = “Bobby”;

char name2[2];

name1=name2; // is not allowed



# Why Terminating Null Character (\0)?

- The array size is not always the size of the string and most often it is much larger than the string stored in it.
- Therefore, the last element of the array need not represent the end of the string.
- To determine the end of the string data terminating null character (\0) is used.
- This is used as “**end-of-string**” marker.



# Read and Write Strings

- Using **scanf()** function,

```
char ara[20];
scanf(“%s”, ara);
```

```
printf(“%s”, ara);
```

Input = Omar Sharif

Output = **Omar**

Think how we could solve this problem.



# Read and Write Strings

- The first solution came in my mind,

```
#include<stdio.h>

int main()
{
    char p1[80], p2[80];

    scanf("%s%s", p1, p2);

    printf("%s %s\n", p1, p2);
}
```

- Is it feasible? What if a person name has 5,6,7,8...20 parts?



# Read and Write Strings

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

```
int main()
{
    char ara[80], ch;
    int i=0;

    while(ch!='\n'){
        ch = getchar();
        //scanf("%c", &ch);
        ara[i]=ch;
        i++;
    }
    ara[i]='\0';
    printf("%s", ara);
}
```

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

```
int main()
{
    char ara[80];

    scanf("%[^\\n]", ara);
    printf("%s", ara);
}
```



# Read and Write Strings

- We have much simpler way to do that,

```
#include<stdio.h>

int main()
{
    char ara[80];
    gets(ara);

    printf("%s\n", ara);

    //Function is similar as printf
    puts(ara);
}
```

- Be careful when you read integer and character array consecutively. Some inconsistency may occur.



# Writing to Screen

- `printf("%w.ds\n", ara)`
- Prints the first **d** characters of the in the field width of **w**. Here, **s** is the format specifier.
  
- `printf("%10.4s",ara)`
- indicates that the first **4 characters** are to be printed in a field width of **10 columns**.

						a	a	a	\0
--	--	--	--	--	--	---	---	---	----

# Writing to Screen

## □ Some features of the %s specifications,

- When the field width is less than the length of the string, the entire string is printed.
- The integer value on the right of the decimal point specifies the number of characters to be printed.
- When the number of characters to be printed is specified as **zero**, nothing is printed.
- The **minus sign** in the specification causes the string to be printed **left-justified**.
- The specification **%.ns** prints the first n characters of the string.



# Example

```
#include<stdio.h>

int main()
{
    char ara[15]="United Kingdom";

    printf("%5s\n", ara);
    printf("%15.6s\n", ara);
    printf("%15.0s\n", ara);
    printf("%-15.6s\n", ara);
    printf("%.3s\n", ara);
}
```

Read the previous rules and match with this code. It will clear your understanding. Carefully read all the examples of the book.

```
United Kingdom
United
United
Uni
Process returned 4 (0x4)
Press any key to continue.
```

# Lesson – 23

Topic	Lesson Learning Outcome <b>(at the end of the lesson students will be able to...)</b>	Teaching Learning Methodology	Assessment Method
String handling functions (B1: Ch-8)	<ul style="list-style-type: none"><li>• Describe strcat(), strcmp(), strcpy() and strlen() functions</li><li>• Apply string-handling functions in program</li></ul>	Multimedia Presentation , Question and Answer	Class Test, Exam, etc.



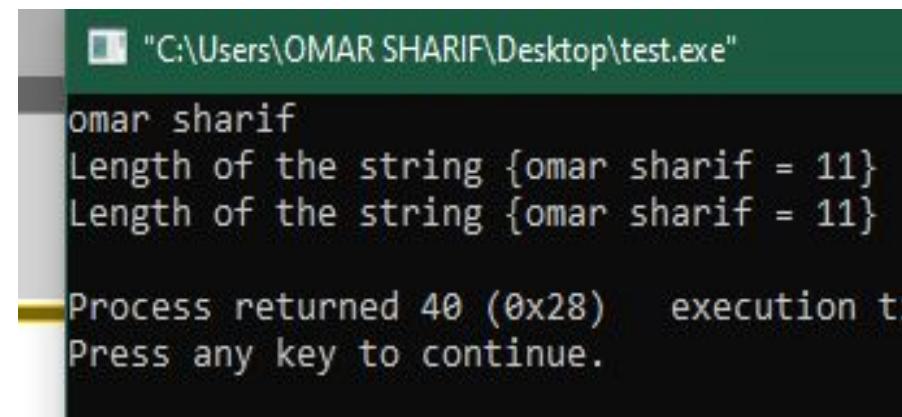
# Find the length of a String

## Write a program to find the length of a string

```
#include<stdio.h>

int main()
{
    char str[100];
    int l=0, x;

    gets(str);
    while(str[l]!='\0'){
        l++;
    }
    printf("Length of the string %s = %d\n", str, l);
    //Using built in function
    x= strlen(str);
    printf("Length of the string %s = %d\n", str, l);
}
```



```
C:\Users\OMAR SHARIF\Desktop\test.exe
omar sharif
Length of the string {omar sharif = 11}
Length of the string {omar sharif = 11}

Process returned 40 (0x28)    execution time:
Press any key to continue.
```

# Compare two strings

- C does not permit comparison of strings directly,

```
if(name1==name2)  
if(name == “abc”)
```

Above statements are not valid in C.

- So, think how can we compare two strings,
  - (concept of loop, null characters and conditional logic)



# Compare two strings

```
int main()
{
    char str1[100], str2[200];
    int i=0, x;

    gets(str1);
    gets(str2);

    while(str1[i]==str2[i] && str1[i]!='\0' && str2[i]!='\0'){
        i++;
    }

    if(str1[i]=='\0' && str2[i]=='\0')
        printf("Strings are equal\n");
    else printf("Strings are not equal\n");
}
```



# Compare two strings

- Compare strings using built in functions,

`strcmp(string1, string2)`

This function will return an integer value x.

- if ( $x==0$ ), both strings are equal
- if( $x<0$ ), string1 is **lexicographically smaller** string2
- if ( $x>0$ ), string2 is **lexicographically smaller** than string1

# Compare two strings

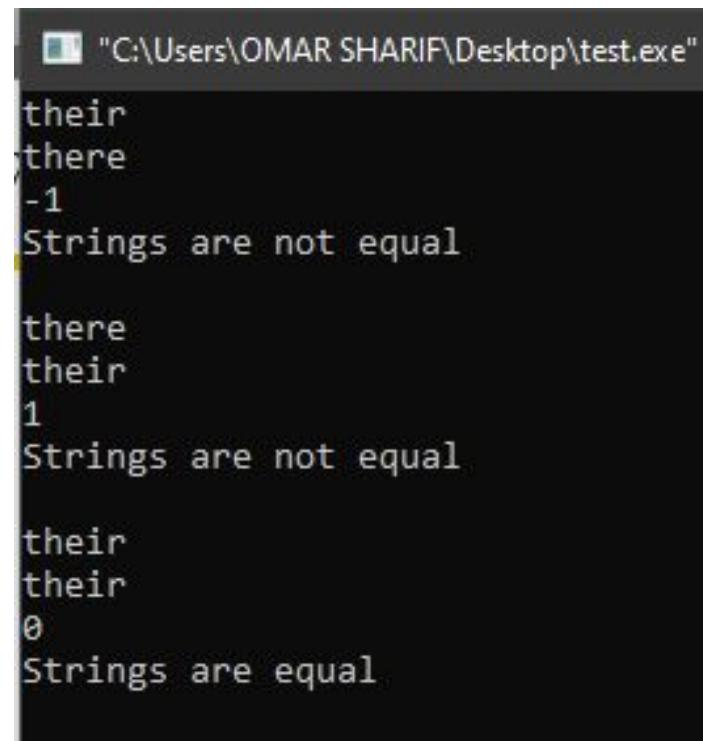
```
#include<stdio.h>
```

```
int main()
{
    char str1[100], str2[200];
    int i=0, x;

    gets(str1);
    gets(str2);

    x= strcmp(str1, str2);
    printf("%d\n", x);

    if(x==0)
        printf("Strings are equal\n");
    else printf("Strings are not equal\n\n");
}
```



The screenshot shows three separate runs of the program in a terminal window. Each run consists of two input strings followed by the output of the strcmp function and a message indicating whether the strings are equal or not.

- Run 1: Input strings are "their" and "there". The output shows x = -1, indicating the strings are not equal.
- Run 2: Input strings are "there" and "their". The output shows x = 1, indicating the strings are not equal.
- Run 3: Input strings are "their" and "their". The output shows x = 0, indicating the strings are equal.

# Copy one string to another

- Write a program to copy one string to another,

**string1= string2**

This statement is not permitted in C.

- We can use built in function to copy one string to another

**strcpy(string1, string2)**

Assigns contents of string2 into string1.



# Concatenation of Strings

- We can not join two strings together by simple arithmetic operation in C,

string3 = string1+string2  
string2= string2+ “hello”

- The characters from **string1** and **string2** should be copied into **string3** in order.
- The size of **string3** should be large enough to hold all the characters.



