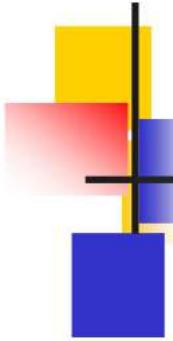


# **PROBLEM SOLVING USING C PROGRAMMING**

**Bachelor of Computer Application  
SEMESTER - I**

**GURU NANAK COLLEGE(Autonomous)  
VELACHERY ROAD, CHENNAI – 600042  
(Re-Accredited 'A' grade by NAAC)**



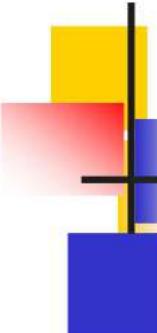
# Syllabus

## ■ **Planning the Computer Program:**

- Problem definition,
- Program design,
- Debugging,
- Types of Errors in programming,
- Techniques of Problem Solving: Flowcharting, Algorithms.

## ■ **C Fundamentals:**

- Character set
- Identifiers and Keywords
- Data Types
- Constants - Variables
- Declarations - Expressions - Statements
- Operators: Arithmetic, Unary, Relational and Logical, Assignment and Conditional.



# Related Books

---

## Recommended Books:

- P. K. Sinha & Priti Sinha, "Computer Fundamentals", BPB Publications, 6<sup>th</sup> Edition.
- Dr. Anita Goel, Computer Fundamentals, Pearson Education, 2010.
- E. Balaguruswamy, 2016, 7<sup>th</sup> Edition, Programming in ANSI C, TMH Publishing Company Ltd.
- Kanetkar Y., 1999, Let us C, BPB Pub., New Delhi.

## Reference Books:

- K.R.Venugopal,Programming with C,1997,McGraw-Hill
- Varalakshmi,Programming using C,2000(Reprint July 2001), V.Ramesh5
- R.Rajaram,C Programming Made Easy,V.Ramesh
- B.W. Kernighan and D.M.Ritchie, 1988, The C Programming Language, 2<sup>nd</sup> Edition, PHI.
- H. Schildt, C,2004, The Complete Reference, 4<sup>th</sup> Edition, TMH
- Gottfried. B.S, 1996, Programming with C, Second Edition, TMH Pub. Co. Ltd., New Delhi.

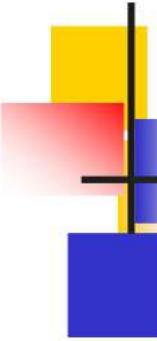
## Websites:

- <http://www.cprogramming.com/>
- <http://www.richardclegg.org/previous/ccourse/>



## 1.1 PROBLEM DEFINITION

- The computer is the symbol- manipulating machine that follows the set of instructions called a ***program***. Any computing has to be performed independently without depending on the programming language and the computer.
- The problem solving techniques involves the following steps
  - Define the problem.
  - Formulate the mathematical model.
  - Develop an algorithm.
  - Write the code for the problem.
  - Test the program.



# 1.1 PROBLEM DEFINITION

---

## 1) Define the Problem

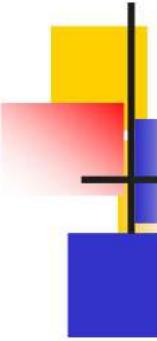
- A clear and concise problem statement is provided.
- The problem definition should specify the input and output.
- Full knowledge about the problem is needed.

**Example:** TO FIND THE AVERAGE OF TWO NUMBERS.

## 2) Formulate the Mathematical Problem

- Any technical problem provided can be solved mathematically.
- Full knowledge about the problem should be provided along with the underlying mathematical concept.

**Example:**  $(\text{data1}+\text{data2})/2$



# 1.1 PROBLEM DEFINITION

---

## 3) Develop an Algorithm

- An algorithm is the sequence of operations to be performed.
- It gives the precise plan of the problem.
- An algorithm can be of flowchart or pseudo code. Example:

*Problem Definition:*

TO FIND THE AVERAGE OF TWO NUMBERS.

*Algorithm:*

1. Set the sum of the data values to 0.
2. Set the count of the data values to zero.
3. As long as the data values exist, add the next data value to the sum and add 1 to the count.
4. To compute the average, divide the sum by the count.
5. Print the average.

*Task:*

To find the average of 20 and 30 manually.

$$20 + 30 = 50 ; 50/2 = 25.$$



# 1.1 PROBLEM DEFINITION

---

## 4) Write the Code for the Problem

- The algorithm developed must be converted to any programming language.
- The compiler will convert the program code to the machine language which the computer can understand.

## 5) Test the Program

- Testing involves checking errors both syntactically and semantically.
- The errors are called as "*bugs*".
- When the compiler finds the bugs, it prevents compiling the code from programming language to machine language.
- Check the program by providing a set of data for testing.



## 1.2 PROGRAM

---

- A ***program*** consists of a series of instructions that a computer processes to perform the required operation.
- Set of computer programs that describe the program are called ***software***.
- The process of software development is called ***Programming*** and the person who develops the computer programs are called ***Programmer***.
- Thus, in order to design a program, a programmer must determine three basic requirements:
  - The instructions to be performed.
  - The order in which those instructions are to be performed.
  - The data required to perform those instructions.

## 1.2 PROGRAM

### Example:

- WRITE A PROGRAM TO ADD TWO NUMBERS

- Input two numbers.
  - Add these two numbers.
  - Display the output.

- Suppose we want to calculate the sum of two numbers, A and B, and store the sum in C, here A and B are the inputs, addition is the process, and C is the output of the program.

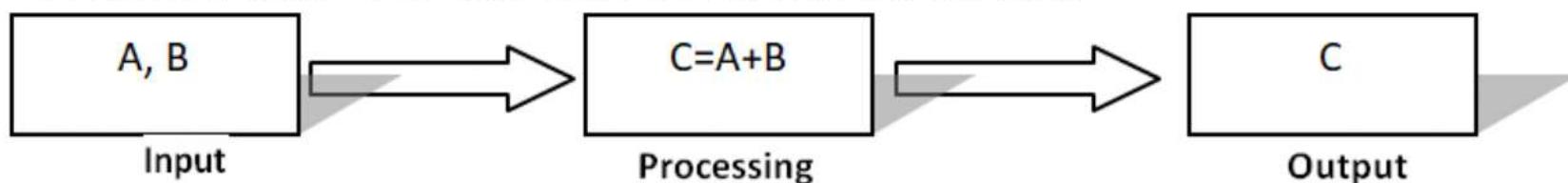
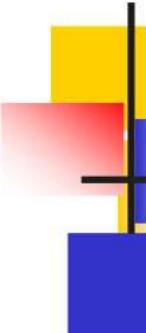


Fig Processing of the data

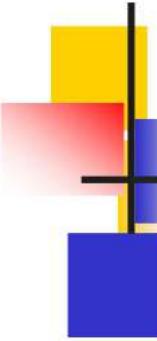


## 1.2 PROGRAM

---

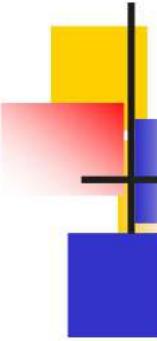
### Characteristics of a Program

- Any computer program should have the following characteristics
- **Accuracy of calculations:** Any calculation made in the program should be correct and accurate.
- **Clarity:** It refers to the overall readability of the program which helps the user to understand the underlying program logic easily without much difficulty.
- **Modularity:** When developing any program, the task is sub divided into several modules or subtasks. These modules are developed independently (i.e.) each task does not depend on the other task.
- **Portability:** Portability is defined as the ability to run the application program on different platforms.



## 1.2 PROGRAM

- **Flexibility:** Any program written should be flexible (i.e.) the program should be developed in such a way that it can handle the most of the changes without rewriting the entire program.
- **Efficiency:** Any program needs certain memory and processing time to process the data. This memory and processing unit should be of least amount. This is the efficiency of the program.
- **Generality:** The program should be in general. If a program is developed for a specific task then it can be used for all the similar tasks in the same domain.
- **Documentation:** any application program developed should be well documented such that even in the absence of the developer and the author, the programmers could be able to understand the concept behind it.



## 1.3 PROGRAM DEVELOPMENT CYCLE

- Any program has to be broken into a series of smaller steps.
- These series are independent of programming language.
- The programmer should have wide knowledge about the problem and the way to solve it.
- Generally any problem solving involves
  - Defining the problem.
  - Understanding the problem.
  - Providing the solution.



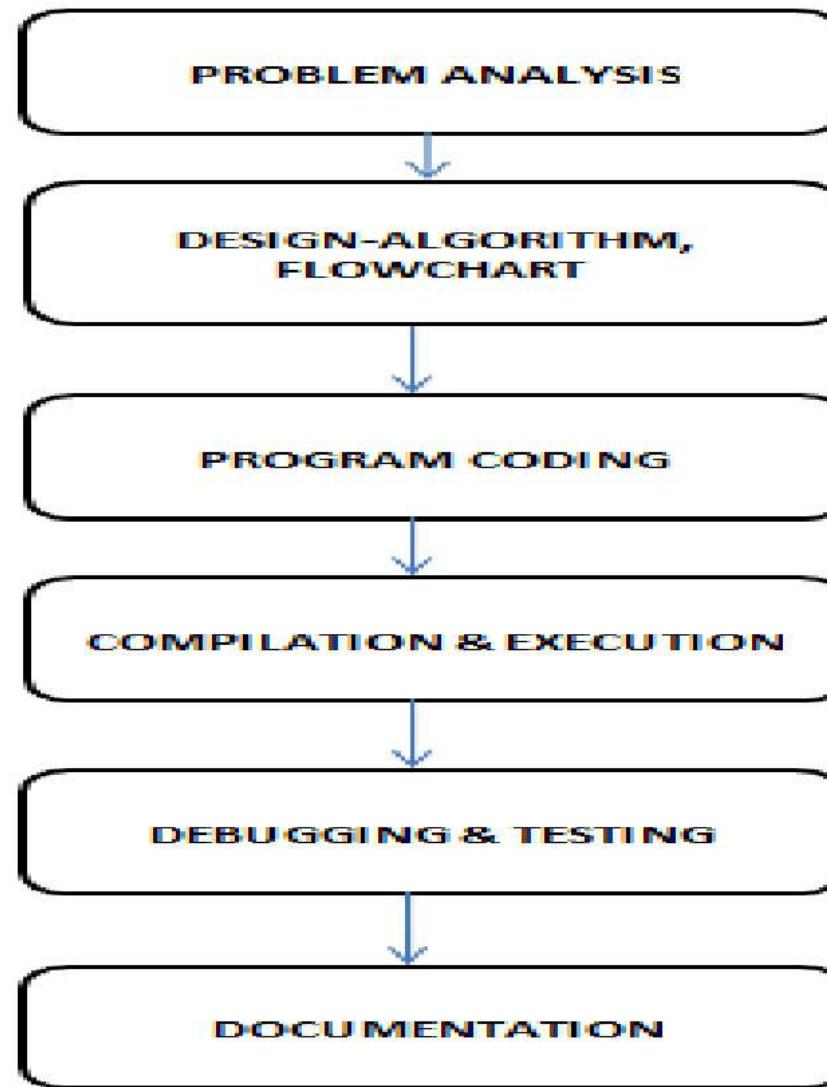
## 1.3 PROGRAM DEVELOPMENT CYCLE

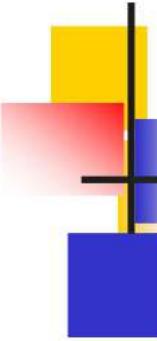
---

- ***The program development cycle involves***

- Problem Analysis.
- Design- Algorithm and Flowchart development.
- Program coding.
- Program compilation and execution.
- Program debugging and Testing.
- Documentation.

# 1.3 PROGRAM DEVELOPMENT CYCLE





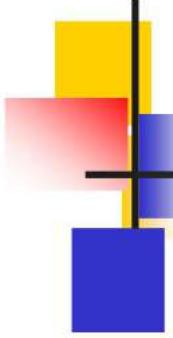
# 1.3 PROGRAM DEVELOPMENT CYCLE

## 1. Problem Analysis

- Define the problem by identifying the input and output of it.
- Variable name is assigned for each item.
- After analyzing, the programmer has to develop various solutions for the given problem.
- Optimal solution is taken from the set of solutions obtained.

## 2. Design – Algorithm & Flowchart Development

- Algorithm and flowchart are developed to provide a sequence of actions to be performed.
- Algorithm provides a basic logic in solving the problem by providing sequence of instructions.
- Algorithm can be of
  - Flow chart
  - Pseudo Code.
- ***Program Design Language (PDL)*** : It has no specific standard rules for defining the PDL statements. PDL is independent of any programming language. It is also called as Pseudo Code.



# 1.3 PROGRAM DEVELOPMENT CYCLE

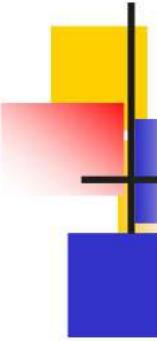
---

## 3. Program Coding

- Code the algorithm in the selected programming language.
- The processes of translating the algorithm or the flowchart into an exact instruction that will make up the program are called ***program coding***.

## 4. Program Compilation and Execution

- After program coding, the program has to be compiled and executed.
- During compilation, if no error is produced, then the program is executed successfully.
- If errors are available, then the errors are displayed in the terminal, and corrected later with correct syntax and then compiled.



# 1.3 PROGRAM DEVELOPMENT CYCLE

## 5. Program Debugging and Testing

- Errors are called as “**bugs**”
- Errors can be categorized as follows
  - *Syntax errors*(during compilation).

**Example:** Program does not compile, missing bracket, bad punctuation.

- *Run time* (during execution)

**Example:** Program crashes, Check input data.

- *Logical* (incorrect or illogical answers) Example: Program runs and give wrong output.

## 6. Documentation

- Once the programmer is free from the errors, it is the duty of the programmer to document all the necessary documents which is provided to the program users as manual.
- Helps the user to operate correctly.



## 1.4 ALGORITHM

---

- Algorithms are one of the most basic tools that are used to develop the problem solving logic.
- An algorithm is defined as a finite sequence of explicit instructions that, when provided with a set of input values produces an output and then terminates.
- In algorithm, after a finite number of steps, solution of the problem is achieved.
- Algorithms can have steps that repeat (iterate) or require decisions (logic and comparison) until the task is completed.
- Different algorithms may accomplish the same task, with a different set of instructions, in more or less the same time, space, and efforts.



## 1.4 ALGORITHM

---

### **EXAMPLE:**

- To determine the largest number out of three numbers A, B, and C
  - Step 1: Start
  - Step 2: Read three numbers say A, B, C
  - Step 3: Find the larger number between A and B and store it in MAX\_AB
  - Step 4: Find the larger number between MAX\_AB and C and store it in MAX Step 5: Display MAX
  - Step 6: Stop
- The above-mentioned algorithm terminates after six steps. This explains the feature of finiteness. Every action of the algorithm is precisely defined; hence, there is no scope for ambiguity. Once the solution is properly designed, the only job left is to code that logic into a programming language.



## 1.4 ALGORITHM

---

### Characteristics of Algorithm

- An algorithm has five main characteristics:
  - It should have finite number of inputs.
  - Terminates after a finite number of steps.
  - Instructions are precise and unambiguous.
  - Operations are done exactly and in a finite amount of time.
  - Outputs are derived from the input by applying the algorithm



## 1.4 ALGORITHM

---

### Quality of Algorithm

- There are few factors that determine the quality of a given algorithm
- An algorithm should be relatively fast.
- Any algorithm should require minimum computer memory to produce the desired output in an acceptable amount of time.

# 1.5 FLOWCHART

- Flowchart is a diagrammatic representation of an algorithm that illustrates the sequence of operations to be performed to get a solution.
- The different boxes are interconnected with the help of arrows.
- The boxes represent operations and the arrows represent the sequence in which the operations are implemented.
- The primary purpose of the flowchart is to help the programmer in understanding the logic of the program.

Symbol	Symbol Name	Description
	Flow Lines	Flow lines are used to connect symbols. These lines indicate the sequence of steps and the direction of flow of control.
	Terminal	This symbol is used to represent the beginning (start), the termination (end), or halt (pause) in the program logic.
	Input/output	It represents information entering or leaving the system, such as customer order (input) and servicing (output).

# 1.5 FLOWCHART

	Processing	Process symbol is used for representing arithmetic and data movement instructions. It can represent a single step ('add two cups of flour'), or an entire sub-process ('make bread') within a larger process.
	Decision	Decision symbol denotes a decision (or branch) to be made. The program should continue along one of the two routes (IF/ELSE). This symbol has one entry and two exit paths. The path chosen depends on whether the answer to a question is yes or no.
	connector	Connector symbol is used to join different flow lines.
	Off-page Connector	This symbol is used to indicate that the flowchart continues on the next page.
	Document	Document is used to represent a paper document produced during the flowchart process.
	Manual Input	Manual input symbol represents input to be given by a developer / programmer.
	Manual Operation	Manual operation symbol shows that the process has to be done by a developer/programmer.
	Online Storage	This symbol represents the online data storage such as hard disks, magnetic drums, or other storage devices.
	Communication Link	Communication link symbol is used to represent data received or to be transmitted from an external system
	Magnetic Disk	This symbol is used to represent data input or output from and to a magnetic disk.



## 1.5 FLOWCHART

### ***Guidelines for preparing flowcharts***

- The following guidelines should be used for creating a flowchart:
  - The flowchart should be clear, neat, and easy to follow.
  - The flowchart must have a logical start and finish.
  - In drawing a proper flowchart, all necessary requirements should be listed in logical order.
  - Only one flow line should come out from a process symbol.
  - Only one flow line should enter a decision symbol. However, two or three flow lines (one for each possible answer) may leave the decision symbol.
  - Only one flow line is used with a terminal symbol.
  - In case of complex flowcharts, connector symbols are used to reduce the number of flow lines.



## 1.5 FLOWCHART

---

### Benefits of Flowcharts

- A flowchart helps to clarify how things are currently working and how they could-be improved. The reasons for using flowcharts as a problem-solving tool are given below.

