

# C Programming (CSC115)

**Sulav Nepal**

Email: [nep.sulav@live.com](mailto:nep.sulav@live.com)

Contact No.: 9849194892

Master's in Computer Information System (MCIS) – Pokhara University  
Bachelor of Science. Computer Science & Information Technology (B.Sc. CSIT) – Tribhuvan University  
Microsoft Technology Associate (MTA): Windows Server Administration Fundamentals  
Microsoft Certified Technology Specialist (MCTS): Windows Server 2008 R2, Server Virtualization  
Microsoft Specialist (MS): Programming in HTML5 with JavaScript and CSS3  
Microsoft Students Partner (MSP) 2012 for Nepal

not **ENGLISH** nor **NEPALI**

**Welcome to C Programming Language**

# Syllabus

- **UNIT 1: Problem Solving with Computer**
  - *Problem Analysis*
  - *Algorithms and Flowchart*
  - *Coding, Compilation, & Execution*
  - *History of C*
  - *Structure of C program*
  - *Debugging, Testing & Documentation*

# UNIT 1

***PROBLEM SOLVING WITH COMPUTER***

# Introduction

- *Number of problems in our daily life.*
- *Suppose we have to calculate Simple Interest.*
- *Suppose we have to prepare a mark sheet.*
- *A computer is a **DUMB** machine.*
- *A computer cannot do anything alone without software. i.e. **Program**.*

# Introduction

- *A software is a set of programs written to solve a particular problem.*
- *Program is a set of instructions on the basis of which computer gives output/result.*
- *If the instructions are not correct, the computer gives wrong result.*

# Never Ever Forget

- *Just writing code is not sufficient to solve a problem.*
- *Program must be planned before coding in any computer language available.*
- *There are many activities to be done before and after writing code.*

# Types of Programming Languages

- **Machine Level Language**

- *Language that a computer actually understands (1's and 0's)*
- *Sequence of instructions written in the form of binary numbers*
- *Executes fast as computer don't need any translation*

- **Assembly Language**

- *Symbolic representation of machine code*
- *Close to machine code but the computer cannot understand*
- *Must be translated into machine code by a separate program called an assembler*

- **High Level Language**

- *Similar to natural language resulting to ease in learning and writing*
- *While execution: translated into assembly language then to machine language*
- *Slow in execution but is efficient for developing programs*

# Overview to C Programming

- *We can assume C as middle level language*
- *This doesn't mean C is less powerful or harder to use or less developed*
- *Instead C combines the advantages of high level language with the functionalism of assembly language*
- *Like high level, C provides block structures, stand-alone functions and small amount of data typing*
- *Like assembly language, C allows manipulations of bits, bytes, pointers and it is mostly used in system programming*
- *Combination of two aspects*

# Stages while solving a problem using computer

- *Problem Analysis*
- *Algorithm Development*
- *Flowcharting*
- *Coding*
- *Compilation & Execution*
- *Debugging & Testing*
- *Documentation*

# Stages while solving a problem using computer

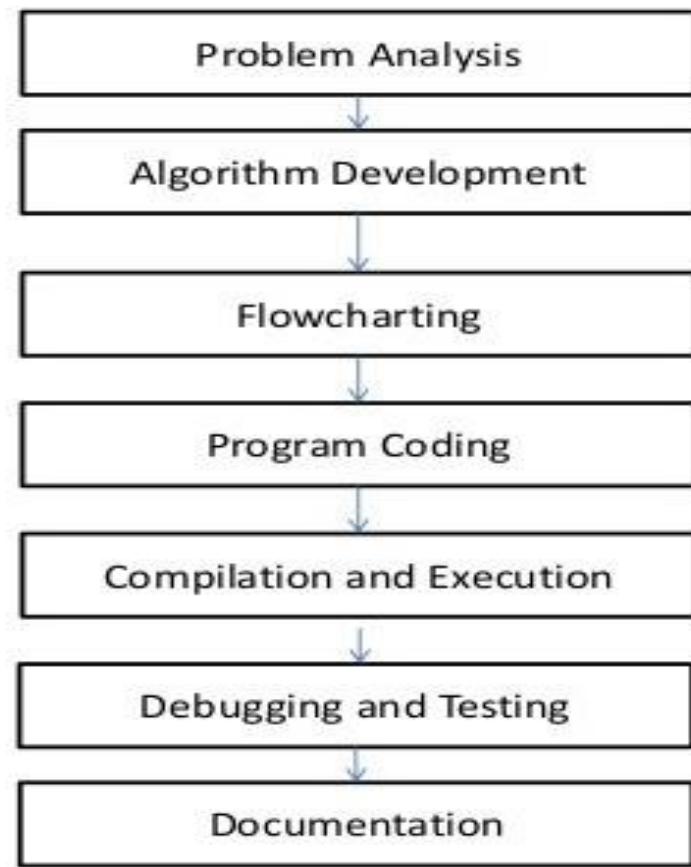


Fig. Steps in problem solving

# Problem Analysis

- *Process of becoming familiar with the problem*
- *We need to analyze and understand it well before solving*
- *The user's requirements cannot be fulfilled without clear understanding of his/her problem in depth*
- *Inadequate identification* of problem may cause program less useful and insufficient
- *Example: Banking Solution, Hospital Medical Study*

# Algorithm Development

- *Step by step description of the method to solve a problem*
- *Effective procedure for solving a problem in finite number of steps*
- *Developing an algorithm is a step of program design*
- *Example: An algorithm to find sum of two numbers:*
  - *Step 1: Start*
  - *Step 2: Declare variables **num1**, **num2**, and **sum***
  - *Step 3: Read values **num1** and **num2***
  - *Step 4: Add **num1** and **num2** and assign the result to **sum***
  - *Step 5: Display **sum** ( $sum \leftarrow num1 + num2$ )*
  - *Step 6: Stop*

# Algorithm Development

- *Three features of Algorithm:*
  - *Sequence*
    - *Each step in the algorithm is executed in specified order. If not algorithm will fail.*
  - *Decision*
    - *We have to make decision to do something*
    - *If the outcome of the decision is true, one thing is done otherwise other*  
*If condition then process1*  
**OR**  
*If condition then process1 Else process2*
  - *Repetition*  
**Repeat**  
*Fill water in kettle*  
**Until** Kettle is full

# Flowcharting

- *Graphical representation of an algorithm using standard symbols*
- *Includes a set of various standard shaped boxes that are interconnected by flow lines*
- *Flow lines have arrows (direction of flow)*
- *Activities are written within boxes in English*
- *Communicates between programmers and business persons*

# Flowcharting - Advantages

- *Communication* — quickly provide logic, ideas, and descriptions of algorithms
- *Effective Analysis* — clear overview of the entire problem
- *Proper Documentation* — documents the steps followed in an algorithm and helps us understand its logic in future
- *Efficient Coding* — more ease with comprehensive flowchart as a guide
- *Easy in Debugging & Program Maintenance* — debugging and maintenance of operating program

# Flowchart Symbols

Symbol	Name	Function
	Start/end	An oval represents a start or end point
	Arrows	A line is a connector that shows relationships between the representative shapes
	Input/Output	A parallelogram represents input or output
	Process	A rectangle represents a process
	Decision	A diamond indicates a decision

# Flowchart – Things to Consider

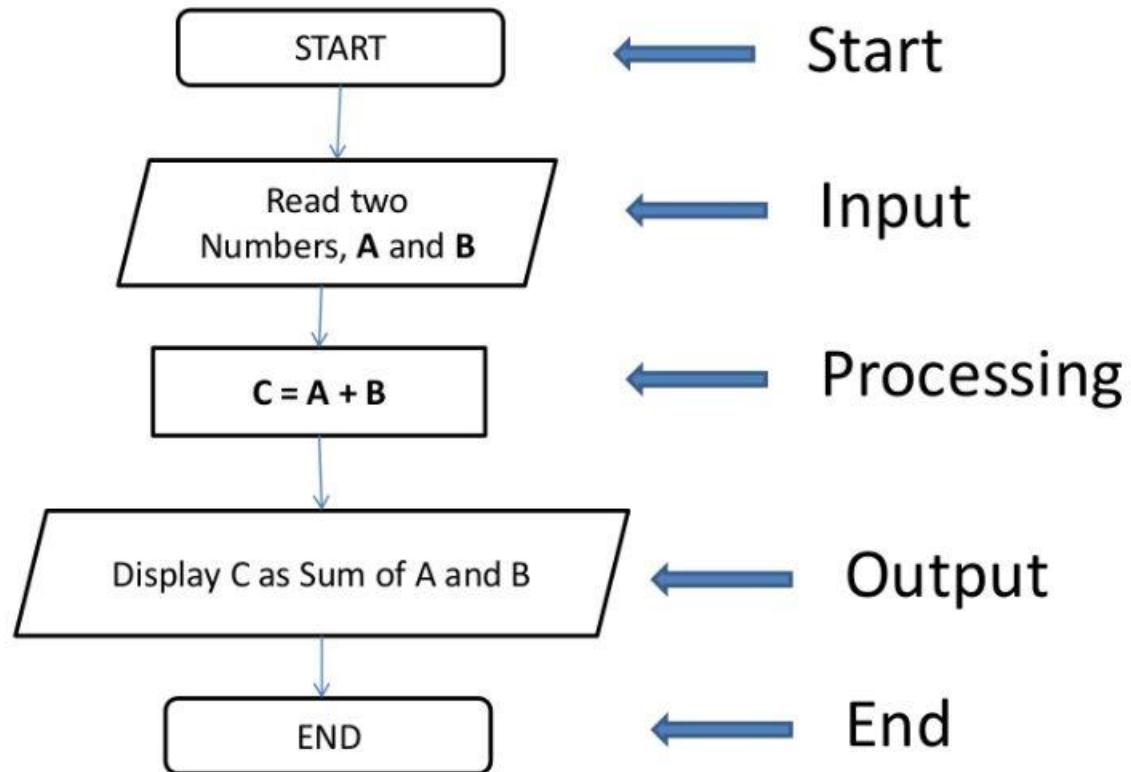
- *There should be **start** and **stop** to the flowchart*
- *Only one flow line should emerge from a process symbol*
- *Only one flow line should enter a decision symbol, but two or three flow lines can leave the decision symbol*

# Example – Sum of Two Numbers

- *Algorithm*
  - *Step 1: Start*
  - *Step 2: Display “Enter two numbers”*
  - *Step 3: Read A and B*
  - *Step 4:  $C = A + B$*
  - *Step 5: Display “C as sum of two numbers”*
  - *Step 6: Stop*

# Example – Sum of Two Numbers

- *Flowchart*



# Coding

- *The process of transforming the program logic design into computer language format*
- *An act of transforming operations in each box of the flowchart in terms of the statement of the program*
- *The code written using programming language is also known as **source code***
- *Coding isn't the only task to be done to solve a problem using computer.*

***Anyone can code. TRUST ME !!***

# Compilation

- *Process of changing high level language into machine level language*
- *It is done by special software, called **Compiler***
- *The compilation process tests the program whether it contains syntax errors or not*
- *If syntax errors are present, compiler can not compile the code.*

# Execution

- Once the compilation is completed then the program is linked with other object programs needed for execution, thereby resulting in a binary program and then the program is loaded in the memory for the purpose of execution and finally it is executed
- The program may ask user for inputs and generates outputs after processing the inputs

# Debugging and Testing

- *Debugging is the discovery and correction of programming errors.*
- *Some errors may remain in the program because the designer/programmer might have never thought about a particular case.*
- *When error appears debugging is necessary.*

# Debugging and Testing

- *Testing ensures that program performs correctly the required task*
- *Verification ensures that program does what the programmer intends to do*
- *Validation ensures that the program produces the correct results for a set of test data*
- *Test data are supplied to the program and output is observed*
- *Expected output = Error free*

# Program Documentation

- Helps to those who *use, maintain, and extend* the program in future
- A program may be difficult to understand even to programmer who wrote the code after some days
- Properly documented program is necessary which will be *useful and efficient in debugging, testing, maintenance, and redesign process*

# Program Documentation - Types

- *Programmer's Documentation (Technical Documentation)*
  - *Maintain, redesign, and upgrade*
  - *Logic, DFD, ER Diagram, algorithm, & flowchart*
- *User Documentation (User Manual)*
  - *Support to the user of the program*
  - *Instructions for installation of the program*

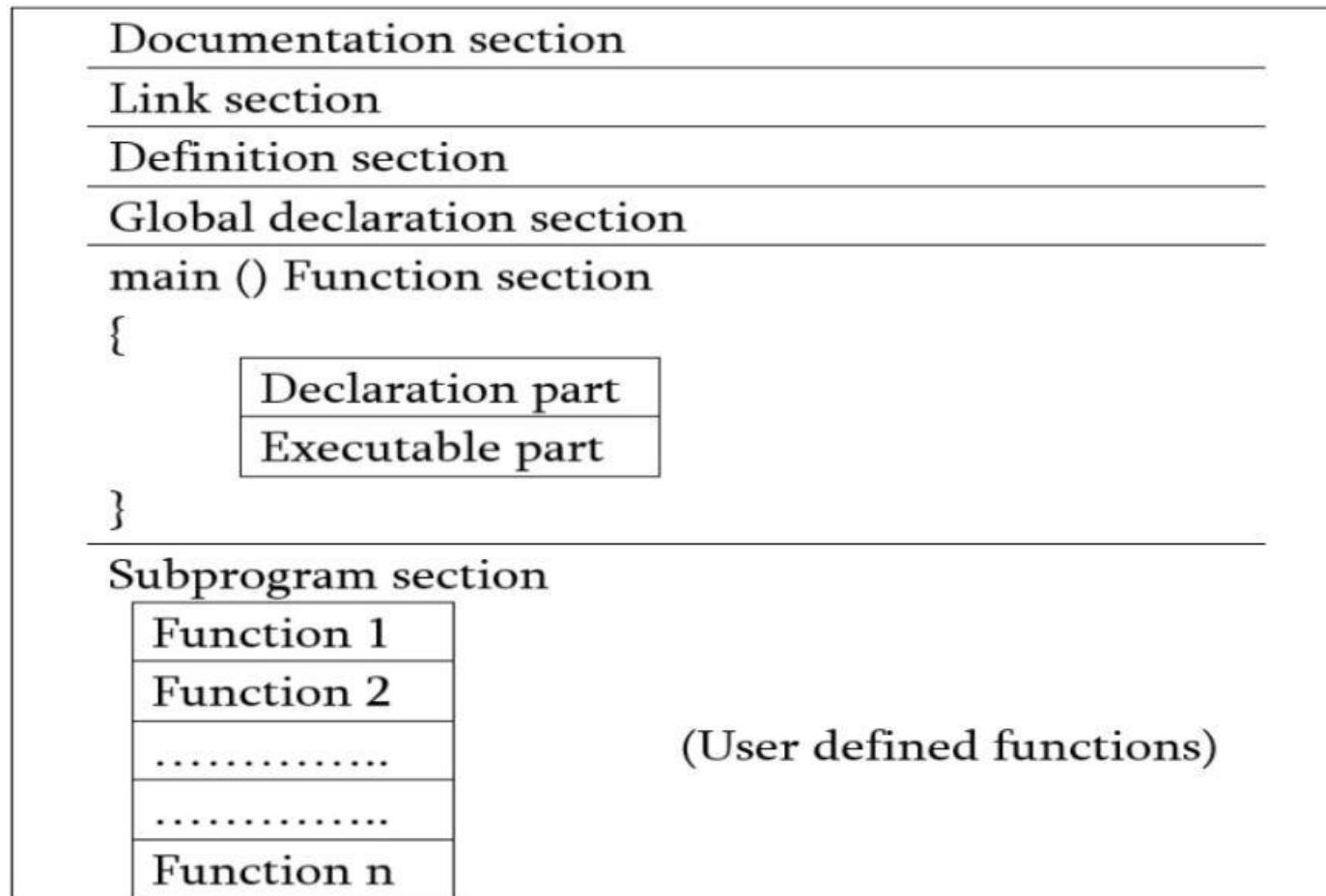
# History of C Programming

- C is a programming language which was born at “AT & T’s Bell Laboratory” of USA in 1972
- C was written by Dennis Ritchie, that’s why he is also called as father of C programming language
- C language was created for a specific purpose i.e. designing UNIX operating system (which is currently base of many UNIX based OS)
- From the beginning, C was intended to be useful to allow busy programmers to get things done because C is such a powerful, dominant and supple language
- Its use quickly spread beyond Bell Labs in the late 70’s

# Why use C?

- *Robust language*
- *Efficient and fast*
- *Highly portable*
- *Structured language*
- *Extendibility*
- *Middle level language*
- *Rich system library*

# Basic Structure of C Program



# Basic Structure of C Program

- **Document Section**

- *Sets of comment line giving the name of program, the author, algorithms, methods used, and other details*
- *Acts as a communication between members of the development team*
- *Acts as user manual*
- *Example: /\* This program adds two numbers \*/*

# Basic Structure of C Program

- **Link Section**

- *Provides instructions to the compiler to link functions with program from the system library*
- *Example: `#include<stdio.h>`*
  - *Links input/output functions like `printf()` and `scanf()` with the program*

# Basic Structure of C Program

- **Definition Section**
  - *In this section all symbolic constants are defined*
  - *This section may be included or excluded while writing a C program*
  - *Example:*  
`#define PI 3.1416`  
`#define FORMULA 3*x*x*x+2*x*x`

# Basic Structure of C Program

- ***Global Declaration Section***

- *The variables which are used in more than one functions or blocks are called global variables*
- *This section also declares all the user-defined functions*
- *This section may be included or excluded while writing a C program*

# Basic Structure of C Program

- ***Main() Function Section***

- *Every C program starts with a **main()** function*
  - *Declaration part and Executable part*
- *Declaration part declares all the variables used in the execution part*
  - *int n1;*
  - *int n2=5;*
- *Execution part has executable operations like*
  - *n1=n1+1;*
  - *n2=n1\*5;*

# Basic Structure of C Program

- ***Subprogram Section***

- *This section contains all the user-defined functions that are called in the main function.*
- *All the sections except the main function section may be absent when they are not required*

# First C Program

```
//First program in C
#include<stdio.h>
#include<conio.h>
void main()
{
    printf("This is my first program in C");
    getch();
}
```

***END OF UNIT ONE***

# Syllabus

- **UNIT 2: Elements of C**

- *C Standards*

*C Character Set*

- *C Tokens*

*Escape Sequence*

- *Delimiters*

*Variables*

- *Data Types*

*Structure of a C Program*

- *Executing a C Program*

*Constants / Literals*

- *Expressions, Statements and Comments*

# UNIT 2

*ELEMENTS OF C*

# First C Program

```
//First program in C
#include<stdio.h>
void main()
{
    printf("This is my first program in C.");
    printf("Welcome to programming world!");
}
```

# C Standards (ANSI C & C99)

- *ANSI C, ISO C, and Standard C refer to the successive standards for the C programming language published by the American National Standards Institute (ANSI).*
- *Historically, the names referred specifically to the original and best-supported version of the standard (known as C89 or C90).*
- *In March 2000, ANSI adopted the ISO/IEC 9899:1999 standard.*
- *This standard is commonly referred to as C99.*

# Character Set

- *Set of characters that are used to form words, numbers and expression in C is called C character set.*
- *Characters in C are grouped into the following four categories:*
  - *Letters and alphabets (A...Z, a...z)*
  - *Digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)*
  - *Special Characters (, . ; : ? “ & ^ \* - + < >)*
  - *White spaces (Blank space, Horizontal tab, etc.)*

# Keywords

- *These are predefined words for a C programming language.*
- *All keywords have fixed meaning and these meanings cannot be changed.*

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

# Identifiers

- Every word used in C program to identify the name of variables, functions, arrays, pointers and symbolic constants are known as *identifiers*.
- Names given by user can consist of a sequence of letters and digits, with a letter as the first character.
- *Example:* myVariable, myName, heyYou, callThisNumber45, add\_this\_number, etc.
- There are certain rules to be followed while naming identifiers.  
**(KEEP THIS IN MIND)**

# Rules for Naming Identifiers

- *It must be a combination of letters and digits and must begin with a letter.*
- *Underscore is permitted between two digits and must begin with a letter.*
- *Only first 31 characters are significant.*
- *Keywords cannot be used.*
- *It is case sensitive. i.e. uppercase and lowercase letters are not interchangeable.*

# Data Types

- 10 is a whole number whereas 10.5 is a fractional/rational number.
- Similarly in C, 10 is an **integer** number whereas 10.5 is a **float** number.
- There are variety of data types available.
- ANSI C supports three classes of data types:
  - Primary/Fundamental data types
  - User-Defined data types
  - Derived data types

# *Primary/Fundamental Data Types*

- *Primary data types are categorized into five types:*
  - *Integer type (int)*
  - *Floating point type (float)*
  - *Double-precision floating point type (double)*
  - *Character type (char)*
  - *Void type (void)*

# Integer Type

- *Integers are whole numbers.*
- *Requires 16 bits of storage. i.e. 2 bytes for 16-bit compiler.*
- *Requires 32 bits of storage. i.e. 4 bytes for 32-bit compiler.*
- *Three classes of integer:*
  - *Integer (int)*
  - *Short integer (short int)*
  - *Long integer (long int)*
- *Both signed and unsigned forms.*
- *Defined as:*

*int a;*

*int myValue=6;*

# Integer Type

Signed Integer	Unsigned Integer
<i>It represents both positive and negative integers</i>	<i>It represents only positive integers</i>
<i>The data type qualifier is <b>signed int</b> or <b>int</b>.</i>  <i>Variables are defined as:</i> <code>signed int a; int b;</code>	<i>The data type qualifier is <b>unsigned int</b> or <b>unsigned</b>.</i>  <i>Variables are defined as:</i> <code>unsigned int a; unsigned b;</code>
<i>By default all int are signed</i>	<i>Unsigned int have to be declared explicitly</i>
<i>It ranges from <math>-2^{15}</math> to <math>+2^{15}</math> i.e. -32768 to 32767</i>	<i>It ranges from 0 to <math>+2^{16}</math> i.e. 0 to 65535</i>
<i>Its conversion character is <b>d</b></i>	<i>Its conversion character is <b>u</b></i>