# Concatenation of Strings

```
int main()
{
    char str1[100], str2[200], str3[200];
    int i=0,j=0;
    gets(str1);
    gets(str2);

    while(str1[j]!='\0'){
        str3[i]=str1[j];
        i++; j++;
    }

    j=0;
    while(str2[j]!='\0'){
        str3[i]=str2[j];
        i++; j++;
    }

    printf("%d\n", i);
    str3[i]='\0';

    puts(str3);
}
```

```
omar
sharif
omar sharif
```

```
Process returned 0 (0x0)
Press any key to continue.
```



# Concatenation of Strings

- `strcat()` function joins two strings together.

**strcat(string1, string2)**

**strcat(string1, “good”)**

- When **strcat** is executed **string2** is appended to **string1**.
- It removes the **null character** at the end of **string1** and place **string2** from there.
- C permits nesting of strings,

**strcat(strcat(string1, string2), string3)**



# Concatenation of Strings

```
#include<stdio.h>

int main()
{
    char str1[100], str2[200];
    int i=0,j=0;

    gets(str1);
    gets(str2);

    strcat(str1, str2);

    puts(str1);
}
```

```
omar
sharif
omar sharif

Process returned 0 (0x0)
Press any key to continue.
```



# Other String Functions

## strncpy():

- Copies only the left-most n characters of the source string to the target string variables.
- This function has 3 parameters

**strncpy(string1, string2, 5)**

## strcmp():

- Compare the left-most n characters of s1 to s2 and returns,

**strcmp(s1, s2, 5)**

- 0 if they are equal
- Negative number, if s1 sub-string is less than s2
- Positive number otherwise



# Other String Functions

## strncat():

- Concatenate the left-most n characters of the source string to the target string variables.

**strncat(s1, s2, 5)**

## strstr():

**strstr(s1, s2)**

- **strstr** searches for string s1 to see whether the string s2 is contained in s2.
- If yes, the function returns the position of the first occurrence of the sub-string.
- Otherwise it returns a NULL pointer.

**\*\*Read the book carefully to get clear understanding of this functions.**



# Some problems on strings

- Count the number of vowels and consonants in a string
- Write a program to check whether a string is palindrome or not
- Write a program to lexicographically sort the characters of an array.
- Write a program to replace a particular word by another word in a given string.





---

# END OF SLIDE

# User Defined Functions

---



Dept. of Computer Science & Engineering

# Lesson – 24

Topic	Lesson Learning Outcome (at the end of the lesson students will be able to...)	Teaching Learning Methodology	Assessment Method
Introduction to Functions	<ul style="list-style-type: none"><li>• Explain the concept of modular programming</li><li>• Identify the necessity of user-defined functions</li><li>• Differentiate between user-defined functions and library functions</li></ul>	Multimedia Presentation, Question and Answer	Class Test, Exam, etc.



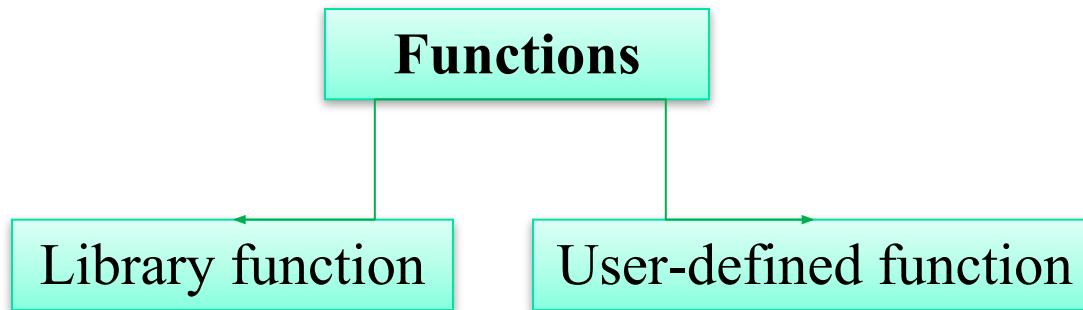
# Introduction (Functions)

- Function: A group of statements put together to perform a specific task
- E.g: printf, scanf, sqrt, strcat, strcpy etc.

This functions are defined in some library

- Can we design our own functions?

Yes! We can :D



# Function Category

## □ Library Function:

- Also known as pre-defined function
- Does not require to be written by us
- E.g: scanf, printf, sqrt, strstr etc.

## □ User-defined Function:

- Developed by the user at the time of writing a program
- They can be called from the main program repeatedly
- It's name is defined by the user
- E.g: addition, calc\_area, my\_func1 etc.

**Q: What type is the ‘main’ function!?**

**A: User-defined!**



# Need for User-defined Function

Calculating nCr:

$$nCr = \frac{n!}{r! * (n - r)!}$$

```
int i, j, k;
int n, r;
scanf("%d %d", &n, &r);
int fact_n=1, fact_r=1, fact_nr=1;
for(i=1; i<=n; ++i){
    fact_n=fact_n*i;
}
for(i=1; i<=r; ++i){
    fact_r=fact_r*i;
}
for(i=1; i<=n-r; ++i){
    fact_nr=fact_nr*i;
}
int nCr=fact_n/(fact_r*fact_nr);
printf("%d\n", nCr);
```

Repeated task

- The program might get too big and becomes hard to debug, test or maintain



# Modular Programming

- Organizing a large program into small, independent program segments
- Independent program segments are known as module
- Program is more readable with modular approach
  
- Properties:
  - Each module performs some task
  - Communication between modules is allowed by function call
  - No direct communication between modules



# Lesson – 25

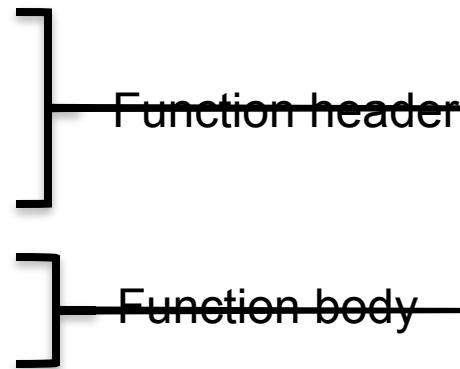
Topic	Lesson Learning Outcome (at the end of the lesson students will be able to...)	Teaching Learning Methodology	Assessment Method
Function declaration	<ul style="list-style-type: none"><li>• Find out the elements of a user-defined functions</li><li>• Define function prototype</li></ul>	Multimedia Presentation, Question and Answer	Class Test, Exam, etc.



# Function Elements

- A function has the following elements:

- Function type
- Function name
- List of parameters
- Function statements
- A return statement



Note: Function type is also known as return type

```
function_type function_name(parameter_list) {  
    ...  
    ...  
    function statements  
    ...  
    ...  
    return statement;  
}
```



# Function Elements (Contd.)

## □ Function for calculating factorial:

```
function_type function_name(parameter_list) {  
    ...  
    ...  
    function statements  
    ...  
    ...  
    return statement;  
}
```

```
int factorial(int n) {  
    int i, ans=1;  
    for(i=1; i<=n; ++i){  
        ans=ans*i;  
    }  
    return ans;  
}
```



# Function Elements (Contd.)

- Function for calculating area of a triangle:

```
double triangle_area(double base, double height) {  
    double ans;  
    ans = 0.5*base*height;  
    return ans;  
}
```

- Printing if a number is even or printing:

```
void even_odd() {  
    int n;  
    scanf("%d", &n);  
    if(n%2) printf("Odd");  
    else printf("Even");  
    return;  
}
```



# Function Call

## □ Calculating nCr:

```
#include<stdio.h>

int factorial(int n) {
    int i, ans=1;
    for(i=1; i<=n; ++i){
        ans=ans*i;
    }
    return ans;
}

int main(){
    int n, r, fact_n, fact_r, fact_nr, nCr;
    scanf("%d %d", &n, &r);
    fact_n = factorial(n);
    fact_r = factorial(r);
    fact_nr = factorial(n-r);
    nCr = fact_n/(fact_r*fact_nr);
    printf("%d", nCr);
}
```



# Function Call (Contd.)

## □ Calculating area:

```
double triangle_area(double base, double height) {  
    double ans;  
    ans = 0.5*base*height;  
    return ans;  
}  
  
int main(){  
    double res = triangle_area(3, 5.5);  
    printf("%lf", res);  
}
```

## □ Even-Odd:

```
void even_odd() {  
    int n;  
    scanf("%d", &n);  
    if(n%2) printf("Odd");  
    else printf("Even");  
    return;  
}  
  
int main(){  
    even_odd();  
}
```



# Similarities between Functions and Variables

---

- Function and variables both are identifiers
- Rules for declaring function name is similar to the rules of variable declaration
- Like variables, functions have a type associated with them
- Like variables, function name and their type must be declared before they are used in a program



# Function Prototyping

- Declaring function without function-body
- Similar to function header with semicolon at the end
- Calculating area:

```
#include<stdio.h>

double triangle_area(double base, double height);

int main(){
    double res = triangle_area(3, 5.5);
    printf("%lf", res);
}

double triangle_area(double base, double height) {
    double ans;
    ans = 0.5*base*height;
    return ans;
}
```



# Prototypes: Yes or No

- Prototype declaration is not essential in C
- If a function has not been declared before it is used, C will assume the return type is integer
- If the assumption is wrong it will give error
- So we must always declare function before it is used

```
#include<stdio.h>

int main(){
    printf("%d\n", func(5));
}

int func(int n){
    return n/2.0;
}
```

Output: 2

```
#include<stdio.h>

int main(){
    printf("%f\n", func(5));
}

float func(int n){
    return n/2.0;
}
```

Error!!