- i. ***Makes Logic Clear.***
- ii. ***Communication.***
- iii. ***Effective Analysis.***
- iv. ***Useful in Coding.***
- v. ***Proper Testing and Debugging.***
- vi. ***Appropriate Documentation.***



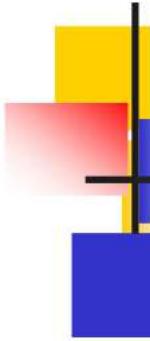
## 1.5 FLOWCHART

---

### Limitations of Flowcharts

- Flowchart can be used for designing the basic concept of the program in pictorial form but cannot be used for programming purposes. Some of the limitations of the flowchart are given as follows:

- i. ***Complex.***
- ii. ***Costly.***
- iii. ***Difficult to Modify.***
- iv. ***No Update.***



## 1.6 C Fundamentals

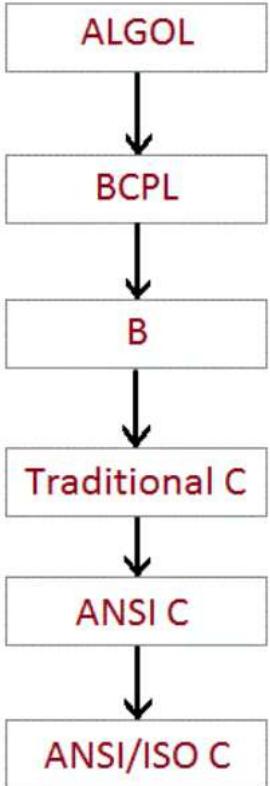
### ■ C Programming,

- Was developed by Dennis Ritchie at AT&T Bell Labs, USA in 1972.
- Is a high-level programming language used to develop applications for high-level business programs and low-level system programs.
- Became popular because of its power, simplicity and ease of use.
- Enables system program writing, using pointers.
- It is reliable, simple and easy to use.

# 1.6 C Fundamentals

- C language has evolved from three different structured language ALGOL, BCPL and B Language. It uses many concepts from these languages and introduced many new concepts such as data types, struct, pointer.

<u>Year</u>		<u>Developed by</u>	<u>Drawbacks</u>
1960	ALGOL	International Group	Too general and too abstract
1967	BCPL	Martin Richard	Less Powerful and too Specific
1970	B	Ken Thompson	Too Specific
1973	Traditional C	Dennis Ritchie	
1989	ANSI C	ANSI committee	
1990	ANSI/ISO C	ISO committee	





## 1.6 C Fundamentals

---

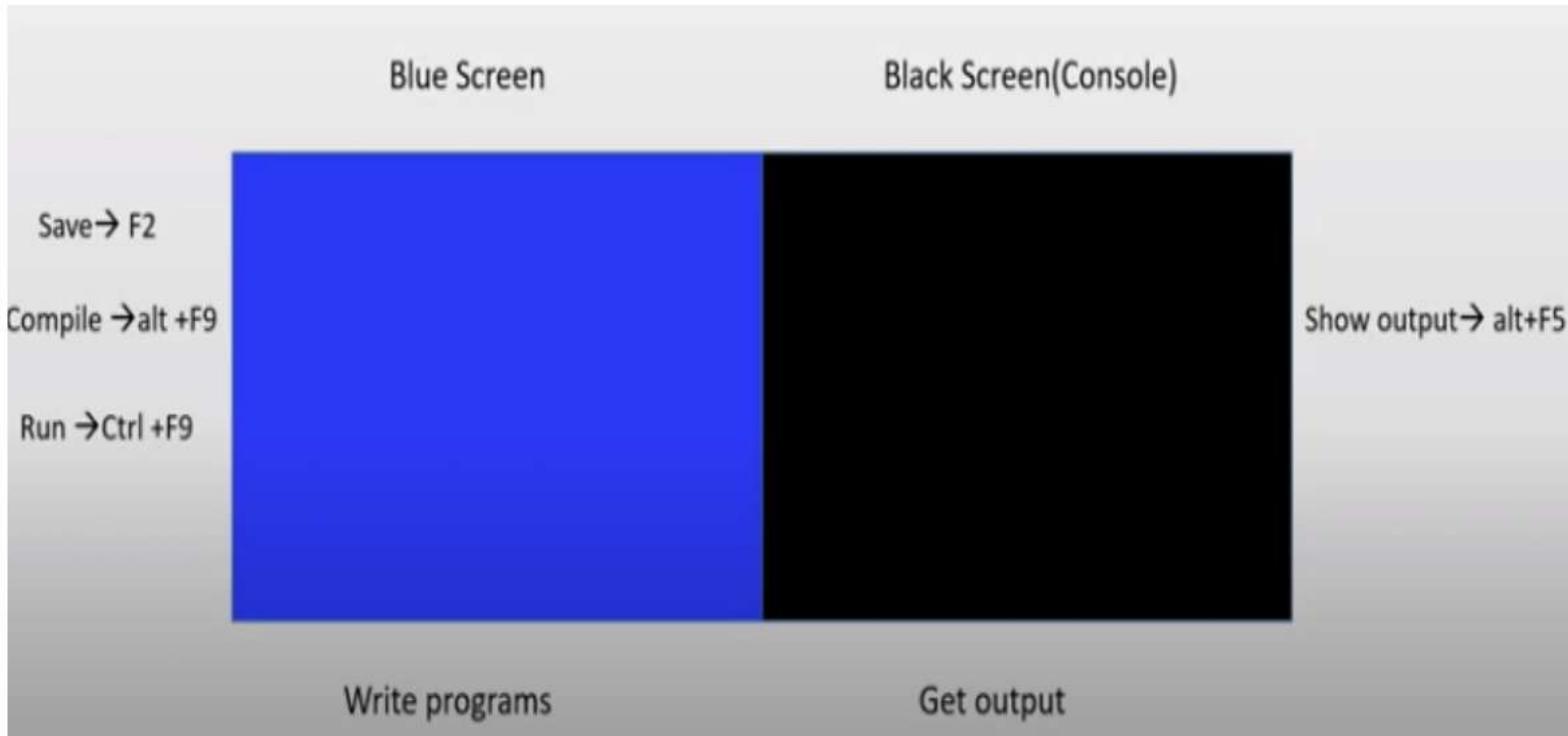
### ■ Features of C

- Robust language, which can be used to write any complex program.
- Has rich set of built-in functions and operators.
- Well-suited for writing both system software and business applications.
- Efficient and faster in execution.
- Highly portable.
- Well-suited for structured programming.
- Dynamic memory allocation.

# 1.6 C Fundamentals

## ■ First C Program

- Initially, learn how to write, compile and run the C program.
- To write the first C program, open the C console.



# 1.6 C Fundamentals

## ■ Example of C Program:

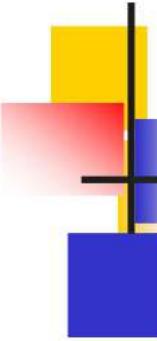
```
#include<stdio.h>
Void main()
{
    printf("Good Morning!");
}
```

The screenshot shows the TURBO C IDE interface. The menu bar includes File, Edit, Search, Run, Compile, Debug, Project, Options, Window, and Help. A toolbar below the menu bar has buttons for F1 Help, F2 Save, F3 Open, Alt-F9 Compile, F9 Make, and F10 Menu. The main window displays the C code:

```
#include<stdio.h>
void main()
{
    printf("Good Morning!");
}
```

The status bar at the bottom shows "1:1" and "Message".

The screenshot shows the DOSBox terminal window. The title bar reads "DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program...". The command prompt is "C:\TURBOC3\BIN>TC" and the output is "Good Morning!".



## 1.6 C Fundamentals

---

- **Header file is included**
  - **#include <stdio.h>** includes the **standard input output** library functions.
  - The printf() function is defined in stdio.h .
- **main()** The **main() function is the entry point of every program** in c language.
- **printf()** The printf() function is **used to print data** on the console.

# 1.6 C Fundamentals

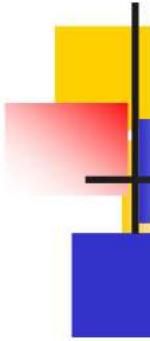
## ■ What is Compilation?

- The compilation is a process of converting the source code into object code. It is done with the help of the compiler.
- The compiler checks the source code for the syntactical or structural errors, and if the source code is error-free, then it generates the object code.

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



```
0100000000000000
0111111111111111
01010101101010
00000011111111
00000011111111
00000010101011
```



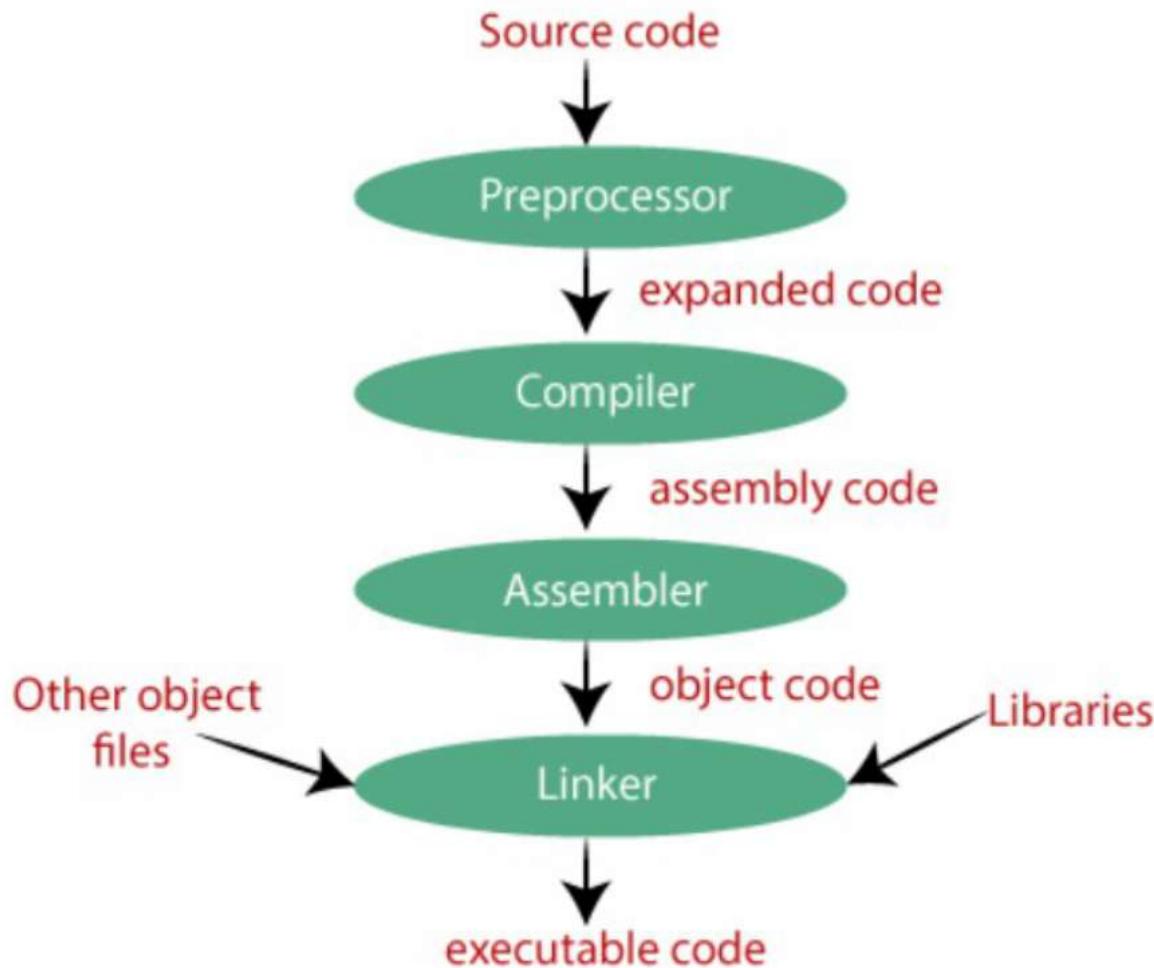
# 1.6 C Fundamentals

## ■ Stages of Compilation

- Like most high-level languages, C also uses compiler to convert its source code (files with the extension .c) to object code (files with the extension .obj).
- The object code will be link-edited by the linker to form the machine language also known as executable codes (files with the extension .exe)
- The compilation process can be divided into four steps
  - **Pre-processor**
  - **Compiler**
  - **Assembler**
  - **Linker**

# 1.6 C Fundamentals

## ■ Stages of Compilation





# 1.6 C Fundamentals

---

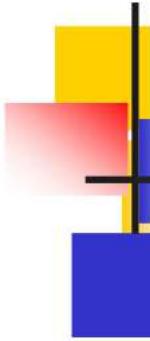
## Stages of Compilation

### ■ Preprocessor

- The **source code** is the code which is written in a text editor and the source code file is given an extension ".c".
- This source code is first passed to the preprocessor, and then the preprocessor expands this code. After expanding the code, the expanded code is passed to the compiler.

### ■ Compiler

- The code which is expanded by the preprocessor is passed to the compiler.
- The compiler converts this code into **assembly code**. Or we can say that the C compiler converts the pre-processed code into assembly code.



## 1.6 C Fundamentals

---

### Stages of Compilation

#### ■ Assembler

- The assembly code is converted into ***object code*** by using an assembler.
- The name of the object file generated by the assembler is the same as the source file.
- The extension of the object file in DOS is '.obj,' and in UNIX, the extension is 'o'.
- If the name of the source file is '**hello.c**', then the name of the object file would be 'hello.obj'.



# 1.6 C Fundamentals

---

## Stages of Compilation

### ■ Linker

- Mainly, all the programs written in C use library functions.
- These library functions are pre-compiled, and the object code of these library files is stored with '.lib' (or '.a') extension.
- The main working of the linker is to combine the object code of library files with the object code of our program.
- The output of the linker is the ***executable file***.
- The name of the executable file is the same as the source file but differs only in their extensions.
- In DOS, the extension of the executable file is '.exe'
- For example, if we are using printf() function in a program, then the linker adds its associated code in an output file.



## 1.6 C Fundamentals

---

### ■ The Structure of a C Program

- Documentation Section
- Header File section
- Definition Section
- Global declaration section
- main( ) section
- Declaration part
- Execution part
- Sub program section



# 1.6 C Fundamentals

## ■ Example:

```
/* Documentation Section */
// File      : Addition.c
// Description      : Addition of Three Numbers
// Author   : Student123

/* Header File Section */ #include<stdio.h> #include<conio.h>
/* Definition Section */
#define c 3
/* Global declaration section */
int calcsum(int,int,int);
/* main( ) section */
int main()
{
    /* Declaration part */
    int a,b,sum;
    /* Execution part */ printf("Enter Two numbers"); scanf("%d %d", &a, &b); sum=calcsum(a,b,c);
    printf("The sum is: %d", sum);
}

/* Sub program section*/
int calcsum(int x,int y, int z)
{
    int d; d=x+y+z; return d;
}
```



## 1.7 C Character Set

- Character set denotes alphabet, digit or special character. Characters combine to form variables.
- Characters in C are grouped into Letters, Digits, Special characters and White spaces. Compiler generally ignores white space when it is not a part of string constant.
- White Space may be used to separate words, but not used between characters of keywords or identifiers.
- The character set in C Language can be grouped into the following categories.
  1. Letters
  2. Digits
  3. Special Characters
  4. White Spaces



## 1.7 C Character Set

### ■ Alphabets

Letters	Letters	Digits
Upper Case A to Z	Lower Case a to z	0 to 9

### ■ Escape Sequences (White Space Characters)

- A character constant represents a single character. It is also called as “Backslash Character Constant”.
- It is used together with input and output statements. It has no meaning but it has the control to decide the way an input has to be displayed.

1. Blank Space	\b	2. Horizontal Tab	\t
3. Carriage Return	\r	4. New line	\n
5. Form Feed	\f	6. Vertical tab	\v

# 1.7 C Character Set

## ■ Special Characters

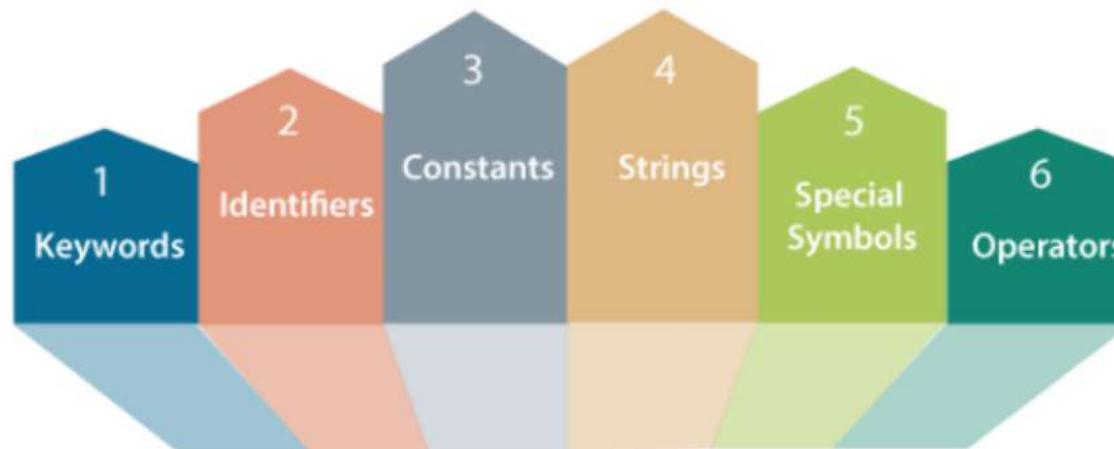
&	.Ampersand	#	.Number Sign
'	.Apostrophe	<	.Opening Angle (Less than sign)
*	.Asterisk	.	.Period ( Dot )
@	At symbol	%	.Percentage Sign
\	.Backslash	+	.Plus Sign
^	.Caret	?	.Question Mark
>	.Closing Angle (Greater than sign)	"	.Quotation Marks
:	.Colon	}	.Right Flower Brace
,	.Comma	)	.Right Parenthesis
\$	.Dollar Sign	]	.Right Bracket
=	Equal to	;	.Semicolon
!	.Exclamation Mark	/	.Slash
(	.Left Parenthesis	~	.Tilde
[	.Left Bracket	_	.Underscore
{	.Left Flower Brace		.Vertical Bar

# 1.8 C TOKENS

## ■ C TOKENS

- Among the group of text individual word, punctuation marks are called Tokens.
- It is a smallest individual unit in a C program.

## ■ Classification of tokens in C



Classification of C Tokens



# 1.9 Keywords

## ■ **Keywords**

- Every word in C language is a keyword or an identifier.
- Keywords in C language cannot be used as a variable name.
- The compiler specifically uses them for its own purpose and they serve as building blocks of a c program.
- The following are the Keyword set of C language.