# Floating Point Type

- *Floating point types are fractional numbers*
- *In C, it is defined by **float***
- *Reserves 32 bits (i.e. 4 bytes)*
- *Variables are defined as:*

*float a;*

*float myValue=6.5;*

# Double Precision Floating Point Type

- *Used to represent the floating point numbers*
- *Reserves 64 bits (i.e. 8 bytes)*
- *In C, it is defined by **double***
- *Variables are defined as:*

*double a;*

*double myValue=4244.546;*

# Character Type

- *A single character can be defined as a character type data*
- *Stored in 8 bits (1 byte)*
- *The qualifier signed or unsigned may be used with **char***
- *The unsigned char has values between 0 and 255*
- *The signed char has values from -128 to 127*

# Character Type

- *Each character is represented by an ASCII (American Standard Code for Information Interchange)*
- *Example: “A” is represented by 65, “B” is represented by 66, “a” is represented by 97, “z” is represented by 122.*
- *With conversion character **d**, it will display ASCII value*
- *With conversion character **c**, it will display character*

# ASCII Table

Dec	Char	Dec	Char	Dec	Char	Dec	Char
0	NUL (null)	32	SPACE	64	@	96	~
1	SOH (start of heading)	33	!	65	A	97	a
2	STX (start of text)	34	"	66	B	98	b
3	ETX (end of text)	35	#	67	C	99	c
4	EOT (end of transmission)	36	\$	68	D	100	d
5	ENQ (enquiry)	37	%	69	E	101	e
6	ACK (acknowledge)	38	&	70	F	102	f
7	BEL (bell)	39	'	71	G	103	g
8	BS (backspace)	40	(	72	H	104	h
9	TAB (horizontal tab)	41	)	73	I	105	i
10	LF (NL line feed, new line)	42	*	74	J	106	j
11	VT (vertical tab)	43	+	75	K	107	k
12	FF (NP form feed, new page)	44	,	76	L	108	l
13	CR (carriage return)	45	-	77	M	109	m
14	SO (shift out)	46	.	78	N	110	n
15	SI (shift in)	47	/	79	O	111	o
16	DLE (data link escape)	48	0	80	P	112	p
17	DC1 (device control 1)	49	1	81	Q	113	q
18	DC2 (device control 2)	50	2	82	R	114	r
19	DC3 (device control 3)	51	3	83	S	115	s
20	DC4 (device control 4)	52	4	84	T	116	t
21	NAK (negative acknowledge)	53	5	85	U	117	u
22	SYN (synchronous idle)	54	6	86	V	118	v
23	ETB (end of trans. block)	55	7	87	W	119	w
24	CAN (cancel)	56	8	88	X	120	x
25	EM (end of medium)	57	9	89	Y	121	y
26	SUB (substitute)	58	:	90	Z	122	z
27	ESC (escape)	59	;	91	[	123	{
28	FS (file separator)	60	<	92	\	124	
29	GS (group separator)	61	=	93	]	125	}
30	RS (record separator)	62	>	94	^	126	~
31	US (unit separator)	63	?	95	_	127	DEL

# NOTE

- *The difference of corresponding uppercase and lowercase character is always 32*
  - *ASCII value of “a” – ASCII value of “A” = 32*
  - *ASCII value of “m” – ASCII value of “M” = 32*
- *Using this logic, we can convert uppercase letter into its lowercase and vice – versa.*

# Void Type

- *This void type has no value*
- *This is usually used to specify a type of function when it does not return any value to the calling function*
- *Example:*  
`void main()`  
`void whatIsThis();`

# Derived Data Types

- C supports a feature called *type definition* which allows users to define an identifier that would represent an existing data type.
- *typedef* statement is used to give new name to an existing data type.
- It allows users to define new data types that are equivalent to an existing data types.
- General form:  
*typedef existing\_data\_type new\_name\_for\_existing\_data\_type;*
- Example:  
*typedef int integer;*

# Constants

- *A **constant** is a quantity that doesn't change during the execution*
- *These fixed values are also called **literals***
- *Constants can be of any of the basic data types like an **integer constant**, a **floating constant**, a **character constant**, or a **string literal***
- *There are enumeration constants as well*

# Integer Literals

- An integer literal can be a **decimal**, **octal**, or **hexadecimal** constant.
- A prefix specifies the base or radix: **0x** or **0X** for hexadecimal, **0** for octal, and nothing for decimal.
- An integer literal can also have a suffix that is a combination of **U** and **L**, for **unsigned** and **long**, respectively.
- Following are other examples of various types of integer literals:

85	/* decimal */
0213	/* octal */
0x4b	/* hexadecimal */
30	/* int */
30l	/* long */

# Character Constants

- *Character literals are enclosed in single quotes*  
*‘x’ can be stored in a simple variable of **char** type*
- *A character literal can be a:*
  - *Plain character (e.g. ‘x’)*
  - *An escape sequence (e.g. ‘\t’)*
  - *Or a universal character (e.g. ‘\u02C0’)*
- *There are certain characters in C that represent special meaning when preceded by a backslash. For example, new line (\n) or tab (\t)*

# String Constants

- *Sequence of characters enclosed in double quotes*
- *May contain letters, numbers, special characters or blank spaces*
- *Example:*

“hello”

“hi”

“2076”

# Variables

- *A symbolic name which is used to store data item. i.e. a numerical quantity or a character constant.*
- *Unlike constant, the value of a variable can change during the execution of a program.*
- *The same variable can store different value at different portion of a program.*
- *Variable name may consist of letters, digits, or underscore characters.*

# Variable Declaration

- Any variable should be defined before using it in a program
- Variable declaration syntax:  
*data-type variable\_name1, variable\_name2, ...*

- Valid declaration are:

*int n1;*

*int centi, temp;*

*float radius;*

*char gender;*

# Rules for Variable Declaration

- *The variable name should start with only letters.*
- *The variable name shouldn't be keyword.*
- *White spaces are not allowed between characters of variable but underscores are approved.*
- *The variable name is case sensitive.*  
***TEMP** and **temp** are different variables.*
- *No two variables of the same name are allowed to be declared in the same scope.*

# Preprocessor Directives

- *Collection of special statements that are executed at the beginning of a compilation process.*
- *Placed in the source program before the main function.*

`#include<stdio.h>` // used for file inclusion

`#define PI 3.1416` // defining symbolic constant PI

`#define TRUE 1` // used for defining TRUE as 1

- *These statements are called preprocessor directives as they are processed before compilation of any other source code in the program.*

# Escape Sequences

- *An escape sequence is a non-printing characters used in C.*
- *Character combination consisting of backslash (\) followed by a letter or by a combination of digits.*
- *Each sequences are typically used to specify actions such as carriage return, backspace, line feed, or move cursors to next line.*

# Escape Sequences

Character	Escape Sequence	ASCII Value
<i>Bell (alert)</i>	\a	007
<i>Backspace</i>	\b	008
<i>Horizontal tab</i>	\t	009
<i>Vertical tab</i>	\v	011
<i>Newline (linefeed)</i>	\n	010
<i>Form feed</i>	\f	012
<i>Carriage return</i>	\r	013
<i>Quotation mark ("")</i>	\“	034
<i>Apostrophe (')</i>	\'	039
<i>Question mark (?)</i>	\?	063
<i>Backslash (\)</i>	\\\	092
<i>Null</i>	\0	000

# Escape Sequence in C

```
#include<stdio.h>

void main()
{
    printf("Hello! \n I am testing an escape sequence");
}
```

## OUTPUT:

*Hello!*

*I am testing an escape sequence*

# Escape Sequence in C

```
#include<stdio.h>

void main()
{
    printf("Hello \tWorld \n");
    printf("He said, \"Hello\"");
}
```

## OUTPUT:

Hello    World

He said, "Hello"

# Tokens in C

- *The basic elements recognized by the C compiler are the “**tokens**”.*
- *In C, tokens are of six types:*
  - *Keywords (e.g. int, while, float, printf)*
  - *Identifiers (e.g. sum, total, num\_of\_hours)*
  - *Constants (e.g 10, 20, -15.4)*
  - *Strings (e.g. “ram”, “hello”)*
  - *Special symbols (e.g. (), {})*
  - *Operators (e.g. +, -, \*, /)*

# Delimiters

- *A delimiter is a unique character or series of characters that indicates the beginning or end of a specific statement, string or function body set.*
- *Delimiter examples include:*
  - *Round brackets or parentheses – ()*
  - *Curly brackets – {}*
  - *Escape sequence or comments – /\**
  - *Double quotes for defining string literals – “ ”*

# Expressions

- *In programming, an expression is any legal combination of symbols that represents a value.*
- *For example, in the C language,  **$x + 5$**  is a legal expression.*
- *Every expression consists of at least one operand and can have one or more operators.*

# Expressions

- Operands are values and operators are symbols that represent particular actions.
- Types of expressions:

Type	Explanation	Example
Infix	Expression in which operator is in between operands	$a+b$
Prefix	Expression in which operator is written before operands	$+ab$
Postfix	Expression in which operator is written after operands	$ab+$

# Comments

- *Used for program documentation.*
- *Comments are not compiled.*
- *The C syntax for writing comment is*  
*/\**  
*anything written in between slash and asterisk and asterisk and slash is a comment*  
*\*/*
- *Another way to write comment in C is*  
*// using double slash (this line only)*

***END OF UNIT TWO***

# Syllabus

- ***UNIT 3: Input and Output***

- *Conversion Specification*
- *Reading a character*
- *Writing a character*
- *I/O Operations*
- *Formatted I/O*

# UNIT 3

***INPUT AND OUTPUT***

# Data Input and Output

- *A program without any input or output has no meaning.*
- *Input → Process → Output*
- *Reading the data from input devices and displaying the result are the two main tasks of any program.*

# Data Input and Output

- *Input/Output functions are the links between the user and the terminal.*
- *Input functions that are used to read data from keyboard are called **standard input functions**. Example: `scanf()`, `getchar()`, `getch()`, etc.*
- *Output functions that are used to display the result on the screen are called **standard output functions**. Example: `printf()`, `putchar()`, `puts()`, etc.*

# Data Input and Output

- In C, the standard library **stdio.h** provides functions for input and output.
- The instruction **#include<stdio.h>** tells the compiler to search for a file named **stdio.h** and places its contents at this point in the program.
- The contents of the header file become part of the source code when it is compiled.
- The input/output functions are classified into two types:
  - Formatted functions
  - Unformatted functions

# Formatted Functions

- Formatted functions allow to read the input from the keyboard or the output displayed on screen to be formatted according to our requirements.

*input function: `scanf()`*

*output function: `printf()`*

- **Example:** Consider the following data – 50, 13.45, Ram (int, float, char)
- This is possible using the **`scanf()`** function which stands for scan formatted.

# Formatted Functions

- The built-in function `scanf()` can be used to enter input data into the computer from a standard input device.
- The general form of `scanf` is  
`scanf("control string", arg1, arg2, ..., argn);`  
*control string* → format in which data is to be entered  
*arg1, arg2, ...* → location where data is stored preceded by ampersand (&)
- The control string consists of individual groups of data formats, with one group for each input data item.
- Each data format must begin with a percentage sign.

# Use of *printf()* & *scanf()*

- *%c* → *character* → *printf*(“%c”, ‘a’);
- *%d* → *decimal integer* → *printf*(“%d”, 100);
- *%f* → *floating point number* → *printf*(“%f”, 1.234);
- *%s* → *string* → *printf*(“%s”, “C-book”);

# Format Specifiers for I/O

Data Type	Format Specifier
<i>int</i>	<code>%d</code>
<i>char</i>	<code>%c</code>
<i>float</i>	<code>%f</code>
<i>double</i>	<code>%lf</code>
<i>short int</i>	<code>%hd</code>
<i>unsigned int</i>	<code>%u</code>
<i>long int</i>	<code>%ld</code>
<i>long long int</i>	<code>%lld</code>

```
#include<stdio.h>  
  
void main()  
{  
    /* Displays the string inside  
quotations */  
    printf("C Programming");  
}
```

## OUTPUT

*C Programming*

# C Output

- All valid C programs must contain the `main()` function. The code execution begins from the start of the `main()` function.
- The `printf()` is a library function to send formatted output to the screen. The function prints the string inside quotations.
- To use `printf()` in our program, we need to include `stdio.h` header file using the `#include<stdio.h>` statement.

# Integer Output

```
#include<stdio.h>

int main()
{
    int testInteger = 5;
    printf("Number = %d", testInteger);
    return 0;
}
```

- We use `%d` format specifier to print `int` types.
- Here, the `%d` inside the quotations will be replaced by the value of `testInteger`.

## OUTPUT

*Number = 5*

# Float & Double Output

```
#include<stdio.h>
int main()
{
    float num1 = 13.5;
    double num2 = 12.4;
    float num3 = 14.62113;
    printf("number1 = %f\n", num1);
    printf("number2 = %lf\n", num2);
    printf("number3 = %.3f", num3);
    return 0;
}
```

## OUTPUT

number1 = 13.500000  
number2 = 12.400000  
number3 = 14.621

- To print float, we use %f format specifier.
- Similarly, we use %lf to print double values
- Also, we use %.3f to print float value with three decimal places.

# Characters Output

```
#include<stdio.h>

int main()
{
    char chr = 'a';
    printf("character = %c", chr);
    return 0;
}
```

- To print char, we use %c format specifier.

## OUTPUT

*character = a*

```
#include <stdio.h>

int main()
{
    int testInteger;
    printf("Enter an integer: ");
    scanf("%d", &testInteger);
    printf("Number = %d", testInteger);
    return 0;
}
```

## OUTPUT

Enter an integer: 4  
Number = 4

# C Input

- In C programming, `scanf()` is one of the commonly used function to take input from the user.
- The `scanf()` function reads formatted input from the standard input such as keyboards.
- Here, we have used `%d` format specifier inside the `scanf()` function to take int input from the user.
- When the user enters an integer, it is stored in the `testInteger` variable.
- Notice, that we have used `&testInteger` inside `scanf()`.
- It is because `&testInteger` gets the address of `testInteger`, and the value entered by the user is stored in that address.

# Float & Double Input/Output

```
#include <stdio.h>
int main()
{
    float num1;
    double num2;
    printf("Enter a number: ");
    scanf("%f", &num1);
    printf("Enter another number: ");
    scanf("%lf", &num2);
    printf("num1 = %f\n", num1);
    printf("num2 = %lf", num2);
    return 0;
}
```

## OUTPUT

Enter a number: 12.523

Enter another number: 10.2

num1 = 12.523000

num2 = 10.200000

- We use %f and %lf format specifier for float and double respectively.

# Characters Input/Output

```
#include <stdio.h>
int main()
{
    char chr;
    printf("Enter a character: ");
    scanf("%c", &chr);
    printf("You entered %c. ", chr);
    return 0;
}
```

## OUTPUT

Enter a character: g  
You entered g

- When a character is entered by the user in the above program, the character itself is not stored.
- Instead, an integer value (ASCII value) is stored.
- And when we display that value using %c text format, the entered character is displayed.
- If we use %d to display the character, its ASCII value is printed.

```
#include <stdio.h>
int main()
{
    char chr;
    printf("Enter a character: ");
    scanf("%c", &chr);
    // When %c is used, a character is displayed
    printf("You entered %c.\n", chr);
    // When %d is used, ASCII value is displayed
    printf("ASCII value is %d.", chr);
    return 0;
}
```

## OUTPUT

Enter a character: g  
You entered g.  
ASCII value is 103.

# ASCII Value

- When a character is entered by the user in the above program, the character itself is not stored.
- Instead, an integer value (ASCII value) is stored.
- And when we display that value using %c text format, the entered character is displayed.
- If we use %d to display the character, it's ASCII value is printed.

# Input/Output Multiple Values

```
#include <stdio.h>
int main()
{
    int a;
    float b;
    printf("Enter integer and then a float: ");
    // Taking multiple inputs
    scanf("%d%f", &a, &b);
    printf("You entered %d and %f", a, b);
    return 0;
}
```

## OUTPUT

*Enter integer and then a float: -3  
3.4*

*You entered -3 and 3.400000*

# *Input & Output of Basic Types in C*

- ***Integer***

- *Input:* `scanf("%d", &intVariable);`
- *Output:* `printf("%d", intVariable);`

- ***Float***

- *Input:* `scanf("%f", &floatVariable);`
- *Output:* `printf("%f", floatVariable);`

- ***Character***

- *Input:* `scanf("%c", &charVariable);`
- *Output:* `printf("%c", charVariable);`

# *Input & Output of Advanced Types in C*

- *String*

- *Input:* `scanf("%s", stringVariable);`
- *Output:* `printf("%s", stringVariable);`

# Example - 1

```
// Program to display user input integer, float, and string values
#include<stdio.h>

void main()
{
    int n1;
    float n2;
    char ch[10];
    printf("Enter an integer number:");
    scanf("%d", &n1);
    printf("Enter a float number:");
    scanf("%f", &n2);
    printf("Enter a string:");
    scanf("%s", &ch);
    printf("\nInteger Number: %d \t Float Number: %f \t String: %s", n1, n2, ch);
}
```

}

## Example – 2

*// Program to add, subtract, multiply, and divide two whole numbers*

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

```
void main()
```

```
{
```

```
    int a, b, c;
```

```
    float d;
```

```
    printf("Enter two whole numbers: \n");
```

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

```
    c = a + b;
```

```
    printf("\nSum = %d", c);
```

```
    c = a - b;
```

```
    printf("\nDifference = %d", c);
```

```
    c = a * b;
```

```
    printf("\nMultiplication = %d", c);
```

```
    d = (float)a / b;
```

```
    printf("\nDivision = %f", d);
```

# Example – 3

```
// Program to convert a temperature given in Celsius to Fahrenheit
#include<stdio.h>

void main()
{
    float c,f;
    printf("Enter temperature in celsius:");
    scanf("%f", &c);
    f = c * 9 / 5 + 32;
    printf("Temperature in fahrenheit = %f",f);
}
```

# Example – 4

```
// Program to find area and circumference of circle
#include<stdio.h>
#include<math.h>
#define PI 3.1415
void main()
{
    float r, a, c;
    printf("Enter radius:");
    scanf("%f", &r);
    a = PI * pow(r, 2);
    c = 2 * PI * r;
    printf("Area = %f\n", a);
    printf("Circumference = %f", c);
}
```

# Unformatted Functions

- *Unformatted I/O functions allow to supply input or display output in user desired format.*
- *getch(), getche(), getchar(), gets(), puts(), putchar(), etc. are the examples of unformatted input/output functions.*
- *Unformatted input and output functions do not contain format specifier in their syntax.*
- *Unformatted I/O functions are used for storing data more compactly.*
- *Mainly, unformatted I/O functions are used for character and string data types.*

# getchar() & putchar() Functions

- *The `getchar()` function reads a character from the terminal and returns it as an integer.*
- *This function reads only single character at a time.*
- *You can use this method in a loop in case you want to read more than one character.*
- *The `putchar()` function displays the character passed to it on the screen and returns the same character.*
- *This function too displays only a single character at a time.*
- *In case you want to display more than one characters, use `putchar()` method in a loop.*

# getchar() & putchar() Functions

```
#include <stdio.h>

void main( )
{
    int c;

    printf("Enter a character: \n");
    // Take a character as input and store it in variable c
    c = getchar();

    // display the character stored in variable c
    printf("The entered character is: ");
    putchar(c);

}
```

# gets() & puts() Functions

- The *gets()* function reads a line from *stdin* (standard input) into the buffer pointed to by *str* pointer, until either a terminating newline or EOF (end of file) occurs.
- The *puts()* function writes the string *str* and a trailing newline to *stdout* (standard output).
- *str* → This is the pointer to an array of chars where the C string is stored. (*Ignore if you are not able to understand this now*)

# gets() & puts() Functions

```
#include<stdio.h>

void main()
{
    /* character array of length 100 */
    char name[100];
    printf("Enter your name: \n");
    gets(name);
    printf("Your name is ");
    puts(name);
}
```

# scanf() & gets() - Difference

- *The main difference between these two functions is that scanf() stops reading characters when it encounters a space, but gets() reads space as character too.*
- *If you enter name as **Sulav Nepal** using scanf() it will only read and store **Sulav** and will leave the **Nepal** part after space.*
- *But gets() function will read it completely.*

***END OF UNIT THREE***

# Syllabus

- *UNIT 4: Operators and Expressions (4 Hrs.)*
  - *Arithmetic Operator*
  - *Relational Operator*
  - *Logical or Boolean Operator*
  - *Assignment Operator*
  - *Ternary Operator*
  - *Bitwise Operator*
  - *Increment or Decrement Operator*
  - *Conditional Operator*
  - *Special Operators (sizeof and comma)*
  - *Evaluation of Expression (implicit and explicit type conversion)*
  - *Operator Precedence and Associativity*

# UNIT 4

***OPERATORS AND EXPRESSIONS***

# Operators

- *Symbol that operates on certain data type or data item.*
- *Used in program to perform certain mathematical or logical manipulations.*
- *For example: In a simple expression **5+6**, the symbol “**+**” is called an operator which operates on two data items **5** and **6**.*
- *The data items that operator act upon are called **operands**.*

# Expression

- *An expression is a combination of variables, constants, and operators written according to syntax of the language.*
- *For example:*

$8+10$

$a+c*d$

$a>b$

$a/c$

# Operators

- *We can classify operators into:*
  - *Unary Operators*
    - *Which requires only one operand*
    - *For example: ++, --*
  - *Binary Operators*
    - *Which requires two operands*
    - *For example: +, -, \*, /, <, >*
  - *Ternary Operators*
    - *Which requires three operands*
    - *For example: “?:” (conditional operator)*

# Arithmetic Operators

- Assume variable  $A$  holds 20 and variable  $B$  holds 10, then

Operator	Description	Example
+	Adds two operands	$A + B = 30$
-	Subtracts second operand from the first	$A - B = 10$
*	Multiplies both operands	$A * B = 200$
/	Divides numerator by de-numerator	$A/B = 2$
%	Modulus operator and remainder of after an integer division	$A \% B = 0$
++	Increment operator increases the integer value by one	$A ++ = 21$
--	Decrement operator decreases the integer value by one	$A -- = 19$