# Some Concepts Related to Function

## □ Some concepts related to functions:

- **Function Declaration:** Declare a function that is to be used later.  
Also known as prototyping

```
double triangle_area(double base, double height);
```

- **Function Definition:** Independent program module where function logic is implemented

```
double triangle_area(double base, double height) {  
    double ans;  
    ans = 0.5*base*height;  
    return ans;  
}
```

- **Function Call:** Program that calls the function

```
int main(){  
    double res = triangle_area(3, 5.5);  
    printf("%lf", res);  
}
```



# Lesson – 26, 27

Topic	Lesson Learning Outcome (at the end of the lesson students will be able to...)	Teaching Learning Methodology	Assessment Method
Categories of Functions	<ul style="list-style-type: none"><li>• Classify different types of functions based on their arguments and return types</li><li>• Apply nesting of functions</li><li>• Describe the concept of scope, visibility and lifetime of variables in functions</li></ul>	Multimedia Presentation, Question and Answer	Class Test, Exam, etc.



# Category of Functions

- Depending on function parameters and return type functions can be one of the following categories:
  - Functions with no arguments and no return values
  - Functions with arguments but no return values
  - Functions with arguments and one return value
  - Functions with no arguments but return a value
  - Functions that return multiple values



# Category of Functions (Contd.)

## □ Functions with no arguments and no return values

```
void even_odd(){
    int n;
    scanf("%d", &n);
    if(n%2) printf("Odd\n");
    else printf("Even\n");
    return;
}
```

```
int main(){
    even_odd();
}
```

Note: Here, Function call is a single statement and can not be used in an expression



# Category of Functions (Contd.)

## □ Functions with arguments but no return values

```
void even_odd(int n){  
    if(n%2) printf("Odd\n");  
    else printf("Even\n");  
    return;  
}
```

```
int main(){  
    int n;  
    scanf("%d", &n);  
    even_odd(n);  
}
```

Note: Changing values inside function will have no effect on the variables of main function. Only the values are passed to the function. (For pointer its different)



# Category of Functions (Contd.)

## □ Functions with arguments and one return values

```
double triangle_area(double base, double height) {  
    double ans;  
    ans = 0.5*base*height;  
    return ans;  
}
```

```
int main(){  
    double res = triangle_area(3, 5.5);  
    printf("%lf", res);  
}
```

Note: Here, return value of the function can be stored in  
a variables or can be used as an expression



# Category of Functions (Contd.)

- No arguments but returns a value

```
int get_id(){  
    printf("What is your ID: ");  
    int id;  
    scanf("%d", &id);  
    return id;  
}
```

```
int main(){  
    printf("ID = %d\n", get_id());  
}
```

Note: Here, return value of the function is directly shown in the output. You can also store it into separate variable.



# Category of Functions (Contd.)

- Functions that return multiple values
  - Can do this by using address operator (&) and indirection operator (\*)

```
void circle(float r, float *area, float *perimeter){  
    float PI=acos(-1); //3.1415...  
    *area = PI*r*r;  
    *perimeter = 2*PI*r;  
}
```

```
int main(){  
    float A, P;  
    circle(1, &A, &P);  
    printf("A = %.2f, P = %.2f\n", A, P);  
}
```

Note: Here, we are passing address location and changing the actual values at that location. This is known as 'pass by pointer'/'call by reference'



# Nesting of Functions

C allows to use function  
inside another function

```
int factorial(int n) {  
    int i, ans=1;  
    for(i=1; i<=n; ++i){  
        ans=ans*i;  
    }  
    return ans;  
}  
  
int calc_nCr(int n, int r){  
    int fact_n, fact_r, fact_nr, nCr;  
    fact_n = factorial(n);  
    fact_r = factorial(r);  
    fact_nr = factorial(n-r);  
    nCr = fact_n/(fact_r*fact_nr);  
    return nCr;  
}  
  
int main(){  
    int n, r;  
    scanf("%d %d", &n, &r);  
    printf("%dC%d = %d\n", n, r, calc_nCr(n, r));  
}
```

Output:

```
8 3  
8C3 = 56
```



# Scope, Visibility and Lifetime

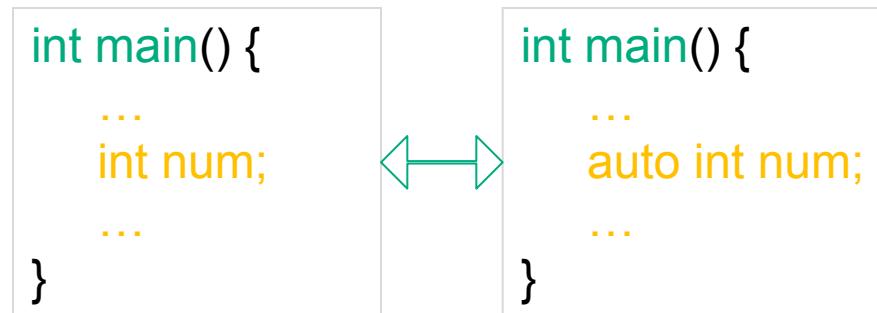
- **Scope:** Region of the program where a variable is actually available for use
- **Visibility:** Refers to the accessibility of a variable from the memory
- **Lifetime:** The time duration for which a variable exists in the memory during execution
- In C, variables have data type and also a storage class. The following storage classes are most relevant to functions:
  - 1) Automatic variables
  - 2) External variables
  - 3) Static variables
  - 4) Register variables



# Storage Class

## □ Automatic variables:

- Also known as private or local variables of a function
- Declared inside a function in which they are to be utilized
- They are created when the function is called
- They are destroyed automatically when the function is exited
- Their values can not be changed outside the function
- Variable inside a function without storage class specification is by default an automatic variable



# Storage Class (Contd.)

## □ External variables:

- Also known as global variables
- They are alive and active throughout the whole program
- Can be accessed by any function in the program
- They are declared outside a function

```
int num;  
  
int func() {  
    extern int num; //Refers to global  
    ...  
}  
  
int main() {  
    ...  
    ...  
}
```



# Storage Class (Contd.)

## □ Static variables:

- Value of static variables persists until the end of the program
- It can be either internal or external depending on the placement
- Static variable remains alive throughout the remainder of the program
- It is initialized once and never initialized again

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

Output:  
1  
2  
3



# Storage Class (Contd.)

## □ Register variables:

- We can tell compilers to store a variable inside registers, instead of memory
- Register access time is much faster than memory access time
- Only a few variables can be stored in the register
- Register variables will be converted to non-register variables if the number of variable limit is reached
- Most compilers allow only **int** or **char** to be placed in the register
- E.g: **register int count;**



# Scope and Lifetime of Each Storage Class

## Scope and Lifetime of Declarations

Storage Class	Where declared	Visibility (Active)	Lifetime (Alive)
extern	Before all functions in a file (cannot be initialized)	Entire file plus other files where variable is declared extern and the file where originally declared as global	Global
Static	Before all functions in a file	Only in that file	Global
None or auto	Inside a function (or a block)	Only in that function or block	Until end of function or block
Register	Inside a function or block	Only in that function or block	Until end of function or block
static	Inside a function	Only in that function	Global



# Lesson – 28

Topic	Lesson Learning Outcome (at the end of the lesson students will be able to...)	Teaching Learning Methodology	Assessment Method
Recursion	<ul style="list-style-type: none"><li>• Know about the concept and importance of recursion</li><li>• Apply recursion to solve complex problems</li><li>• Pass array as function argument</li></ul>	Multimedia Presentation, Question and Answer	Class Test, Exam, etc.



# Recursion

- **Recursion:** Special case of nested function where a function call itself
  - Require base case

```
void func() {  
    base case  
    ...  
    func();   ↙ Call to itself  
    ...  
    return;  
}  
  
int main() {  
    func();  
}
```



# Recursion (Contd.)

## Example: Calculating factorial

```
factorial(0) = 1  
factorial(n) = n*factorial(n-1)
```

```
int factorial(int n){  
    int ans;  
    if(n==0) ans=1;  
    else ans=n*factorial(n-1);  
    return ans;  
}
```

```
int main(){  
    printf("%d\n", factorial(5));  
}
```



# Recursion (Contd.)

## □ Example: Calculating Fibonacci

```
Fib(0) = 1  
Fib(1) = 1  
Fib(n) = Fib(n-1) + Fib(n-2)
```

```
int fib(int n){  
    int ans;  
    if(n==0 || n==1) ans=1;  
    else ans=fib(n-1) + fib(n-2);  
    return ans;  
}  
  
int main(){  
    printf("%d\n", fib(5));  
}
```