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



## 1.10 Identifiers

---

### ■ Identifiers

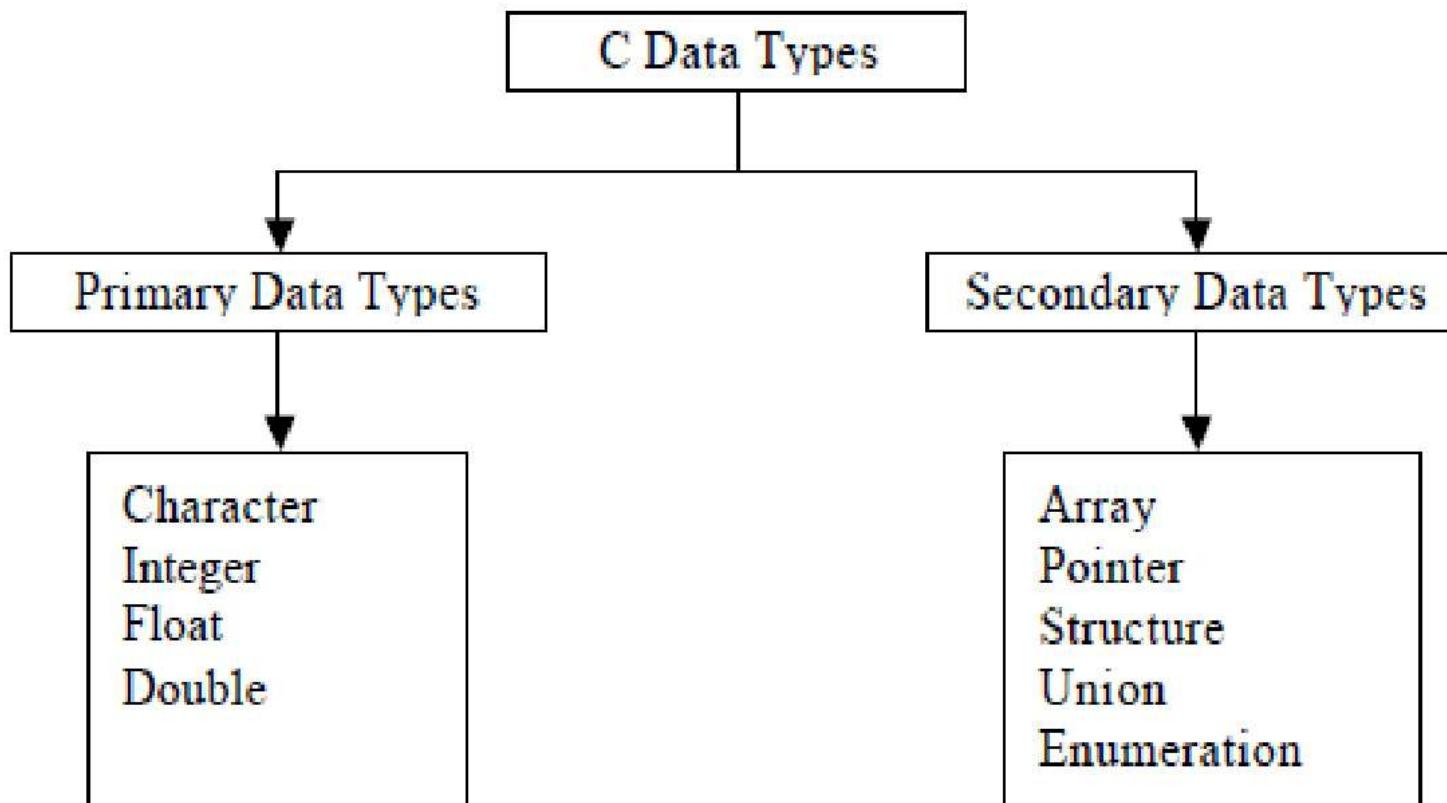
- Identifier is a name given to program elements such as variables, functions, procedure, arrays and soon.
- First character must be an Alphabet or Underscore.
- Identifier consists of sequence of letters, digits or combination of both.

### ■ Rules for Identifier

- First character must be an Alphabet or Underscore( \_ ).
- Special characters and embedded commas are not allowed.
- First 31 characters are given high priority or preference.
- Keyword cannot be used as Identifier.
- There should not be any white space.
- Both uppercase and lowercase letters are permitted.
- The underscore character is also permitted in identifiers.

# 1.11 Data Types

- Data Type is used to define the type of value to be used in a Program.
- Based on the type of value specified in the program specified amount of required Bytes will be allocated to the variables used in the program.





# 1.11 Data Types

## ■ Primary Data Type

- Integers are represented as int.
- Character are represented as char.
- Floating point value are represented as float, double precision floating point are represented as double and finally void are primary data types.
- Primary data type offers extended data types.
- longint, longdouble are extended data types.

### i. Character Data Type

- Normally a single character is defined as char data type. It is specified by the keyword char. Char data type uses 8 bits for storage. Char may be signed char or unsigned char.

Name	C Representation	Size (bytes)	Range	Format Delimiter
Character	char	1	-128 to 127	%c
Signed Character	signed char	1	-128 to 127	%c
Unsigned Character	unsigned char	1	0 to 255	%c

# 1.11 Data Types

## ii. Integer Data Type

- Integer data type can store only the whole numbers.

Name	C Representation	Size (bytes)	Range	Format Delimiter
Integer	int	2	-32768 to 32767	%d
Short Integer	short int / short	2	-32768 to 32767	%d
Long Integer	long int / long	4	-2147483648 to 2147483647	%ld
Signed Integer	signed int	2	-32768 to 32767	%u
Unsigned Integer	unsigned int	2	0 to 65535	%d
Signed Short Integer	unsigned short int / short	2	-32768 to 32767	%d
Unsigned Short Integer	unsigned short int / short	2	0 to 65535	%u
Signed Long Integer	signed long int / long	4	-2147483648 to 2147483647	%ld
Unsigned Long Integer	unsigned long int / long	4	0 to 4294967295	%lu



## 1.11 Data Types

### iii. Floating Point Data Type

- Floating Point data types are also known as Real Numbers. It can store only the real numbers (decimal numbers) with 6 digits precision.

Name	C Representation	Size (bytes)	Range	Format Delimiter
Float	float	4	3.4 e-38 to 3.4 e+38	%f
Double	double	8	1.7e-308 to 1.7e+308	%f
Long Double	long double	10	3.4 e-4932 to 3.4 e+4932	%lf

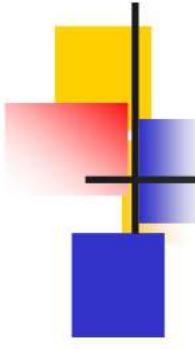


## 1.12 Constants

---

### ■ Constant

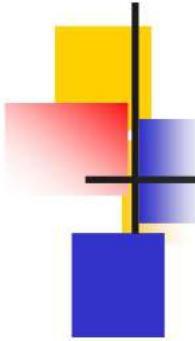
- Constant is a fixed value that doesn't or does not change during the execution of a program.
- It remains same throughout the program. Its value cannot be altered or modified in a program.
- A variable is declared by the Keyword const. Constant is classified into three main types.
- They are described as follows.
  - i. Numeric Constant
  - ii. Character Constant
  - iii. Symbolic Constant.



# 1.12 Constants

## i. Numeric Constant

- Numeric Constant is further classified into two main types. They are
  - **Integer Constants**
    - Integer constant consist of sequence of digit without decimal point. It is normally a whole number.
  - **Real Constants**
    - Real Constants are otherwise known as Floating Point Constants.
    - Real Constant consists of decimal number, exponent or combination of both.
    - Real constant consist of two forms. They are Fractional form and Exponential form.



# 1.12 Constants

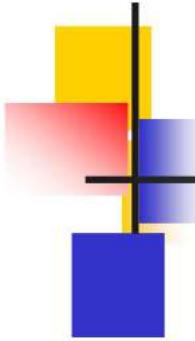
## ii. Character Constant

- Character Constant is also known as single character constant. The Character constant is mainly classified into three types. They are

- Single character constants**

- Single character constant is also known as character constant.
- It contains single character or number enclosed within a pair of single quotes.
- If number specified within single quotes it is considered as character constant.
- Each character is stored in one byte or length of single character constant is one.

<b>Example:</b>	<b>Valid</b>	'A'	'm'	' '	'3'
	<b>Invalid</b>	"A"	"m"		



# 1.12 Constants

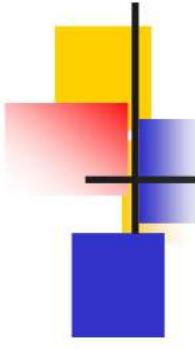
## ii. Character Constant

- Character Constant is also known as single character constant. The Character constant is mainly classified into three types. They are

- Single character constants**

- Single character constant is also known as character constant.
- It contains single character or number enclosed within a pair of single quotes.
- If number specified within single quotes it is considered as character constant.
- Each character is stored in one byte or length of single character constant is one.

<b>Example:</b>	<b>Valid</b>	'A'	'm'	' '	'3'
	<b>Invalid</b>	"A"	"m"		



# 1.12 Constants

---

## ii. Character Constant

### ■ String constant

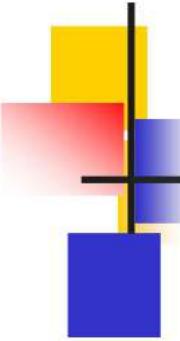
- String constant consist of sequence of characters enclosed with double quotes.
- The character may be letters, numbers, special characters and blank space.

### ■ Rules for String constant

- String constant must be a single alphabet, special character or sequence of alphabets or digits enclosed in double quotes.
- Every string constant ends with NULL character. It is automatically assigned.

■ **Example:**      **Valid**    "I am a Student"

■ **Invalid**            'I am a Student'



# 1.12 Constants

## iii. **Symbolic Constant**

- It represent numeric/character string constant. It is defined as follows.

```
#define<scname><sctext>.
```

- Here scname stands for Symbolic Constant name and sctext stands for Symbolic Constant text.
- scname is usually written in capital letters.

**Example :-**

```
#define PI 3.14  
const int pi=3.14
```



## 1.13 Variables

- Variable is a name given to Memory location in which data is stored.
- Variable acts as value, which changes during the execution of program.
- Variable may take different value at different times during execution.
- The general format of any declaration

*datatype v1, v2, v3, ... vn*

- where v1, v2, v3 are variable names. Variables are separated by commas.
- A declaration statement must end with a semicolon.

int sum;

double average, mean;



# 1.13 Variables

---

## 1. Variable Assignment

- It is another format of variable declaration. The general form is
  - *data-type variable name [=value];*
- Here data type refers to the type of value used in the C program. Variable name is a valid name.

**Example:**      int a;    (or)    int a=10;

- Here if the value of a is assigned 10 the value remains the same throughout the program. If no value is specified then the value keeps on changing.

## 2. Simple Variable Assignment

- To assign a single value to a variable '=' equality operator is used. The syntax is

*variable-name = value;*

**Example:**      X=10; y=30;



# 1.13 Variables

### 3. Compound Variable Assignment

- This type of variable declaration can be used together with combination of characters, numbers or an expression. Here '=' equality operator is used.

*variable-name = expression*

**Example:** `Y=x+5;`      `Z=((x+1)(y-2)*x)`

### 4. Declaring a variable as constant

- Variable is declared as constant by using the keyword or qualifier '**const**' before the variable name. This can be done at the time of initialization.

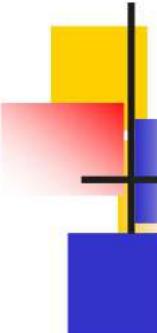
**Example:**      `Const int class_size = 40;`

### 5. Volatile Variable

- A variable is volatile if the value gets changed at any time by some of the external sources.

**Example:**      `volatile int num;`

- when we declare a variable as volatile the compiler will examine the value of the variable each time it is encountered to see if an external factor has changed the value.

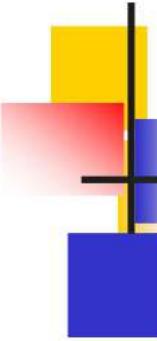


## 1.14 Declaration in C

- A *declaration* is a C language construct that introduces one or more identifiers into the program and specifies their meaning and properties.
- Declarations may appear in any scope. Each declaration ends with a semicolon (just like a statement) and consists of two distinct parts:  
***specifiers-and-qualifiers declarators-and-initializers ;***
- specifiers-and-qualifiers:
  - void
  - the name of an arithmetic type
  - the name of an atomic type
  - a name earlier introduced by a typedef declaration
  - struct, union, or enum specifier
- declarators-and-initializers
  - Declarators may be accompanied by initializers.
- **Example:**

```
int a, *b=NULL;
```

// "int" is the type specifier, // "a" is a declarator // "\*b" is a declarator and  
NULL is its initializer



## 1.15 Expression in C

- An expression followed by a semicolon is a statement.  
***expression(optional);***
- Most statements in a typical C program are expression statements, such as assignments or function calls.
- An expression statement without an expression is called a *null statement*. It is often used to provide an empty body to a for or while loop. It can also be used to carry a label in the end of a compound statement or before a declaration:

```
puts("hello"); // expression statement
char *s;
while (*s++ != '\0')
    ; // null statement
```



## 1.16 Statement in C

- Statements are fragments of the C program that are executed in sequence. The body of any function is a compound statement, which, in turn is a sequence of statements and declarations:

```
int main(void)
{ // start of a compound statement
    int n = 1; // declaration (not a statement)
    n = n+1; // expression statement
    printf("n = %d\n", n); // expression statement
    return 0; // return statement
} // end of compound statement, end of function body
```

- There are five types of statements:
  - 1) compound statements
  - 2) expression statements
  - 3) selection statements
  - 4) iteration statements
  - 5) jump statements



## 1.17 OPERATORS

- Operator is a Symbol that tells or instructs the Compiler to perform certain Mathematical or Logical manipulations (Calculations).
- Operators are used in a program to work on data and variables.
- Operators in general form a part of Mathematical or Logical expression. Operators are generally classified into different types. They are described one below another as follows.
  1. Arithmetic Operators
  2. Relational Operators
  3. Logical Operators
  4. Assignment Operators
  5. Conditional Operators
  6. Special Operators
  7. Bitwise operators
  8. Increment and decrement operators
  9. Unary operators
  10. Equality operators
  11. Size of operators



# 1.17 OPERATORS

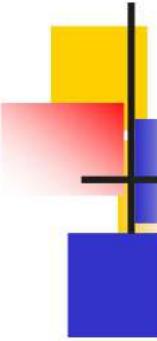
## 1. Arithmetic Operators

- Arithmetic operators are used to perform Arithmetic operations.
- They form a part of program. Programs can be written with or without operators.
- But calculations are performed only using operators.
- Operators act along Operand to produce result.

Operator	Meaning	Details
+	Addition	Performs addition on integer numbers, floating point numbers. The variable name which is used is the <b>operand</b> and the symbol is <b>operator</b> .
-	Subtraction	Subtracts one number from another.
*	Multiplication	Used to perform multiplication
/	Division	It produces the Quotient value as output.
%	Modulo	It returns the remainder value as output.

Examples of arithmetic operators are

- $x + y$        $x - y$        $x * y$        $x/y$        $x \% y$
- Here  $x$ ,  $y$  are known as operands. The modulus operator is a special operator in C language that evaluates the remainder of the operands after division.



# 1.17 OPERATORS

## 2. Relational Operator

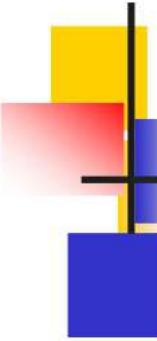
- Relational Operators are used to compare two same quantities.
- There are six relational operators. They are mentioned one below another as follows.

Operator	Meaning	Operator	Meaning
<	is less than	$\geq$	is greater than or equal to
$\leq$	is less than or equal to	$=$	is equal to
>	is greater than	$\neq$	is not equal to

- The general form is  
$$(\text{exp1 relational operator exp2})$$
- where exp1 and exp2 are expressions, which may be simple constants, variables or combination of them. Given below is a list of examples of relational expressions and evaluated values.

$6.5 \leq 25$  TRUE       $-65 > 0$  FALSE       $10 < 7 + 5$  TRUE

- Relational expressions are used in decision making statements of C language such as if, while and for statements to decide the course of action of a running program.



# 1.17 OPERATORS

## 3. Logical Operators

- Logical Operators are used when we need to check more than one condition.
- It is used to combine two or more relational expressions.
- Logical Operators are used in decision making. Logical expression yields value 0 or 1 i.e.,( Zero or One) .
- 0 indicates that the result of logical expression is TRUE and 1 indicates that the result of logical expression is FALSE.

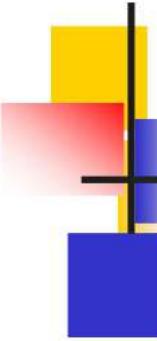
Logical Operator	Meaning
&&	Logical AND
	Logical OR
!	Logical NOT

### i) Logical AND ( && )

- The result of the Logical AND operator will be TRUE If both value is TRUE. If any one of the value is false then the result will be always False. The result is similar to basic Binary multiplication.

Example:  $a > b \&\& x = = 10$

- The expression is true only if both expressions are true i.e., if a is greater than b and x is equal to 10.



# 1.17 OPERATORS

## 3. Logical Operators

### ii) Logical OR ( || )

- If any of the expression is true the result is true else false otherwise. The result is similar to basic Binary addition.
- The logical OR is used to combine 2 expressions or the condition evaluates to true if any one of the 2 expressions is true.

Example:  $a < m \mid\mid a < n$

- It evaluates to true if a is less than either m or n and when a is less than both m and n.

### iii) Logical NOT ( ! )

- It acts upon single value. If the value is true result will be false and if the condition is false the result will be true. The logical not operator takes single expression and evaluates to true if the expression is false and evaluates to false if the expression is true.

Example:  $(!a)$



# 1.17 OPERATORS

## 4. Equality Operators

- Equality operator ( = ) is used together with condition.
- The value of the expression is one or zero. If the expression is true the result is one, if false result is zero.

Operator	Meaning
<code>= =</code>	Equal to
<code>!=</code>	Not equal to

Example :

- $x = 1$  and  $y = 2$  then
- $x == 2$  is false ,  $x == 1$  true ,  $y != 3$  true

# 1.17 OPERATORS

## 5. Assignment Operators

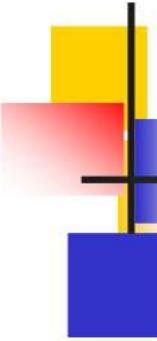
- The Assignment Operator evaluates an expression on the right of the expression and substitutes it to the value or variable on the left of the expression. The value of the expression is one or zero. If the expression is true the result is one, if false result is zero.
- The general form is

identifier = expression

Example:  $x = a + b;$

- Here the value of  $a + b$  is evaluated and substituted to the variable  $x$ . '=' equal to is used in Assignment operators.