# Integer Arithmetic

- *Division Rule*
  - $\text{int/int} = \text{int}$
  - $\text{float/float} = \text{float}$
  - $\text{int/float} = \text{float}$
  - $\text{float/int} = \text{float}$

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a=21; int b=10; int c;
    c=a+b;
    printf("Line 1 - Value of c is %d\n",c);
    c=a-b;
    printf("Line 2 - Value of c is %d\n",c);
    c=a*b;
    printf("Line 3 - Value of c is %d\n",c);
    c=a/b;
    printf("Line 4 - Value of c is %d\n",c);
    c=a%b;
    printf("Line 5 - Value of c is %d\n",c);
    a++;
    printf("Line 6 - Value of c is %d\n",a);
    a--;
    printf("Line 7 - Value of c is %d\n",a);
    getch();
}
```

# Relational Operators

- Assume variable  $A$  holds 10 and variable  $B$  holds 20, then

Operator	Description	Example
$==$	Checks if the values of two operands are equal or not. If yes, then the condition becomes true.	$(A == B)$ is not true
$!=$	Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true.	$(A != B)$ is true
$>$	Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true.	$(A > B)$ is not true
$<$	Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true.	$(A < B)$ is true
$>=$	Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true.	$(A >= B)$ is not true
$<=$	Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true.	$(A <= B)$ is true

# EXAMPLE - TWO

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a=21, int b=10, int c;
    if(a==b)
    {
        printf("Line 1 - a is equal to b\n");
    }
    else
    {
        printf("Line 1 - a is not equal to b\n");
    }
    if(a<b)
    {
        printf("Line 2 - a is less than b\n");
    }
    else
    {
        printf("Line 2 - a is not less than b\n");
    }
    if(a>b)
    {
        printf("Line 3 - a is greater than b\n");
    }
    else
    {
        printf("Line 3 - a is not greater than b\n");
    }
    if(a<=b)
    {
        printf("Line 4 - a is either less than or equal to b\n");
    }
    if(a>=b)
    {
        printf("Line 4 - a is either greater than or equal to b\n");
    }
    getch();
}
```

# Logical Operators

- Assume variable  $A$  holds 1 and variable  $B$  holds 0, then

Operator	Description	Example
$\&\&$	Called “Logical AND Operator”. If both the operands are non-zero, then the condition becomes true.	$(A \&\& B)$ is false
$\ $	Called “Logical OR Operator”. If any of the two operands is non-zero, then the condition becomes true.	$(A \  B)$ is true
!	Called “Logical NOT Operator”. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT Operator will make it false.	$!(A \&\& B)$ is true $!(A \  B)$ is false

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a=5, b=20;
    if(a && b)
    {
        printf("Line 1 - Condition is true\n");
    }
    else
    {
        printf("Line 1 - Condition is not true\n");
    }
    if(a || b)
    {
        printf("Line 2 - Condition is true\n");
    }
    else
    {
        printf("Line 2 - Condition is not true\n");
    }
    // Let us change the value of a and b
    a=0; b=10;
    if(a && b)
    {
        printf("Line 3 - Condition is true\n");
    }
    else
    {
        printf("Line 3 - Condition is not true\n");
    }
    if(!(a && b))
    {
        printf("Line 4 - Condition is true\n");
    }
    getch();
}
```

# Assignment Operators

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand	$C = A + B$ will assign the value of $A + B$ to $C$
+=	Add AND assignment operator. It adds the right operand to the left operand and assign the results to the left operand	$C += A$ is equivalent to $C = C + A$
-=	Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand	$C -= A$ is equivalent to $C = C - A$
*=	Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand	$C *= A$ is equivalent to $C = C * A$
/=	Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand	$C /= A$ is equivalent to $C = C / A$
%=	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand	$C \% = A$ is equivalent to $C = C \% A$

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a=1, b=2, c;
    c=a+b;
    printf("Line 1 - c=%d",c);
    c+=a;
    printf("\nLine 2 - c=%d",c);
    c-=a;
    printf("\nLine 3 - c=%d",c);
    c*=a;
    printf("\nLine 4 - c=%d",c);
    c/=a;
    printf("\nLine 5 - c=%d",c);
    c%=a;
    printf("\nLine 6 - c=%d",c);
    getch();
}
```

# Increment & Decrement Operators

- *Increment operator is used to increase the value of an operand by 1*
- *Decrement operator is used to decrease the value of an operand by 1*

Operator	Description	Example
<code>++</code>	<code>++variable</code> (prefix notation)	<code>variable = variable + 1</code>
<code>++</code>	<code>variable ++</code> (postfix notation)	<code>variable = variable + 1</code>
<code>--</code>	<code>--variable</code> (prefix notation)	<code>variable = variable - 1</code>
<code>--</code>	<code>variable --</code> (postfix notation)	<code>variable = variable - 1</code>

# Increment & Decrement Operators

- **Pre-Increment or Pre-Decrement ( $++a$  or  $--a$ )**
  - **CHANGE** the value of the variable
  - **USE** the new value
- **Post-Increment or Post-Decrement ( $a++$  or  $a--$ )**
  - **USE** the original value of the variable
  - **CHANGE** the value of the variable

# EXAMPLE - FIVE

```
/* Use of pre increment & post increment Operators */
#include<stdio.h>
#include<conio.h>
void main()
{
    int a=20, b=10;
    printf("INCREMENT OPERATORS\n");
    printf("%d\n",a);
    printf("%d\n",++a);
    printf("%d\n",a++);
    printf("%d\n",a);
    printf("\nDECREMENT OPERATORS\n");
    printf("%d\n",b);
    printf("%d\n",--b);
    printf("%d\n",b--);
    printf("%d\n",b);
    getch();
}
```

# Conditional Operator

- *The operator pair “?:” is known as conditional operator.*
- *It takes three operands.*
- *Also called as **ternary operator**.*
- *General form:*  
*expression1? expression2: expression3*  
*expression1 is evaluated first*  
*If expression1 is true*  
*then value of expression2 is the value of condition expression*  
*Else*  
*the value of expression3 is the value of conditional expression*

# EXAMPLE – SIX

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int num1, num2, larger;
    printf("Enter two numbers:\n");
    scanf("%d %d", &num1, &num2);
    larger=num1>num2?num1:num2;
    printf("\nThe larger number is %d", larger);
    getch();
}
```

# Bitwise Operator

- *Bitwise operators are used for manipulating data at bit level.*
- *These operators are used for testing the bits or shifting them to the left or to the right.*
- *Can be applied only to integer-type operands and not to float or double.*
- *Three types of bitwise operators*
  - *Bitwise Logical Operators*
  - *Bitwise Shift Operators*
  - *One's Complement Operator*

# Bitwise Logical Operator

- *Performs logical tests between two integer-type operands.*
- *These operators work on their operands bit-by-bit starting from the least significant (i.e. rightmost) bit.*
- *Three logical bitwise operators:*
  - *Bitwise AND (&)*
  - *Bitwise OR (|)*
  - *Bitwise Exclusive OR (^)*

# Bitwise AND (&)

- *Logical ANDing between two operands.*
- *The result of ANDing operation is 1 if both the bits have a value of 1; otherwise it is 0.*
- Consider  $num1 = 45$  and  $num2 = 25$ 
  - $num1 \rightarrow 0000\ 0000\ 0010\ 1101$
  - $num2 \rightarrow 0000\ 0000\ 0001\ 1001$
- If  $num3 = num1 \& num2$ 
  - $num3 \rightarrow 0000\ 0000\ 0000\ 1001$

# Bitwise OR (I)

- *Logical ORing between two operands.*
- *The result of ORing operation is 1 if either of the bits have a value of 1; otherwise it is 0.*
- Consider  $num1 = 45$  and  $num2 = 25$ 
  - $num1 \rightarrow 0000\ 0000\ 0010\ 1101$
  - $num2 \rightarrow 0000\ 0000\ 0001\ 1001$
- If  $num3 = num1 \mid num2$ 
  - $num3 \rightarrow 0000\ 0000\ 0011\ 1101$

# Bitwise Exclusive XOR (^)

- *Logical Exclusive ORing between two operands.*
- *The result of Exclusive ORing operation is 1 only if one of the bits have a value of 1; otherwise it is 0.*
- Consider  $num1 = 45$  and  $num2 = 25$ 
  - $num1 \rightarrow 0000\ 0000\ 0010\ 1101$
  - $num2 \rightarrow 0000\ 0000\ 0001\ 1001$
- If  $num3 = num1 \wedge num2$ 
  - $num3 \rightarrow 0000\ 0000\ 0011\ 0100$

# EXAMPLE – SEVEN

```
#include<stdio.h>

void main()
{
    int num1=45, num2=25, AND, OR, XOR;

    AND = num1 & num2;

    OR = num1 | num2;

    XOR = num1 ^ num2;

    printf("AND=%d\n", AND);

    printf("OR=%d\n", OR);

    printf("XOR=%d\n", XOR);

}
```

# Bitwise Shift Operators

- *These operators are used to move bit patterns either to the left or to the right.*
- *There are two bitwise shift operators:*
  - *Left shift (<<)*
  - *Right shift (>>)*

# Bitwise Left Shift (<<) Operators

- It causes the operand to be shifted to the left by  $n$  positions.  
 $operand \ll n$
- The leftmost  $n$  bits in the original bit pattern will be lost and the rightmost  $n$  bits empty positions will be filled with 0's.
- Example:  $num1 = 45$ ; execute  $num2 = num1 \ll 3$ ;

<b>num1</b>	0000 0000 0010 1101
Shift 1	0000 0000 0101 1010
Shift 2	0000 0000 1011 0100
Shift 3	0000 0001 0110 1000 (num2)

# Bitwise Right Shift (>>) Operators

- It causes the operand to be shifted to the right by  $n$  positions.  
 $operand >> n$
- The empty leftmost  $n$  bits position will be filled with 0's, if the operand is an unsigned integer.
- Example: **Unsigned int num1 = 45**; execute **num2 = num1 >> 3;**

<b>num1</b>	0000 0000 0010 1101
Shift 1	0000 0000 0001 0110
Shift 2	0000 0000 0000 1011
Shift 3	0000 0000 0000 0101 (num2)

# Bitwise One's Complement Operator

- It is a unary operator which inverts all the bits represented by its operand. i.e. all 0's becomes 1's and all 1's becomes 0's.
- For any integer  $n$ , bitwise one's complement of  $n$  will be  $-(n + 1)$ .
- Example: If  $num1 = 45$ , then we execute the statement;  $num2 = \sim num1$ ;
- The resulting bit pattern represents the decimal  $-46$ .

<b>num1</b>	0000 0000 0010 1101
$\sim num1$	1111 1111 1101 0010 (num2)

# Bitwise One's Complement Operator

## Bitwise One's Complement Operator

45	0	0	0	0	0	0	0	0	0	1	0	1	1	0	1	=	45
----	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	----

Converts 0 to 1 and 1 to 0

$\sim 45$	1	1	1	1	1	1	1	1	1	0	1	0	0	1	0	=	-46
-----------	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	-----

# Bitwise One's Complement Operator

How negative integer is stored in memory?															
+46	0	0	0	0	0	0	0	0	0	0	1	0	1	1	1
<i>One's complement of 46</i>															0
<i>Two's Complement of 46 (add 1)</i>															1
-46	1	1	1	1	1	1	1	1	1	1	0	1	0	0	1
<i>Equivalent to ~45</i>															+
~45	1	1	1	1	1	1	1	1	1	1	0	1	0	0	1

# EXAMPLE – EIGHT

```
#include<stdio.h>

void main()
{
    int num1=45, left, right, comp;
    left = num1<<3;
    right = num1>>3;
    comp = ~num1;
    printf("%d\n", left);
    printf("%d\n", right);
    printf("%d\n", comp);
}
```

# Special Operators

- C supports some special operators like **sizeof** operator and **comma** operator.
- **sizeof Operator:**
  - It is used with an operand to return the number of bytes it occupies.
  - The operand may be constant, variable or a data type qualifier.

# Special Operators

- C supports some special operators like **sizeof** operator and **comma** operator.
- **comma Operator:**
  - It can be used to link related expressions together.
  - A comma-linked list of expressions are evaluated from left-to-right and the value of the rightmost expression is the value of the combined expressions.
  - For example:  $num3 = (num1 = 45, num2 = 25, num1 + num2)$ 
    - At first, 45 is assigned to num1
    - Then, 25 is assigned to num2
    - Then finally, sum of num1 and num2 is assigned to num3

# EXAMPLE – NINE

```
#include <stdio.h>

int main()
{
    short int a = 0;
    printf("Size of variable a : %d\n", sizeof(a));
    printf("Size of int data type : %d\n", sizeof(int));
    printf("Size of char data type : %d\n", sizeof(char));
    printf("Size of float data type : %d\n", sizeof(float));
    printf("Size of double data type : %d\n", sizeof(double));
    return 0;
}
```

# Expressions

- *Expressions are the representation of code inside C text editor which is recognized by compiler.*
- *An expression in C consists of syntactically valid combination of operators and operands that computes to a value.*
- *The C expressions are not a statement.*
- *The expressions are the basic building blocks of statement.*
- *The C expressions are slightly differ than mathematical expressions.*
- *An expression is a combination of variables, constants, and operators written according to the syntax of C language.*

# *Evaluation of Expressions*

- *The changing an entity of one data type into another is called type casting or coercion.*
- *This is performed to take advantage of certain features of type representations.*
- *In general, fundamental data types can be converted.*
- *The word coercion is used to denote an implicit type casting.*
- *There are two types of casting:*
  - *Implicit Type Casting*
  - *Explicit Type Casting*

# Implicit Type Casting

- The conversion of data is performed either during compilation or run time is called implicit type casting.
- It is the automatic type conversion process performed by compiler itself.
- The data can be lost during this type of casting.
- The conversion from *float* to *int* can cause loss of higher order bits.
- For example:

```
int p, t, r, i;  
i=p*t*r/100.00;
```

Here, the variables are declared as integer but in the calculation  $i = p * t * r / 100.00$ , after dividing by 100.00, the value of *i* will change in to float variable.

# Explicit Type Casting

- *Explicit type conversion can either be performed by built-in functions or by a special syntax generated by coder.*
- *These syntax changes one data type to other by using conversion keyword.*
- *It is a secure manner of changing variables from one data type to other.*
- *For example:*  
`int a=97, b=65;  
printf("%c %c", (char)a, (char)b);`
- *Here, the ASCII value of a and A are 97 and 65 respectively. So, we print the value after type casting from integer to character.*

# Operator Precedence and Associativity

- *The precedence is used to determine how an expression involving more than one operator is evaluated.*
- *There are distinct level of precedence.*
- *The operators at the higher level of precedence are evaluated first.*
- *Operators of same precedence are evaluated either from “left to right” or “right to left” depending on the level also known as **associativity**.*

Category	Operator	Associativity
Postfix	( ) [] -> . ++ --	Left to Right
Multiplicative	* / %	Left to Right
Additive	+ -	Left to Right
Shift	<< >>	Left to Right
Relational	< <= > >=	Left to Right
Equality	== !=	Left to Right
Bitwise AND	&	Left to Right
Bitwise OR		Left to Right
Bitwise XOR	^	Left to Right
Logical AND	&&	Left to Right
Logical OR		Left to Right
Comma	,	Left to Right
Unary	+ - ! ~ ++ -- (type) * & sizeof	Right to Left
Conditional	? :	Right to Left
Assignment	= += -= *= /= %= >>= <<= &= ^=  =	Right to Left

***END OF UNIT FOUR***

# Syllabus

- *UNIT 5: Control Statement*
  - *Conditional Statements*
  - *Decision Making and Branching (if, if else, nested if else, else if ladder, switch statements)*
  - *Decision Making and Looping (for, while, do while loops)*
  - *Exit function*
  - *Break and Continue*

# UNIT 5

***CONTROL STATEMENT***

# Error Types in C Programming

- *While writing C programs, errors may occur which are also known as bugs.*
- *Errors may occur unwillingly which may prevent the program to compile and run correctly as per the expectation of the programmer.*
- *Basically there are three types of errors in C programming:*
  - *Runtime Errors*
  - *Compile Errors*
  - *Logical Errors*

# Runtime Errors in C

- *C runtime errors are those errors that occur during the execution of a C program and generally occur due to some illegal operation performed in the program.*
- *Examples of some illegal operations that may produce runtime errors are:*
  - *Dividing a number by zero*
  - *Trying to open a file which does not exist*
  - *Lack of free memory space*

# Compile Errors in C

- *Compile errors are those errors that occur at the time of compilation of the program.*
- *C compile errors may be further classified as:*
  - *Syntax Errors*
    - *Example: `int a, b:`*
    - *This will produce syntax error as the statement is terminated with `:` rather than `;`*
  - *Semantic Errors*
    - *Example: `b + c = a;`*
    - *Here, we are trying to assign a value of `a` in the value obtained by adding `b` and `c` which has no meaning in C.*
    - *The correct statement will be: `a = b + c;`*

# Logical Errors in C

- *Logical errors are the errors in the output of the program.*
- *The presence of logical errors leads to undesired or incorrect output.*
- *These errors are caused due to error in the logic applied in the program to produce the desired output.*
- *Logical errors could not be detected by the compiler, and thus, programmers has to check the entire coding of a C program line by line.*

# Control Statements

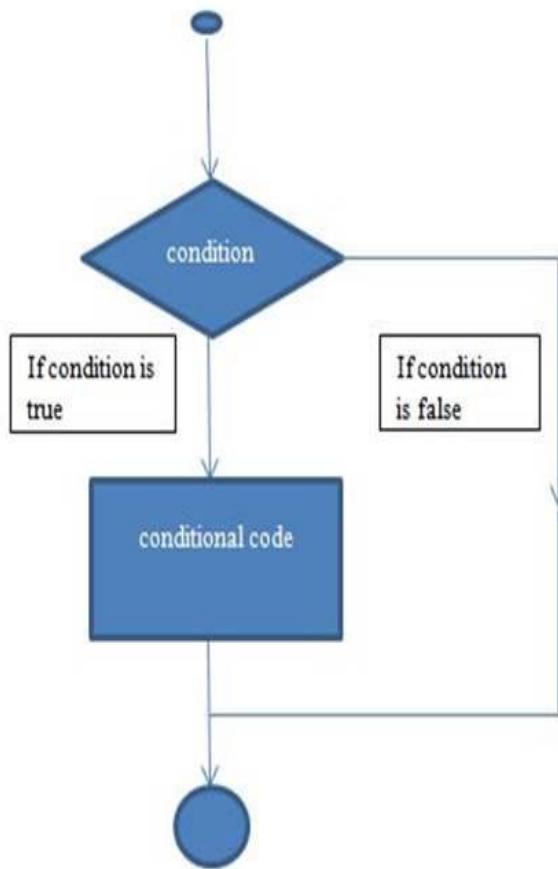
- *The statement which alter the flow of execution of the program are known as **control statements**.*
- *Sometimes we have to do certain tasks depending on whether a condition is true or not.*
- *Similarly, it is necessary to perform repeated actions or skip some statements.*
- *For these operations, control statements are needed.*
- *There are two types of control statements:*
  - *Decision Making (or branching) Statements*
  - *Loop or Repeating Construct*

# Control Statements

- *Decision Making (or branching) Statements*
  - *If statement*
  - *If...Else statement*
  - *Else...If statement*
  - *Nested If...Else statement*
  - *Switch statement*
- *Loop or Repeating Construct*
  - *For loop*
  - *While loop*
  - *Do...While loop*
- ***NOTE:***
  - *Branching* is deciding what actions to take
  - *Looping* is deciding how many times to take a certain action

# Decision Making in C

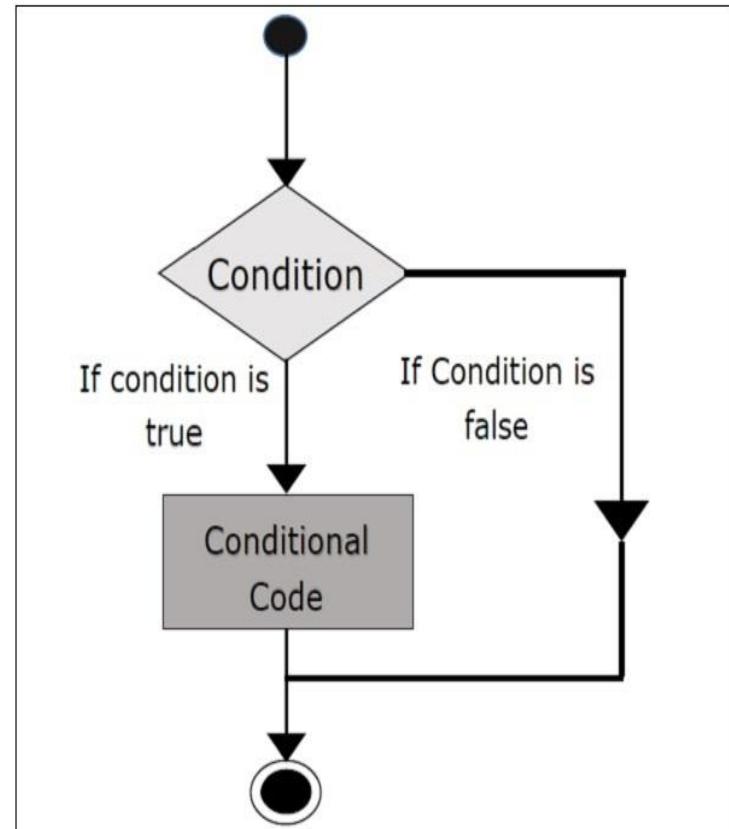
- Programmer specifies one or more conditions to be evaluated.
- If the condition is determined to be true, a statement is executed.
- If the condition is determined to be false, the other statements is to be executed.
- Typical decision making structure found in most of the programming language is:



# if Statement

- An *if statement* consists of a Boolean expression followed by one or more statements.
- *Syntax:*

```
if(boolean_expression)
{
    /* statement(s) will execute if the
    Boolean expression is true */
}
```



# if Statement

- If the Boolean expression evaluates to *true*, then the block of code inside the “if” statement will be executed.
- If the Boolean expression evaluates to *false*, then the first set of code after the end of “if” statement will be executed.
- C programming language assumes any *non-zero* and *non-null* values as *true* and if it is either *zero* or *null*, then it is assumed as *false* value.

# Example - 1

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a=10;
    if(a<20)
    {
        printf("a is less than 20\n");
    }
    printf("Value of a is: %d\n", a);
    getch();
}
```

# if...else Statement

- An *if* statement can be followed by an optional *else* statement, which executes when the Boolean expression is *false*.