# Passing Array to Functions

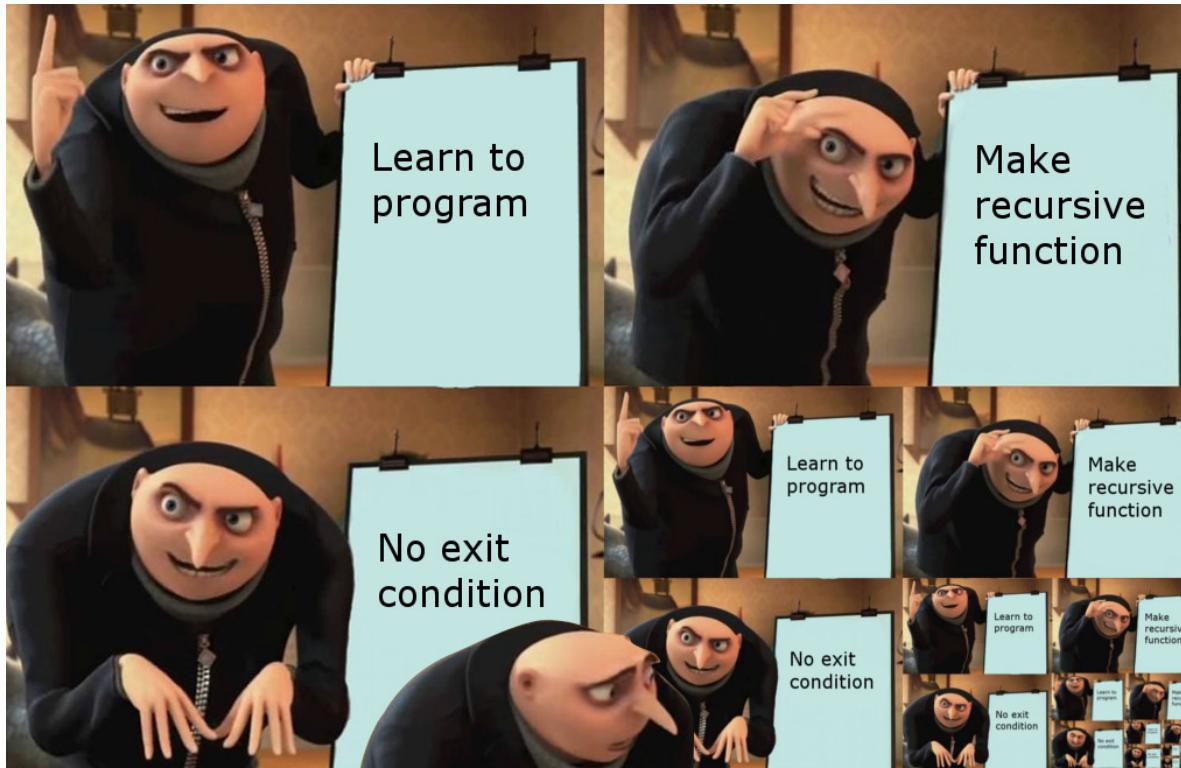
- Function ‘adder’ tries the add the elements of the array:
  - Function is called by passing only the name of the array
  - In the definition the formal parameter must be of array type.  
Size of the array does not need to specify
  - Can also use in terms of pointer: int \*ara

```
int adder(int ara[], int sz){  
    int tot=0;  
    for(int i=0; i<sz; ++i) tot+=ara[i];  
    return tot;  
}  
  
int main(){  
    int ara[]={1, 5, 7, 10, -4};  
    int sum=add(ara, 5);  
    printf("Sum = %d\n", sum);  
}
```



# Thank you

- End of Function and Recursion
- See the **examples** and **exercises** of your book
- Make sure you have clear understanding of working procedure of Function



# Structures and Unions

---



Dept. of Computer Science & Engineering

# Lesson – 29, 30

Topic	Lesson Learning Outcome (at the end of the lesson students will be able to...)	Teaching Learning Methodology	Assessment Method
Structure	<ul style="list-style-type: none"><li>• Define structure variable</li><li>• Compare between array and structure</li><li>• Apply structure in a function</li></ul>	Multimedia Presentation, Question and Answer	Class Test, Exam, etc.



# Introduction

- We want to store information of 100 students. E.g: ID, Dept., CGPA
- How to do this? **Can use array to store the info**
- But array represents data items that belongs to same type such as ‘int’ or ‘float’. But not both together.
  - int id[100];
  - char dept[100][10];
  - float cgpa[100];



# Introduction (Contd.)

## □ Storing info using array:

```
int main0{
    int i;
    int id[100];
    char dept[100][10];
    float cgpa[100];
    printf("Input:\n");
    for(i=0; i<3; ++i){
        scanf("%d %s %f", &id[i], &dept[i], &cgpa[i]);
    }
    printf("\nOutput:\n");
    for(i=0; i<3; ++i){
        printf("%d %s %0.2f\n", id[i], dept[i], cgpa[i]);
    }
}
```

Input:  
1304027 CSE 3.89  
1304003 CSE 3.90  
1301001 CE 3.50

Output:  
1304027 CSE 3.89  
1304003 CSE 3.90  
1301001 CE 3.50

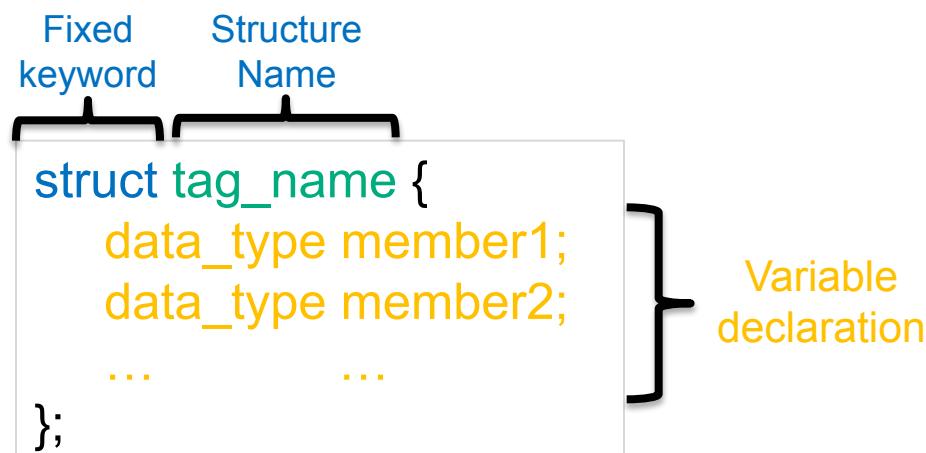
Process returned 0 (0x0) execution time : 0.484 s  
Press any key to continue.

Can we put different data in a single variable?  
That's where structure come into action



# Structures

- ‘C’ supports a constructed data type known as structure, a mechanism for packing data of different types
- **Structures:** A structure is a convenient tool for handling a group of logically related data items.
- E.g: It can be used to represent a set of attributes together, such as: ID, Dept., CGPA
- Format of a structure:



# Defining a Structures

The diagram illustrates the syntax of defining a structure in C. It shows the keyword 'struct' followed by a user-defined tag name 'tag\_name'. Inside the curly braces, there are declarations for members: 'data\_type member1;' and 'data\_type member2;'. Ellipses indicate additional members. A large brace on the right side groups all the declarations under the heading 'Variable declaration'.

```
struct tag_name {
    data_type member1;
    data_type member2;
    ...
};
```

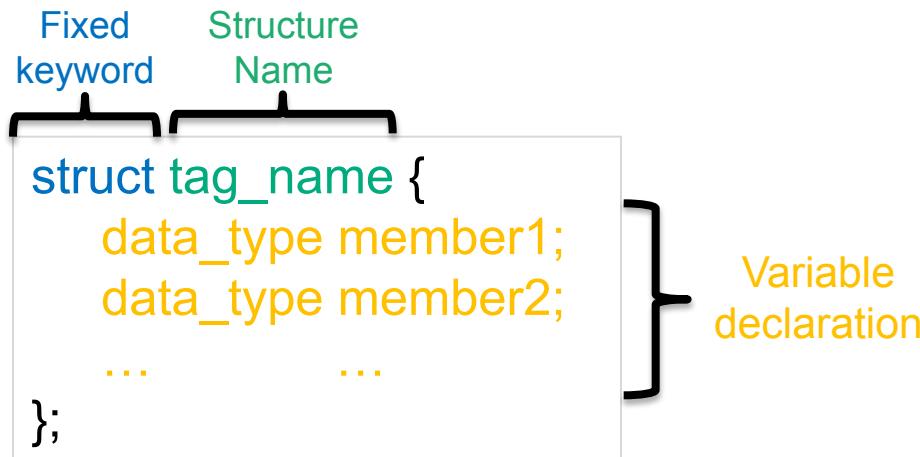
Fixed keyword      Structure Name

Variable declaration

- Structures format must be defined before you can use them
- ‘**struct**’ keyword indicates starting of the structure
- The ‘**tag\_name**’ is a user defined name (follows rule of variable declaration)
- Inside structure, each ‘**members**’ are declared independently
- Must end with a semicolon



# Declaring variables and Accessing members



- Declare structure variables:

```
struct tag_name variable1, variable2;  
struct tag_name ara[100];
```

- Accessing structure members:

```
variable1.member1;  
variable1.member2;  
ara[i].member1;
```



# Structure Example

- Representing single student information:

```
struct student{  
    int id;  
    char dept[10];  
    float cgpa;  
};  
  
int main(){  
    struct student info;  
    printf("Input:\n");  
    scanf("%d %s %f", &info.id, &info.dept, &info.cgpa);  
  
    printf("\nOutput:\n");  
    printf("%d %s %0.2f\n", info.id, info.dept, info.cgpa);  
}
```

Input:  
1304027 CSE 3.89

Output:  
1304027 CSE 3.89

Process returned 0 (0x0) execution time : 5.512 s  
Press any key to continue.