Operator	Meaning	Example Simp. Assign	Example Shorthand
$+=$	Assign sum	$x = x + 1$	$x+ = 1$
$-=$	Assign difference	$y = y - 1$	$y-=1$
$*=$	Assign product	$z=z*(x+y)$	$z*=(x+y)$
$/=$	Assign quotient	$y=y/(x+y)$	$y/=(x+y)$
$\%=$	Assign remainder	$x=x\%z$	$x\%=z$
$\sim=$	Assign one's complement		
$<<=$	Assign left shift	$x = x << z$	$x << = z$
$>>=$	Assign right shift		
$\&=$	Assign bitwise AND	$y = y \& x$	$y\&=x$
$ =$	Assign bitwise OR		
$^=$	Assign bitwise X - OR	$z = z ^ y$	$z ^= y$



# 1.17 OPERATORS

## 6. Conditional Operators

- Conditional Operator Ternary operator is also known as "Ternary operator". The general form of Conditional Operator is

$$(exp1)?(exp2):(exp3);$$

- where exp1 is the condition which is to be checked and exp2 is the true value and exp3 is the false value. If the condition in exp1 is false then statement in exp3 will be automatically executed.

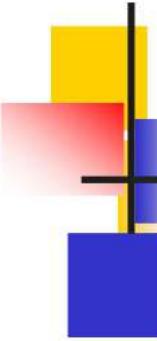
### Example:

```
#include<stdio.h> void main()
{
    int x,y,z;
    clrscr();
    printf("Enter the value of a and b :");
    scanf("%d %d",&x,&y);
    z=((x>y)?x:y);
    printf("The biggest value is %d",z); getch();
}
```

#### Output:

Enter the value of a and b: 125 100  
The biggest value is 125

Enter the value of a and b: 25 100  
The biggest value is 100



# 1.17 OPERATORS

## 7. Special Operators

- Special operators are known as separators or punctuators. Special operators are Ampersand ( & ) Braces ( { } ) Colon ( : ) Ellipsis ( ... ) Asterisk ( \* ) Brackets ( [] ) Comma ( , ) Hash ( # ) Parenthesis ( () ) Semicolon ( ; )

### i. Ampersand ( & )

- It is also known as addresss operator. It is specified before the identifier name. i.e., variable name. It indicates memory location of the identifier.

### ii. Asterisk ( \* )

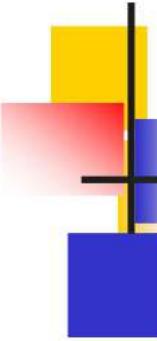
- Asterisk ( \* ) is also known as indirection operator. It is specified before identifier name. It indicates creation of pointer variable. It is also a unary operator.

### iii. Braces ( { } )

- The opening brace ( { } ) and closing brace ( } ) specify the start and end of compound statement in a function. Here semicolon ( ; ) is not necessary. Semicolon ( ; ) is used only in structure declaration.

### iv. Brackets

- Brackets [] also referred as array subscript operator. It is used to indicate single and multi dimensional arrays.
- Example: int x[10]; float l[10][20];



# 1.17 OPERATORS

## 7. Special Operators

### v. Colon ( : )

- Colon ( : ) is used in labels. It is used in unconditional control statement i.e., in goto statement.
- Example: goto d;

### vi. Comma Operator ( , )

- It is used to link expressions together. It is used together with variables to separate one variable from another. It is used in for loop. It has the lowest precedence among operators
- Example: for(n=1,m=10;n<=m; n++, m++) int a,b,c;
- sum= (x=5,y=3,x+y);

### vii. Ellipsis ( ... )

- Ellipsis ( ... ) are three continuous dots with no white spaces between them. It is used in function prototype. It indicates that the function can have any number of arguments.
- Example: void fun(char s,int n, float f, ...);



# 1.17 OPERATORS

## 7. Special Operators

### viii. Hash ( # )

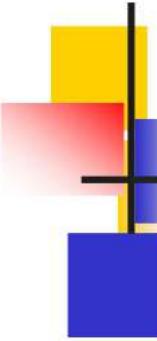
- Hash ( # ) is also known as pound sign. It is used to indicate preprocessor directives.
- Example: #include“stdio.h”

### ix. Parenthesis ( () )

- Parenthesis ( () ) is also known as function call operator. It is used to indicate the open and end of function prototypes, function call, function parameters, .... Parentheses are used to group expressions.

### x. Semicolon ( ; )

- Semicolon ( ; ) is a statement delimiter. It is used to end a C statement.
- Example: g=d+h;



# 1.17 OPERATORS

## 8. Bitwise Operators

- Bit wise operator is used to manipulate with bits within a word of memory. Bit wise operator operates only on integer and character but not on float and double.

Operator	Meaning	Operator	Meaning
<code>~</code>	One's Complement	<code>&amp;</code>	Bitwise AND
<code>&lt;&lt;</code>	Left shift	<code> </code>	Bit wise OR
<code>&gt;&gt;</code>	Right shift	<code>^</code>	Bit wise x-or

### I. One's Complement Operator ( `~` )

- One's Complement makes the bits of operand inverted. Here one becomes zero and zero becomes one.

**Example:**       $x = 7$  i.e.,  $x = 0000\ 0111$

One's complement of  $t$  is 248 ( i.e.,  $\sim x = 1111\ 1000 = 248$  )



# 1.17 OPERATORS

---

## 8. Bitwise Operators

### II. Left shift Operator ( << )

- Left shift operator ( << ) shifts each bit of the operand to left. The general form is
- *Variable << number of bit positions*

**Example:**       $x = 7$  ( i.e., 0000 0111 = 7 )

$y = x << 1$  is 14. ( i.e., 0000 1110 = 14 )

### III. Right shift Operator ( >> )

- Right shift operator shifts each bit of the operand to right. The general form is
- *Variable >> number of bit positions*

**Example:**       $x = 7$  ( 0000 0111 = 7 )

22

$y = x >> 1$  is 3 ( i.e., 0000 011 = 3 )



# 1.17 OPERATORS

## 9. Size-Of Operator

- This operator is used to return the size of string or character. It cannot be used in together with Integers. The syntax is

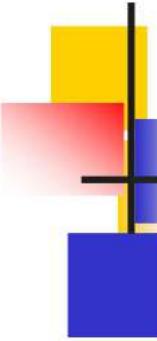
sizeof(variable-name);

Example: #include<stdio.h> void main()

```
{  
int x; clrscr();  
printf("The value of x is %d",sizeof(x)); getch();  
}
```

**Output:**

The value of x is 2



# 1.17 OPERATORS

## 10. Increment & Decrement Operators

### i) ***Increment Operators***

- It is used to increase the value of the Operand by 1. There are two types of Increment Operators in C Language. They are pre Increment operator and post Increment operator.

### ii) ***Pre Increment Operator***

- It is used to increase the value of the variable by 1. Here the value of 1 is added to the variable first along with the given variable value.

Example:            $++g$       ->      pre increment

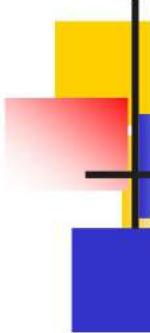
### iii) ***Post Increment Operators***

- It is used to increase the value of the variable by 1. Here the value of 1 is added to the variable first along with the given variable value.

Example:            $g++$       ->      post increment

### iv) ***Decrement Operators***

- It is used to decrease the value of the Operand by 1. There are two types of Decrement Operators in C Language. They are pre decrement operator and post decrement operator.



# 1.17 OPERATORS

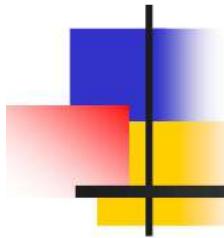
## 11. Unary Operators

- Unary operators act on single operand to produce new value.
- It proceeds with operands. Unary minus denotes subtraction.
- Subtraction operators require two operands but unary minus require one operand.
- The operand used with this operator must be a single variable.

Operator	Meaning
-	Unary minus
++	Increment by 1
--	Decrement by 1
sizeof	Return the size of operand

Example:

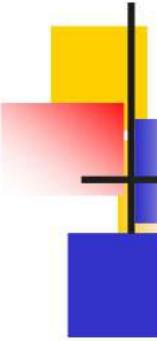
- $-786.064$      $-5e-8$      $-(a+b)$      $-6*(f+b)$      $-45.878$
- When operator is used before variable then it is prefix notation. When operator is used after variable then it is postfix notation.



# **PROBLEM SOLVING USING C PROGRAMMING**

**Bachelor of Computer Application  
SEMESTER - I**

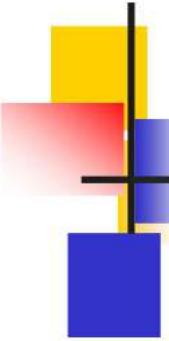
**GURU NANAK COLLEGE(Autonomous)  
VELACHERY ROAD, CHENNAI – 600042  
(Re-Accredited 'A' grade by NAAC)**



## Syllabus – UNIT 2

---

- Data input output functions
- Simple C programs
- Flow of control
  - If
  - if- else
  - While
  - do-while
  - for loop
  - nested control structures
  - Switch
  - break and continue
  - go to statements
  - comma operator.

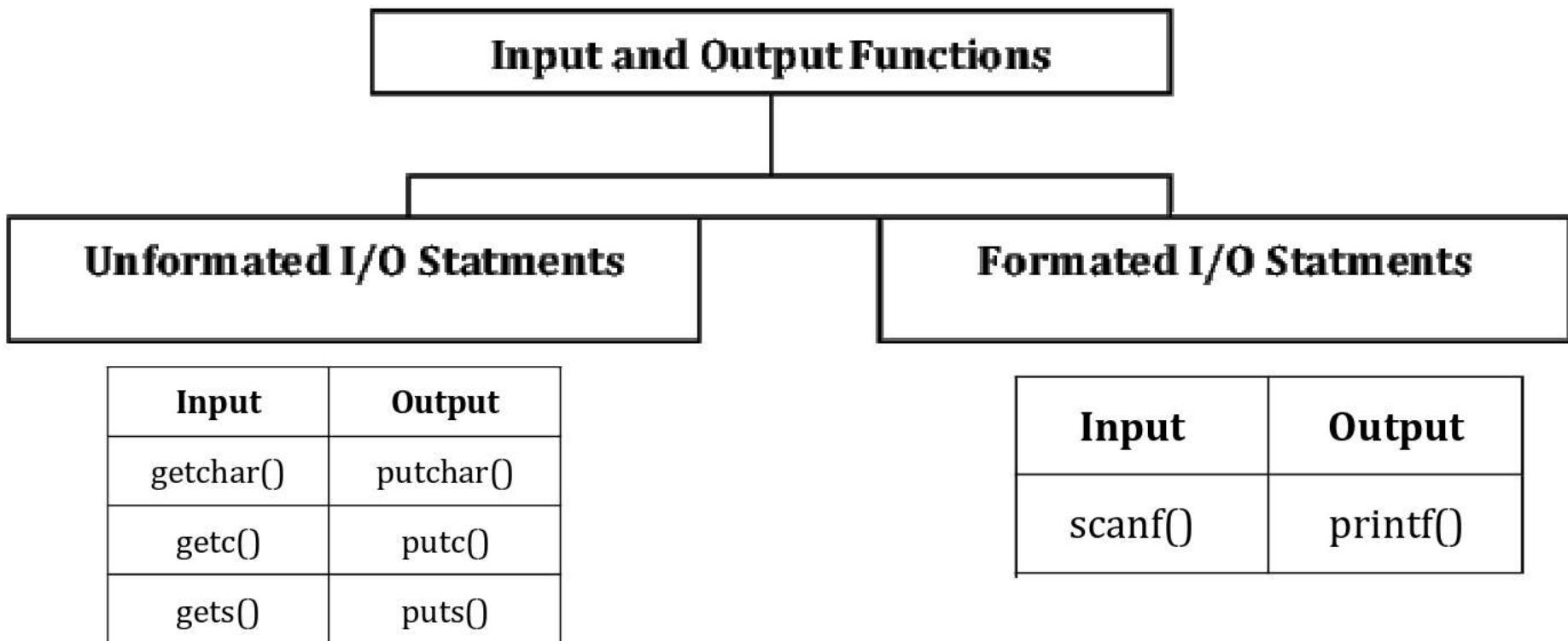


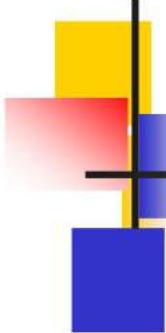
## 2.1 INPUT OUTPUT FUNCTIONS

- We know that input, process, output are the three essential features of computer program.
- The program takes some input data, processes it and gives the output.
- We have two methods for providing data to the program
  - Assigning the data to the variable in a program
  - By using the Input/output statements
- In 'C' language, two types of Input/output statements are available, and all input and output operation are carried out through function calls.
- Several functions are available for input/output operations in 'C'.

# 2.1 INPUT OUTPUT FUNCTIONS

- These functions are collectively known as the standard I/O library
  - Unformatted Input/output Statements
  - Formatted Input/output Statements



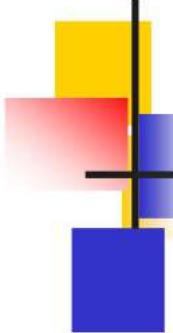


## 2.1 INPUT OUTPUT FUNCTIONS

### 1. Unformatted Input/Output Statements

- These statements are used to Input/output a single/group of characters form/ to the input/output devices. Here the user cannot specify the type of data that is going to be input/output.
- The following are the unformatted Input/Output statements available in 'C'

Input	Output
getchar()	putchar()
getc()	putc()
gets()	puts()



## 2.1 INPUT OUTPUT FUNCTIONS

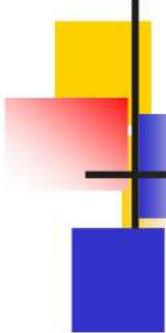
i.

### Single Character Input-getchar( ) function

- A Single character can be given to the computer using 'C' input library function getchar()

<b>Syntax</b>	char variable=getchar();
<b>Description</b>	char : datatype; variable : Any valid 'C' variable
<b>Example</b>	char x; x=getchar();

- The getchar() function is written in standard I/O library.
- It reads a single character from a standard input device.
- This function do not require any arguments, through a pair of empty parentheses, must follow the statements getchar().
- The first statement declares x as a character type variable.
- The second statement causes a single character to be entered from the standard input device and then assigned to variable x.



## 2.1 INPUT OUTPUT FUNCTIONS

### ii. Single Character Output-putchar() function

- The putchar() function is used to display one character at a time on the standard output device. This function does the reverse operation of the single character input function as discussed above

<b>Syntax</b>	putchar(character variable);
<b>Description</b>	Character variable is the valid 'C' variable of the type of char data type
<b>Example</b>	char x; putchar(x);



## 2.1 INPUT OUTPUT FUNCTIONS

### iii. The **getc()** function

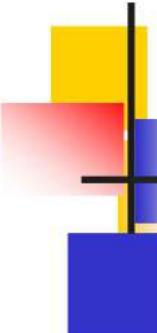
- This is used to accept a single character from the standard input to a character variable

<b>Syntax</b>	character variable=getc();
<b>Description</b>	Character variable is the valid 'C' variable of the type of char data type
<b>Example</b>	char c; c=getc();

### iv. The **putc()** function

- This is used to display a single character in a character variable to standard output device

<b>Syntax</b>	putc(character variable);
<b>Description</b>	Character variable is the valid 'C' variable of the type of char data type
<b>Example</b>	char c; put(c);



## 2.1 INPUT OUTPUT FUNCTIONS

v.

### The gets() and puts() function

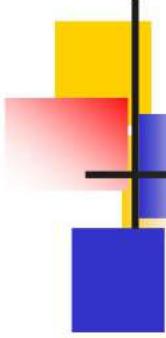
- The gets() function is used to read the string (string is a group of characters) from the standard input device (keyboard)

<b>Syntax</b>	gets(char type of array variable)
<b>Description</b>	Valid 'c' variable declared as one dimension char type
<b>Example</b>	gets(s);

- The puts() function is used to display/write the string to the standard output device.

<b>Syntax</b>	puts(char type of array variable)
<b>Description</b>	Valid 'c' variable declared as one dimension char type
<b>Example</b>	put(s);

- The gets() and puts() function are similar to scanf() and printf() function but the difference is in the case of scanf() input statement, when there is a blank space in input text, then it takes the space as an ending of the string the remaining string are not been taken.



## 2.1 INPUT OUTPUT FUNCTIONS

### 2. Formatted Input/Output Statements

- Formatted Input/output refers to input and output, that has been arranged in a particular format. **Example :** LAK 3977
- This line contains two parts of data that is arranged in a format, such data can be read to the format of its appearance, as the first data should be read into a variable char, the second into int.
- Such operation can be made possible in 'C' language by using the formatted Input/output statements. Using these statements, the user must specify the type of data, that is going to be accessed(Input/output).
- The following are the formatted Input/output statements

Input	Output
scanf()	printf()

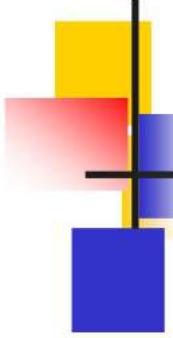


## 2.1 INPUT OUTPUT FUNCTIONS

### i. The **scanf()** function

- Input data can be entered into the computer using the standard input 'C' library function called **scanf()**.
- This function is used to enter any combination of input.
- The **scanf()** function is used to read information from the standard input device(keyboard), **scanf()** function starts with a string arguments and may contain additional arguments.
- Any additional argument must be pointer.

<b>Syntax</b>	<code>scanf("control string",&amp;var1,&amp;var2,.....&amp;varn);</code>
<b>Description</b>	The Control String consists of character groups, Each character group must begin with a percentage sign '%'. The character group contains percentage sign, followed by conversion character as specified in table below.  var1,var2....varn – are the arguments or variables in which the data is going to be accepted
<b>Example</b>	<code>int n;  scanf("%d",&amp;n);</code>



## 2.1 INPUT OUTPUT FUNCTIONS

### i. The `scanf()` function

#### **Control String:**

- It is the type of data that the user going to accept via the input statements, this can be formatted and always preceded with a '%' sign. The below table illustrates code formats (control strings) in Input/Output statements.
- Each variable name (argument) must be preceeded by an ampersand(&).
- The ('&') symbol gives the meaning " address of" the variable.
- The scanf control string or placeholder consists of % at the beginning and type indicator at the end.
- Apart from that it can have \*, a maximum field width indicator and a type indicator modified.