- *Syntax:*

*if(boolean\_expression)*

{     */\* statement(s) will execute if the Boolean expression is true \*/*

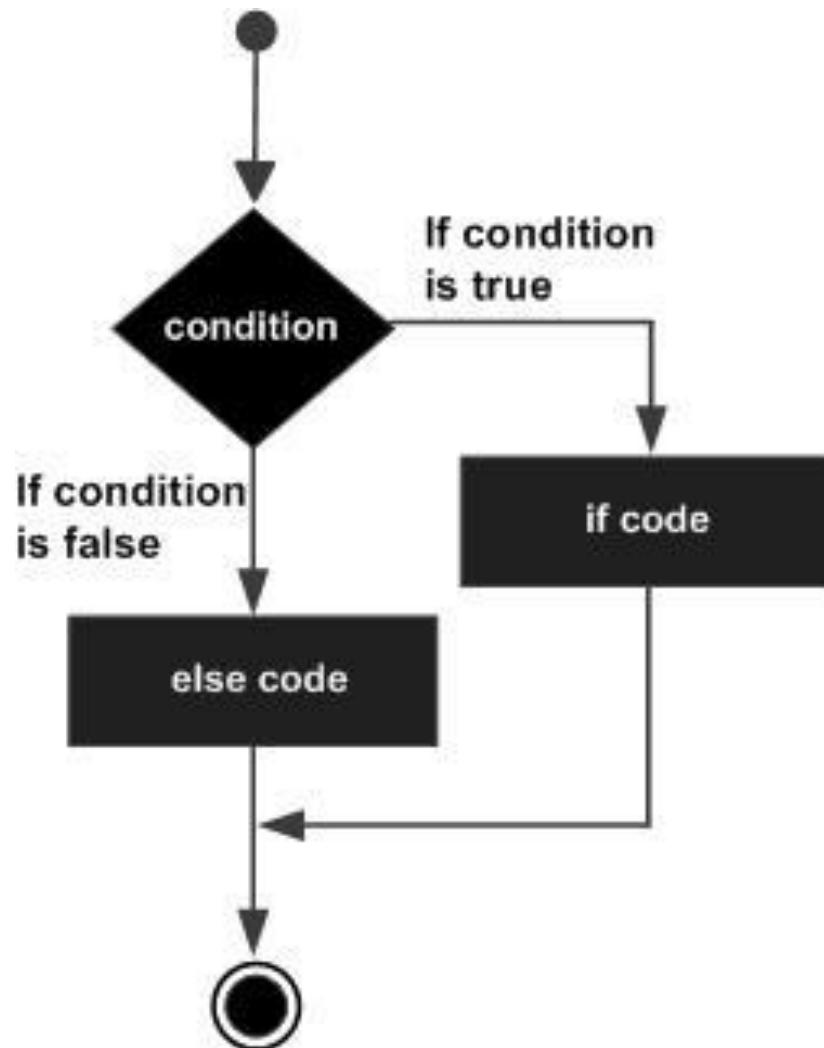
}

*else*

{     */\* statement(s) will execute if the Boolean expression is false \*/*

}

# *if...else Statement*



# **if...else Statement**

- *If the Boolean expression evaluates to **true**, then the **if block** will be executed, otherwise, the **else block** will be executed.*
- *C programming language assumes any **non-zero** and **non-null** values as **true** and if it is either **zero** or **null**, then it is assumed as **false** value.*

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a=100;
    if(a<20)
    {
        printf("a is less than 20\n");
    }
    else
    {
        printf("a is not less than 20\n");
    }
    printf("Value of a is %d", a);
    getch();
}
```

## Example - 2

# if...else if...else Statement

- An *if* statement can be followed by an optional *else if . . . else* statement, which is very useful to test various conditions.
- Syntax:

```
if(boolean_expression1)
{
    /* statement(s) will execute if the Boolean expression 1 is true */
}
else if(boolean_expression2)
{
    /* statement(s) will execute if the Boolean expression 2 is true */
}
else
{
    /* statement(s) will execute if both Boolean expressions are false */
}
```

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a=100;
    if(a==10)
    {
        printf("Value of a is 10\n");
    }
    else if(a==20)
    {
        printf("Value of a is 20\n");
    }
    else if(a==30)
    {
        printf("Value of a is 30\n");
    }
    else
    {
        printf("None of the values are matching\n");
    }
    printf("Exact value of a is %d\n",a);
    getch();
}
```

# Example - 3

# Nested if Statement

- *It is always legal in C programming to nest if-else statements, i.e. you can use one if or else if statement inside another if or else if statement(s).*
- *Syntax:*

```
if(boolean_expression1)
{
    /* statement(s) will execute if the Boolean expression 1 is true */
    if(boolean_expression2)
    {
        /* statement(s) will execute if the Boolean expression 2 is true */
    }
}
```

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a=100, b=200;
    if(a==100)
    {
        if(b==200)
        {
            printf("Value of a is 100 &b is 200\n");
        }
    }
    printf("Exact value of a is: %d\n",a);
    printf("Exact value of b is: %d\n",b);
    getch();
}
```

# Example - 4

# Loop Control Statements

- *Loop control statements change execution from its normal sequence.*
- *When execution leaves a scope, all automatic objects that were created in that scope are destroyed.*
- *C supports the following control statements:*
  - *break statement*
  - *continue statement*
  - *goto statement*

# Break Statement

- The *break* statement in C programming has the following two usages:
  - When a *break* statement is encountered inside a loop, the *loop is immediately terminated* and the program control resumes at the next statement following the loop.
  - It can be used to terminate a case in the *switch* statement.
- Syntax:  
*break;*

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a=10;
    while(a<20)
    {
        printf("Value of a: %d\n",a);
        a++;
        if(a>15)
        {
            break;
        }
    }
    getch();
}
```

# Continue Statement

- *The `continue` statement in C programming works somewhat like the `break` statement.*
- *Instead of forcing termination, it forces the next iteration of the loop to take place, skipping any code in between.*
- *For the `for` loop, `continue` statement causes the conditional test and increment portions of the loop to execute.*
- *For the `while` and `do...while` loops, `continue` statement causes the program control to pass to the conditional tests.*
- *Syntax:*  
`continue;`

# Example - 6

```
#include<stdio.h>

int main()
{
    int i=1;          // initializing a local variable
    // starting a loop from 1 to 10
    for(i=1;i<=10;i++)
    {
        if(i==5)
        {   // if value of i is equal to 5, it will continue the loop
            continue;
        }
        printf("%d \n",i);
    }  // end of for loop
    return 0;
}
```

# break vs. continue

<b><i>break</i></b>	<b><i>continue</i></b>
<p><i>The break statement is used to terminate the control from the switch...case structure as well in the loop.</i></p>	<p><i>The continue statement is used to bypass the execution of the further statements.</i></p>
<p><i>When the break statement is encountered, it terminates the execution of the entire loop or switch case.</i></p>	<p><i>When the continue statement is encountered, it bypasses single pass of the loop.</i></p>
<p><i>It doesn't bypass the current loop though it transfers the control out of the loop.</i></p>	<p><i>It is used to bypass current pass through a loop.</i></p>
<p><i>The loop terminates when a break is encountered.</i></p>	<p><i>The loop doesn't terminate when continue is encountered.</i></p>
<p><i>Syntax: break;</i></p>	<p><i>Syntax: continue;</i></p>

# goto Statement

- *To alter the normal sequence of program execution by unconditionally transferring control to some other part of the program.*
- *General expression:*
  - *goto label:*
    - *Here, label is an identifier used to label the target statement to which the control would be transferred.*
- *Generally the use of **goto** statement is avoided as it makes program illegible.*

# goto Statement

- *This statement is used in unique situations like:*
  - *Branching around statements or group of statements under certain conditions.*
  - *Jumping to the end of a loop under certain conditions, thus bypassing the remainder of the loop during current pass.*
  - *Jumping completely out of the loop under certain conditions, terminating the execution of a loop.*

# Example - 7

```
/* Program to calculate the sum and average of positive numbers. If the user enters a negative number, the sum and average are displayed */
#include <stdio.h>
int main()
{
    int maxInput = 100, i;
    double number, average, sum = 0.0;
    for (i = 1; i <= maxInput; ++i)
    {
        printf("%d. Enter a number: ", i);
        scanf("%lf", &number);
        if (number < 0.0)
        {
            goto jump; // go to jump if the user enters a negative number
        }
        sum += number;
    }
    jump:
    average = sum / (i - 1);
    printf("Sum = %.2f\n", sum);
    printf("Average = %.2f", average);
    return 0;
}
```

# Switch Statement

- *Switch statement allows a program to select one statement for execution of a set of alternatives.*
- *Only one of the possible statements will be executed, the remaining statements will be skipped.*
- *The multiple usage of if...else statement increases the complexity of the program, hard to read, and difficult to follow the program.*
- *Switch statement removes these disadvantages by using a simple and straight forward approach.*
- *Syntax:*

```
switch(expression)
{
    case caseConstant1:
        statement(s);
        break;
    case caseConstant2:
        statement(s);
        break;
    .
    .
    .
    default:
        statement;
}
```

# Switch Statement

- The following rules apply to a switch statement:
  - The **expression** used in a **switch** statement must have an **integral** or **enumerated** type.
  - You can have any number of **case** statements within a **switch**. Each **case** is followed by the **value** to be compared to and a colon.
  - The **caseConstant** for a **case** must be the same data type as the variable in the **switch**, and it must be a **constant** or a **literal**.
  - When the variable being switched on is equal to a **case**, the statements following that **case** will execute until a **break** statement is reached.
  - When a **break** statement is reached, the **switch** terminates, and the flow of control jumps to the next line following the **switch** statement.
  - Not every **case** needs to contain a **break**. If no **break** appears, the flow of control will fall through to subsequent **cases** until a **break** is reached.
  - A **switch** statement can have an optional **default** case, which must appear at the end of the **switch**. The **default** case can be used for performing a task when none of the **cases** is true. No **break** is needed in the **default** case.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int choice;
    LOOP:
    printf("Select 1 for file, 2 for Edit, 3 for Save\n");
    printf("1==>File\n2==>Edit\n3==>Save\n");
    scanf("%d",&choice);
    switch(choice)
    {
        case 1:
            printf("\nYou have chosen File Menu Item\n");
            break;
        case 2:
            printf("\nYou have chosen Edit Menu Item\n");
            break;
        case 3:
            printf("\nYou have chosen Save Menu Item\n");
            break;
        default:
            printf("\nINVALID OPTION CHOSED\n");
            goto LOOP;
    }
    getch();
}
```

# Example - 8

# Loop or Repeating Construct

- You may encounter situations, when a block of code needs to be executed several number of times.
- Programming languages provide various control structures that allow for more complicated execution paths.
- A loop statement allows us to execute a statement or group of statements multiple times.

# for Loop

- A *for loop* is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.
- The syntax for a *for loop* in C programming language is:  
`for(init ; condition ; increment)  
{  
 statement(s);  
}`
- The *init* step is executed first, and only once. This step allows you to declare and initialize any loop control variables.
- Next, the *condition* is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and the flow of control jumps to the next statement just after the *for loop*.
- After the body of the *for* loop executes, the flow of control jumps back up to the *increment* statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.
- The condition is now evaluated again. If it is true, the loop executes and the process repeats itself. After the condition becomes false, the *for loop* terminates.

# Example - 9

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a;
    for(a=10;a<=20;a++)
    {
        printf("Value of a is %d\n",a);
    }
    getch();
}
```

# while Loop

- A **while** loop in C programming repeatedly executes a target statement as long as a given condition is true.
- The syntax for a **while** loop in C programming language is:  
*while(condition)  
{     statement(s);  
}*
- **Statement(s)** may be a single statement or a block of statements.
- The **condition** may be any expression, and true is any nonzero value.
- The loop iterates while the condition is true.

# Example - 10

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a=10;
    while(a<20)
    {
        printf("Value of a is %d\n",a);
        a++;
    }
    getch();
}
```

# do...while Loop

- Unlike for and while loops, which test the loop condition at the top of the loop, the do...while loop in C programming checks its condition at the bottom of the loop.
- A do...while loop is similar to a while loop, except the fact that it is guaranteed to execute at least one time.
- The syntax for a **do...while** loop in C programming language is:

```
do
{
    statement(s);
} while(condition);
```
- Conditional expression appears at the end of the loop, so the statement(s) in the loop executes once before the condition is tested.

# Example - 11

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a=10;
    do
    {
        printf("Value of a is %d\n",a);
        a++;
    } while(a<20);
    getch();
}
```

# for vs. while

<i>for</i>	<i>while</i>
<i>It is definite loop</i>	<i>It may be definite and may not be definite</i>
<i>The loop expression is within a one block</i>	<i>The loop expression is scattered throughout the program</i>
<i>It contains three expressions</i>	<i>It contains only one expression</i>
<i>Syntax:</i> <i>for(init; condition; increment / decrement)</i> <i>{ statement(s); }</i>	<i>Syntax:</i> <i>while(test expression)</i> <i>{ body of loop }</i>

# while vs. do...while

while	do...while
<p><i>while loop is entry-controlled loop. i.e. test condition is evaluated first and body of loop is executed only if this test is true.</i></p>	<p><i>do...while loop is exit-controlled loop. i.e. the body of the loop is executed first without checking condition and at the end of body of loop, the condition is evaluated for repetition of next time.</i></p>
<p><i>The body of the loop may not be executed at all if the condition is not satisfied at the very first attempt.</i></p>	<p><i>The body of loop is always executed at least once.</i></p>
<p><i>Loop is not terminated with semicolon.</i></p>	<p><i>Loop is terminated with semicolon.</i></p>
<p><i>Syntax:</i> <i>while(test expression)</i> <i>{ body of loop }</i></p>	<p><i>Syntax:</i> <i>do {</i> <i>Body of loop</i> <i>} while(test expression);</i></p>

# Nested loops in C

- *C programming allows to use one loop inside another loop.*
- *The inner loop is said to be nested within the outer loop.*
  - Nested *for* loop
  - Nested *while* loop
  - Nested *do...while* loop

# Nested for loop

```
for(init; condition; increment)
{
    for(init; condition; increment)
    {
        statement(s);
    }
    statement(s);
}
```

# Nested for loop

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int i, j;
    for(i=1; i<=9; i++)
    {
        for(j=1; j<=i; j++)
            printf("%d", i);
        printf("\n");
    }
    getch();
}
```

# Nested while loop

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

# Nested while loop

```
#include <stdio.h>

int main()
{
    int a = 1, b = 1;
    while(a <= 5)
    {
        b = 1;
        while(b <= 5)
        {
            printf("%d ", b);
            b++;
        }
        printf("\n");
        a++;
    }
    return 0;
}
```

# Nested do...while loop

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

# Nested do while loop

```
#include <stdio.h>

int main()
{
    do
    {
        printf("I'm from outer do-while loop ");
        do
        {
            printf("\nI'm from inner do-while loop ");
        }
        while(1 > 10);
    }
    while(2 > 10);
    return 0;
}
```

# exit function

- The C library function **void exit(int status)** terminates the calling process immediately.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    printf("Start of the program . . . \n");
    printf("Exiting the program . . . \n");
    exit(0);
    printf("End of the program . . . \n");
    getch();
}
```

***END OF UNIT FIVE***

# Syllabus

- **UNIT 6: Arrays**
  - *Introduction to Array*
  - *Types of Array (Single, Dimensional, and Multidimensional)*
  - *Declaration and Memory Representation of Array*
  - *Initialization of Array*
  - *Character Array and Strings*
  - *Reading and Writing Strings*
  - *Null Character*
  - *String Library Functions (string length, string copy, string concatenation, string compare)*

# UNIT 6

**ARRAYS**

# Arrays

- An array is used to store a collection of data, but it is often more useful to think of an array as a *collection of variables of the same type*.
- Array is a data structure that store a number of data items as a single entity (object).
- The individual data items are called elements and all of them have same data types.
- Array is used when multiple data items have common characteristics.

# Arrays

- Instead of declaring, `number0`, `number1`, ..., and `number99`; we declare one array variable such as `numbers` and use `numbers[0]`, `numbers[1]`, ..., and `numbers[99]` to represent individual variables.
- A specific element in an array is accessed by an index.
- All arrays consist of contiguous memory locations.
- The lowest address corresponds to the first element and the highest address to the last element.



# Defining an Array

- An array is defined as following:

*<storage\_class> <type\_of\_array> <name\_of\_array> [<number of elements in array>];*

*int num[35]; // an integer array with 35 elements*

*char ch[5]; // an array of five characters*

- **storage\_class**: it may be *auto*, *register*, *static*, and *extern* (optional).
- **type\_of\_array**: it is the type of elements that an array stores. For example: ‘char’, ‘int’, etc.
- **name\_of\_array**: this is the name that is given to array. At least the name should be in context with what is being stored in the array.
- **[number of elements]**: this value in subscripts [] indicates the number of elements the array stores.

# Initializing an Array

- You can initialize an array in C either one by one or using a single statement as follows:

*double balance[5] = {1234.5, 6.7, 8.9, 1.0, 11.12};*

- The number of values between braces {} cannot be larger than the number of elements that we declare for the array between square brackets [].

- If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write

*double balance[] = {1234.5, 6.7, 8.9, 1.0, 11.12};*

- The above statement shows that the 5<sup>th</sup> element in the array has a value of 11.12 (i.e. *balance[4]=11.12*).

# Initializing an Array

- All arrays have 0 as the index of their first element which is also called the base index and the last index of an array will be total size of the array minus 1.
- Shown below is the pictorial representation of the array we discussed:  
 $double balance[5] = \{1234.5, 6.7, 8.9, 1.0, 11.12\};$

	0	1	2	3	4
balance	1234.5	6.7	8.9	1.0	11.12

# Accessing an Array Elements

- *An element is accessed by indexing the array name.*
- *This is done by placing the index of the element within a square brackets after the name of the array. For example:*  
*double num = balance[4];*
- *This above statement will take the 5<sup>th</sup> element from an array and assign the value to **num** variable.*

# Example - 1

```
#include<stdio.h>
#include<conio.h>
void main()
{
    double balance[5] = {1234.5, 6.7, 8.9, 1.0, 11.12};
    double num=balance[4];
    printf("%d",num);
    getch();
}
```

# Example – 2 (Sorting)

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int num[30],i,j,n,temp;
    printf("How many numbers are there?\n");
    scanf("%d",&n);
    printf("Enter %d numbers:\n",n);
    for(i=0;i<n;i++)
    {
        scanf("%d",&num[i]);
    }
    for(i=0;i<n-1;++)
    {
        for(j=i+1;j<n;j++)
        {
            if(num[i]>num[j])
            {
                temp=num[i];
                num[i]=num[j];
                num[j]=temp;
            }
        }
    }
    printf("The numbers are: ");
    for(i=0;i<n;i++)
    {
        printf("%d\t",num[i]);
    }
    getch();
}
```

# Characteristics of an Array

- *The declaration of `int a[5]` is nothing but creation of 5 variables of integer type in the memory.*
- *All the elements of an array share the same name, distinguished from one another by element number or array index.*
- *Any element can be modified separately without disturbing another element.*
- *Element basis operation should be carried out rather than taking as a whole.*

# Types of Array: Single Dimensional

- *A list of items can be given one variable name using only one subscript (or dimension or index) and such a variable is called a single-subscripted variable or a one-dimensional array.*
- *The value of the single subscript or index from 0 to  $n - 1$  refers to the individual array elements; where  $n$  is the size of the array.*
- *For example: The declaration `int a[5];` is a 1-D array of integer data type with 5 elements:  $a[0], a[1], a[2], a[3], \& a[4]$ .*

# Types of Array: Multi Dimensional

- Those arrays having more than one dimension are called multi dimensional arrays.
- Multi-dimensional arrays are defined in the same way as one dimensional arrays, except that a separate pair of square brackets is required for each subscript or dimension or index.
- Thus a 2-D array requires two pairs of square brackets (also called as matrix); a 3-D array requires three pairs of square brackets; and so on.

# Types of Array: Multi Dimensional

- Syntax for defining multidimensional array is:  
*storage\_class data\_type array\_name [dim1] [dim2] ... [dimN];*
- Here,  $dim1, dim2, \dots, dimN$  are positive valued integer expressions that indicate the number of array elements associated with each subscript.
- Thus, total number of elements is equal to  $dim1 * dim2 * \dots * dimN$
- For example: *int survey[3][5][12]*

# 2 Dimensional Array

- $m * n$  – Two dimensional array can be thought as tables of values having  $m$  rows and  $n$  columns.
- For example:  $int x[3][3]$  can be shown as follows:

	Col 1	Col 2	Col 3
Row 1	$x[0][0]$	$x[0][1]$	$x[0][2]$
Row 2	$x[1][0]$	$x[1][1]$	$x[1][2]$
Row 3	$x[2][0]$	$x[2][1]$	$x[2][2]$

# Declaration of 2 Dimensional Array

- Like one dimensional array, two dimensional arrays must also be declared before using it.

- Syntax:

*storage\_class data\_type array\_name [row\_size] [col\_size];*

- For example:

```
int matrix[2][3]; //matrix having 2 rows and 3 columns  
float m[10][20];  
char students[10][15];
```

# Initialization of 2-D Array

- For example:

*int matrix[2][3] = {{2,4,6},{8,10,12}};*

*is equivalent to*

*matrix[0][0] = 2;*

*matrix[1][0] = 8;*

*matrix[0][1] = 4;*

*matrix[1][1] = 10;*

*matrix[0][2] = 6;*

*matrix[1][2] = 12;*

# Accessing of 2-D Array Elements

- *In 2-D array, the first dimension specifies number of rows and second specifies columns.*
- *A row is 1-D array. 2-D array contains multiple rows (i.e. 1-D arrays).*
- *2-D array is traversed row by row (i.e. every column elements in first row are traversed first and then column elements of second row are traversed and so on).*
- *Nested loop is used to traverse the 2-D array.*

# Accessing of 2-D Array Elements

- Let us consider a 2-D array **marks** of size  $4 * 3$  (i.e. matrix having 4 rows and 3 columns).

35	10	11
34	90	76
13	8	5
76	4	1

- The array is traversed in the following order:  
 $35 \rightarrow 10 \rightarrow 11 \rightarrow 34 \rightarrow 90 \rightarrow 76 \rightarrow 13 \rightarrow 8 \rightarrow 5 \rightarrow 76 \rightarrow 4 \rightarrow 1$
- i.e.  $marks[0][0] \rightarrow marks[0][1] \rightarrow marks[0][2] \rightarrow marks[1][0] \rightarrow marks[1][1] \rightarrow marks[1][2] \rightarrow marks[2][0] \rightarrow marks[2][1] \rightarrow marks[2][2] \rightarrow marks[3][0] \rightarrow marks[3][1] \rightarrow marks[3][2]$

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int matrix[2][3],i,j;
    for(i=0;i<2;i++)
    {
        for(j=0;j<3;j++)
        {
            printf("Enter Matrix [%d][%d]:",i,j);
            scanf("%d",&matrix[i][j]);
        }
    }
    printf("The entered matrix is:\n");
    for(i=0;i<2;i++)
    {
        for(j=0;j<3;j++)
        {
            printf("%d\t",matrix[i][j]);
        }
        printf("\n");
    }
    getch();
}
```

# Example – 3

# *Addition & Subtraction of Matrix - WAP*



# Bubble Sort

```
for(i=0;i<n-1;i++)  
{  
    for(j=0;j<n-1-i;j++)  
    {  
        if(num[j]>num[j+1])  
        {  
            temp=num[j];  
            num[j]=num[j+1];  
            num[j+1]=temp;  
        }  
    }  
}
```

# Selection Sort

```
for(i = 0; i < n - 1; i++)  
{  
    min=i;  
    for(j = i + 1; j < n; j++)  
    {  
        if(a[min] > a[j])  
            min=j;  
    }  
    if(min != i)  
    {  
        temp=a[i];  
        a[i]=a[min];  
        a[min]=temp;  
    }  
}
```

# Sequential (Linear) Search

```
#include <stdio.h>

int main()
{
    int array[100], element, i, n, found=0;
    printf("Enter number of elements in
array\n");
    scanf("%d", &n);
    printf("Enter %d integer(s)\n", n);
    for (i = 0; i < n; i++)
        scanf("%d", &array[i]);
    printf("Enter a number to search\n");
    scanf("%d", &element);
    for(i=0; i<n; i++)
    {
        if(array[i]==element)
        {
            printf("Found at location %d", i);
            found=1;
            break;
        }
    }
    if(!found)
        printf("Element not found");
    return 0;
}
```

# Arrays and Strings

- In C programming, an array of character are called strings.
- A string is terminated by null character \0.  
For example: “C Strings”  
Here, “C Strings” is a string. When compiler encounters strings, it appends null character at the end of string.