# Structure Example (Contd.)

- Representing all students information:

```
struct student{  
    int id;  
    char dept[10];  
    float cgpa;  
};  
  
int main(){  
    struct student info[100];  
    printf("Input:\n");  
    for(int i=0; i<3; ++i){  
        scanf("%d %s %f", &info[i].id, &info[i].dept, &info[i].cgpa);  
    }  
    printf("\nOutput:\n");  
    for(int i=0; i<3; ++i){  
        printf("%d %s %0.2f\n", info[i].id, info[i].dept, info[i].cgpa);  
    }  
}
```

Input:  
1304027 CSE 3.89  
1304003 CSE 3.90  
1301001 CE 3.50

Output:  
1304027 CSE 3.89  
1304003 CSE 3.90  
1301001 CE 3.50

```
Process returned 0 (0x0) execution time : 0.484 s  
Press any key to continue.
```



# Comparison between Array and Structure

Array	Structure
Collection of related data elements of same type	Can have elements of different data types
It's a derived data type	It's a programmer-defined one
Behaves like built-in data types	Behaves like user-defined data types
We only need to declare the variable before using them	We need to design and declare structures before they can be used



# Application of Structure in Function

- Information of CSE student's only:

```
struct student{
    int id;
    char dept[10];
    float cgpa;
};

void show_cse(struct student info[6], int sz){
    printf("\nCSE Students are:\n");
    for(int i=0; i<sz; ++i){
        if(strcmp(info[i].dept, "CSE")==0){
            printf("%d %s %.2f\n", info[i].id, info[i].dept, info[i].cgpa);
        }
    }
}

int main(){
    struct student info[6];
    printf("Input:\n");
    for(int i=0; i<5; ++i){
        scanf("%d %s %f", &info[i].id, &info[i].dept, &info[i].cgpa);
    }
    show_cse(info, 5);
}
```

```
Input:
27 CSE 3.89
3 CSE 3.90
1 CE 3.5
5 ME 2.9
54 CSE 3.5

CSE Students are:
27 CSE 3.89
3 CSE 3.90
54 CSE 3.50
```



# Application of Structure in Function (Contd.)

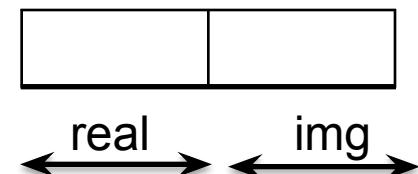
## □ Adding two complex numbers:

```
struct cmplx{  
    float real;  
    float img;  
};
```

```
struct cmplx add(struct cmplx c1, struct cmplx c2){  
    struct cmplx ans;  
    ans.real=c1.real+c2.real;  
    ans.img=c1.img+c2.img;  
    return ans;  
}
```

```
int main(){  
    struct cmplx c1, c2, c3;  
    c1.real=1, c1.img=2;  
    c2.real=0.5, c2.img=1.6;  
    c3=add(c1, c2);  
    printf("% .2f + % .2fi\n", c3.real, c3.img);  
}
```

Storage:



Output:

```
1.50 + 3.60i
```

# Lesson – 31

Topic	Lesson Learning Outcome (at the end of the lesson students will be able to...)	Teaching Learning Methodology	Assessment Method
Union and enumeration	<ul style="list-style-type: none"><li>• Describe the concept of union and enumeration</li><li>• Differentiate between structure, union and enumeration</li><li>• Know about bit-fields</li></ul>	Multimedia Presentation, Question and Answer	Class Test, Exam, etc.

# Union

- Its format is similar to the format of structure
- Major distinction between them in terms of storage
- For structure, each member variable has its own storage
- But for union all member variables use the same location
- Which means it can declare many variables but can store value in only one of them



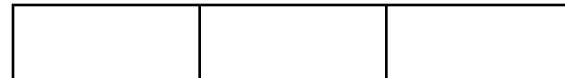
# Union Example

## □ Information of an item:

```
union item{  
    char ch;  
    int id;  
    float price;  
};  
  
int main(){  
    union item x;  
    x.id=10;  
    printf("ID = %d\n", x.id);  
    x.price=100.5;  
    printf("ID = %d\n", x.id);  
    printf("Price = %.2f\n", x.price);  
}
```

Output: ID = 10  
ID = 1120468992  
Price = 100.50

Storage:



ch

id

price

# Structure vs. Union

## DIFFERENCE

<u>Structure</u>	<u>Union</u>
Structure allocates storage space for all its members separately.	Union allocates one common storage space for all its members. Union finds that which of its member needs high storage space over other members and allocates that much space
Structure occupies higher memory space.	Union occupies lower memory space over structure.
We can access all members of structure at a time.	We can access only one member of union at a time.
All members may be initialized.	Only first member may be initialized.
Consume more space than union.	Conservation of memory is possible.
<b>Syntax:</b> struct name { data-type member-1; data-type member-2; ----- data-type member-n; }	<b>Syntax:</b> Union tag-name { data-type member1; data-type member 2; ----- data-type member n; }



# Pointers

---



Dept. of Computer Science & Engineering

# Lesson – 32, 33

Topic	Lesson Learning Outcome (at the end of the lesson students will be able to...)	Teaching Learning Methodology	Assessment Method
Pointers	<ul style="list-style-type: none"><li>• Know about the underlying concept of pointers</li><li>• Define pointer variables</li><li>• Apply pointers to access other variables</li><li>• Relate between pointers and arrays</li><li>• Pass pointers as function arguments</li></ul>	Multimedia Presentation, Question and Answer	Class Test, Exam, etc.



# Introduction

- Pointers are one of the most distinct and exciting features of C language
- Pointers contain memory addresses as their values
- It can be used to manipulate data stored in the memory
- Pointer is one kind of derived data type which is built from one of the fundamental data types in C

**Derived Data Type:** Data types that are derived from fundamental data types are called derived data types. Derived data types don't create a new data type, instead they add some functionality to the basic data types.  
E.g: Array, Structure, Pointer etc.



# Benefits of Pointers

- Can efficiently handle arrays
- Can be used to return multiple values from function
- Permits referencing function and can be used to pass functions as the parameter of another function
- Allow dynamic memory management
- Helpful for handling dynamic data structures like linked list, queue, tree etc.
- Sometimes reduce length and complexity of a program
- More faster and thus reduces execution time



# Understanding Pointers

- Computer's memory is a sequential collection of storage cells
- Each memory cell is 1 byte and has an address
- Addresses are numbered consecutively, starting from zero
- The number of addresses depends on the memory size
- A computer system having 64KB memory will have its last address as 65,535

Memory Organization  
of 64KB Memory  
( $64 \times 1024$ ) addresses

Address	Memory Cell
0	-----
1	
2	
3	
...	
...	
...	
65,534	
65,535	



# Representation of a Variable

- Example: `int num = 179;`

Here,

`num` <= Variable name

`179` <= Value

`5000` <= Address (Assume)



How to get this address??

Addresses are also numbers and can be stored in a variable.

Can use pointer! A pointer variable can store the address of another variable.  
Let us consider a pointer variable `ptr`.

<u>Variable name</u>	<u>Contents</u>	<u>Location</u>
<code>num</code>	179	5000
<code>ptr</code>	5000	6429

# Declaring Pointer Variables

`data_type *pointer_name;`

- ‘`data_type`’ indicates the data type whose location we are going to store
- Asterisk (`*`) indicates that we are going to declare a pointer variable
- ‘`pointer_name`’ points to a variable of type ‘`data_type`’
- Example:

`int *x;`

Here, `x` is a pointer variable pointing to integer data type

`float *z;`

Here, `z` is a pointer variable pointing to float data type



# Accessing the Address of a Variable

□ Previously:

```
int num = 179;
```

Here,

num <= Variable name

179 <= Value

5000 <= Address (Assume)



How to get this address??

i) Declare a integer type variable,

```
int num = 179;
```

ii) Declare a pointer variable,

```
int *ptr;
```

iii) To get the address of variable ‘num’,

```
ptr = &num; //Here, '&' is referred as 'address of' operator
```

iv) We can get the value addressed by the pointer variable ‘ptr’,

```
printf("%d\n", *ptr); //output: 179
```



# Example of pointers

## □ Example:

```
#include<stdio.h>
int main(){
    int num = 179;
    int *ptr;
    ptr = &num;
    printf("num = %d\n", num);
    printf("ptr = %d\n", ptr);
    printf("*ptr = %d\n", *ptr);
}
```

Output:

```
num = 179
ptr = 272695032
*ptr = 179
```

Here, ‘ptr’ is addressing to the variable ‘num’



# Example of pointers (Contd.)

- Changing contents of a variable using pointer:

```
#include<stdio.h>
int main(){
    int num = 179;
    int *ptr;
    ptr = &num;
    printf("num = %d\n", num);
    printf("ptr = %d\n", ptr);
    printf("*ptr = %d\n\n", *ptr);

    *ptr = 45;
    printf("num = %d\n", num);
    printf("ptr = %d\n", ptr);
    printf("*ptr = %d\n", *ptr);
}
```

Output:

```
num = 179
ptr = 272695032
*ptr = 179

num = 45
ptr = 272695032
*ptr = 45
```