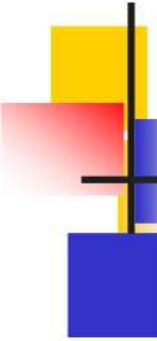


# 2.1 INPUT OUTPUT FUNCTIONS

## i. The **scanf()** function

### Control String:

Format String	Meaning
%c	Read single character.
%d	Read a decimal integer.
%e	Read a floating point value in Exponential form.
%f	Read a floating point value.
%g	Read a floating point value.
%h	Reads a short integer.
%i	Read a decimal, hexadecimal or octal integer.
%o	Reads an octal integer.
%s	Reads an string.
%u	Reads an unsigned decimal integer.
%x	Reads an Hexadecimal integer. (Unsigned) using lower case a – f
%X	Reads a hexadecimal integer (Unsigned) using upper case A – F
%u	Reads a unsigned integer.
%U	Reads a unsigned long integer.
%p	Reads a pointer value
%hx	Reads hex short
%lo	Reads octal long
%ld	Reads long



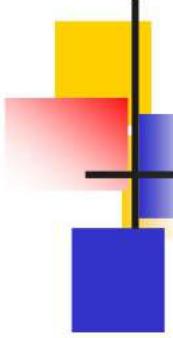
## 2.1 INPUT OUTPUT FUNCTIONS

---

### i. The `scanf()` function

#### **Rules for writing `scanf()` function:**

1. The control string must be preceded with (%) sign and must be within quotations, i.e., the address of variable should be passed.
2. If there is an number of input data items, items must be separated by commas and must be preceded with (&) sign except for string input.
3. The control string and the variables going to input should match with each other.
4. It must have termination with semicolon.
5. The `scanf()` reads the data values until the blank space in numeric input or maximum number of character have been read or an error is detected.

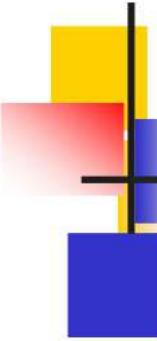


## 2.1 INPUT OUTPUT FUNCTIONS

### ii. The printf() function

- Output data or result of an operation can be displayed from the computer to a standard output device using the library function printf().
- This function used to output any combination of data.
- It is similar to the input function scanf(), except that it display data rather than input.

<b>Syntax</b>	printf("control string",var1,var2,.....varn);
<b>Description</b>	<p>Control string is any of the following.</p> <ul style="list-style-type: none"><li>a) Format code character</li><li>b) Execution character set or Escape sequences</li><li>c) Characters/String that will be displayed</li></ul> <p>var1,var2,varn are the arguments or variables from which the data is going to output. Here the variables need not to be preceded with '&amp;' sign.</p>
<b>Example</b>	printf("Result is ....%d",n); printf("%f",f); printf("%s",s);



## 2.1 INPUT OUTPUT FUNCTIONS

---

### ii. **The printf() function**

#### **Rules for writing printf() function:**

1. Place appropriate heading in the output.
2. The variables must be separated by commas, and need not be preceded with '&' sign.
3. The control string and the variables must match in their order.
4. The control string must be in quotations and there we can also use any other text to print with data.
5. Provide blank space in between the numbers for better readability.
6. Print special messages wherever required in output.



## 2.1 INPUT OUTPUT FUNCTIONS

i.

### Difference between **scanf** and **printf** function.

Scanf	printf
Used to accept data	Used to display data
Control string and & operator is used	Control string only used.
It end with semicolon	It end with semicolon
Number of input specified with format string and enclosed within double quotes.	Number of input specified with format string and separated by commas.
The Input variables are specified using Address operator (&) is separated by commas.	The output variables are specified by their name and separated by commas.

# 2.1 INPUT OUTPUT FUNCTIONS

ii.

## Difference between **getchar()** and **gets()**

<b>getchar()</b>	<b>gets()</b>
Used to receive a single character	Used to receive a string with white spaces and blanks
Does not require any argument	It requires a single argument

iii.

## Difference between **scanf()** and **gets()**

<b>scanf()</b>	<b>gets()</b>
Strings with white spaces cannot be accessed until ENTER key is pressed	Strings with any number of spaces can be accessed
Spaces and tabs are not acceptable as a part of the input string	Spaces and tabs are acceptable as a part of input string.
Can receive any number of characters and integers.	Only one string can be received at a time.
Format string and input variable name is specified.	Format string and address is specified.
All data types can be accessed	Only character data types can be accessed

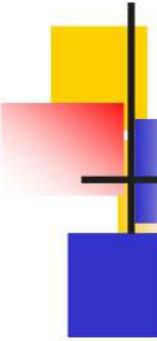


## 2.1 INPUT OUTPUT FUNCTIONS

iv.

### Difference between puts() and printf()

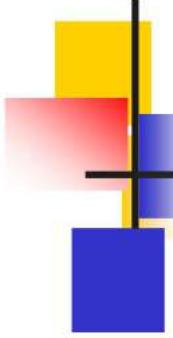
puts()	printf()
It can display only one string at a time.	It can display any number of characters, integers or strings at a time.
All data types are considered as characters.	Each data type is considered separately, depending upon the conversion specifications.



## 2.2 FLOW OF CONTROL -DECISION MAKING STATEMENTS

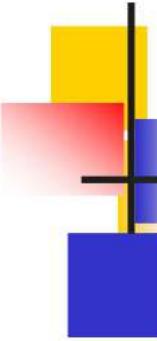
---

- Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.
- 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.
- C programming language provides following types of decision making statements. Click the following links to check their detail.



## 2.2 FLOW OF CONTROL -DECISION MAKING STATEMENTS

STATEMENT	DESCRIPTION
if statement	An if statement consists of a Boolean expression followed by one or more statements.
if...else statement	An if statement can be followed by an optional else statement, which executes when the Boolean expression is false.
nested if statements	You can use one if or else if statement inside another if or else if statement(s).
switch statement	A switch statement allows a variable to be tested for equality against a list of values.
nested switch statements	You can use one switch statement inside another switch statement(s).



## 2.2 FLOW OF CONTROL -DECISION MAKING STATEMENTS

### 1. Simple IF Statement

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

#### Syntax

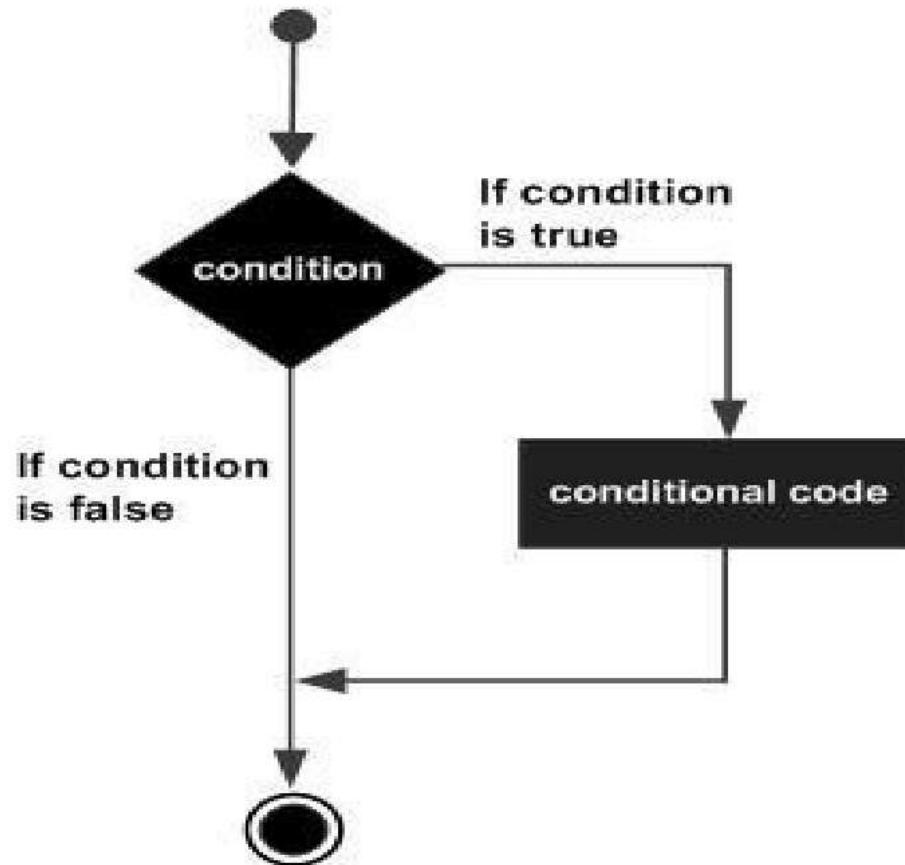
- The syntax of an **if** statement in C programming language is:

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

- If the boolean expression evaluates to true then the block of code inside the if statement will be executed.
- If boolean expression evaluates to false then the first set of code after the end of the if statement(after the closing curly brace) 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.

## 2.2 FLOW OF CONTROL -DECISION MAKING STATEMENTS

### 1. Simple IF Statement





## 2.2 FLOW OF CONTROL -DECISION MAKING STATEMENTS

---

### 1. Simple IF Statement

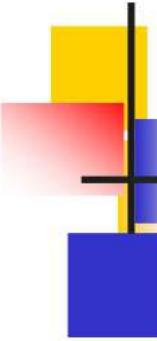
**Example:**

```
#include <stdio.h>
int main ()
{
/* local variable definition */
int a = 10;
/* check the boolean condition using if statement */
if( a < 20 )
{
/* if condition is true then print the following */
printf("a is less than 20\n");
}
printf("value of a is : %d\n", a);
return 0;
}
```

■ Output

**a is less than 20**

**value of a is : 10**



## 2.2 FLOW OF CONTROL -DECISION MAKING STATEMENTS

### 2. IF ELSE Statement

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

#### Syntax

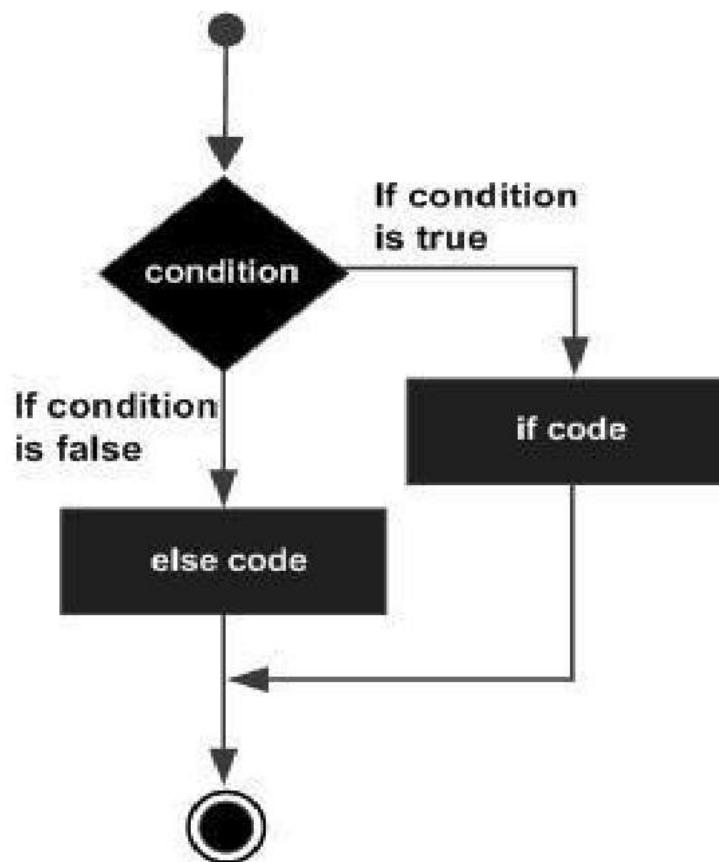
- The syntax of an **if...else** statement in C programming language is:

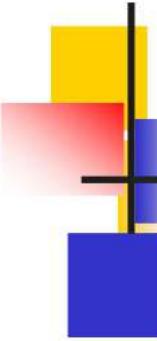
```
if(boolean_expression)
{
    /* statement(s) will execute if the boolean expression is true */
}
else
{
    /* statement(s) will execute if the boolean expression is false */
}
```

- If the boolean expression evaluates to true then the if block of code will be executed otherwise else block of code 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.

## 2.2 FLOW OF CONTROL -DECISION MAKING STATEMENTS

### 2. IF ELSE Statement





## 2.2 FLOW OF CONTROL -DECISION MAKING STATEMENTS

---

### 2. IF ELSE Statement

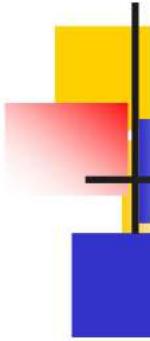
**Example:**

```
#include <stdio.h>
int main ()
{ /* local variable definition */
int a = 100;
/* check the boolean condition */
if( a < 20 )
{ /* if condition is true then print the following */
printf("a is less than 20\n");
}
else
{ /* if condition is false then print the following */
printf("a is not less than 20\n");
}
printf("value of a is : %d\n", a); return 0;
}
```

■ Output

**a is not less than 20**

**value of a is : 100**



## 2.2 FLOW OF CONTROL -DECISION MAKING STATEMENTS

---

### 3. Nested IF Statement

- An if statement can be followed by an optional else if...else statement, which is very useful to test various conditions using single if...else if statement.
- When using if , else if , else statements there are few points to keep in mind.
  - An if can have zero or one else's and it must come after any else if's.
  - An if can have zero to many else if's and they must come before the else.
  - Once an else if succeeds, none of the remaining else if's or else's will be tested.



## 2.2 FLOW OF CONTROL -DECISION MAKING STATEMENTS

### 3. Nested IF Statement

#### Syntax

- The syntax of an if...else if...else statement in C programming language is:

```
if(boolean_expression 1)
{
    /* Executes when the boolean expression 1 is true */
}
else if( boolean_expression 2)
{
    /* Executes when the boolean expression 2 is true */
}
else if( boolean_expression 3)
{
    /* Executes when the boolean expression 3 is true */
}
else
{
    /* executes when the none of the above condition is true */
}
```

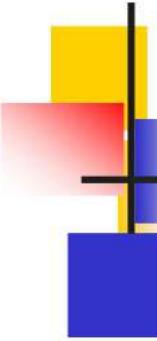


## 2.2 FLOW OF CONTROL -DECISION MAKING STATEMENTS

### 3. Nested IF Statement

#### Example:

```
#include <stdio.h> int main ()
{ /* local variable definition */ int a = 100;
/* check the boolean condition */ if( a == 10 )
{
    /* if condition is true then print the following */ printf("Value of a is 10\n");
}
else if( a == 20 )
{
    /* if else if condition is true */ printf("Value of a is 20\n");
}
else if( a == 30 )
{
    /* if else if condition is true */ printf("Value of a is 30\n");
}
else
{
    /* if none of the conditions is true */ printf("None of the values is matching\n");
}
printf("Exact value of a is: %d\n", a );
return 0;
}
```



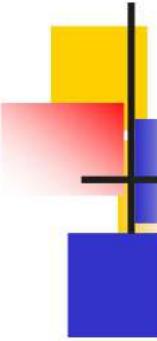
## 2.2 FLOW OF CONTROL -DECISION MAKING STATEMENTS

### 4. Switch Statements

- A switch statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each switch case.

- **Syntax:**

```
switch(expression)
{
    case constant-expression :
        statement(s);
        break; /* optional */
    case constant-expression :
        statement(s);
        break; /* optional */
    /* you can have any number of case statements */
    default : /* Optional */ statement(s);
}
```



## 2.2 FLOW OF CONTROL -DECISION MAKING STATEMENTS

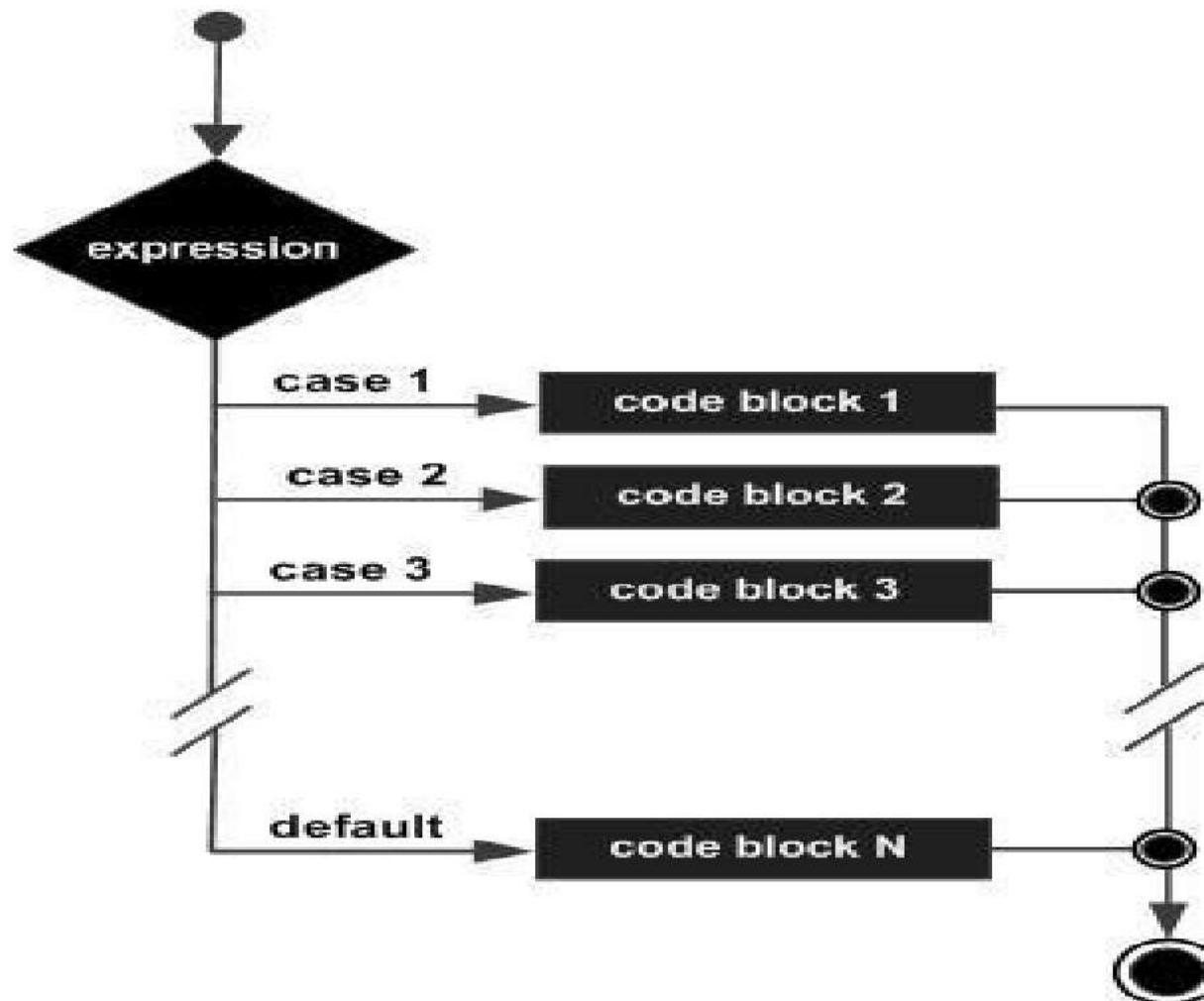
---

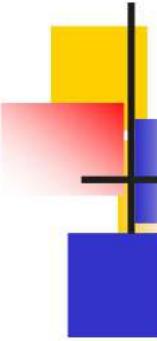
### 4. Switch Statements

- The following rules apply to a switch statement:
  - The **expression** used in a **switch** statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to 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 constant-expression 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.