- Strings are actually one-dimensional array of characters terminated by a null character.
- If you follow the rule of array initialization, then you can write the above statement as follow:

`char greeting[] = "Hello";`

# Arrays and Strings

*char greeting[] = "Hello";*

- Following is the memory presentation of the above defined string in C:

Index	0	1	2	3	4	5
Variable	H	e	l	l	o	\0
Address	0x98765	0x98764	0x98763	0x98762	0x98761	0x98760

- Actually, we do not place the null character at the end of a string constant but the C compiler automatically places the \0 at the end of the string when it initializes the array.

# Initialization of Strings

- In C, string can be initialized in different number of ways.

*char c[ ] = "hello";*

*OR*

*char c[ ] = {'h', 'e', 'l', 'l', 'o', '\0'};*

c[0]	c[1]	c[2]	c[3]	c[4]	c[5]
h	e	l	l	o	\0

# Declaration of Strings

- *Strings are declared in C in similar manner as arrays.*
- *Only difference is that, strings are of char type.*  
 $\text{char } c[5];$

$c[0]$	$c[1]$	$c[2]$	$c[3]$	$c[4]$
h	e	l	l	o

- *Strings can also be declared using pointer.*  
 $\text{char } * p;$

## Example – 4

```
#include<stdio.h>
#include<conio.h>
void main()
{
    char greeting[]={'H','e','l','l','o','\0'};
    printf("Greeting message is %s\n",greeting);
    char greetings[]="Hello";
    printf("Greeting message is %s\n",greetings);
    getch();
}
```

## Example – 5

```
#include<stdio.h>
#include<conio.h>
void main()
{
    char name[20];
    printf("Enter your name:");
    scanf("%s",&name);
    printf("Your name is %s",name);
    getch();
}
```

# Example – 6

```
#include<stdio.h>
#include<conio.h>
void main()
{
    char name[30],ch;
    int i=0;
    printf("Enter your name:");
    while(ch!='\n')
    {
        ch=getchar();
        name[i]=ch;
        i++;
    }
    name[i]='\0';
    printf("Your name is %s",name);
    getch();
}
```

# **gets() and puts() function**

- The *gets()* function is used to read a string of text, containing whitespaces, until a newline character is encountered.
  - Syntax: *gets(variable\_name);*
- It takes a string from user and stores in a string variable *variable\_name*
- The *puts()* function is used to display the string onto the screen.
  - Syntax: *puts(variable\_name);*

## Example - 7

```
#include<stdio.h>
#include<conio.h>
void main()
{
    char name[30];
    puts("Enter your name: ");
    gets(name);
    printf("Your name is ");
    puts(name);
    getch();
}
```

# String Library Functions

- The header file `<string.h>` contains some string manipulation functions.

String Functions	Its use
strlen	Returns number of characters in string
strlwr	Converts all the characters in the string into lower case characters
strcat	Adds one string at the end of another string
strcpy	Copies a string into another
strcmp	Compares two strings and returns zero if both are equal
strdup	Duplicates a string
strstr	Finds the first occurrence of given string in another string
strchr	Finds the first occurrence of given character in a string
strset	Sets all the characters of string to given character or symbol
strrev	Reverse a string

```
#include<stdio.h>
#include<conio.h>
void main()
{
    char str1[20]={"Hello "};
    char str2[20]={"world!"};
    char str3[20];
    int len;
    strcpy(str3,str1);
    printf("Displaying str3: %s\n",str3);
    /*
    strcpy(str3,strcat(str1,str2));
    printf("Displaying str3: %s\n",str3);
    */
    len=strlen(str3);
    printf("Length of str3 is %d",len);
    /*
    getch();
    */
}
```

```
#include <stdio.h>
#include <conio.h>
#define M 3
#define N 3
void main()
{
    int matrix[M][N], transpose[N][M], i, j;
    printf("\nEnter the elements of matrix:\n");
    for(i=0; i<M; i++)
    {
        for(j=0; j<N; j++)
        {
            scanf("%d", &matrix[i][j]);
        }
    }
    printf("\nThe matrix to be transposed is:\n");
    for(i=0; i<M; i++)
    {
        for(j=0; j<N; j++)
        {
            printf("%d\t", matrix[i][j]);
        }
        printf("\n");
    }
}
```

```
/*finding transpose matrix*/
for(i=0; i<M; i++)
{
    for(j=0; j<N; j++)
    {
        transpose[j][i] = matrix[i][j];
    }
}
printf("\nThe transpose matrix is:\n");
for(i=0; i<M; i++)
{
    for(j=0; j<N; j++)
    {
        printf("%d\t", transpose[i][j]);
    }
    printf("\n");
}
getch();
}
```

# Multiplication of a Matrix

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
void main()
{
    int matrix1[10][10],matrix2[10][10],i,j,k,product[10][10],M,N,P,Q;
    int row_mul_col=0;
    printf("\nEnter order of first matrix (less than 10*10):\n");
    scanf("%d %d",&M,&N);
    printf("\nEnter order of second matrix (less than 10*10):\n");
    scanf("%d %d",&P,&Q);
    if(N!=P)
    {
        printf("\nThe matrices are unsuitable for multiplication.\n");
        getch();
        exit(0);
    }
}
```

# Multiplication of a Matrix

```
printf("\nEnter the elements of first matrix:\n");
for(i=0;i<M;i++)
{
    for(j=0;j<N;j++)
    {
        scanf("%d",&matrix1[i][j]);
    }
}
printf("\nThe first matrix is:\n");
for(i=0;i<M;i++)
{
    for(j=0;j<N;j++)
    {
        printf("%d\t",matrix1[i][j]);
    }
}
printf("\n");
```

# Multiplication of a Matrix

```
printf("\nEnter the elements of second matrix:\n");
for(i=0;i<P;i++)
{
    for(j=0;j<Q;j++)
    {
        scanf("%d",&matrix2[i][j]);
    }
}
printf("\nThe second matrix is:\n");
for(i=0;i<P;i++)
{
    for(j=0;j<Q;j++)
    {
        printf("%d\t",matrix2[i][j]);
    }
}
printf("\n");
```

# Multiplication of a Matrix

```
/*multiply two matrices*/  
for(i=0;i<M;i++) //first row  
{  
    for(j=0;j<Q;j++) //second col  
    {  
        for(k=0;k<N;k++) //first col  
        {  
            row_mul_col += matrix1[i][k]*matrix2[k][j];  
        }  
        product[i][j]=row_mul_col;  
        row_mul_col=0;  
    }  
}
```

# Multiplication of a Matrix

```
printf("\nThe matrix after multiplication is:\n");
for(i=0;i<M;i++)
{
    for(j=0;j<Q;j++)
    {
        printf("%d\t",product[i][j]);
    }
    printf("\n");
}
getch();
}
```

***END OF UNIT SIX***

# Syllabus

- **UNIT 7: Functions**

- *Library Functions*
- *User defined functions*
- *Function prototype, Function call, and Function Definition*
- *Nested and Recursive Function*
- *Function Arguments and Return Types*
- *Passing Arrays to Function*
- *Passing Strings to Function*
- *Passing Arguments by Value, Passing Arguments by Address*
- *Scope visibility and lifetime of a variable, Local and Global Variable*

# UNIT 7

## **FUNCTIONS**

# Function

- *A function is a group of statements that together perform a task.*
- *Every C program has at least one function, which is `main()`, and all the most trivial programs can define additional functions.*
- *You can divide up your code into separate functions.*
- *How you divide up your code among different function is up to you – but logically the division should be such that each function performs a specific task.*

# Function

- Suppose a program where a set of operations has to be repeated often, though not continuously.
- *Loops* seems like a better option.
- Instead of inserting the program statements for these operations at so many places, a separate program segment is written and compiled it separately.
- As many times as it is needed, the segment program is called.
- The separate program segment is called a **function**.
- The C functions can be classified into two categories:
  - User Defined Function
  - Library Function

# Advantages of Function

- *Manageability*
  - *Easier to write and keep track of.*
  - *Easier to understand and maintain.*
- *Code Reusability*
  - *Can be used multiple times.*
- *Non-redundant programming*
  - *Same function can be called when needed.*
- *Logical Clarity*
  - *Reduced number of code in main function.*
- *Easy to divide work*
  - *Large work can be divided by writing different functions.*

# Library Functions

- *Also known as built-in function.*
- *These are the functions which are already written, compiled and placed in C library and they are not required to be written by a programmer.*
- *The functions name, its return type, their argument number and types have been already defined.*
- *We can use these functions as required.*
- *For example: `printf()`, `scanf()`, `sqrt()`, `getch()`, etc.*

# User - *Defined Functions*

- *These are functions which are defined by user at the time of writing a program.*
- *The user has choice to choose its name, return type, arguments and their types.*
- *The job of each user-defined functions is as defined by the user.*
- *A complex C problems can be divided into a number of user-defined functions.*
- *The function **main()** is an user-defined function.*

# *main() Function*

- *The function **main()** is an user defined function except that the name of function is defined or fixed by the language.*
- *The return type, argument and body of the function are defined by the programmer as required.*
- *The function is executed first, when the program starts execution.*

# main() Function

- The function **main()** has two arguments that traditionally are called **argc** and **argv** and return a signed integer.

Argument	Name	Description
<i>argc</i>	<i>Argument count</i>	<i>Length of the argument vector</i>
<i>argv</i>	<i>Argument vector</i>	<i>Array of character pointers</i>

# *Components associated with Function*

- *Function Definition*
- *Function Declaration (or Prototype)*
- *Return Statement*
- *Accessing/Calling a function*

# Function Definition

- *The collection of program statements that describes the specific task to be done by the function is called function definition.*
- *It consists of **function header**, which defines functions name, its **return type** and its **argument** list and a **function body**, which is a block of code enclosed in parenthesis.*

- *Syntax:*

```
return_type function_name(data_type variable1, data_type variable2, ...)  
{      ...  
      statements;  
      ...  
}
```

# Function Definition Syntax

- **Return Type:**

- *A function may return a value.*
- *The **return\_type** is the data type of the value the function returns.*
- *Some functions perform the desired operations without returning a value.*
- *In this case, the **return\_type** is the keyword **void**.*

# Function Definition Syntax

- **Function Name:**

- *This is the actual name of the function.*
- *The function name and the parameter list together constitute the function signature.*

- **Function Body:**

- *The function body contains a collection of statements that define what the function does.*

# Function Definition Syntax

- **Parameters**
  - *Parameters include data\_type variable1, data\_type variable2, ...*
  - *When a function is invoked, you pass a value to the parameter.*
  - *This value is referred to as actual parameter or argument.*
  - *The parameter list refers to the type, order, and number of the parameters of a function.*
  - *Parameters are optional; i.e. a function may contain no parameters.*

# Syntax – Example

```
int add(int a, int b)          // function header
{
    int sum;                  // function body (statements)
    sum=a+b;                  // function body (statements)
    return sum;                // returns value of sum to whoever called
}
```

\*\*\*\*\*

```
float areaOfCircle(float radius)      // function header
{
    return 3.1428*radius*radius;        // returns radius
}
```

# Function Declaration (or Prototype)

- *The function declaration or prototype is model or blueprint of the function.*
- *If functions are used before they are defined, then function declaration or prototype is necessary which provides the following information to the compiler.*
  - *The name of the function.*
  - *The type of the value returned by the function.*
  - *The number and the type of arguments that must be supplied when calling the function.*
- *A function declaration tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.*

# Function Declaration (or Prototype)

- A function declaration has the following parts:

***return\_type function\_name(parameter list);***

- For the above defined function `add()` & `areaOfCircle()`, the function declaration is as follow:

***int add(int num1, int num2);***

***float areaOfCircle(float radius);***

# Return Statement

- *It is the statement that is executed just before the function completes its job and control is transferred back to the calling function.*
- *The job of return statement is to hand over some value given by function body to the point where the call was made.*
- *Two purposes of return statement:*
  - *Immediately transfer the control back to the calling program*
  - *Returns the value to the calling function*

# Return Statement

- *Syntax:*

***return (expression);***

- *A function may or may not return a value. If a function doesn't return a value the return type in the function definition and declaration is specified as void.*

# Accessing/Calling a Function

- *A function can be called or accessed by specifying its name, followed by a list of arguments enclosed in parenthesis and separated by commas/*
- *For example:*
  - *A function **add()** with two arguments is called by **add(a,b)** to add two numbers.*
  - *If function call doesn't require any arguments any empty pair of parenthesis must follow the name of function.*

# Accessing/Calling a Function

- *General Form:*

- *If function has parameters but it doesn't return value.*

*function\_name(variable1,variable2,...);*

- *If function has no arguments and it doesn't return value.*

*function\_name();*

- *If function has no arguments and it does return value.*

*variable\_name=function\_name();*

```
#include<stdio.h>
#include<conio.h>
int add(int a, int b)
{
    int sum;
    sum=a+b;
    return sum;
}
float areaOfCircle(float radius)
{
    return 3.1428*radius*radius;
}
void main()
{
    int a,b,mySum;
    float myArea;
    printf("Enter two numbers: \n");
    scanf("%d%d", &a, &b);
    mySum=add(a,b);
    printf("The sum is %d\n",mySum);
    myArea=areaOfCircle(mySum);
    printf("Area of circle is %f\n",myArea);
    getch();
}
```

# Example - 1

```
#include<stdio.h>
#include<conio.h>
int add(int,int);
float areaOfCircle(float);
void main()
{
    int a,b,mySum;
    float myArea;
    printf("Enter two numbers:\n");
    scanf("%d%d",&a,&b);
    mySum=add(a,b);
    printf("The sum is %d\n",mySum);
    myArea=areaOfCircle(mySum);
    printf("Area of circle is %f\n",myArea);
    getch();
}
int add(int a, int b)
{
    int sum;
    sum=a+b;
    return sum;
}
float areaOfCircle(float radius)
{
    return 3.1428*radius*radius;
}
```

## Example – 2

# Function Arguments & Return Type

- *There are three category of functions according to the return value and arguments:*
  - *Functions with no arguments and no return value*
  - *Functions with arguments and no return value*
  - *Function with arguments and return value*

# Function Arguments & Return Type

- **Functions with no arguments and no return value:**
  - When function has no arguments, it does not receive any data from the calling function.
  - Similarly when it doesn't return a value, the calling function does not receive any data from the called function.
  - Thus, in such type of functions there is no data transfer between the calling function and the called function.
  - Keyword “**void**” means the function doesn't return any value.
  - There is no arguments within parenthesis which implies function has no argument and it doesn't receive any data from the called function.

# Function Arguments & Return Type

- *Functions with no arguments and no return value:*

```
#include<stdio.h>
#include<conio.h>
void add()
{
    int a,b,sum;
    printf("Enter any two numbers: \n");
    scanf("%d%d",&a,&b);
    sum=a+b;
    printf("\n The sum is %d \n",sum);
}
void main()
{
    add();
    getch();
}
```

# Function Arguments & Return Type

- *Functions with arguments but no return value:*
  - *We pass arguments while calling a function but nothing is returned to the calling function.*

```
#include<stdio.h>
#include<conio.h>
void add(int a,int b)
{
    int sum=0;
    sum=a+b;
    printf("\nThe sum is %d\n",sum);
}
void main()
{
    int a,b;
    printf("Enter two numbers:\n");
    scanf("%d%d",&a,&b);
    add(a,b);
    getch();
}
```

# Function Arguments & Return Type

- *Functions with arguments and return value:*

- *We pass arguments and we expect a return value*

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

# Passing Arrays to Function

- *It is possible to pass the value of an array element and even an entire array as an argument to a function.*
- *To pass an entire array to a function, the array name must appear by itself, without brackets or subscripts, as an actual argument in function call statement.*
- *When declaring a one-dimensional array as a formal argument, the array name is written with a pair of empty square brackets.*
- *The size of the array is not specified within the formal argument declaration.*

# Passing Arrays to Function

- *Syntax for function call passing array as argument:*  
*function\_name(array\_name)*
- *Syntax for function prototype which accepts array:*  
*return\_type function\_name(data\_type array\_name[]);*  
*Or*  
*return\_type function\_name(data\_type \*pointer\_variable);*
- *When array is passed to a function, the values of the array elements are not passed to the function rather the array name is interpreted as the address of the first array element.*
- *The address is assigned to the corresponding formal argument when the function is called.*
- *The formal argument therefore becomes a pointer to the first array element.*

```
#include<stdio.h>
#include<conio.h>
void display(int n)
{
    printf("%d\t",n);
}
void main()
{
    int i, num[5]={12,23,34,45,56};
    printf("\n The elements of an array are: ");
    for(i=0;i<5;i++)
    {
        display(num[i]);
    }
    getch();
}
```

# Passing Entire Array to Function

```
// Program to calculate the sum of array elements by passing to a function
#include <stdio.h>
float calculateSum(float num[]);
int main()
{
    float result, num[] = {23.4, 55, 22.6, 3, 40.5, 18};
    // num array is passed to calculateSum()
    result = calculateSum(num);
    printf("Result = %.2f", result);
    return 0;
}
float calculateSum(float num[])
{
    float sum = 0.0;
    for (int i = 0; i < 6; ++i)
    {
        sum += num[i];
    }
    return sum;
}
```

# Passing Strings to Function

```
#include<stdio.h>
#include<conio.h>
void display(char ch[])
{
    printf("String output: ");
    puts(ch);
}
void main()
{
    char c[50];
    printf("Enter String:\n");
    gets(c);
    display(c);
    getch();
}
```

# Passing Arguments by Value

- When values of actual arguments are passed to the function as arguments, it is known as passing by value.
- Here, the value of each actual argument is copied into corresponding formal argument of the function definition.
- **Note:** The contents of the actual arguments in the calling function are not changed, even if they are changed in the called function.

# Passing Arguments by Value

```
#include<stdio.h>
#include<conio.h>
void swap(int a,int b)
{
    int t;
    t=a;
    a=b;
    b=t;
    printf("\nValues within swap: a=%d & b=%d",a,b);
}
void main()
{
    int x=2,y=3;
    printf("Before swap function call: x=%d & y=%d",x,y);
    swap(x,y);
    printf("\nAfter swap function call: x=%d & y=%d",x,y);
    getch();
}
```

# Passing Arguments by Address

- When addresses of actual arguments are passed to the function as arguments (instead of values of actual arguments), it is known as *passing by address*.
- Here, the address of each actual argument is copied into corresponding formal argument of the function definition.
- In this case, the formal arguments must be of type pointers.
- **Note:** The values contained in addresses of the actual arguments in the calling function are changed, if they are changed in the called function.

# Passing Arguments by Address

```
#include<stdio.h>
#include<conio.h>
void swap(int *x, int *y)
{
    int temp;
    temp=*x;
    *x=*y;
    *y=temp;
    printf("Values within swap: x=%d &y=%d\n",*x, *y);
}
void main()
{
    int a=23, b=34;
    printf("Before swap function call: a=%d &b=%d\n",a,b);
    swap(&a,&b);
    printf("After swap function call: a=%d &b=%d\n",a,b);
    getch();
}
```

# Recursive Function

- *Within a function body, if the function calls itself, the mechanism is known as “**Recursion**” and the function is known as “**Recursive Function**”.*
- *To solve a problem using recursive method, two conditions must be satisfied:*
  - *Problem could be written or defined in term of its previous result.*
  - *Problem statement must include a stopping condition.*

# WAP to find factorial