Here, '\*ptr' is equivalent to the variable 'num'



# Pointer Increment/Decrement

- We can add or subtract certain values to move the pointer address either to front or back:

```
ptr++;
ptr = ptr + 1;
ptr = ptr - 4;
```

- Example:

```
#include<stdio.h>
int main(){
    int val = 34;
    int *ptr;
    ptr = &val;
    printf("ptr = %d\n", ptr);
    ptr++;
    printf("ptr = %d\n", ptr);
    ptr = ptr - 2;
    printf("ptr = %d\n", ptr);
}
```

Output:

```
ptr = 272695032
ptr = 272695036
ptr = 272695028
```



# Pointer Increment/Decrement (Contd.)

□ Example:

```
int main(){
    double val = 179.0445;
    double *ptr;

    ptr = &val;
    printf("ptr = %d\n", ptr);

    ptr++;
    printf("ptr = %d\n", ptr);

    ptr++;
    printf("ptr = %d\n", ptr);

    ptr = ptr - 2;
    printf("ptr = %d\n", ptr);
}
```

Data types and Sizes  
(for 64-bit system)

Data Type	Size
char	1 byte (8-bit)
int	4 byte (32-bit)
long long	8 byte (64-bit)
float	4 byte (32-bit)
double	8 byte (64-bit)

Output:

```
ptr = 272695024
ptr = 272695032
ptr = 272695040
ptr = 272695024
```



# Pointers and Arrays

## □ Example:

```
int main0{  
    int ara[] = {-1, 5, 2, 0, -7};  
    int *ptr;
```

Note: The array name points to the location of first element

```
ptr = &ara[0]; ← This line is equivalent to: ptr = ara;  
printf("%d\n", ptr); //272695016
```

```
ptr = &ara[1];  
printf("%d\n", ptr); //272695020
```

```
ptr++;  
printf("%d\n", ptr); //272695024
```

```
ptr+=2;  
printf("%d\n", ptr); //272695032
```

```
ptr-=3;  
printf("%d\n", ptr); //272695020
```

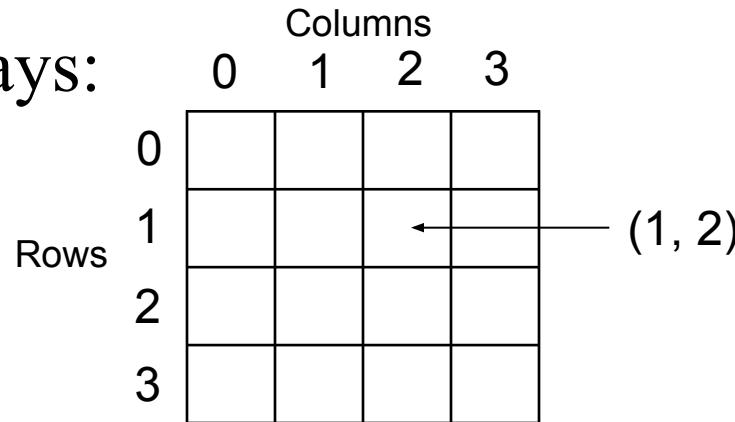
```
printf("%d\n", *ptr); //5
```

```
}
```



# Pointers and Arrays (Contd.)

## □ Two Dimensional Arrays:



ptr	ptr+4	ptr+8	...						
(0,0)	(0,1)	(0,2)	(0,3)	(1,0)	(1,1)	(1,2)	.....	(3,2)	(3,3)

ptr	Pointer to first row
ptr+i	Pointer to i'th row
*(ptr+i)	Pointer to first element of the i'th row
*(ptr+i)+j	Pointer to j'th element of the i'th row
*(*(ptr+i)+j)	Value stored in cell (i, j)

Note: If we do not specify j it will be assumed as 0. So, it will point to first column

# Pointers and Arrays (Contd.)

## □ Example:

```
int main(){
    int ara[3][3]={ {1,2,3},
                    {4,5,6},
                    {7,8,9} };
    int *ptr;
    int i=1, j=2;
    ptr=*(ara+i)+j;
    printf("(%d, %d) = %d\n", i, j, *ptr);

    int col_size=3;
    ptr=&ara[0][0]+i*col_size+j;
    printf("(%d, %d) = %d\n", i, j, *ptr);
}
```

Output:

```
(1, 2) = 6
(1, 2) = 6
```



# Pointers and Character Strings

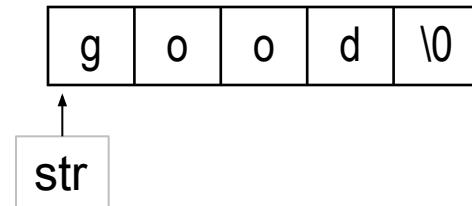
- Strings are also treated as character arrays:

```
char str[5] = "good";
```

- Compiler automatically insert '\0' at the end of the string
- We can create string using pointer as follows:

```
char *str;  
str = "good";  
printf("%s", str); ← No need to use '*' for string
```

- Note that it's not a copy of the string, because the variable 'str' is a pointer, not a string.



# Pointers and Character Strings (Example)

## □ Finding length of the string

```
int main(){
    char *str="good";
    int length=0;
    while(*str != '\0'){
        printf("%c is located at %d\n", *str, str);
        length++;
        str++;
    }
    printf("Length = %d\n", length);
}
```

Output:

```
g is located at 4206628
o is located at 4206629
o is located at 4206630
d is located at 4206631
Length = 4
```

Note:

‘str’ is pointer, ‘\*str’ is the character value at pointer location,

printf(“%d”, str); <= prints location

printf(“%c”, \*str); <= prints the character at that location

printf(“%s”, str); <= keeps printing characters until ‘\0’ is found



# Pointers as Function Arguments

- We can pass the address of a variable as an argument to a function
- When we pass address to a function, the parameter that receives the address should be a pointer
- Changing values within the function will also change the values in calling function if called using address
- Array can also be passed as the function argument
- In this case the first location is passed, we also need to pass the array size
- The process of calling a function using pointers to pass the addresses of variable is known as ‘call by reference’



# Pointers as Function Arguments (Example)

## □ Calculating area and perimeter of a circle

```
void circle(float r, float *area, float *perimeter){  
    float PI=acos(-1); //3.1415...  
    *area = PI*r*r;  
    *perimeter = 2*PI*r;  
}  
  
int main(){  
    float A, P;  
    circle(1, &A, &P);  
    printf("A = %.2f, P = %.2f\n", A, P);  
}
```

Note: Here, we are passing address location and changing the actual values at that location. This is known as ‘pass by pointer’/‘call by reference’



# Pointers as Function Arguments (Example)

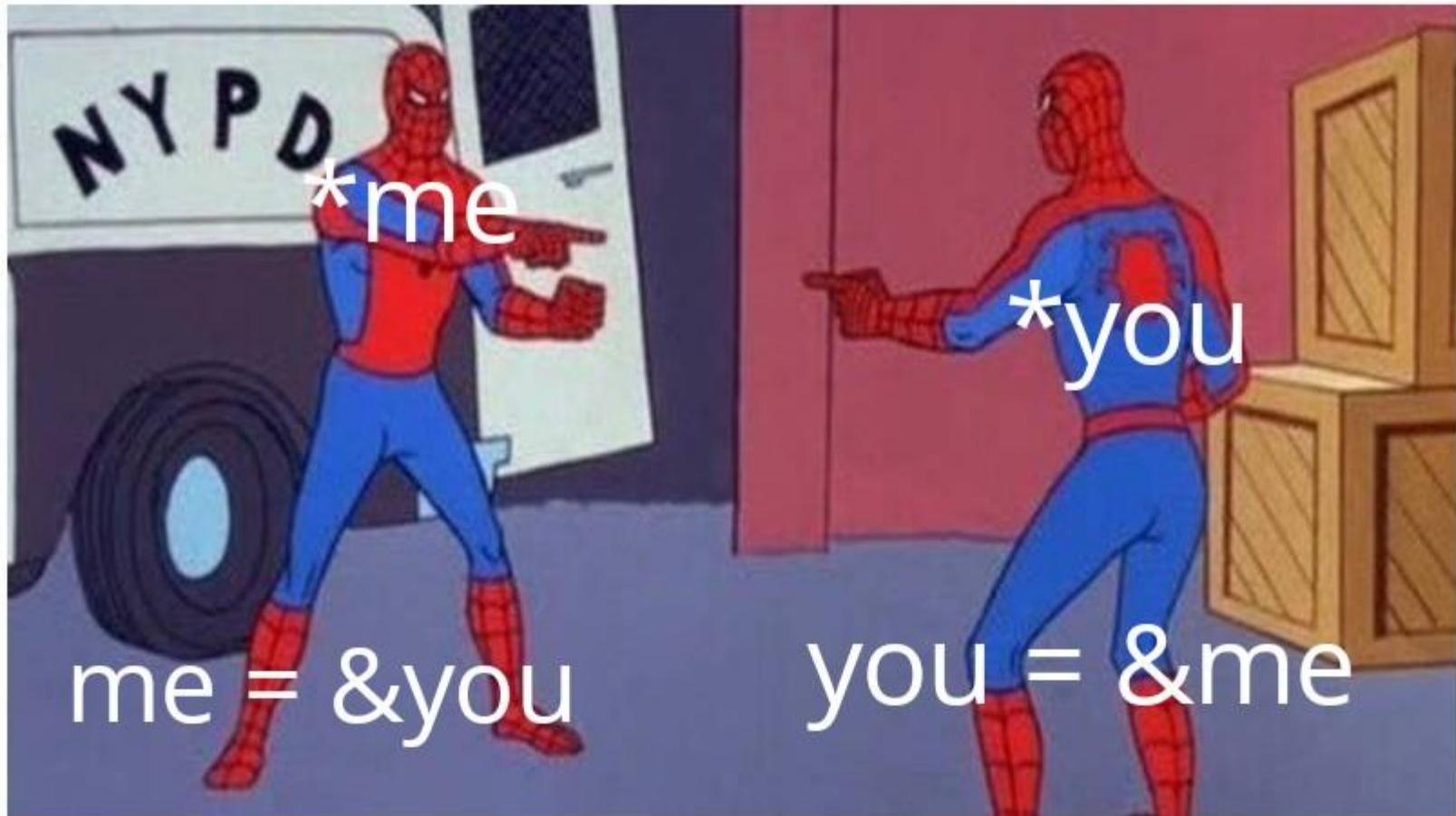
- Getting sum of elements of an array:

```
int adder(int *ara, int sz){  
    int tot=0;  
    for(int i=0; i<sz; ++i){  
        tot+=ara[i];  
    }  
    return tot;  
}  
  
int main(){  
    int ara[]={1, 2, -5, 8, -2};  
    int sum = adder(ara, 5);  
    printf("Sum = %d\n", sum);  
}
```