## 2.2 FLOW OF CONTROL -DECISION MAKING STATEMENTS

### 4. Switch Statements





## 2.2 FLOW OF CONTROL -DECISION MAKING STATEMENTS

### 4. Switch Statements

```
#include <stdio.h> int main ()
{ /* local variable definition */
char grade = 'B';
switch(grade)
{
    case 'A' :
        printf("Excellent!\n");
        break;
    case 'B' :
        printf("Good\n");
        break;
    case 'C' :
        printf("Well done\n");
        break;
    case 'D' :
        printf("You passed\n");
        break;
    case 'F' :
        printf("Better try again\n");
        break;
    default :
        printf("Invalid grade\n");
}
printf("Your grade is %c\n", grade );
return 0;
}
```

**Output:**

Well done  
Your grade is C



## 2.3 FLOW OF CONTROL - BRANCHING AND LOOPING

- There may be a situation when you need to execute a block of code several number of times.
- In general statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.
- Programming languages provide various control structures that allow for more complicated execution paths.
- C programming language provides following types of loop to handle looping requirements.

LOOP TYPE	DESCRIPTION
while loop	Repeats a statement or group of statements until a given condition is true. It tests the condition before executing the loop body.
for loop	Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
do...while loop	Like a while statement, except that it tests the condition at the end of the loop body
nested loops	You can use one or more loop inside any another while, for or do..while loop.

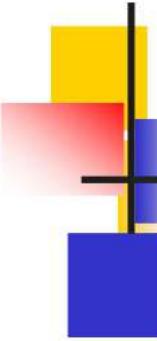


## 2.3 FLOW OF CONTROL - BRANCHING AND LOOPING

### ■ 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. Click the following links to check their detail.

CONTROL STATEMENT	DESCRIPTION
break statement	Terminates the <b>loop</b> or <b>switch</b> statement and transfers execution to the statement immediately following the loop or switch.
continue statement	Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
goto statement	Transfers control to the labelled statement. Though it is not advised to use goto statement in your program.



## 2.3 FLOW OF CONTROL - BRANCHING AND LOOPING

---

### 1. The Infinite Loop

- A loop becomes infinite loop if a condition never becomes false.
- The **for** loop is traditionally used for this purpose. Since none of the three expressions that form the for loop are required, you can make an endless loop by leaving the conditional expression empty.

```
#include <stdio.h> int main ()  
{  
    for(;;)  
    {  
        printf("This loop will run forever.\n");  
    }  
    return 0;  
}
```

- When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C programmers more commonly use the for(;;) construct to signify an infinite loop.
- **NOTE:** You can terminate an infinite loop by pressing Ctrl + C keys.



## 2.3 FLOW OF CONTROL - BRANCHING AND LOOPING

### 2. while Loop

- A **while** loop statement in C programming language repeatedly executes a target statement as long as a given condition is true.

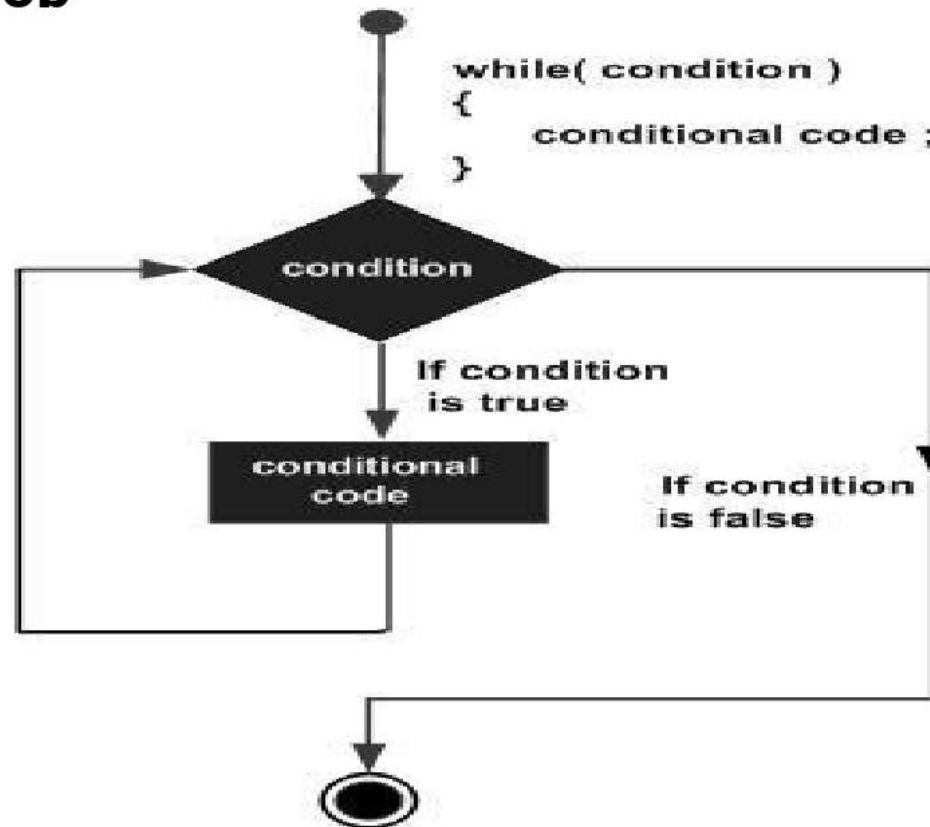
- **Syntax:**

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

- Here 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.
- When the condition becomes false, program control passes to the line immediately following the loop.

## 2.3 FLOW OF CONTROL - BRANCHING AND LOOPING

### 2. while Loop



- Here key point of the while loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.



## 2.3 FLOW OF CONTROL - BRANCHING AND LOOPING

---

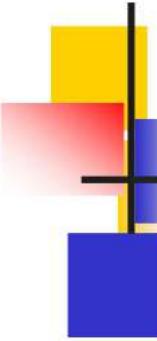
### 3. while Loop

**Example:**

```
#include <stdio.h> int main ()  
{  
/* local variable definition */ int a = 10;  
  
/* while loop execution */  
while( a < 20 )  
{  
printf("value of a: %d\n", a); a++;  
}  
return 0;  
}
```

**OUTPUT:**

```
value of a: 10  
value of a: 11  
value of a: 12  
value of a: 13  
value of a: 14  
value of a: 15  
value of a: 16  
value of a: 17  
value of a: 18  
value of a: 19
```



## 2.3 FLOW OF CONTROL - BRANCHING AND LOOPING

### 3. 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 language checks its condition at the bottom of the loop.
- A do...while loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

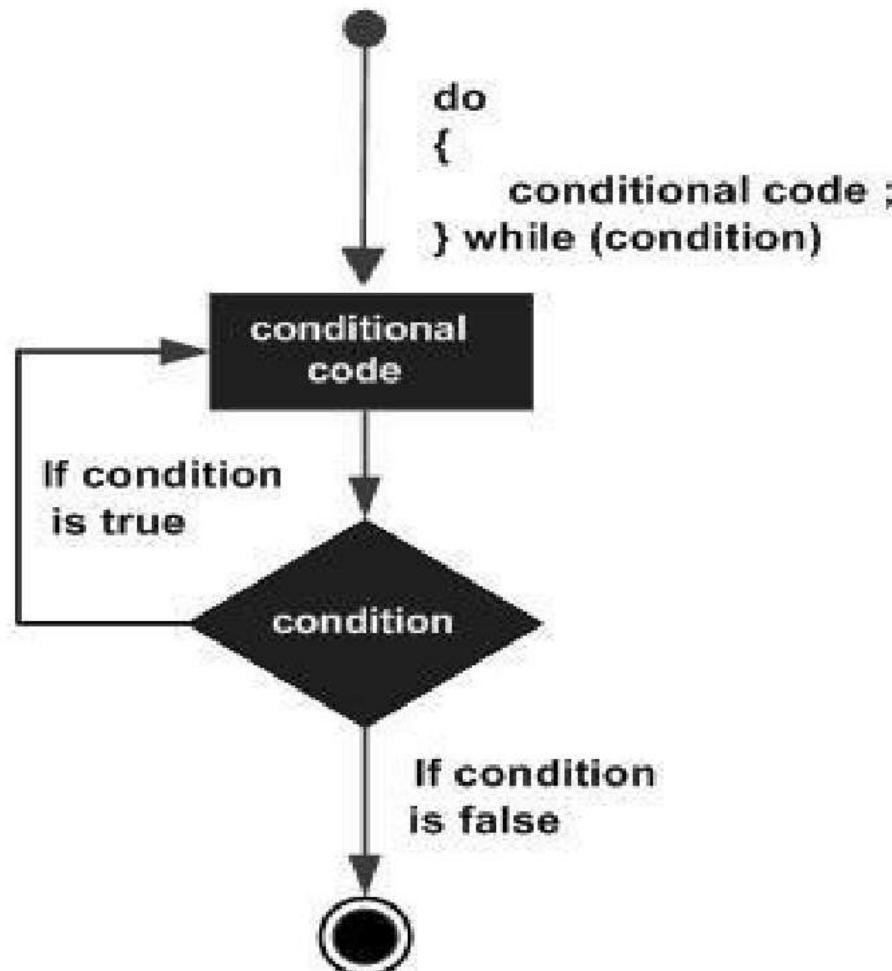
- Syntax:**

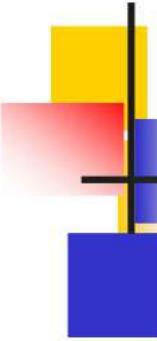
```
do  
{  
    statement(s);  
}while( condition );
```

- Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop execute once before the condition is tested.
- If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop execute again. This process repeats until the given condition becomes false.

## 2.3 FLOW OF CONTROL - BRANCHING AND LOOPING

### 3. do..while loop





## 2.3 FLOW OF CONTROL - BRANCHING AND LOOPING

---

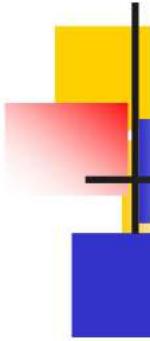
### 3. do..while loop

**Example:**

```
#include <stdio.h> int main ()  
{  
/* local variable definition */ int a = 10;  
  
/* do loop execution */ do  
{  
printf("value of a: %d\n", a); a = a + 1;  
}while( a < 20 ); return 0;  
}
```

**OUTPUT:**

```
value of a: 10  
value of a: 11  
value of a: 12  
value of a: 13  
value of a: 14  
value of a: 15  
value of a: 16  
value of a: 17  
value of a: 18  
value of a: 19
```



## 2.3 FLOW OF CONTROL - BRANCHING AND LOOPING

---

### 4. 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.

- **Syntax:**

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



## 2.3 FLOW OF CONTROL - BRANCHING AND LOOPING

---

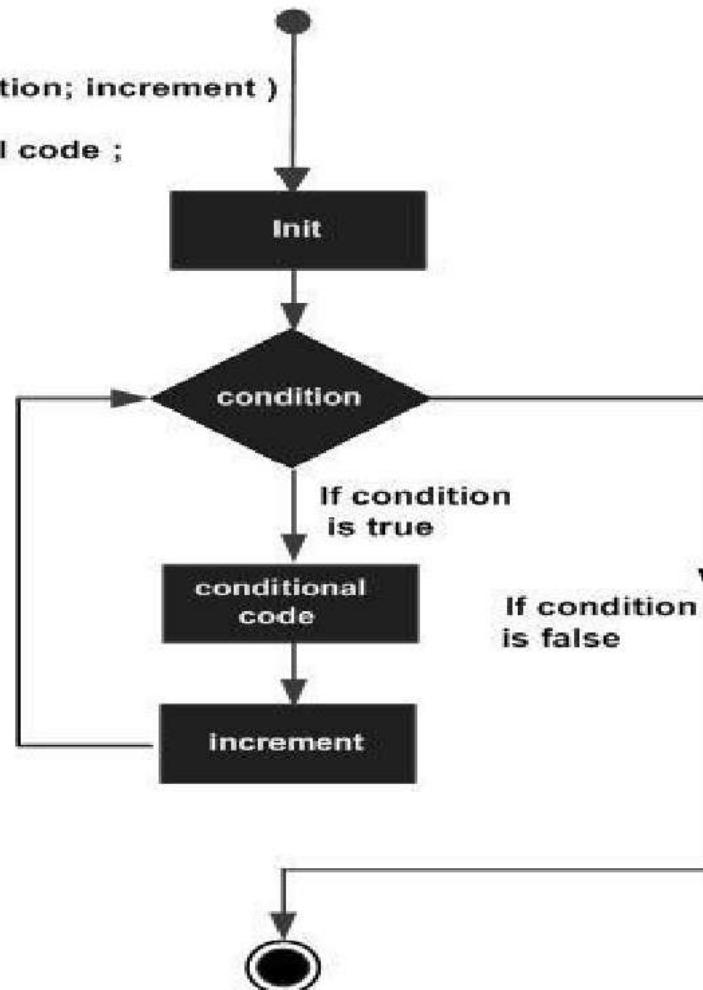
### 4. for loop

- Here is the flow of control in a for loop:
  - The **init** step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
  - 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 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 (body of loop, then increment step, and then again condition). After the condition becomes false, the for loop terminates.

## 2.3 FLOW OF CONTROL - BRANCHING AND LOOPING

### 4. for loop

```
for( init; condition; increment )  
{  
    conditional code ;  
}
```





## 2.3 FLOW OF CONTROL - BRANCHING AND LOOPING

---

### 4. for loop

#### Example:

```
#include <stdio.h> int main ()  
{  
/* for loop execution */  
for( int a = 10; a < 20; a = a + 1 )  
{  
printf("value of a: %d\n", a);  
}  
return 0;  
}
```

#### OUTPUT:

```
value of a: 10  
value of a: 11  
value of a: 12  
value of a: 13  
value of a: 14  
value of a: 15  
value of a: 16  
value of a: 17  
value of a: 18  
value of a: 19
```



## 2.3 FLOW OF CONTROL - BRANCHING AND LOOPING

---

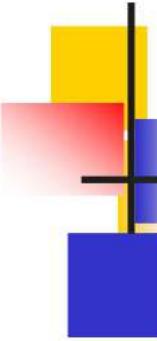
### 5. Nested loop

- C programming language allows to use one loop inside another loop. Following section shows few examples to illustrate the concept.
- A final note on loop nesting is that you can put any type of loop inside of any other type of loop. For example a for loop can be inside a while loop or vice versa.

- Syntax:

- The syntax for a nested for loop statement in C is as follows:

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



## 2.3 FLOW OF CONTROL - BRANCHING AND LOOPING

---

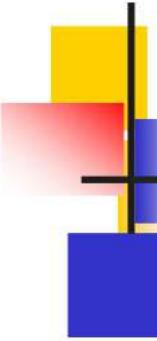
### 5. Nested loop

- The syntax for a nested while loop statement in C programming language is as follows:

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

- The syntax for a nested do...while loop statement in C programming language is as follows:

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



## 2.3 FLOW OF CONTROL - BRANCHING AND LOOPING

### 5. Nested loop

#### Example:

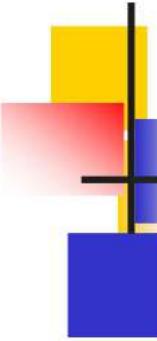
##### ■ Example

- The following program uses a nested for loop to find the prime numbers from 2 to 100:

```
#include <stdio.h>
int main ()
{
/* local variable definition */
int i, j;
for(i=2; i<100; i++)
{
    for(j=2; j <= (i/j); j++)
        if(!(i%j))
            break; // if factor found, not prime
        if(j > (i/j))
            printf("%d is prime\n", i);
}
return 0;
}
```

#### OUTPUT:

```
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime
53 is prime
59 is prime
61 is prime
67 is prime
71 is prime
73 is prime
79 is prime
83 is prime
89 is prime
97 is prime
```



## 2.3 FLOW OF CONTROL - Loop Control Statements

### 1. **break Statement**

- The **break** statement in C programming language has following two usage:

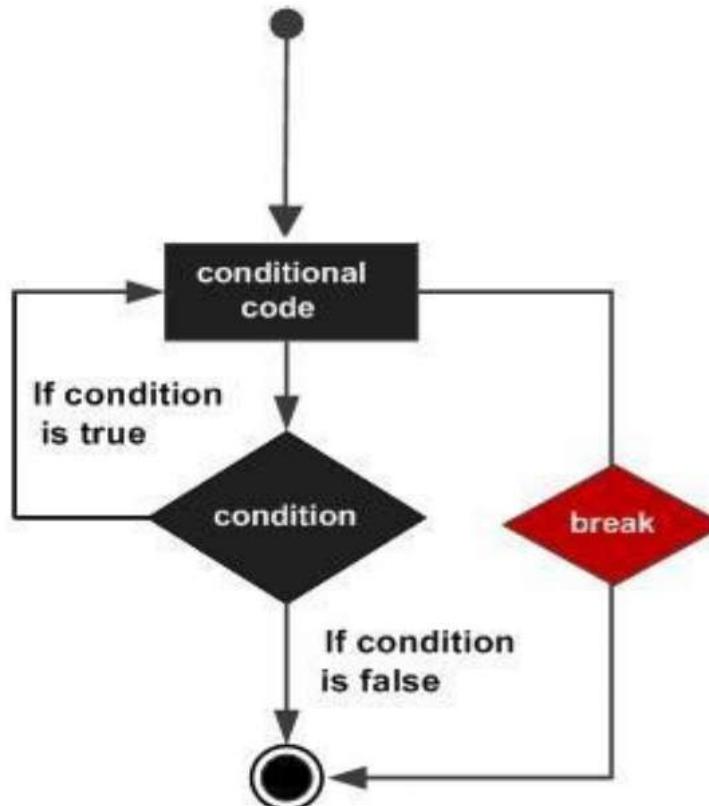
- When the **break** statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.
- It can be used to terminate a case in the **switch** statement
- If you are using nested loops ( ie. one loop inside another loop), the break statement will stop the execution of the innermost loop and start executing the next line of code after the block.