```
#include<stdio.h>
#include<conio.h>
int fact(int n)
{
    int res=1, i;
    for(i=n;i>=1;i--)
    {
        res=res*i;
    }
    return res;
}
void main()
{
    int n,factorial;
    int fact(int);
    printf("Enter any number:\n");
    scanf("%d",&n);
    factorial=fact(n);
    printf("Factorial is %d\n",factorial);
    getch();
}
```

# WAP to find factorial – recursion

```
#include<stdio.h>
#include<conio.h>
int fact(int n)
{
    int res;
    if(n==1)
        return(1);
    else
        res=n*fact(n-1);
    return res;
}
void main()
{
    int n,factorial;
    int fact(int);
    printf("Enter any number:\n");
    scanf("%d",&n);
    factorial=fact(n);
    printf("Factorial is %d\n",factorial);
    getch();
}
```

Recursion	Iteration
<i>A function is called from the definition of the same function to do repeated task.</i>	<i>Loop is used to do repeated task.</i>
<i>Recursion is a top-down approach to problem solving; it divides the problem into pieces.</i>	<i>Iterations is like bottom-up approach; it begins with what is known and from this it constructs the solution step by step.</i>
<i>In recursion, a function calls to itself until some condition will be satisfied.</i>	<i>In iteration, a function does not call to itself.</i>
<i>Problem could be written or defined in term of its previous result to solve a problem using recursion.</i>	<i>It is not necessary to define a problem in term of its previous result to solve using iteration.</i>
<i>All problems cannot be solved using recursion.</i>	<i>All problems can be solved using iteration.</i>

# Scope visibility and lifetime of a variable

- Variables in C are categorized into four different storage classes according to the **scope** and **lifetime** of variables.
  - Local (Automatic) variables
  - Global (External) variables
  - Static variables
  - Register variables
- The **scope** of variable determines over what part(s) of the program a variable is actually available for use (**active**).
- **Lifetime** refers to the period of time during which a variable retains a given value during execution of a program (**alive**).

# Variables

- *Variables can also be broadly categorized, depending on the place of their declaration, as **internal (local)** or **external (global)**.*
- *Local variables are those which are declared within a particular function.*
- *Global variables are declared outside of any function.*

# Local Variables

- Also known as *Automatic variables* or *Internal variables*.
- They are always declared inside a function or block in which they are to be used.
- They are **created** when the function is called and **destroyed** automatically when the function is exited, hence the name is *automatic*.
- The keyword **auto** is used for storage class specification although it is *optional*.
- **Initial value**: garbage.
- **Scope**: local to the block where the variable is defined.
- **Lifetime**: till the control remains within the block where variable is defined.

# Global Variables

- *Also known as External variables.*
- *These variables are both alive and active throughout the entire program.*
- *Unlike local variables, global variables can be accessed by any function in the program.*
- *Initial value:* zero
- *Scope:* global (i.e. throughout the program)
- *Lifetime:* throughout program execution

<i>Local Variable</i>	<i>Global Variable</i>
<pre>#include&lt;stdio.h&gt; #include&lt;conio.h&gt; void main() {     int a;     printf("%d",a);     getch(); }</pre>	<pre>#include&lt;stdio.h&gt; #include&lt;conio.h&gt; int a; void main() {     printf("%d",a);     getch(); }</pre>

# Local & Global Variables – Example

```
#include<stdio.h>
#include<conio.h>
int a=100;
void function()
{
    printf("%d\n",a);
    a=200;
    printf("%d\n",a);
}
void main()
{
    printf("%d\n",a);
    function();
    getch();
}
```

# Static Variables

- As the name suggests, the value of static variables persists till the end of the program.
- A variable is declared static using the keyword **static**. i.e. **static int x;**
- A static variable may be either an internal type or an external type, depending on the place of declaration.
- **Initial value**: zero (initialized only once)
- **Scope**: local (within the function) or global (within the program)
- **Lifetime**: throughout the program for local as well as global static variables.

```
#include<stdio.h>
#include<conio.h>
increment()
{
    int i=1;
    printf("%d\t",i);
    i++;
}
void main()
{
    increment();
    increment();
    increment();
    increment();
    increment();
    getch();
}
```

```
#include<stdio.h>
#include<conio.h>
increment()
{
    static int i=1;
    printf("%d\t",i);
    i++;
}
void main()
{
    increment();
    increment();
    increment();
    increment();
    increment();
    getch();
}
```

# Register Variables

- Normally, variables are stored in memory.
- However, we can instruct the compiler that a variable should be kept in one of the CPU's registers, instead of keeping in the memory.
- **Use:** A register access is much faster than a memory access. So keeping frequently accessed variables (e.g. loop control variables) in the register will help faster execution of the program.
- A register variable is declared using **register** keyword: i.e. **register int x;**
- Register variables are always declared inside a function or block.
- They are allocated space upon entry to a function; and the storage is freed when the function is exited.
- **Initial value:** garbage
- **Scope:** local (within the function)
- **Lifetime:** until the end of function

# Register Variables

```
#include<stdio.h>
#include<conio.h>
void main()
{
    register int i;
    for(i=0;i<100;i++)
    {
        printf("%d\t",i);
    }
    getch();
}
```

# Local, Global, & Static Variables

Local Variables	Global Variables	Static Variables
<p><i>The variables are declared within function or blocks. The scope is only within the function or block in which they are defined.</i></p>	<p><i>Global variables are defined outside the functions so that their scope is throughout the program.</i></p>	<p><i>Static variables are special case of local variables. Thus, static variables are also defined inside the functions or blocks.</i></p>
<p><i>The initial value is unpredictable or garbage value.</i></p>	<p><i>The initial value is zero.</i></p>	<p><i>The initial value is zero.</i></p>
<p><i>The life time is till the control remains within the block or function in which the variable is defined. The variable is destroyed when function returns to the calling function or block ends.</i></p>	<p><i>The life time is till programs execution doesn't come to an end. i.e. variables are created when program starts and destroyed when program ends.</i></p>	<p><i>Its value persists between different function calls. Thus life time is same as global variables.</i></p>
<p><i>The keyword <b>auto</b> is used.</i></p>	<p><i>The keyword <b>extern</b> is used.</i></p>	<p><i>The keyword <b>static</b> is used.</i></p>

***END OF UNIT SEVEN***

# UNIT 8

**STRUCTURE AND UNION**

# Syllabus

- **UNIT 8: Structure and Union**
  - *Introduction*
  - *Array of structure*
  - *Passing structure to function*
  - *Passing array of structure to function*
  - *Structure within structure (nested structure)*
  - *Union*
  - *Pointer to structure*

# Structure

- *A structure is a collection of variables under a single name.*
- *Arrays allow to define type of variables that can hold several data items of the same kind.*
- *Similarly structure is another user defined data type available in C that allows to combine data items of different kinds.*
- *A structure is a convenient way of grouping several pieces of related information together.*
- *A structure is a convenient tool for handling a group of logically related data items.*

# Structure

- For example: name, roll, fee, marks, address, gender, and phone are related information of a student.
- To store information about a student, we would require to store the name of the student which is array of characters, roll of student which is integer type of data and so on.
- These attributes of students can be grouped into a single entity, **student**. Here, student is known as structure which organizes different data types in a more meaningful way.

# Defining a Structure

- *Syntax:*

```
struct structure_name {  
    data_type member_variable1;  
    data_type member_variable2;  
    data_type member_variable3;  
    .....  
    .....  
    data_type member_variableN;  
};
```

- Once **structure\_name** is declared as new data type, then variables of that type can be declared as

```
struct structure_name structure_variable;
```

# Defining a Structure

- So from earlier example, `student` is new data type and various variables of type **struct student** can be declared as:  
`struct student st;`
- Similarly multiple variables can also be declared:  
`struct student st1, st2, st3;`
- The member variables are accessed using **dot(.)** operator.
- Each member variable of structure has its own copy of member variables.
- For example:
  - `st1.name` is member variable, **name** of `st1` structure variable.
  - `St2.roll` is member variable, **roll** of `st2` structure variable.

```
struct data{  
int day;  
int month;  
int year;  
};
```

```
struct book{  
char title[25];  
char author[20];  
int pages;  
float price;  
}b1, b2;
```

```
struct employee{  
int emp_id;  
char  
emp_name[20];  
int age;  
char gender;  
float salary;  
}e1, e2;
```

```
struct account{  
int acc_no;  
char acc_type[20];  
float balance;  
};
```

# Examples

# Declaration of a Structure

```
#include<stdio.h>
#include<conio.h>
void main()
{
    struct student{
        char name[25];
        int roll_no;
        float marks;
        char gender;
        long long int phone_no;
    };
    struct student st1={"Ram Thapa", 4, 89.5, 'M', 9841234567};
    printf("Name: \t\t%s\n", st1.name);
    printf("Roll No.: \t%d\n", st1.roll_no);
    printf("Marks: \t\t%f\n", st1.marks);
    printf("Gender: \t\t%c\n", st1.gender);
    printf("Phone No.: \t%lld\n", st1.phone_no);
    getch();
}
```

# Declaration of a Structure

```
#include<stdio.h>
#include<conio.h>
void main()
{
    struct student
    {
        char name[20];
        int roll;
        float marks;
        char remarks;
    };
    struct student s;
    printf("Enter name:\t");
    gets(s.name);
    printf("\nEnter roll:\t");
    scanf("%d", &s.roll);
    printf("\nEnter marks:\t");
    scanf("%f", &s.marks);
    printf("\nEnter remarks(P for pass or F for fail):\t");
    scanf(" %c", &s.remarks);
    printf("\n Name \t Roll \t Marks \t Remarks\n");
    printf("\n.....\n");
    printf("\n%s\t%d\t%f\t%c", s.name, s.roll, s.marks, s.remarks);
    getch();
}
```

# Array of Structure

- *A collection of similar type of structure placed in a common variable name is called array of structure.*
- *In our previous structure example, if we want to keep record of 50 students, we have to make 50 structure variables like st1, st2, ..., st50. (which is the worst technique)*
- *To tackle this we can use array of structure to store records of 50 students.*

# Array of Structure

- An array of structure can be declared in two ways as illustrated below:

```
struct Employee{  
    char name[20];  
    int empID;  
    float salary;  
} emp[10];
```

```
struct Employee{  
    char name[20];  
    int empID;  
    float salary;  
};  
struct Employee emp[10];
```

```
/* C program to store record of 10 books */
#include<stdio.h>
#include<conio.h>
struct book
{
    char name[50];
    float price;
    int pages;
};
void main()
{
    struct book b[10];
    int i;
    for(i=0; i<10; i++)
    {
        printf("\nEnter name, price and page number:\n");
        scanf("%s%f%d", &b[i].name, &b[i].price, &b[i].pages);
    }
    for(i=0; i<10; i++)
    {
        printf("\n Name: %s \n Price:%f \n Pages:%d", b[i].name, b[i].price, b[i].pages);
    }
    getch();
}
```

# Array of Structure

# Passing Structure to Function

- *Like any ordinary variable, a structure variable can also be passed to a function.*
- *One may either pass individual structure elements or the entire structure at once.*

# Passing Structure to Function

```
#include<stdio.h>
#include<conio.h>
void display(char[],float,int);
struct book
{
    char name[25];
    float price;
    int pages;
};
void main()
{
    struct book b={"C Program", 345.5, 154};
    display(b.name, b.price, b.pages);
}
void display(char n[25],float pr, int pg)
{
    printf("\nName: %s\nPrice: %f\nPage No.: %d",n, pr, pg);
    getch();
}
```

# Passing Array of Structure to Function

- It is possible to define an array of structures for example if we are maintaining information of all the students in the college and if 100 students are studying in the college.
- We need to use an array than single variables.

- We can define an array of structures as shown in example below:

*struct information*

{

```
int id_no;  
char name[25];  
char address[25];  
char combination[3];  
int age;
```

} student[100];

# Passing Array of Structure to Function

```
#include<stdio.h>
#include<conio.h>
struct Example
{ int num1; int num2; } s[10];
void accept(struct Example s[],int n)
{
    int i;
    for(i=0;i<n;i++)
    {
        printf("\nEnter num1 : "); scanf("%d",&s[i].num1);
        printf("\nEnter num2 : "); scanf("%d",&s[i].num2);
    }
}
void print(struct Example s[],int n)
{
    int i;
    for(i=0;i<n;i++)
    {
        printf("\nNum1 : %d",s[i].num1); printf("\nNum2 : %d",s[i].num2);
    }
}
void main()
{
    int i; accept(s,3); print(s,3); getch();
}
```

# Structure within Structure (Nested Structure)

- Nested structures are nothing but structure within a structure.

```
struct emp_add
{
    int house_no;
    char streetname[25];
};
```

- The above definition could be used as a template to declare a variable address in the employee structure.

```
struct employee
{
    char name[25];
    int age;
    struct emp_add address;
    char address[50];
    float salary;
};
```

# Structure within Structure (Nested Structure)

```
#include<stdio.h>
#include<conio.h>
struct student_college_detail
{
    int college_id;
    char college_name[25];
};
struct student_detail
{
    int id;
    char name[25];
    float percentage;
    struct student_college_detail clg_data;
}stu_data;
void main()
{
    struct student_detail stu_data={1, "Ram", 90.5, 711, "TIC"};
    printf("ID is:\t%d\n", stu_data.id);
    printf("Name is:\t%s\n", stu_data.name);
    printf("Percentage is:\t%f\n", stu_data.percentage);
    printf("College ID is:\t%d\n", stu_data.clg_data.college_id);
    printf("College Name is:\t%s\n", stu_data.clg_data.college_name);
    getch();
}
```

# Union

- *User defined data type such that every union member takes memory that contains a variety of objects.*
- *Union members share space thereby resulting, conserves storage.*
- *Size required by a union variable is the size required by the member requiring the largest size.*
- *Only the last data member defined can be accessed in union.*

# Union

*union item*

```
{    int m;  
    float p;  
    char c;  
} code;
```

- *The above code declares a variable code of type union item.*
- *Union contains three members each with different data types.*
- *However only one of them can be used at a time.*
- *Syntax to access variable is similar to structure. i.e.*  
*code.m=456;*  
*code.p=456.78;*  
*printf("%d", code.m);*

```
#include<stdio.h>
#include<conio.h>
void main()
{
    union student
    {
        int roll;
        float marks;
    }stud;
    stud.roll=45;
    printf("Roll No. - \t%d\n", stud.roll);
    stud.marks=72.5;
    printf("Marks - \t%f", stud.marks);
}
```

```
#include<stdio.h>
#include<conio.h>
void main()
{
    union student
    {
        int roll;
        float marks;
    }stud;
    stud.roll=45;
    stud.marks=72.5;
    printf("Roll No. - \t%d\n", stud.roll);
    printf("Marks - \t%f", stud.marks);
}
```

# Union – Example

```
#include<stdio.h>
#include<conio.h>
union student
{
    char name[20];
    int roll_no;
    float marks;
    char section;
};
void main()
{
    union student s;
    printf("Size of union=%d\n", sizeof(s));
    strcpy(s.name, "Hari");
    printf("Name=%s\t", s.name);
    printf("Location=%u", s.name);
    s.roll_no=10;
    printf("\nRoll=%d \tLocation=%u", s.roll_no, &s.roll_no);
    s.marks=91.5;
    printf("\nMarks=%.2f \tLocation=%u", s.marks, &s.marks);
    s.section='A';
    printf("\nSection=%c \tLocation=%u", s.section, &s.section);
    printf("\n\nErroneous output");
    printf("\n\nName\tRoll\tMarks\tSection\n");
    printf("%s\t%d\t%.2f\t%c", s.name, s.roll_no, s.marks, s.section);
    getch();
}
```

# Pointer to Structure

- Pointers can be used also with structure.
- To store address of a structure type variable, we can define a structure type pointer variable as normal way.
- A structure type pointer variable can be declared as:

```
struct book
{
    char name[20];
    int pages;
    float price;
};
```

Struct book \*bptr;

# Pointer to Structure

- *This declaration for a pointer to structure does not allocate any memory for a structure but allocates only for a pointer.*
- *So to access structures members through pointer **bptr**, we must allocate the memory using **malloc()** function.*
- *Now, individual structure members are accessed as:*

<b><i>bptr -&gt; name</i></b>	<i>or</i>	<b><i>(*bptr).name</i></b>
<b><i>bptr -&gt; pages</i></b>	<i>or</i>	<b><i>(*bptr).pages</i></b>
<b><i>bptr -&gt; price</i></b>	<i>or</i>	<b><i>(*bptr).price</i></b>
- *Here, **->** is called **arrow operator** and there must be a pointer to the structure on the left side of this operator.*

# Pointer to Structure - Example

```
#include <stdio.h>

struct person
{
    int age;
    float weight;
};

int main()
{
    struct person *personPtr, person1;
    personPtr = &person1;
    printf("Enter age: ");
    scanf("%d", &personPtr->age);
    printf("Enter weight: ");
    scanf("%f", &personPtr->weight);
    printf("\nDisplaying:\n");
    printf("Age: %d\n", personPtr->age);
    printf("weight: %f", personPtr->weight);
    return 0;
}
```

# Example

*Write a menu driven program which performs the following jobs:*

- i. Create a structure named student with member variables: roll, name, grade, gender, age.*
- ii. Define ten structure variables of type student and using loop take input for all ten structure variables.*
- iii. Display names of all male students.*
- iv. Exit.*

***END OF UNIT EIGHT***

# UNIT 9

**POINTERS**

# Syllabus

- **UNIT 9: Pointers**
  - *Introduction*
  - *The & and \* operator*
  - *Declaration of Pointer*
  - *Chain of Pointers*
  - *Pointer Arithmetic*
  - *Pointers and Arrays*
  - *Array of Pointers*
  - *Pointers as Function Arguments*
  - *Function Returning Pointers*
  - *Pointers and Structures*
  - *Dynamic Memory Allocation*

# Background

- So before knowing what pointer is let us know about memories.
- All computer have primary memory, also known as **RAM** (Random Access Memory)
- RAM holds the programs that the computer is currently running along with the data (i.e. variables) they are currently manipulating.
- All the variables used in a program reside in the memory when the program is executed.
- RAM is divided into a number of small units or locations and each location is represented by some unique number known as memory address.

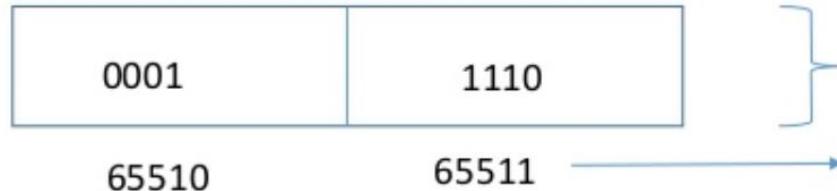
# Background

- *Each memory location is capable of storing small number, which is known as byte.*
- *A **char** data is one byte in size and hence needs one memory location of the memory.*
- *Similarly, integer data is two byte in size and hence needs two memory locations of the memory.*
- **Key concept:**  $8 \text{ bits} = 1 \text{ byte}$  ;  $1024 \text{ bytes} = 1 \text{ Kilo Byte (KB)}$  ;  $1024 \text{ KB} = 1 \text{ Mega Byte (MB)}$  ;  $1024 \text{ MB} = 1 \text{ Giga Bytes (GB)}$
- *A computer having 1 GB RAM has  $1024*1024*1024$  (i.e. 1073741824) bytes and these 1073741824 bytes are represented by 1073741824 different address.*
- *For example: The memory address 65524 represents a byte in memory and it can store data of one byte.*

# Background

- *Every variable in C program is assigned a space in memory.*
- *When a variable is declared, it tells computer the type of variable and name of the variable.*
- *According to the type of variable declared, the required memory locations are reserved.*
- *For example: int requires two bytes, float requires four bytes and char requires one byte.*

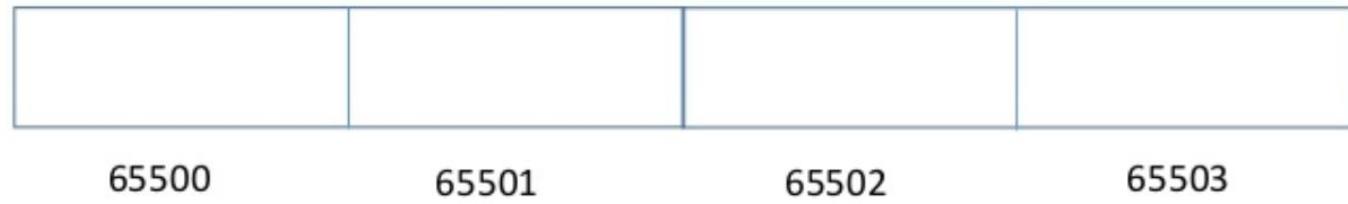
```
int a=30;
```



Value equivalent to 30

Address

```
float b;
```



```
char c;
```



65510

# A program to display memory location reserved by a variable

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a=30;
    printf("The address of a is %u\n", &a);
    printf("The value of a is %d\n", a);
    getch();
}
```

- **Output:**

*The address of a is 2752268*

*The value of a is 30*

# *A program to illustrate address reserved by different data types*

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a=3, b=4;
    float c=50.5;
    char d='A';
    printf("The Base Address of a is %u\n", &a);
    printf("The Base Address of b is %u\n", &b);
    printf("The Base Address of c is %u\n", &c);
    printf("The Base Address of d is %u\n", &d);
    getch();
}
```

- **Output:**

*The Base Address of a is 2752268  
The Base Address of b is 2752264  
The Base Address of c is 2752260  
The Base Address of d is 2752259*

# Pointer

- *A pointer is a variable that contains a memory address of variable.*
- *A pointer variable is declared to some type, like any other variable.*
- *Each pointer variable can point only to one specific type.*
- *Pointer is declared in the same fashion like other variables but is always preceded by '\*' (asterisk operator).*
- *Integer variable declared as:*  
`int a; (Here, a is considered as integer variable)`
- *Similarly,*  
`int *a;`  
*Now variable a is a pointer variable, it now can store address of integer variable.*  
*The address of float variable can not be stored in it.*

# Pointer

- *Valid Example:*