Output: 4



# Thank you



# File Management

---



Dept. of Computer Science & Engineering

# Lesson – 34, 35

Topic	Lesson Learning Outcome (at the end of the lesson students will be able to...)	Teaching Learning Methodology	Assessment Method
File Management	<ul style="list-style-type: none"><li>• Define files in a c program</li><li>• Know about the process of opening and closing a file</li><li>• Perform input/output operations on files</li><li>• Perform error handling operations during I/O</li><li>• Write codes for randomly accessing files</li></ul>	Multimedia Presentation, Question and Answer	Class Test, Exam, etc.



# Introduction

- We have seen ‘`scanf`’ and ‘`printf`’ function for reading and writing data using console
- It works fine when the input/output data is small
- However, for large volume of data the console oriented input/output has two major problems:
  - It becomes time consuming and difficult to understand
  - If user makes a mistake during entering data, whole data has to be re-entered
  - The entire data is lost after the program is terminated

Solution: **Use File for Input/Output**

- **Data File:** Allows us to store information permanently and alter information when necessary. (E.g: `input.txt`)



# Some High Level I/O Functions

- There are two ways to perform file operations in C:
  1. **Low-level I/O:** Provide direct access to files, more complex but faster
  2. **High-level I/O:** More flexible and easy to use

Function	Operation
fopen()	Creates a new file / opens an existing file
fclose()	Closes a file which has been opened for use
getc()	Reads a character from the file
putc()	Writes a character to the file
fprintf()	Write data values to a file
fscanf()	Reads a set of data values from a file
getw()	Reads an integer from the file
putw()	Writes an integer to a file
fseek()	Sets the position to the desired point in the file
ftell()	Gives the current position in the file
rewind()	Sets the position to the beginning of the file



# Defining and Opening a File

- General format for declaring and opening a file:

```
FILE *fp;
```

```
fp = fopen("filename", "mode");
```

- Here, the first statement declares the variable ‘fp’ as a “pointer to the data type FILE”.
- Second statement opens the file named “filename” with the purpose “mode” and the beginning address of the file is located by the file pointer ‘fp’.
- Note that, both “filename” and “mode” are specified as strings.

(FILE data type is defined in the I/O library)



# File Opening Modes

FILE \*fp;  
fp = fopen("input.txt", "w");

mode

Mode	Description
"r"	Opens a file for reading. The file must exist.
"w"	Creates an empty file for writing. If a file with the same name exists, its content is erased and the file is assumed as a new empty file.
"a"	Attaches to a file. Writing operations, attaches data at the end of the file. The file is created if it does not present.
"r+"	Opens a file to update both reading and writing. The file must present.
"w+"	Creates an empty file for both reading and writing.
"a+"	Opens a file for reading and appending.



# Closing a File

- A file must be closed as soon as all operations on it have been completed
- It ensures all information associated with the file is flushed out from the buffer and all links to the file are broken
- If there is limit on number of files we can open, then we can close the current file to open a new one
- Or if we want to open same file in different mode
- It takes the following form: `fclose(filename);`
- E.g:
  - `FILE *fp;`
  - `fp = fopen("input.txt", "r");`
  - `fclose(fp);`



# Input/Output Operations on Files [getc(), putc()]

- **getc(file\_pointer):** used to read a character from a file that has been opened in read mode, pointed by the file\_pointer. E.g:  
                `getc(fp);`

- **putc(character, file\_pointer):** used to write a character to the file that has been opened in write mode, pointed by file\_pointer. E.g:  
                `putc('a', fp);`

Note: the file pointer moves by one character for every operation of getc() and putc(). The getc() will return an end-of-file marker (EOF) when the end of the file is reached.



# End of File (EOF)

- It is a marker that indicates that we have reached to the end of the file
- Therefore, one should stop reading characters when the ‘EOF’ is encountered
- An attempt to read after ‘EOF’ file may result in termination of the program with an error or infinite loop
- Actual value of EOF is system dependent and not part of the standard
- Usually it's an integer with negative value

```
printf("%d", EOF); //Outputs -1 in my computer
```



# File Example with getc() and putc()

- Read characters from a file named ‘input.txt’
- Save all the characters in ‘output.txt’ except for the vowels

```
int main(){  
FILE *fp1, *fp2;  
fp1=fopen("input.txt", "r");  
fp2=fopen("output.txt", "w");  
char ch;  
while(1){  
    ch=getc(fp1);  
    if(ch==EOF) break;  
    else if(ch=='a' || ch=='e' || ch=='i' || ch=='o' || ch=='u') continue;  
    putc(ch, fp2);  
}  
fclose(fp1);  
fclose(fp2);  
}
```

input.txt - Notepad  
File Edit Format View Help  
hello world!  
how are you?

output.txt - Notepad  
File Edit Format View Help  
hll wrld!  
hw r y?

# Input/Output Operations on Files [getw(), putw()]

- **getw(file\_pointer):** used to read an integer number from a file that has been opened in read mode, pointed by the file\_pointer. E.g:

```
getw(fp);
```

- **putw(integer, file\_pointer):** used to write an integer to the file that has been opened in write mode, pointed by file\_pointer. E.g:

```
putw(x, fp);
```

Similar to getc() and putc() except they deal with integers



# File Example with getw() and putw()

- Read a file ‘number.txt’ that contains some number
- Save all the even numbers in ‘even.txt’ and all the odd numbers in ‘odd.txt’

Exercise:

Try this at home and  
let me know if it works

```
#include<stdio.h>
int main(){
    FILE *fp_num, *fp_even, *fp_odd;
    fp_num=fopen("number.txt", "r");
    fp_even=fopen("even.txt", "w");
    fp_odd=fopen("odd.txt", "w");
    int n;
    while(1){
        n=getw(fp_num);
        if(n==EOF) break;
        if(n%2==0) putw(n, fp_even);
        else putw(n, fp_odd);
    }
    fclose(fp_num);
    fclose(fp_even);
    fclose(fp_odd);
}
```



# Formatted Input/Output

- Using formatted input/output we can read/write numbers, characters or strings as per our requirement
- **fprintf(file\_pointer, “control string”, variables):** It is a formatted output function which is used to write data to a file. E.g:

```
fprintf(fp, "%s %d", name, age);
```

- **fscanf(file\_pointer, “control string”, &variables):** It is a formatted input function which is used to read data from a file. E.g:

```
fscanf(fp, "%s %d", name, &age);
```

Note: Similar to ‘scanf’ and ‘printf’ here the function ‘fscanf’ and ‘fprintf’ return the number of items that are successfully read. When end of file is encountered it returns EOF.



# Formatted Input/Output Example

- Read ‘number.txt’, save even numbers in ‘even.txt’ and odd numbers in ‘odd.txt’

number.txt - Notepad

```
File Edit Format View  
10  
124  
33  
12  
15  
2234
```

```
#include<stdio.h>  
int main(){  
    FILE *fp_num, *fp_even, *fp_odd;  
    fp_num=fopen("number.txt", "r");  
    fp_even=fopen("even.txt", "w");  
    fp_odd=fopen("odd.txt", "w");  
    int n;  
    while(1){  
        int ok=fscanf(fp_num, "%d", &n);  
        if(ok==EOF) break;  
        if(n%2==0) fprintf(fp_even, "%d\n", n);  
        else fprintf(fp_odd, "%d\n", n);  
    }  
    fclose(fp_num);  
    fclose(fp_even);  
    fclose(fp_odd);  
}
```

odd.txt - Notepad

```
File Edit Format View  
33  
15
```

even.txt - Notepad

```
File Edit Format View  
10  
124  
12  
2234
```



# Error handling in Files

- Some error situations during I/O operations on a file:
  - Trying to read beyond end-of-file mark
  - Device overflow (Exceeds memory limit)
  - Trying to use a file that has not been opened
  - Trying to perform an operation on a file, when it is opened for another type of operation
  - Opening a file with an invalid filename
  - Attempting to write in a write-protected file
- Programs may behave abnormally when such error occurs
- I/O errors can be detected using two status-inquiry library functions: feof() and ferror()

If for any case we can not open a file then fopen() will return NULL.