- **Syntax:**

```
break;
```

## 2.3 FLOW OF CONTROL - Loop Control Statements

### 1. break Statement



## 2.3 FLOW OF CONTROL - Loop Control Statements

### 1. break Statement

**Example:**

#### ■ Example

```
#include <stdio.h>
int main ()
{
/* local variable definition */ int a = 10;

/* while loop execution */ while( a < 20 )
{
printf("value of a: %d\n", a); a++;
if( a > 15)
{
/* terminate the loop using break statement */
break;
}
}
return 0;
}
```

**OUTPUT:**

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
```



## 2.3 FLOW OF CONTROL - Loop Control Statements

---

### 2. continue Statement

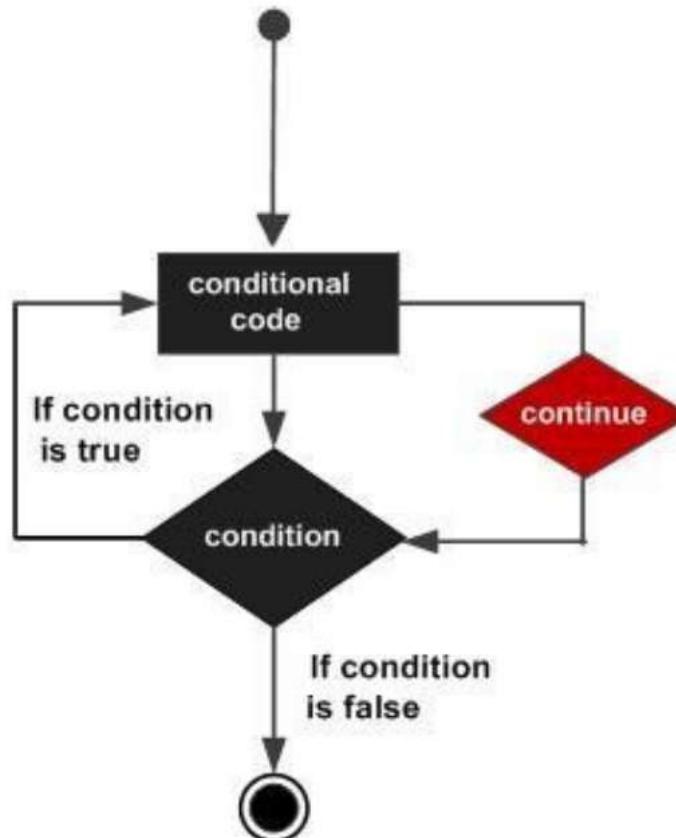
- The **continue** statement in C programming language works somewhat like the **break** statement. Instead of forcing termination, however, continue forces the next iteration of the loop to take place, skipping any code in between.
- For the **for** loop, **continue** statement causes the conditional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

- **Syntax:**

`continue;`

## 2.3 FLOW OF CONTROL - Loop Control Statements

### 2. continue Statement





## 2.3 FLOW OF CONTROL - Loop Control Statements

### 2. continue Statement

**Example:**

```
#include <stdio.h>
int main ()
{
/* local variable definition */ int a = 10;
/* do loop execution */ do
{
if( a == 15)
{
/* skip the iteration */ a = a + 1;
continue;
}
printf("value of a: %d\n", a); a++;
}

}while( a < 20 );

return 0;
}
```

**OUTPUT:**

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```



## 2.3 FLOW OF CONTROL - Loop Control Statements

---

### 3. **goto**

- A **goto** statement in C programming language provides an unconditional jump from the goto to a labeled statement in the same function.
- **NOTE:** Use of **goto** statement is highly discouraged in any programming language because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a goto can be rewritten so that it doesn't need the goto.

- Syntax:

`goto label;`

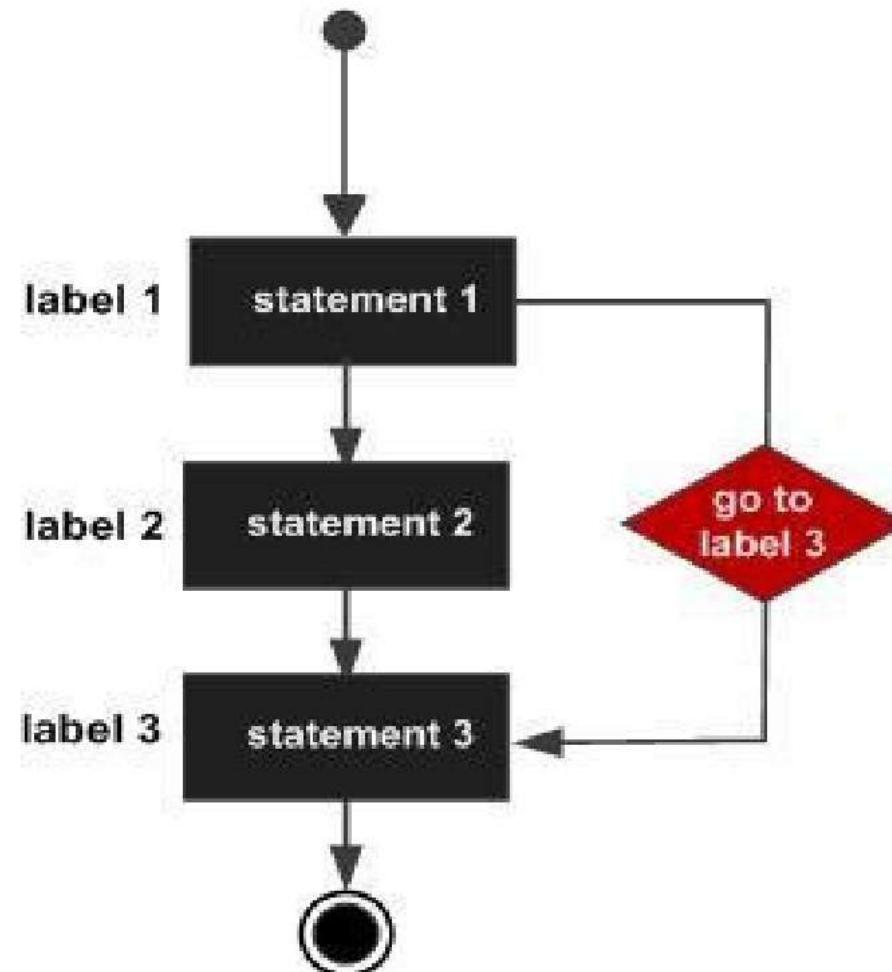
`..`

`.`

`label: statement;`

## 2.3 FLOW OF CONTROL - Loop Control Statements

### 3. goto





## 2.3 FLOW OF CONTROL - Loop Control Statements

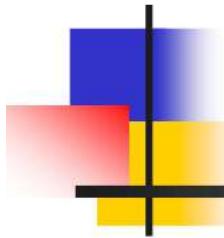
### 3. goto

**Example:**

```
#include <stdio.h> int main ()
{
    /* local variable definition */ int a = 10;
    /* do loop execution */
    LOOP:do
    {
        if( a == 15)
        {
            /* skip the iteration */
            a = a + 1;
            goto LOOP;
        }
        printf("value of a: %d\n", a);
        a++;
    }while( a < 20 );
    return 0;
}
```

**OUTPUT:**

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```



# **PROBLEM SOLVING USING C PROGRAMMING**

**Bachelor of Computer Application**

**SEMESTER - I**

**GURU NANAK COLLEGE(Autonomous)**  
**VELACHERY ROAD, CHENNAI – 600042**  
**(Re-Accredited 'A' grade by NAAC)**



# Syllabus – UNIT 3

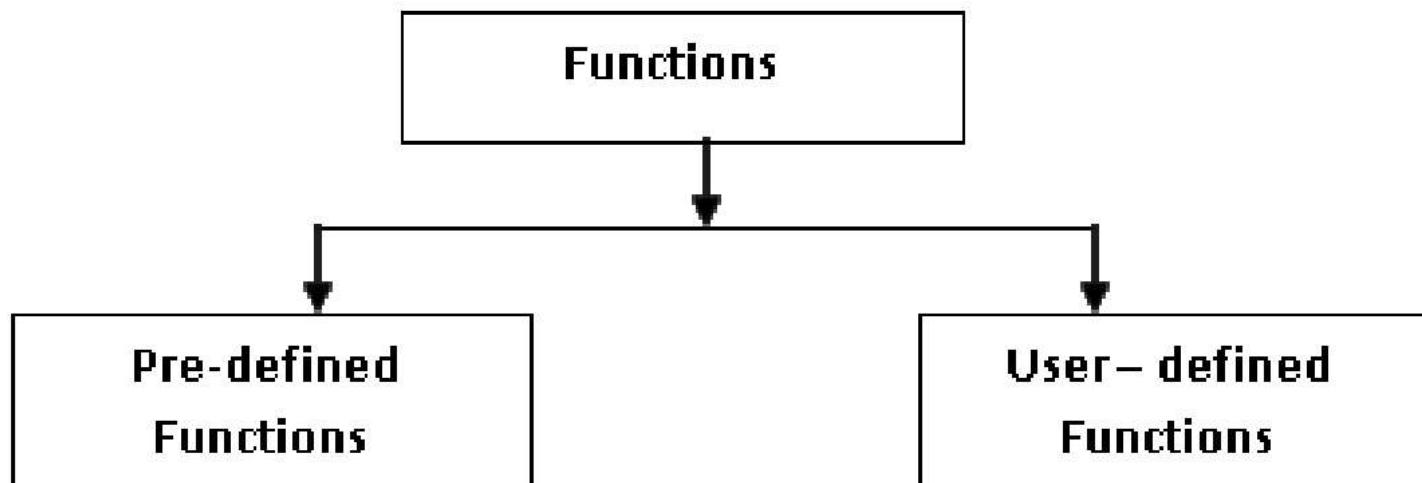
---

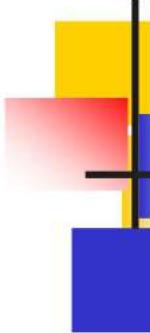
- Functions:

- Definition
- Proto-types
- Passing arguments
- Recursions.
- Storage Classes -
  - Automatic
  - External
  - Static
  - register variables
- Library functions

## 3.1 FUNCTIONS

- Functions are self-contained blocks of programs that perform some specific, well-defined task.
- It break large complicated computing tasks into smaller and simpler ones.
- It helps in maintenance and enhancement of programs.
- It also helps programmers to build their own functions and tying them to the existing library.





## 3.1 FUNCTIONS

### 1) Predefined standard library functions

- Standard library functions are also known as built-in functions.
- These functions are already defined in header files (files with .h extensions are called header files such as stdio.h), so we just call them whenever there is a need to use them.
- For example, printf() function is defined in <stdio.h> header file so in order to use the printf() function, we need to include the <stdio.h> header file in our program using #include <stdio.h>.
- Some of the header files in C are:

Header File	Description
<ctype.h>	Character testing and conversion functions
<math.h>	Mathematical functions
<stdio.h>	Standard I/O functions
<stdlib.h>	Utility functions
<string.h>	String handling functions
<time.h>	Time manipulation functions

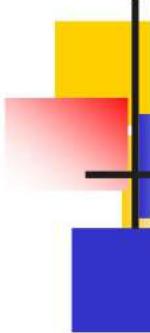


## 3.1 FUNCTIONS

### 1) Predefined standard library functions

- Some of the predefined functions available in **ctype.h** header file are:

Function	Return Type	Use
isalnum(c)	int	Determine if the argument is alphanumeric or not
isalpha(c)	int	Determine if the argument is alphabetic or not
isascii(c)	int	Determine if the argument is ASCII character or not
isdigit(c)	int	Determine if the argument is a decimal digit or not.
toascii(c)	int	Convert value of argument to ASCII
tolower(c)	int	Convert character to lower case
toupper(c)	int	Convert letter to uppercase

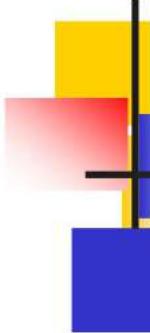


## 3.1 FUNCTIONS

### 1) Predefined standard library functions

- Some of the predefined functions available in **math.h** header file are:

Function	Return Type	Use
ceil(d)	double	Returns a value rounded up to next higher integer
floor(d)	double	Returns a value rounded up to next lower integer
cos(d)	double	Returns the cosine of d
sin(d)	double	Returns the sine of d
tan(d)	double	Returns the tangent of d
exp(d)	double	Raise e to the power of d
fabs(d)	double	Returns the absolute value of d
pow(d1, d2)	double	Returns d1 raised to the power of d2
sqrt(d)	double	Returns the square root of d

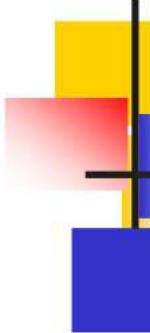


## 3.1 FUNCTIONS

### 1) Predefined standard library functions

- Some of the predefined functions in **stdlib.h** header file are:

Function	Return Type	Use
abs(i)	int	Return the absolute value of i
exit(u)	void	Close all file and buffers, and terminate the program
rand(void)	int	Return a random positive integer
calloc(u1, u2)	void*	Allocate memory for an array having u1 elements, each of length u2 bytes
malloc(u)	void*	Allocate u bytes of memory
realloc(p,u)	void*	Allocate u bytes of new memory to the pointer variable p
free(p)	void	Free a block of memory whose beginning is indicated by p



## 3.1 FUNCTIONS

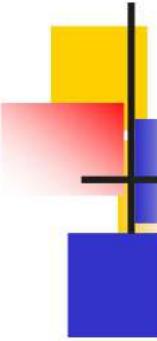
### 1) Predefined standard library functions

- Some of the predefined functions in **string.h** header file are:

Function	Return Type	Use
strcmp(s1,s2)	int	Compare two strings
strcpy(s1,s2)	char*	Copy string s2 to s1
strlen(s)	int	Return the number of characters in string s
strrev(s)	char*	Return the reverse of the string s

- Some of the predefined functions available in **time.h** header file are:

Function	Return type	Use
difftime(11,12)	double	Return the difference between 11 ~ 12.
time(p)	long int	Return the number of seconds elapsed beyond a designated base time



## 3.1 FUNCTIONS

---

### Example:

```
#include <stdio.h>
#include <math.h>
int main()
{
    float num, root;
    printf("Enter a number: ");
    scanf("%f", &num);

    // Computes the square root of num and stores in root.
    root = sqrt(num);

    printf("Square root of %.2f = %.2f", num, root);
    return 0;
}
```

### Output

Enter a number: 12  
Square root of 12.00 = 3.46

# 3.1 FUNCTIONS

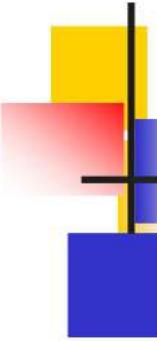
## 2) User-Defined functions

**Example:**

```
#include <stdio.h>
/* function declaration */
void introduction();
int main()
{
    /*calling function*/
    introduction();
    return 0;
}
void introduction()
{
    printf("Hi\n");
    printf("My name is Chaitanya\n");
    printf("How are you?");
    /* There is no return statement inside this function, since its
     * return type is void
    */
}
```

**Output**

Hi  
My name is Chaitanya  
How are you?

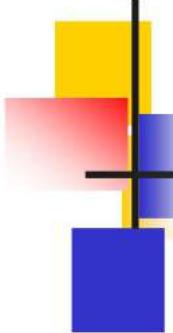


## 3.2 FUNCTIONS

---

### 2) User-Defined functions

- The functions that we create in a program are known as user defined functions or in other words you can say that a function created by user is known as user defined function.
- **return\_type:** Return type can be of any data type such as int, double, char, void, short etc. Don't worry you will understand these terms better once you go through the examples below.
- **function\_name:** It can be anything, however it is advised to have a meaningful name for the functions so that it would be easy to understand the purpose of function just by seeing it's name.
- **argument list:** Argument list contains variables names along with their data types. These arguments are kind of inputs for the function. For example – A function which is used to add two integer variables, will be having two integer argument.
- **Block of code:** Set of C statements, which will be executed whenever a call will be made to the function.



## 3.2 FUNCTIONS PROTOTYPE

### 1. Function Declaration

- Function declaration is also called as function prototype, since they provide model or blueprint of the function.

#### Syntax:

```
return_type function_name(parameter list);
```

#### Example:

```
int cube(int);
```

### 2. Function Definition

- It is the process of specifying and establishing the user defined function by specifying all of its elements and characteristics.
- A function that does not return any value, but only performs some operation, is declared to be void.

#### Syntax:

```
return_type function_name(parameters declaration)
{
```

```
}
```



## 3.2 FUNCTIONS PROTOTYPE

---

### 3. Function Call

- Function Call invokes the function that is defined in the program.

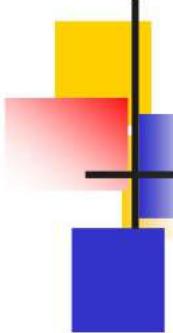
A function call is an expression of the form:

*Syntax:*

function\_name (argument-list);

*Example:*

addValue(index);



## 3.2 FUNCTIONS PROTOTYPE

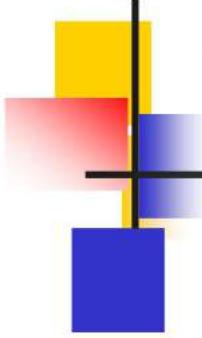
### 4. Return Statement

- To return some values to calling function main(), return statement is required; otherwise, it is optional.

#### Syntax:

```
return(expression);
```

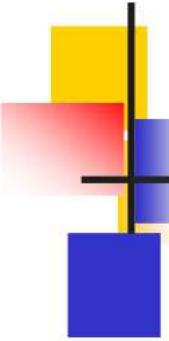
- The expression can be a constant, a variable, a user defined data structure, a general expression or a function call.
- If the datatype of the expression returned does not match the return type of the function, it is converted to the return type of the function.
- If there is no value in a return statement, the calling function will receive the control, but no value. Such a type of function is known as a void function.



## 3.2 FUNCTIONS PROTOTYPE

- ***Example:***

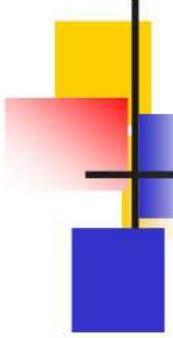
```
int factorial(int n)
{
    int i, result; if(n<0)
        return -1; if(n==0)
        return 1; for(i=1,result=1;i<=n;i++)
            result*=i; return result;
}
```



## 3.2 FUNCTIONS PROTOTYPE

### 4. Function Parameters

- Function parameters are the means of communication between the calling and the called functions.
- There is no limitation on the number of parameters passed to a function.
- There are two types of parameters,
  - ***Actual parameters*** – These are the parameters transferred from the calling program (main program) to the called program (function)
  - ***Formal parameters***- These are the parameters, transferred into the calling function (main program) from the called program (function)



## 3.2 FUNCTIONS PROTOTYPE

### 4. Function Parameters

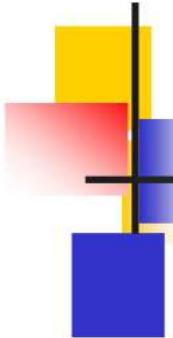
- **Example:**

```
main()
{
..... fun1(a,b);
.....
.....
25
}
fun1(x,y)
{
.....
.....
}
```

**Where,**

**a,b are the Actual parameters**

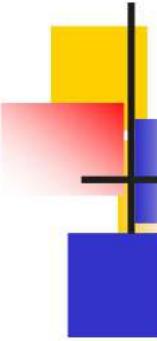
**x,y are the Formal parameter**



## 3.3 PASSING FUNCTION ARGUMENTS

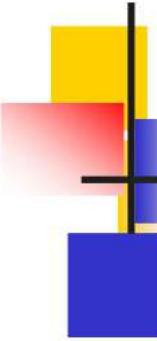
- If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the formal parameters of the function.
- The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.
- While calling a function, there are two ways that arguments can be passed to a function:

Call Type	Description
Call by value	This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
Call by reference	This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.



## 3.3 PASSING FUNCTION ARGUMENTS

- Default, C uses call by value to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function and above mentioned example while calling max() function used the same method.



## 3.3 PASSING FUNCTION ARGUMENTS

---

### i. Call By Value

- In the **call by value** method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
- By default, C programming language uses call by value method to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function. Consider the function **swap()** definition as follows.

# 3.3 PASSING FUNCTION ARGUMENTS

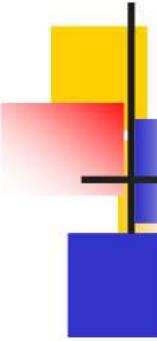
## i. Call By Value

```
/* function definition to swap the values */
void swap(int x, int y)
{
    int temp;
    temp = x; /* save the value of x */
    x = y;      /* put y into x */
    y = temp; /* put temp into y */
    return;
}
#include <stdio.h>

/* function declaration */ void swap(int x, int y); int main ()
{
    /* local variable definition */ int a = 100;
    int b = 200;
    printf("Before swap, value of a : %d\n", a );
    printf("Before swap, value of b : %d\n", b );
    /* calling a function to swap the values */ swap(a, b);
    printf("After swap, value of a : %d\n", a );
    printf("After swap, value of b : %d\n", b );
    return 0;
}
```

### Result:

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :100
After swap, value of b :200
```



## 3.3 PASSING FUNCTION ARGUMENTS

---

### ii. **Call By Reference**

- The **call by reference** method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.