```
int *p;
```

```
int num;
```

```
p=&num;
```

- *Invalid Example:*

```
int *p;
```

```
float num;
```

```
p=&num;
```

# Advantages of Pointer

- *Pointers increase the execution speed and thus reduce the program execution time.*
- *With the help of pointer we can handle any type of data structure.*
- *By using pointers we can access a data which is available outside a function.*
- *By using pointer variable we can implement dynamic memory allocation.*
- *Pointers provide an efficient tool for manipulating dynamic data structures such as structures, linked lists, queues, stacks and trees.*
- *Pointers reduce length and complexity of programs.*

# Pointer Declaration

- *Pointer variable can be declared as follows:*

*data\_type \*variable\_name;*

- *Example:*

*int \*x; // x integer pointer, holds address of any int variable*

*float \*y; // y integer pointer, holds address of any float variable*

*char \*z; // z integer pointer, holds address of any char variable*

# Pointer Declaration

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int v=10, *p;
    p=&v;
    printf("Address of v is %u\n",&v);
    printf("Address of v is %u\n",p);
    printf("Value of v is %d\n",v);
    printf("Value of v is %d\n",*p);
    printf("Address of p is %u\n",&p);
    getch();
}
```

# Indirection or Dereference Operator

- The operator `*`, used in front of a variable, is called *pointer* or *indirection or dereference operator*.
- Normal variable provides direct access to their own values whereas pointer provides indirect access to the values of the variable whose address it stores.
- When the pointer is declared, the star indicates that it is a pointer, not a normal variable.
- The indirection operator indicates “*the value at the memory location stored in the pointer*” or “*the content of the location pointed by pointer variable*”

# Address Operator

- *The operator **&** is known as **address operator**.*
- ***&a*** denotes the address of variable ***a***.
- *For example:*

```
int a=10, *p;  
p=&a;
```

# Initializing Pointer

- *Address of some variable can be assigned to a pointer variable at the time of declaration of the pointer variable.*

- *For example:*

*int num;*

*int \*ptr=&num;*

- *These two statements above are equivalent to following statements.*

*int num;*

*int \*ptr;*

*ptr=&num;*

# So what actually is bad pointer?

- *When a pointer is first declared, it doesn't have a valid address.*
- *Each pointer must be assigned a valid address before it can support dereference operators. Before that, the pointer is bad and must not be used.*
- *Every pointer contains garbage value before assignment of some valid address.*
- *Correct code overwrites the garbage value with a correct reference to an address and thereafter the pointer works fine.*
- *We have to program carefully.*

# Void Pointer

- *Void pointer is a special type of pointer.*
- *It can point to any data type, from an integer value to a float to a string of characters.*
- *Using void pointer, the pointed data can not be referenced directly (i.e. \* operator can not be used on them)*
- *Type casting or assignment must be used to change the void pointer to a concrete data type to which we can refer.*

# Void Pointer in C: Definition

- Suppose we have to declare integer pointer, character pointer and float pointer then we need to declare 3 pointer variables.
- Instead of declaring different types of pointer variable it is feasible to declare single pointer variable which can act as integer pointer, character pointer.
- Declaration of void pointer:  
`void *pointer_name;`
- In C, general purpose pointer is called as void pointer.
- It does not have any data type associated with it.
- It can store address of any type of variable.
- A void pointer is a C convention for a raw address.
- The compiler has no idea what type of object a void pointer really points to.

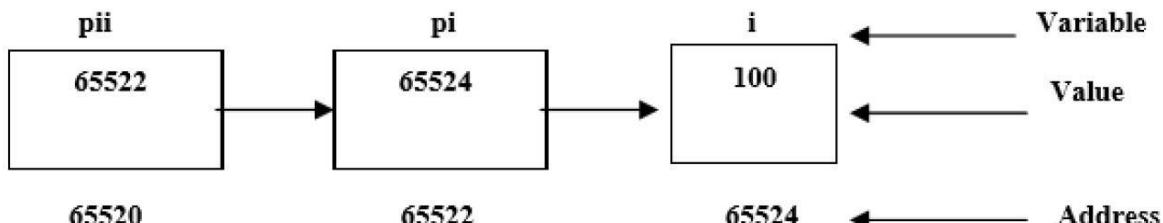
# Example

```
void *ptr;           // declaration of void pointer
char cnum;          // declaration of character type variable
int inum;           // declaration of integer type variable
float fnum;          // declaration of float type variable
ptr=&cnum;          // ptr has address of character data
ptr=&inum;          // ptr has address of integer data
ptr=&fnum;          // ptr has address of float data
```

# Chain of Pointers

- A pointer variable can be assigned the address of an ordinary variable.
- Now, this variable itself could be another pointer (i.e. a pointer can contain address of another pointer)

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int i = 100;
    int *pi;
    int **pii;
    pi = &i;
    pii = &pi;
    printf("Address of i = %u \n", &i);
    printf("Address of i = %u \n", pi);
    printf("Address of i = %u \n", *pii);
    printf("Address of pi = %u \n", &pi);
    printf("Address of pi = %u \n", pii);
    printf("Address of pii = %u \n", &pii);
    printf("Value of i = %d \n", i);
    printf("Value of i = %d \n", *(&i));
    printf("Value of i = %d \n", *pi);
    printf("Value of i = %d", **pii);
    getch();
}
```



# Pointer Arithmetic

- Pointer address can be manipulated by arithmetic operators with integer values.
- These arithmetic operators are only `+` and `-` and increment and decrement operators.
- The increment and decrement of pointer variable depends on its data types.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int *p;
    int age=17;
    p=&age;
    printf("\n Value of age is %d", age);
    printf("\n Increment on age is %d", ++age);
    printf("\n Address of age is %u", p);
    printf("\n Increment in pointer is %u", ++p);
    getch();
}
```

# NULL Pointer

- *A null pointer is a special pointer value that points nowhere or nothing.*
- *The predefined constant NULL in stdio.h is used to define null pointer.*

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

```
#include<conio.h>
```

```
#define NULL 0
```

```
void main()
```

```
{
```

```
    int *ptr=NULL; //double/float/char/void *ptr=NULL
```

```
    if(ptr==NULL)
```

```
        printf("Change NULL to 1 and there will be warnings!!!");
```

```
        getch();
```

```
}
```

# Double Pointer (Pointer to Pointer)

- C allows the use of pointers that point to other pointers and these in turn, point to data.
- For pointers to do that, we only need to add asterisk (\*) for each level of reference. For example:

*int a=20;*

*int \*p;*

*int \*\*q;*

*p=&a;*

*q=&p;*

# Double Pointer (Pointer to Pointer)

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int var=25;
    int *ptr;
    int **pptr;
    ptr=&var;
    pptr=&ptr;
    printf("Value of var = %d\n",var);
    printf("Value available at *ptr = %d\n",*ptr);
    printf("Value available at **pptr = %d\n",**pptr);
    getch();
}
```

# Pointers and Arrays

- Pointers and arrays are so closely related.
- An array declaration such as `int arr[5]` will lead the compiler to pick an address to store a sequence of 5 integers, and `arr` is a name for that address.
- The value is not the first integer in the sequence, nor is it the sequence in its entirety. It is just an address.
- Now, if `arr` is a one-dimensional array, then the address of the first array element can be written as `&arr[0]` or simply `arr`.
- Moreover, the address of the second array element can be written as `&arr[1]` or simply `(arr+1)`.
- It is not possible to assign an arbitrary address to an array name or to an array element. (i.e. expressions such as `arr`, `(arr+i)`, and `arr[i]` cannot appear on the left side of an assignment statement.

`&arr[0]=&arr[1]; //invalid`

# Pointers and Arrays

```
/*Program that accesses array elements of a one-dimensional array
using pointers*/
#include<stdio.h>
#include<conio.h>
void main()
{
    int arr[5] = {10, 20, 30, 40, 50};
    int i;
    for (i = 0; i < 5; i++)
    {
        printf ("i=%d\t arr[i]=%d\t *(arr+i)=%d\n", i, arr[i], *(arr+i));
        printf ("&arr[i]=%u\t arr+i=%u\n", &arr[i], (arr+i));
    }
    getch();
}
```

# Pointers and Arrays

```
/*WAP to calculate average marks of 5 students in a subject using pointer*/
#include<stdio.h>
#include<conio.h>
void main()
{
    float marks[5],sum=0;
    int i;
    float avg;
    printf("Enter marks of 5 students:\n");
    for(i=0;i<5;i++)
    {
        scanf("%f",marks+i);
        sum+=*(marks+i);
    }
    avg=sum/5;
    printf("\nThe average is %f.",avg);
    getch();
}
```

# Pointers and Character Strings

- As we have seen in strings, a string in C is an array of characters ending in the null character (written as '\0'), which specifies where the string terminates in memory.
- Like in one-dimensional arrays, a string can be accessed via a pointer to the first character in the string.
- The value of a string is a (constant) address of its first character.
- Thus, it is appropriate to say that a string is a constant pointer.
- A string can be declared as a character array or a variable of type `char *`.

# Pointers and Character Strings

- The declarations can be done as shown below:

*char country[] = "NEPAL";*

*char \*country = "NEPAL";*

- The second declaration creates a pointer variable *country* that points to the letter **N** in the string "NEPAL" somewhere in memory.
- Once the base address is obtained in the pointer variable *country*, *\*country* would yield the value at this address, which gets printed through,  
*printf("%s", \*country);*

```
#include<stdio.h>
#include<conio.h>
void main()
{
    char name[10] = "Mr. Nepal";
    int i;
    printf("String Element\t Address\n");
    printf(".....\n");
    for(i=0;i<10;i++)
        printf("name[%d]=%c \t %u\n", i, name[i], &name[i]);
    printf("\n\nString Element\t Address\n");
    printf(".....\n");
    for(i=0;i<10;i++)
        printf("name[%d]=%c \t %u\n", i, *(name + i), name + i);
    printf("\n%s \t %s \t %s \t %s\n", name, name+1, name+2, name+3);
    getch();
}
```

# Array of Pointers to Strings

```
#include<stdio.h>
#include<conio.h>
void main()
{
    char *names[] = {"jump", "walk", "red", "green", "talk", "move", "look", "feel"};
    int i, j;
    char *temp;
    for(i=0; i<7; i++)
    {
        for(j=i+1; j<8; j++)
        {
            if(strcmp(names[i], names[j])>0)
            {
                temp=names[i];
                names[i]=names[j];
                names[j]=temp;
            }
        }
    }
    printf("\nSorted List:\n\n");
    for(i=0; i<8; i++)
        puts(names[i]);
    getch();
}
```

# Array of Pointers

- The way there can be an array of integers, or an array of float numbers, similarly, there can be array of pointers too.
- Since a pointer contains an address, an array of pointers would be a collection of addresses.
- For example, a multidimensional array can be expressed in terms of an array of pointers rather than a pointer to a group of contiguous arrays.
- Two dimensional array can be defined as a one-dimensional array of integer pointers by writing:  
`int *arr[3];`  
rather than the conventional array definition,  
`int arr[3][5];`
- Similarly, an  $n$ -dimensional array can be defined as  $(n-1)$ -dimensional array of pointers by writing:  
`data-type *arr[subscript 1] [subscript 2] ... [subscript n-1];`
- The subscript 1, subscript 2, ..., subscript  $n-1$  indicates the maximum number of elements associated with each subscript.

# Pointers as Function Arguments

- Pointers are often used to pass value on the function.
- The value and address can be passed as an argument or parameter to the function and receive as pointer.
- The value of pointer variable can be manipulated as other functions.

# Pointers as Function Arguments

```
/*WAP to pass pointer variable to function sum them and display after returning it.*/
#include<stdio.h>
#include<conio.h>
int add(int *i, int *j);
void main()
{
    int a,b,c=0;
    printf("\n Enter two numbers:");
    scanf("%d%d", &a, &b);
    c=add(&a, &b);
    printf("\nSum of two numbers %d and %d is %d", a,b,c);
    getch();
}
int add(int *p, int *q)
{
    int temp;
    temp=*p+*q;
    return(temp);
}
```

# Function Returning Pointers

- A function can return a single value by its name or return multiple values through pointer parameters.
- Since pointers are a data type in C, we can force a function to return a pointer to the calling function.

```
#include<stdio.h>
#include<conio.h>
int * larger(int *,int *);
void main()
{
    int a=100,b=50;
    int *p;
    p=larger(&a,&b); //function call
    printf("%d",*p);
    getch();
}
int *larger(int *x,int *y)
{
    if(*x>*y)
        return (x); // address of a
    else
        return (y); // address of b
}
```

- Note: The address returned must be the address of a variable in the calling function. It is an error to return a pointer to a local variable in the called function.

# Dynamic Memory Allocation (DMA)

- *The process of allocating and freeing memory at run time is known as dynamic memory allocation.*
- *This reserves the memory required by the program and returns valuable resources to the system once the use of reserved space is utilized.*
- *Though arrays can be used for data storage, they are of fixed size.*

```
#define m 5
void main()
{
    int x[m];
    printf("%lu\n", sizeof(x));
}
```

# Dynamic Memory Allocation (DMA)

- Consider an array size of 100 to store marks of 100 student.
- If the number of students is less than 100 say 10, only 10 memory locations will be used and rest 90 locations are reserved but will not be used. i.e. wastage of memory will occur.
- In such situation DMA will be useful.
- Since an array name is actually a pointer to the first element within the array, it is possible to define the array as a pointer variable rather than as a conventional array.
- While defining conventional array, system reserves fixed block of memory at the beginning of program execution which is inefficient but this does not occur if the array is represented in terms of a pointer variable.
- The use of pointer variable to represent an array requires some type of initial memory assignment before the array elements are processed.
- This is known as Dynamic Memory Allocation (DMA).

# Dynamic Memory Allocation (DMA)

- At *execution time*, a program can request more memory from a free memory pool and frees if not required using DMA.
- Thus, DMA refers allocating and freeing memory at execution time or run time.
- There are four library functions for memory management:
  - *malloc()*
  - *calloc()*
  - *free()*
  - *realloc()*
- These functions are defined within header file **stdlib.h** and **alloc.h**

# DMA – *malloc()*

- It allocates requested size of bytes and returns a pointer to the first byte of the allocated space.

*ptr=(data\_type\*) malloc(size\_of\_block);*

- Here, *ptr* is a pointer of type *data\_type*. The *malloc()* returns a pointer to an area of memory with size *size\_of\_block*.

*x=(void\*) malloc(100\*sizeof(int));*

- A memory space equivalent to 100 times the size of an integer is reserved and the address of the first byte of the memory allocated is assigned to the pointer *x* of type *int*. (i.e. *x* refers to the first address of allocated memory).

# Example – malloc()

```
#include<stdio.h>
#include<stdlib.h>
struct Emp
{
    int eno;
    char ename[20];
    float esal;
};
void main()
{
    struct Emp *ptr;
    ptr=(struct Emp*)malloc(sizeof(struct Emp));
    if(ptr==NULL)
    {
        printf("Memory allocation failed\n");
    }
    else
    {
        printf("Enter Employee Number:\n");
        scanf("%d",&ptr->eno);
        printf("Enter Employee Name:\n");
        scanf("%s",ptr->ename);
        printf("Enter Employee Salary:\n");
        scanf("%f",&ptr->esal);
        printf("\nEmployee Number = %d\nEmployee Name = %s\nEmployee Salary = %f",ptr->eno,ptr->ename,ptr->esal);
    }
}
```

# DMA - `calloc()`

- The function provides access to the C memory heap, which is available for dynamic allocation of variable-sized blocks of memory.
  - Unlike `malloc()`, it accepts two arguments: `no_of_blocks` and `size_of_each_block` specifies the size of each item.
  - The function `calloc` allocates multiple blocks of storage, each of the same size and then sets all bytes to zero.
  - `Calloc` initializes all bytes in the allocated block to zero.  
`ptr=(data_type*) calloc(no_of_blocks, size_of_each_block);`
  - For example:  
`x=(int*) calloc(5, 10* sizeof(int));`      **OR**  
`x=(int*) calloc(5, 20);`
  - The above statement allocates contiguous space for 5 blocks, each of size 20 bytes. i.e. we can store 5 arrays, each of 10 elements of integer types.

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
    int n, *arr, i;
    printf("Enter size:\n");
    scanf("%d", &n);
    arr=(int*)calloc(n,sizeof(int));
    if(arr==NULL)
        printf("Memory allocation FAILED !!");
    else
    {
        printf("Array addresses are:\n");
        for(i=0;i<n;i++)
        {
            printf("%u\n", arr+i);
        }
    }
}
```

# Example – calloc()

# DMA – *realloc()*

- *This function is used to modify the size of previously allocated space.*
- *Sometimes the previously allocated memory is not sufficient; we need additional space and sometime the allocated memory is much larger than necessary.*
- *We can change memory size already allocated with the help of function **realloc()**.*
- *If the original allocation is done by the statement:*  $\text{ptr} = \text{malloc}(\text{size});$
- *Then, reallocation of space may be done by the statement:*  $\text{ptr} = \text{realloc}(\text{ptr}, \text{newsized});$
- *This function allocates a new memory space of size **newsized** to the pointer variable **ptr** and returns a pointer to the first byte of the new memory block and on failure the function return **NULL**.*

# Example - *realloc()*

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
    int n, *arr, i;
    printf("Enter size:\n");
    scanf("%d", &n);
    arr=(int*)calloc(n,sizeof(int));
    n++;
    arr=(int*)realloc(arr,n*sizeof(int));
    if(arr==NULL)
        printf("Memory allocation FAILED !!");
    else
    {
        printf("Address of array elements are:\n");
        for(i=0;i<n;i++)
        {
            printf("%u\n", (arr+i));
        }
    }
}
```

# DMA – *free()*

- *Frees previously allocated space by `calloc`, `malloc` or `realloc` functions.*
- *The memory dynamically allocated is not returned to the system until the programmer returns the memory explicitly. This can be done using `free()` function.*
- *This function is used to release the space when not required.*
- *Syntax:*  
*`free(ptr);`*

# Example – free()