# Random Access to Files

- Till now, reading and writing data from/to a file has been done sequentially
- But we may need to access a particular data item placed in any location without starting from the beginning
- This is known as random access or direct access
- **ftell(file\_pointer):** Takes a file\_pointer as argument and returns a number of type long, that indicates the current position of the file\_pointer within the file.
  - E.g: n = ftell(**fp**);
  - Here, n will store number of bytes that have already been read
  - Useful for saving the current position, which can be used later in the program



# Random Access to Files (Contd.)

- **rewind(file\_pointer):** Takes a file\_pointer and resets the position to the start of the file.
  - E.g: `rewind(fp);`
  - Now, if we use '`n=f.tell(fp)`' it will show 0. Because file has been set to the start of the file by `rewind()`
- **fseek(file\_pointer, offset, position):** This function is used to move the file\_pointer to a desired position within a file.
  - E.g: `fseek(fp, 5, 0);`
  - Moves the position after 5'th byte from beginning
  - Position can take value of 0, 1 or 2

Value	Meaning
0	Beginning of the file
1	Current location
2	End of file



# Random Access to Files (Contd.)

## fseek()...

- The *offset* may be positive, meaning move forwards, or negative, meaning move backwards.
- Examples:

Statement	Meaning
<code>fseek(fp, 0L, 0);</code>	<b>Move file pointer to beginning of file. (Same as rewind.)</b>
<code>fseek(fp, 0L, 1);</code>	<b>Stay at the current position. (File pointer is not moved.)</b>
<code>fseek(fp, 0L, 2);</code>	<b>Move file pointer past the last character of the file. (Go to the end of file.)</b>
<code>fseek(fp, m, 0);</code>	<b>Move file pointer to (m+1)th byte in the file.</b>
<code>fseek(fp, m, 1);</code>	<b>Move file pointer forwards by m bytes.</b>
<code>fseek(fp, -m, 1);</code>	<b>Move file pointer backwards by m bytes from the current position.</b>
<code>fseek(fp, -m, 2);</code>	<b>Move file pointer backwards by m bytes from the end. (Positions the file pointer to the mth character from the end.)</b>



# Thank you

---

?



# **Command Line Arguments, Dynamic Memory Allocation, Preprocessor**

---



Dept. of Computer Science & Engineering

# Lesson – 36, 37

Topic	Lesson Learning Outcome (at the end of the lesson students will be able to...)	Teaching Learning Methodology	Assessment Method
Command line arguments and dynamic memory allocation, Header files, Preprocessor	<ul style="list-style-type: none"><li>• Explain the concept of command line arguments and dynamic memory allocation</li><li>• Know about memory allocation functions: malloc, calloc, free, realloc</li><li>• Explain the importance of header files</li><li>• Use different preprocessors in program</li></ul>	Multimedia Presentation, Question and Answer	Class Test, Exam, etc.



# Command Line Arguments



# Command Line Arguments (Example)

- Taking some characters from command line and converting them into lower case

```
#include<stdio.h>
int main(int argc, char *argv[]){
    int i;
    printf("argc = %d\n", argc);
    for(i=0; i<argc; ++i){
        printf("%d = %s\n", i, argv[i]);
    }
    printf("\n");
    for(i=1; i<argc; ++i){
        if(argv[i][0]>='A' && argv[i][0]<='Z') argv[i][0]+='a'-'A';
        printf("%d = %s\n", i, argv[i]);
    }
    return 0;
}
```

```
F:\Code\Exercise>gcc to_lower.c
F:\Code\Exercise>a.exe a B C t
argc = 5
0 = a.exe
1 = a
2 = B
3 = C
4 = t
1 = a
2 = b
3 = c
4 = t
```



# Dynamic Memory Allocation

## □ Some problems with arrays:

- Requires number of elements to be specified at compile time
- It is difficult to assume the maximum size of the array in advanced.
- We have to give highest possible size value when initializing.
- This is a wastage of memory for smaller input

## □ Solution: Use dynamic memory allocation

- The process of allocating and freeing memory at run time (or execution time) is called dynamic memory allocation
- In C, we have following four functions for memory management process: malloc(), calloc(), free(), realloc()



# Memory Allocation: malloc()

- It allocates requested size of bytes and returns a pointer to the first byte of the allocated space
- Syntax:  
`ptr = (data_type *) malloc(size_of_block);`
- Here, ‘ptr’ is a pointer of type ‘data\_type’. The malloc() returns a pointer to the first byte to an area of memory with size ‘size\_of\_block’
- E.g.:  
`ptr = (int *) malloc(100*sizeof(int));`  
`ptr = (int *) malloc(200);`



# Clear Allocation: calloc()

- We have seen malloc() allocates a single block of storage space
- calloc() allocates multiple blocks of storage, each of the same size and then sets all bytes to zero
- It acts like an array of blocks
- Syntax:

```
ptr = (data_type *) calloc(no_of_blocks, size_of_each_block);
```

- E.g:

```
ptr = (int *) calloc(5, 10*sizeof(int));
```



# Re-Allocation: realloc()

- It is used to modify the size of the previously allocated space
- When the size of previously allocated memory space is not enough or when the size of previously allocated memory space is larger than needed, we need to use realloc()
- Syntax: If original allocation has been done as:  
`ptr = (data_type *) malloc(size);`  
Then reallocation is done as:  
`ptr = (data_type *) realloc(ptr, new_size);`



# Free Allocation: free()

- It frees previously allocated memory space by malloc(), calloc(), or realloc() function
- Syntax:

```
free(pointer_name);
```

- E.g:

```
free(ptr);
```

Note: We need to add #include<stdlib.h> header file when using dynamic memory allocation functions

Note: If in any case memory can not be allocated it will return null



# Dynamic Memory Allocation Example

- Write a C program to find summation of n elements entered by the user. Use dynamic memory allocation.

```
#include<stdio.h>
#include<stdlib.h>
int main(){
    int i, n;
    scanf("%d", &n);
    int *ptr;
    ptr=(int *)malloc(n*sizeof(int));
    for(i=0; i<n; ++i){
        scanf("%d", ptr+i);
    }
    int sum=0;
    for(i=0; i<n; ++i){
        sum+=*(ptr+i);
    }
    printf("Sum = %d\n", sum);
    free(ptr);
}
```

Output:

```
3
10 15 -5
Sum = 20
```



# The Preprocessor

- It is a program that processes the source code before its passes through the compiler



- Preprocessing occurs before the program is compiled
- Preprocessors start with ‘#’ and does not end with ‘;’
- Preprocessors work under the control of preprocessor command line or directives
- They are divided into three categories
  - 1) Macro substitution
  - 2) File inclusion
  - 3) Compiler control

# Macro Substitution

- It is a process where an identifier in a program is replaced by a predefined string composed of one or more tokens

`#define identifier string`

- The preprocessor replaces every occurrences of the ‘identifier’ in the source code by the ‘string’
- The ‘string’ can be any text while the ‘identifier’ must be a valid C name
- E.g:

```
#define N 10
#define X 10
#define sqr(x) ((x)*(x))
#define max(a,b) (a>b?a:b)
#define cube(x) ((x)*sqr(x))
```



# Macro Substitution (Example)

- What is the output of the following program:

```
#include<stdio.h>
#define ADD(x) 1+x
```

```
int main(){
    int res=2*ADD(3);
    printf("%d\n", res);
}
```

Output: 5



# File Inclusion

- An external file containing functions or macro definitions can be included as a part of a program so that we need not rewrite those functions or macro definitions
- This is achieved by the ‘#include’ directives

```
#include <filename>
```

- E.g:

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

```
#include <ctype.h>
```

```
#include <test.c>
```



# Compiler Control Directives

- C preprocessor offers a feature known as conditional compilation
- It can be used to switch on or off a particular line or group of lines in a program
- This can be done using ‘`#ifdef`’, ‘`#ifndef`’ and ‘`#endif`’
- ‘`#ifdef`’ Test for a macro definition
- ‘`#endif`’ Specifies the end of ‘`#if`’
- ‘`#ifndef`’ Tests whether a macro is not defined



# Compiler Control Directives (Example)

## □ Executing test portion

```
#include<stdio.h>
#define TEST 1

int main(){
    #ifdef TEST
        printf("Testing\n");
        //Test code...
    #endif // TEST

    int x, y;
    //Rest of the code
}
```

Output:  
Testing

Here, if we remove TEST definition then #ifdef portion wont get executed



# Types of Errors in a Program

- **Syntax Errors:** Violation of rules of the language
  - E.g: Missing semicolon, wrongly declared variables
- **Run-time Errors:** The program followed the rules of the syntax but produced erroneous results during execution
  - E.g: division by zero:  $\text{res}=\text{p}/\text{q}$  where  $\text{q}=0$
- **Logical Error:** Incorrect implementation of logic in a program
  - E.g: ‘`if(x=y)`’ instead of writing ‘`if(x==y)`’



Thank you

---

THE END

return 1;