```
#include<stdlib.h>
#include <stdio.h>
void main()
{
    int* ptr = malloc(10 * sizeof(*ptr));
    if(ptr != NULL)
    {
        *(ptr + 2) = 50;
        printf("Value of the 3rd integer is %d\n", *(ptr + 2));
    }
    free(ptr);
    printf("Value of the 3rd integer is %d", *(ptr + 2));
}
```

# Static Memory & Dynamic Memory

S.N.	STATIC MEMORY	DYNAMIC MEMORY
1.	<i>Memory is allocated during compile time.</i>	<i>Memory is allocated during run time.</i>
2.	<i>Memory is of fixed size throughout the program.</i>	<i>Memory keeps on changing with the change in insertion and deletion of memory.</i>

***END OF UNIT NINE***

# UNIT 10

*FILE HANDLING IN C*

# Syllabus

- **UNIT 10: File Handling in C**
  - *Concept of File*
  - *Opening and Closing of File (naming, opening, and closing a file)*
  - *Input Output Operations in File (reading data from file, writing data to a file)*
  - *Random Access in File (ftell(), fseek(), rewind())*
  - *Error Handling in Files (feof(), ferror())*  
(Note: address some of the functions associated with file handling, e.g. fopen(), fclose(), fgetc(), fputc(), fprintf(), fscanf())

# Introduction

- *The input output functions like `printf()`, `scanf()`, `getchar()`, `putchar()`, etc are known as **console oriented I/O functions** which always use input devices and computer screen or monitor for output devices.*
- *Using these library function, the entire data is lost when either the program is terminated or the computer is turned off.*
- *This problem invites concept of data files in which data can be stored on the disks and read whenever necessary, without destroying data.*

# Introduction

- *A file is a place on the disk where a group of related data is stored.*
- *The data file allows us to store information and to access and alter the information whenever necessary.*
- *C has various library functions for creating and processing data files.*
- *Mainly there are two types of data files, one is **stream oriented** or standard or high level and the other is **system oriented** or low level data files.*

# Introduction

- The *standard data files* (i.e. *stream oriented*) are again subdivided into *text files* and *binary files*.
- The *text files* consists of consecutive characters and these characters are interpreted as individual data item.
- The *binary files* organize data into blocks containing contiguous bytes of information.
- For each *binary* and *text files*, there are number of *formatted* and *unformatted* library functions in C.

# Introduction

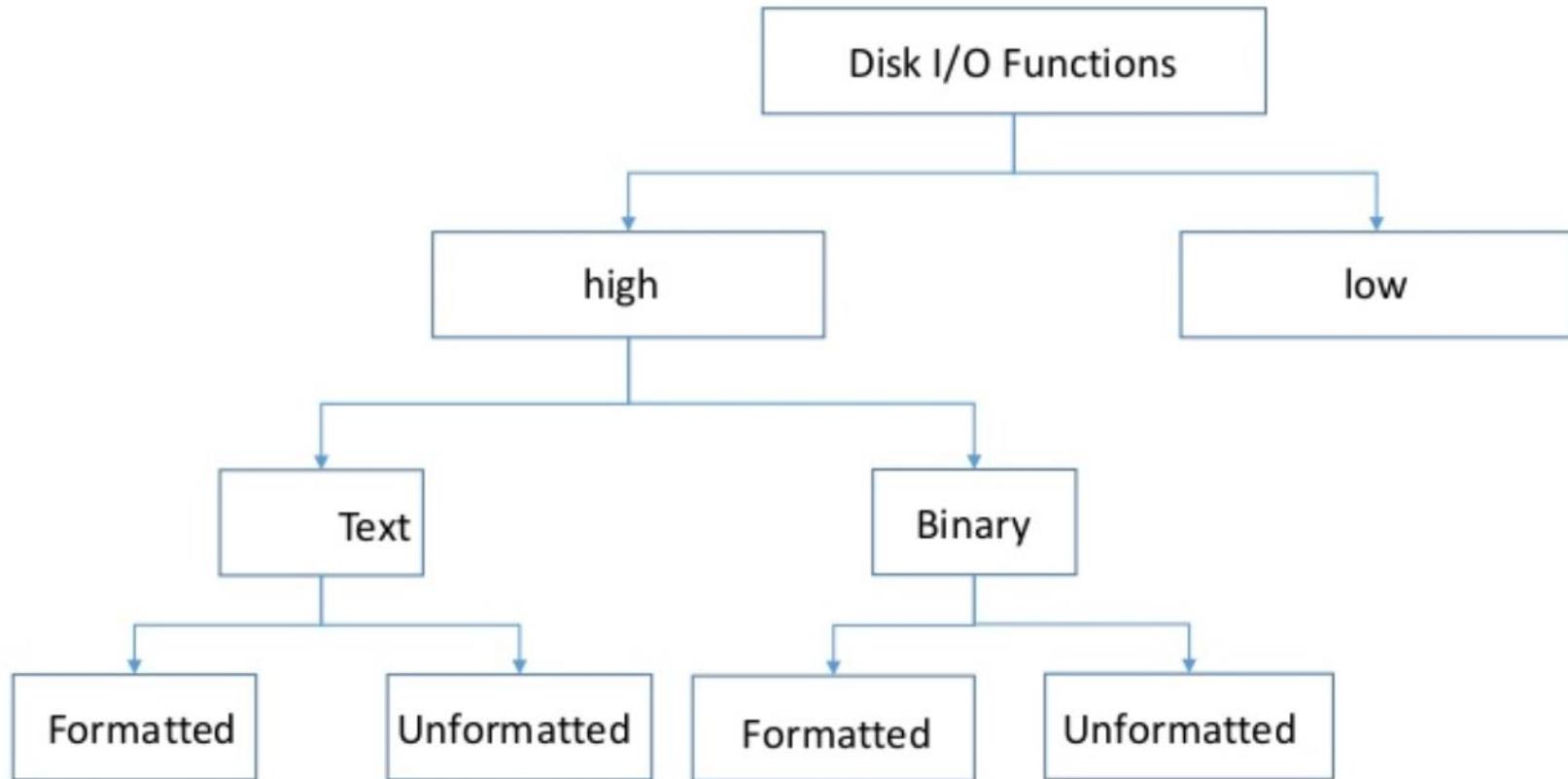


Fig. Classification of disk I/O functions

# Opening & Closing File

- *Before a program can write to a file or read from a file, the program must open it.*
- *Opening a file establishes a link between the program and the OS. This provides OS the name of the file and the mode in which the file is to be opened.*
- *While working with high level data file, we need buffer area where information is stored temporarily in the course of transferring data between computer memory and data file.*
- *The buffer area is established as:*  
***FILE \*ptr\_variable;***

# Opening & Closing File

- *The buffer area is established as:*  
*FILE \*ptr\_variable;*
- *Here, FILE is a special structure, declared in header file stdio.h*
- *The ptr\_variable is a pointer “pointer to the data type FILE” that stores the beginning address of the buffer area allocated after a file has been opened.*
- *This pointer contains all the information about the file and it is used as a communication link between the system and the program.*
- *A data file is opened using syntax:*  
*ptr\_variable = fopen(file\_name, file\_mode);*

# Opening & Closing File

- A data file is opened using syntax:  
*ptr\_variable = fopen(file\_name, file\_mode);*
- The function *fopen* returns a pointer to the beginning of the buffer area associated with the file.
- A NULL value is returned if the file cannot be opened due to some reasons.
- After opening a file, we can process data and finally we close it.
- Closing a file ensures that all outstanding information associated with the file is flushed out from the buffers.
- For example:  
*fclose(ptr\_variable);*

# File Opening Modes

Mode	Description
r	<i>Opens an existing text file for reading purpose.</i>
w	<i>Opens a text file for writing. If it does not exist, then a new file is created. Here your program will start writing content from the beginning of the file.</i>
a	<i>Opens a text file for writing in appending mode. If it does not exist, then a new file is created. Here your program will start appending content in the existing file content.</i>
r+	<i>Opens a text file for both reading and writing. If the file exists, loads it into memory and set up a pointer to the first character in it. If the file doesn't exist it returns null.</i>
w+	<i>Opens a text file for both reading and writing. If file is present, it first destroys the file to zero length, otherwise creates a file if it does not exist.</i>
a+	<i>Opens a text file for both reading and writing. It creates the file if it doesn't exist. The reading will start from the beginning but writing can only be appended.</i>

# Input Output Operations – String

- Using string I/O functions *fgets()* & *fputs()*, data can be read from a file or written to a file in the form of array of characters.

- *fgets()* is used to read string from file.

*fgets(string, int\_value, fp);*

*// here, int\_value denotes the no. of characters in the string*

- *fputs()* is used to write string to file.

*fputs(string, fp);*

# Example – I

```
/* Program to create a file named test.txt and write some text "I study B.Sc. CSIT" to the file. */
#include<stdio.h>
#include<conio.h>
void main()
{
    FILE *fp;
    fp=fopen("D:\\test.txt", "w");
    if(fp==NULL)
    {
        printf("\n Cannot create file. ");
        exit(0);
    }
    else
    {
        printf("\n File is created. ");
    }
    fputs("I study B.Sc. CSIT",fp);
    fclose(fp);
    getch();
}
```

# Example - II

```
/* Program to open the file named test.txt, read its content and display it to screen */
#include <stdio.h>
#include<conio.h>
void main()
{
    FILE *fp;
    char s[100];
    fp=fopen("D:\\test.txt", "r");
    if(fp==NULL)
    {
        printf("\n Cannot open file. ");
        exit(0);
    }
    else
    {
        printf("\nFile is opened. ");
    }
    fgets(s,19,fp);
    printf("\nText from file is: %s", s);
    fclose(fp);
    getch();
}
```

# Input Output Operations – Character

- Using character I/O functions `fgetc()` & `fputc()`, data can be read from a file or written to a file one character at a time.
- `fgetc()` is used to read a character from a file.  
`char_variable = fgetc(fp);`
- `fputc()` is used to write a character to a file.  
`fputc('character' or character_variable, fp);`

# End Of File (EOF)

- *EOF is a special character (an integer with ASCII value 26) that indicates that the end-of-file has been reached.*
- *This character can be generated from the keyboard by typing **Ctrl+Z***
- *It is defined in `<stdio.h>`*
- *When we are creating a file, the special character EOF is inserted after the last character of the file by the Operating System.*
- *Thus, the last point of file is detected using EOF while reading data from file.*
- **Caution:** *An attempt to read after EOF might either cause the program to terminate with an error or result in an infinite loop situation.*

```
/*C program to read a file and display file contents character by character using fgetc() */

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
void main()
{
    /* File pointer to hold reference to our file */
    FILE *fPtr;
    char ch;
    /*Open "test.txt" file in r (read) mode.*/
    fPtr = fopen("D://test.txt", "r");
    /*fopen() return NULL if last operation was unsuccessful */
    if(fPtr == NULL)
    {
        /* Unable to open file hence exit */
        printf("Unable to open file.\n");
        printf("Please check whether file exists and you have read privilege.\n");
        exit(0);
    }
    /* File open success message */
    printf("File opened successfully. Reading file contents character by character. \n\n");
    do
    {
        /* Read single character from file */
        ch = fgetc(fPtr);
        /* Print character read on console */
        putchar(ch);
    } while(ch != EOF); /* Repeat this if last read character is not EOF */
    /* Done with this file, close file to release resource */
    fclose(fPtr);
    getch();
}
```

# Example - I

# **Input Output Operations – Formatted**

- Using formatted I/O functions *fprintf()* & *fscanf()*, numbers, characters or string can be read from a file or written to a file according to our requirement format.
- *fprintf()* is formatted output function which is used to write integer, float, char or string value to a file.  
*fprintf(fp, “control\_string”, list\_of\_variables);*
- *fscanf()* is formatted input function which is used to read integer, float, char or string value from a file.  
*fscanf(fp, “control\_string”, &list\_of\_variables);*

# Example – I

```
/* Program to create a file named student.txt and write name, roll, address and marks of a student to this file*/
#include<stdio.h>
#include<conio.h>
void main()
{
    FILE *fp;
    char name[20];
    int roll;
    char address[20];
    float marks;
    fp=fopen("D:\\student.txt", "w");
    if(fp==NULL)
    {
        printf("\n File cannot be created or opened.");
        exit(0);
    }
    printf("\n Enter name of student:\t");
    gets(name);
    printf("\n Enter roll number of %s:\t", name);
    scanf("%d", &roll);
    fflush(stdin);
    printf("\n Enter address of %s:\t", name);
    gets(address);
    printf("\n Enter marks of %s:\t", name);
    scanf("%f", &marks);
    printf("\n Now writing data to file... ");
    fprintf(fp, "Name=%s\n Roll=%d\n Address=%s\n Marks=%f", name, roll, address, marks);
    printf("\n Completed");
    fclose(fp);
    getch();
}
```

# Example - II

```
/*Given a text file, create another text file deleting the following words "three", "bad", and "time".*/
#include <stdio.h>
#include<conio.h>
void main()
{
    FILE *fp,*fpp;
    char c[10];
    fp=fopen("D:\\test.txt","r");
    if(fp==NULL)
    {
        printf("Cannot open file");
        exit(0);
    }
    fpp=fopen("D:\\hello.txt","w");
    if(fpp==NULL)
    {
        printf("Cannot create file");
        exit(0);
    }
    while(fscanf(fp,"%s",&c)!=EOF)
    {
        if(strcmp(c,"three")!=0)&&(strcmp(c,"bad")!=0)&&(strcmp(c,"time")!=0))
        {
            fprintf(fpp,"%s ",c);
        }
    }
    fclose(fp);
    fclose(fpp);
    getch();
}
```

# Random Access in File

- Till now, reading and writing data from/to a file has been done sequentially.
- While reading data from a file, the data items are read from the beginning of the file in sequence until the end of file.
- Also, while writing data to a file, the data items are placed one after the other in a sequence.
- This is called sequential access.
- But we may need to access a particular data item placed in any location without starting from the beginning.
- This is called *random access* or *direct access*.

# Use of file pointer in random access file

- *A file pointer is a pointer to a particular byte in a file.*
- *While opening a file in write mode, the file pointer is at the beginning of the file, and whenever we write to a file, the file pointer moves to the end of the data items written so that writing can continue from that point.*
- *While opening a file in read mode, the file pointer is at the beginning of the file, and whenever we read from a file, the file pointer moves to the beginning of the next data item so that reading can continue from that point.*
- *While opening a file in append mode, the file pointer is at the end of the existing file, so that new data items can be written from there onwards.*
- *So, if we are able to move the file pointer according as our need, then any data item can be read from a file or written onto a file randomly.*

# Functions used in random access

## ***ftell()***

- *This function takes a file pointer as an argument and returns a number of type long, that indicates the current position of the file pointer within the file.*
- *This function is useful in saving the current position of a file, which can be used later in the program.*
- *Syntax:*  
 $n = \text{ftell}(fp);$
- *Here, n would give the relative offset (in bytes) of the current position.*
- *This means that n bytes have already been read (or written).*

# Functions used in random access

## rewind()

- Do you remember that one way of positioning the file pointer to the beginning of the file is to close the file and then reopen it?
- The `rewind()` function can do this without closing the file.
- This function takes a file pointer as argument and resets the current position of the file pointer to the start of the file.
- Syntax:  
`rewind(fp);`  
`rewind(fp);`  
`n=fseek(fp);`
- Here, `n` would be assigned to 0, because file position has been set to the start of the file by `rewind()`.
- The first byte in the file is numbered as 0, second as 1, and so on.

# Functions used in random access

## fseek()

- *This function is used to move the file pointer to a desired position within a file.*
- *Syntax:*  
*fseek(fp, offset, position);*  
*Where, fp is a file pointer, offset is a number or variable data type long, and position is an integer number.*
- *The offset specifies the number of positions (bytes) to be moved from the location specified by position.*
- *The position can have one of the following 3 values:*
  - *0 – Beginning of file*
  - *1 – Current position*
  - *2 – End of file*

# Error Handling in Files

- *Error situations during I/O operations:*
  - Trying to read beyond the end-of-file mark
  - Trying to use a file that has not been opened
  - Trying to perform an operation on a file, when the file is opened for another type of operation
  - Opening a file with an invalid filename.
- *Error handling functions:*
  - I/O errors can be detected using two status-inquiry library functions: *feof()* and *ferror()*.

# Error Handling in Files – feof()

- *It is used to test for an end-of-file condition.*
- *It takes a FILE pointer as its only argument and returns a nonzero integer value if all of the data from the specified file has been read, and returns zero otherwise.*
- *If fp is a pointer to a file that has just been opened for reading, then the statement*  
*if(feof(fp))*  
*printf("End of data");*  
*would display the message "End of data" on reaching the end-of-file condition.*

# Error Handling in Files – *ferror()*

- *This function reports the status of the file indicated.*
- *It takes a FILE pointer as its argument and returns a nonzero integer if an error has been detected up to that point, during processing.*
- *It returns zero otherwise.*
- *So, the statement*  
*if(*ferror(fp)*!=0)*  
*printf("An error has occurred");*  
*Would print the error message, if the reading is not successful.*

# Error Handling Strategies

- *Addition of new records should take place at the end of existing records in the file. i.e. in the same way new records are added in a register manually.*
- *While modifying records, first we must ask the user which record he intends to modify. Instead of asking the record number to be modified, it would be more meaningful to ask for the name of the student whose record is to be modified. On modifying the record, the existing record gets overwritten by the new record.*
- *In deleting records, except for the record to be deleted, rest of the records must first be written to a temporary file, then the original file must be deleted, and the temporary file must be renamed back to original.*
- *Displaying all records means displaying the existing records on the screen. Naturally, records should be displayed from first record to last record.*

***END OF UNIT TEN***

# UNIT 11

***INTRODUCTION TO GRAPHICS***

# Syllabus

- **UNIT 11: Introduction to Graphics**
  - Concepts of Graphics (*graphics.h* header file)
  - Graphics Initialization and Modes (*graphics driver* & *graphics mode*)
  - Graphics Function (Basic functions of *graphics.h* – example: *line()*, *arc()*, *circle()*, *ellipse()*, *floodfill()*, *getmaxx()*, *getmaxy()*)

# Introduction

- *There are two modes of the standard output device:*
  - *Text Mode*
  - *Graphics Mode*
- *The programming for video games, animation and multimedia is difficult in text mode as they predominantly work with computer graphics.*
- *In graphics mode we work with tiny dots on the screen called pixels (picture elements).*

# Introduction

- *The pixels are even present in text mode as they are used to form characters that appear on the screen with only difference that they are pre-defined pattern of pixels.*
- *However, the graphics mode provides the ability to manipulate the individual pixels.*
- *To work with graphics, we can use **graphics.h** header file.*

# Graphics Characteristics – Pixels

- Short for picture element, a pixel is a single point (i.e. dot) in graphic image.
- Graphics monitor displays pictures by dividing the display screen into thousands of pixels arranged in rows and columns.
- The pixels are so close together that they appear connected.

# Graphics Characteristics – Pixels

- The computer screen is a two dimensional; each pixel on the screen has some location, illustrated by  $x$  and  $y$  values.
- $x$  is the horizontal offset from the left side of the screen.
- $y$  is the vertical offset from the top of the screen.
- So,  $x=0$  and  $y=0$  represent top left corner of the computer screen.

# Graphics Characteristics – Resolution

- *The number of pixels used on the screen is called resolution.*
- *There are fixed number of rows and each rows contains certain numbers of pixels.*
- *The frequently used resolutions supported by most of adapters are **640X480, 800X600, 1024X768, 1152X864, 1220X1024**, etc.*
- *The resolution **640X480 (640 by 480)** means that there are **640 pixels** in horizontal direction (i.e. x-axis) and **480 pixels** in vertical direction (i.e. y-axis).*
- *In general for higher resolution, the picture is more pleasing.*

# Graphics Characteristics – Colors

- *Some graphics modes support more colors than other ranging from 2 to million colors.*
- *A particular mode may support only two colors at a time while other may support 256 colors.*
- *These groups of colors are known as color palettes.*

# Graphics Characteristics – Adapters

- *Video adapters are drivers for display.*
- *Each video adapter handles graphics in different way.*
- *Once a video adapter is initialized by the program for particular graphics mode then it can use it to plot various elements as well as to display text in different fonts.*
- *Some examples of video adapters are **CGA** (Color Graphics Adapter), **VGA** (Video Graphics Array), and **EGA** (Enhanced Graphics Adapter).*

# Graphics Initialization & Modes

- Having known about adapters, now let us start knowing on how to start switching to graphics mode from text mode.
- In other words, how to start using pixel and resolution concepts.
- This is done by a function called `initgraph()`.
- This `initgraph()` takes in it 2 main arguments as input namely `gd` and `gm`. The 3<sup>rd</sup> argument specifies the path to driver.
- `gd` has the number of mode which has the best resolution.

# Graphics Initialization & Modes

- *This is very vital for graphics since the best resolution only gives a sharper picture.*
- *This value is obtained by using the function called as `getgraphmode()` in C graphics.*
- *The other argument `gm` gives insight about the monitor used, the corresponding resolution of that, the colors that are available since this varies based on adapters supported.*
- *This value is obtained by using the function named as `getmodename()` in C graphics.*

# Auto-Initialization of Graphics Hardware

- In the use of `initgraph()` function we have explicitly told `initgraph()` what graphics driver and mode to use by assigning the values to `driver` and `mode` arguments.
- It is possible to let the program to find out the video adapter installed in the computer and use the best driver and mode. i.e. the combination that gives the highest resolution.

# Auto-Initialization of Graphics Hardware

- There are two approaches for auto initialization of graphics hardware.
  - **DETECT** is used for driver argument. In this method program doesn't know in advance what mode will be used and cannot assume anything about the resolution. For example:

```
int gd, gm;  
gd=DETECT;  
initgraph(&gd, &gm, "C:\\TC\\BGI");
```

- A function called **detectgraph()** is used that returns value for the best driver and mode. For example:

```
int gd, gm;  
detectgraph(&gd, &gm);  
initgraph(&gd, &gm, "C:\\TC\\BGI");
```

# **Closing Graphic Mode**

- Once the program has finished its job using the graphics facilities, then it should restore the system to the mode that was previously in use.
- If graphics mode is not closed explicitly by the programmer, undesirable effects may be felt.
- The **closegraph()** function is used to restore the screen to the mode it was in before we called **initgraph()** and **deallocates** all the memory allocated by the graphics system.

# Graphics Function

- *There are numerous graphics functions available in C*
- *But let us see some to have an understanding of how and where a pixel is placed in each when each of the graphics function gets invoked.*

# Graphics Function

- **Function:** *putpixel(x, y, color);*
  - **Purpose:**
    - *The functionality of this function is it puts a pixel or in other words a dot at position x, y given in inputted argument.*
    - *Here one must understand that whole screen is imagined as a graph.*
    - *In other words, the pixel at the top left hand corner of the screen represents the value (0, 0).*
    - *Here, the color is the integer value associated with colors and when specified the picture element or the dot is placed with the appropriate color associated with that integer value.*

# Graphics Function

- **Function:** `line(x1, y1, x2, y2);`
  - **Purpose:**
    - *The functionality of this function is to draw a line from  $(x1, y1)$  to  $(x2, y2)$ .*
    - *Here also the coordinates are passed taking the pixel  $(0, 0)$  at the top left hand corner of the screen as the origin.*
    - *And also one must note that the line formed is by using number of pixels placed near each other.*

# Graphics Function

- **Function:** *getpixel(x, y);*
- **Purpose:**
  - *This function when invoked gets the color of the pixel specified.*
  - *The color got will be integer value associated with that color and hence the function gets an integer value as return value.*

COLOR	INT VALUES
BLACK	0
BLUE	1
GREEN	2
CYAN	3
RED	4
MAGENTA	5
BROWN	6
LIGHTGRAY	7
DARKGRAY	8
LIGHTBLUE	9
LIGHTGREEN	10
LIGHTCYAN	11
LIGHTRED	12
LIGHTMAGENTA	13
YELLOW	14
WHITE	15

# Graphics Function

- **Function:** `circle(x, y, r);`
  - **Purpose:**
    - Draws a circle having center point  $(x, y)$  and radius  $r$  with current color.
- **Function:** `ellipse(x, y, startAngle, endAngle, xRadius, yRadius);`
  - **Purpose:**
    - Draws an ellipse with current color.
- **Function:** `arc(x, y, startAngle, endAngle, radius);`
  - **Purpose:**
    - Draws a circular arc in a portion of circle.

# Graphics Function

- **Function:** `rectangle(x1, y1, x2, y2);`
  - **Purpose:**
    - Draws rectangle from two end points of a diagonal of the rectangle.
- **Function:** `getmaxx();`
  - Returns max `x` value for current graphics driver and mode.
- **Function:** `getmaxy();`
  - Returns max `y` value for current graphics driver and mode.

# Example - 1

```
//Drawing a Circle
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
void main()
{
    int gd= DETECT, gm;
    initgraph(&gd,&gm, "c:\\tc\\bgi");
    circle(200,100,10);
    setcolor(WHITE);
    getch();
    closegraph();
}
```

## Example – 2

```
//Drawing Lines
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
void main()
{
    int gd= DETECT, gm;
    initgraph(&gd,&gm, "c:\\tc\\bgi");
    line(90,70,60,100);
    line(200,100,150,300);
    setcolor(WHITE);
    getch();
    closegraph();
}
```

# Example – 3

```
// Drawing rectangle
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
void main()
{
    int gd= DETECT, gm;
    initgraph(&gd, &gm, "c:\\tc\\bgi");
    rectangle(200,100,150,300);
    setcolor(WHITE);
    getch();
    closegraph();
}
```

# Example – 4

*// Constructing Triangle*

```
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
void main()
{
    int gd= DETECT, gm;
    initgraph(&gd, &gm, "c:\\tc\\bgi");
    line(200, 100, 10, 20);
    line(10, 20, 50, 60);
    line(50, 60, 200, 100);
    setcolor(WHITE);
    getch();
    closegraph();
}
```

***END OF UNIT ELEVEN***

... **BEST WISHES** ...

**KEEP LEARNING, HOW TO CODE ☺**