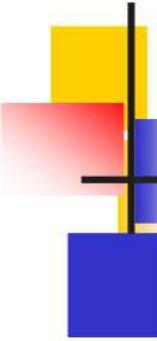
# 3.3 PASSING FUNCTION ARGUMENTS

## i. Call By Value

```
/* function definition to swap the values */
void swap(int *x, int *y)
{
    int temp;
    temp = *x; /* save the value at address x */
    *x = *y; /* put y into x */
    *y = temp; /* put temp into y */
    return;
}
#include <stdio.h>
/* function declaration */
void swap(int *x, int *y);
int main ()
{
    /* local variable definition */ int a = 100;
    int b = 200;
    printf("Before swap, value of a : %d\n", a );
    printf("Before swap, value of b : %d\n", b );
    /* calling a function to swap the values.
     *          &a indicates pointer to a ie. address of variable a and
     *          &b indicates pointer to b ie. address of variable b. */
    swap(&a, &b);
    printf("After swap, value of a : %d\n", a );
    printf("After swap, value of b : %d\n", b );
    return 0;
}
```

### Result:

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :100
After swap, value of b :200
```

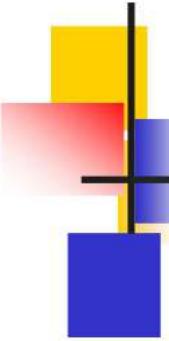


## 3.4 Recursion

- Recursion is the process of calling the same function itself again and again until some condition is satisfied. This process is used for repetitive computation in which each action is satisfied in terms of a previous result.

```
#include <stdio.h>
int factorial(unsigned int i)
{
    if(i <= 1)
    {
        return 1;
    }
    return i * factorial(i - 1);
}
int main()
{
    int i = 15;
    printf("Factorial of %d is %d\n", i, factorial(i));
    return 0;
}
```

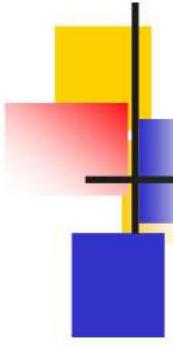
Output: Factorial of 15 is 2004310016



## 3.5 Scope of Variables

---

- The part of the program within which variable/constant can be accessed is called as its **scope**.
- By default, the scope of a variable is **local** to the function in which it is defined.
- **Local variables** can only be accessed in the function in which they are defined.
- A variable defined outside any function is called as **External Variable**.
- Scope of an external variable will be the whole program, and hence such variables are referred to as **Global variables**.



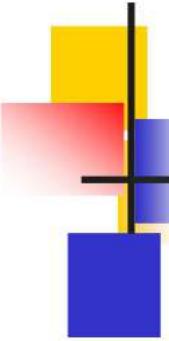
## 3.6 STORAGE CLASSES

---

- All variables have a datatype, they also have a '**Storage Class**'.
- The storage class determines the lifetime of the storage associated with the variable.
- If we don't specify the storage class of a variable in its declaration, the compiler will assume a storage class dependent on the context in which variable is used.

A variable's storage class gives the following information

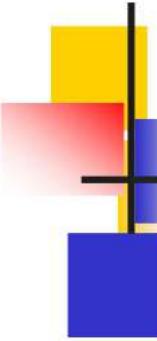
- Where the variable would be stored.
- What will be the default initial value.
- What is the scope of the variable.
- What is the life of the variable that is, how long would the variable exist.



## 3.6 STORAGE CLASSES

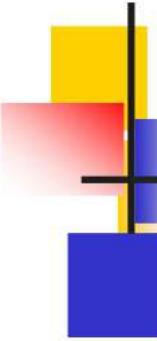
---

- Following four types of storage classes are provided in C:
  - Automatic Storage Class
  - Static Storage Class
  - External Storage Class
  - Register Storage Class



## 3.6.1 Automatic Storage Class

- Variables declared inside a block and local to block in which declared are said to be Automatic variables. These variables can be accessed by block in which they are declared. Another block cannot access its value.
- These variables created as new variable each time when function is called and destroyed automatically when the function is exited.
- Compiler treat variable declared inside block as automatic variable by default. Automatic variables are stored in memory. All variables declared inside the function is **auto** by default.
- Auto variables are safe that is, they cannot be accessed directly by other functions.



## 3.6.1 Automatic Storage Class

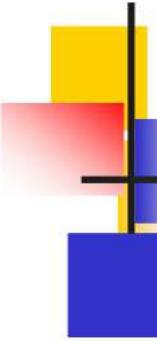
Example

```
main()
{
    int number;
    -----
    -----
}
```

- We may also use the keyword auto to declare automatic variables explicitly. main()

```
{
    auto int number;
    -----
    -----
}
```

- One important feature of automatic variables is that their value cannot be changed accidentally by what happens in some other function in the program.
- This assures that we may declare and use the same variable name in different functions in the same program without causing any confusion to the compiler.

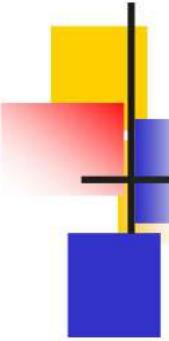


## 3.6.1 Automatic Storage Class

### Example: Program to illustrate how automatic variables work

```
#include<stdio.h>
#include<conio.h>
void function1(void);
void function2(void);
main()
{
    int m=2000;
    function2();
    printf("%d\n",m);
}
void function1(void)
{
    int m=10;
    printf("%d\n",m);
}
void function2(void)
{
    int m=100;
    function1();
    printf("%d\n",m);
}
```

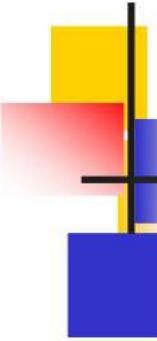
**Output:**  
10  
100  
2000



## 3.6.2 Static Storage Class

---

- Static variable may be either an internal type or an external type depending on the place of declaration.
- Static variables declared within individual block. They are local to the block in which declared. It exists until the end of the program.
- Variable can be declared using the keyword static.
- Global and Local variable can be declared static. Static variables are initialized only once when they are compiled in a program.
- When the program is closed the function associated with that program is also excited and whenever it is visited again the same value exists.
- Internal static variable is declared inside a function.
- External static variable is declared outside a function. It is made available to all functions in a C program.



## 3.6.2 Static Storage Class

### Example: Program to illustrate how static variables work

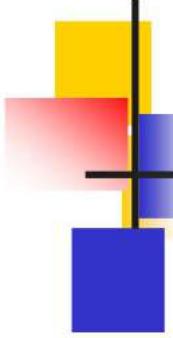
```
#include<stdio.h>
#include<conio.h>
void start(void);
void main()
{
    int i;
    clrscr();
    for(i=1;i<3;i++)
        stat();
    getch();
}
void stat(void)
{
    static int x=0;
    x=x+1;
    printf("x=%d\n",x);
}
```

**Output:**

x=1

x=2

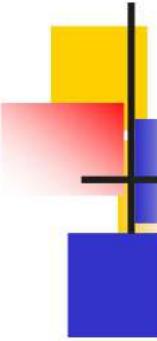
x=3



### 3.6.3 External Storage Class

---

- Variables that are active throughout the entire program are called as external variables (global variables).
- External variables are declared outside the function body. This storage class is created when variable is declared global.
- No memory is reserved for the variable. Variable retain the value throughout the execution of a program.
- This storage class can be accessed by any function in same or different program file and change its value.



### 3.6.3 External Storage Class

Example: the external declaration of integer number and float length might appear as

```
int number; float length=6.2; main()
```

```
{  
    -----;  
}  
function1()  
{  
    -----;  
}  
function2()  
{  
    -----;  
    -----;  
}
```

### 3.6.3 External Storage Class

#### Example: Program to illustrate how static variables work

```
#include<stdio.h>
#include<conio.h>
int fun1(void);
int fun2(void);
int fun3(void);
int x;      /*GLOBAL*/
main()
{
    x=10;      /*GLOBAL */
    printf("x=%d\n", x);
    printf("x=%d\n", fun1());
    printf("x=%d\n", fun2());
    printf("x=%d\n", fun3());
}
fun1(void)
{
    x=x+10;  /*GLOBAL*/
}
```

```
int fun2(void)
{
    int x ;      /*LOCAL*/
    x=1 ;
    return(x) ;
}
fun3(void)
{
    x=x+10;  /* GLOBAL*/
}
```

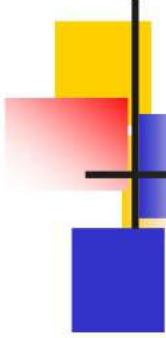
#### Output:

x=10

x=20

x=1

x=30

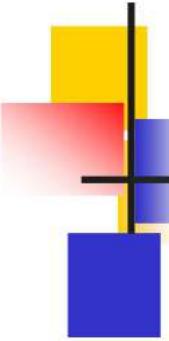


## 3.6.3 External Storage Class

### External Declaration

- In the above program, the main cannot access the variable y as it has been declared after the main function. This problem can be solved by declaring the variable with the storage class named as "**extern**".

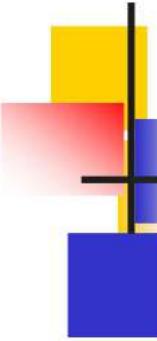
```
main()
{
    extern int y;          /*External declaration*/
    -----
}
func1()    /*External declaration*/
{
    extern int y;
}
int y;      /*Definition*/
```



## 3.6.4 Register Storage Class

---

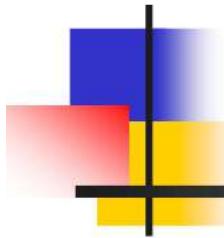
- Register is a special storage area in a Central Processing Unit (CPU). There are 8 registers available inside a Computer.
- Register variable can be accessed only by block in which it is declared. It cannot be accessed by any other function.
- Register variable declared using keyword register. Both Local variable and formal parameter can be declared as a register.
- Register is used to increase the execution speed. Only integer or char variables are declared as register in most of the compilers but ANSI C supports all the data types.



## 3.6.4 Register Storage Class

**Example: Program to illustrate how Register variables work**

```
#include<stdio.h>
#include<conio.h>
void main()
{
register int x;
clrscr();
for(x=1;x<=10;x++)
printf("%d",x);
getch();
}
```

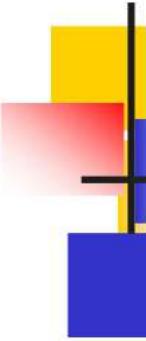


# **PROBLEM SOLVING USING C PROGRAMMING**

**Bachelor of Computer Application**

**SEMESTER - I**

**GURU NANAK COLLEGE(Autonomous)**  
**VELACHERY ROAD, CHENNAI – 600042**  
**(Re-Accredited 'A' grade by NAAC)**



# Syllabus – UNIT 4

---

- Arrays

- Defining and Processing
- Passing arrays to functions
- Multi-dimension arrays

- Arrays and String

- Structures

- User defined data types

- Unions



## 4.1 Arrays

### 1. Defining And Processing Arrays

- Many applications require the processing of multiple data items that have common characteristics. In such a situation it is convenient to place such data item in an **Array**
- An array is a collection of similar data items that are stored under a common name. A value in an array is identified by **index or subscript** enclosed in square brackets with **array name**.
- The individual data items can be integers, floating point numbers, and characters and so on, but they must be the same type and same storage class.
- Each array element is referred by specifying the array name with subscripts each subscripts enclosed in square brackets and each subscript must be a non-negative integer.

```
int A [5] =5  
A[0],A[1],A[2],A[3],A[4]
```



## 4.1 Arrays

---

### 1. Defining And Processing Arrays

- Thus 'n' elements array 'A' and the elements are

**A[0],A[1],A[2].....A[N-1]**

- The value of each subscript can be expressed as an integer constant or an integer variable or an integer expression.
- The arrays can be used to represent not only simple list of value but also task of data items in two and three or more dimensions.
- Arrays can be classified into
  - One-Dimensional arrays
  - Two-Dimensional arrays
  - Multi-Dimensional arrays



## 4.1 Arrays

---

### Array Declaration

- Arrays are declared in the same manner as an ordinary variables except that each array name must have the size of the array i.e., number of elements accommodate in that array.
- Like variables, the array must be declared before they are used.

#### ***Syntax :***

**data\_type array\_variable [Size or Subscript of the array];**

eg: char a[20];

- **data type** – Specifies the type of the data that will be contained in the array
- **array\_variable**- Specifies the name of the array
- **Size or Subscript** – Specifies the maximum number of elements that the array can hold

## 4.1 Arrays

- The subscript of an array can be integer constant, integer variable or an expression that yields an integer value

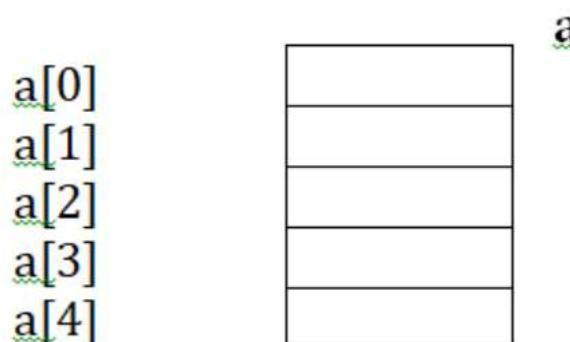
**Example:**      int a[5];

                  int n=20;

                  int a[n];

                  int a[x+y+z];

- Where, 'a' is the name of the array with 5 subscripts of integer data types and the computer reserves five storage location as shown below.





## 4.1 Arrays

---

### Processing an Array

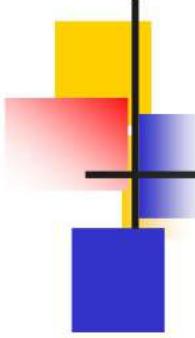
- The entire array cannot be accessed with single operation.
- So, the array elements must be accessed on an element-by element basis.
- This can be usually done with the help of the loop, when each pass of the loop is used to access an array element, thus the number of passes through the loop will therefore equal the number of array elements to be processed.



## 4.1 Arrays

---

- The entire array cannot be accessed with single operation.
- So, the array elements must be accessed on an element-by element basis.
- This can be usually done with the help of the loop, when each pass of the loop is used to access an array element, thus the number of passes through the loop will therefore equal the number of array elements to be processed.



## 4.2 Array Initialization

- The values can be initialized to an array, when they are declared like ordinary variable, otherwise they hold garbage values.
- The array can be initialized in the following two ways:
  - **At compile time**
  - **At run time**



## 4.2 Array Initialization

### i. At Compile Time

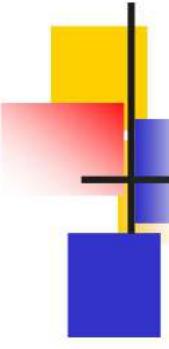
#### Syntax:

- `data_type array name [size]={List of values};` Where,
- The list of values must be separated by commas.

#### Example:

- `int marks[3]={70,50,86};`
- This statement declares the variable `marks` as an array of 3 elements and will be assigned to values specified in list as shown below

70	<code>marks[0]</code>
50	<code>marks[1]</code>
86	<code>marks[2]</code>



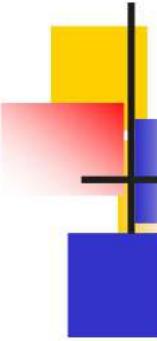
## 4.2 Array Initialization

---

### i. At Compile Time

//Program to read marks using array initialization

```
#include<stdio.h>
void main()
{
    int studmark[5]={99,97,87,89,92};
    int i;
    printf("mark of the student is:\n");
    for(i=0;i<=4;++i)
    {
        printf("%d\t",studmark[i]);
    }
    getch();
}
```



## 4.2 Array Initialization

---

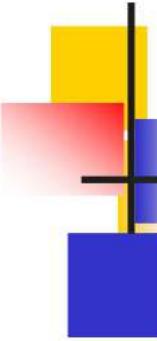
### ii. At Run Time

- The array can be explicitly initialized at run time

#### **Example :**

```
int n[2];
scanf("%d%d",&n[0],&n[1]);
```

- Like, the array can also be initialized by reading data items from the input



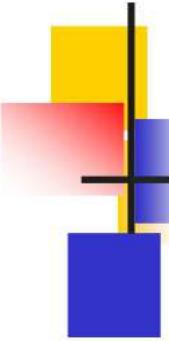
## 4.2 Array Initialization

---

### ii. At Run Time

//Input n numbers and display n numbers

```
#include<stdio.h>
void main()
{int a[100];
int i,n;
printf("number of elements in array\n");
scanf("%d",&n);
printf("enter the elements\n");
for(i=0;i<n;++i)
{
scanf("%d",&a[i]);
}
printf("elements in array\n");
for(i=0;i<=n-1;++i)
{
printf("%d\t",a[i]);
}
getch();
}
```

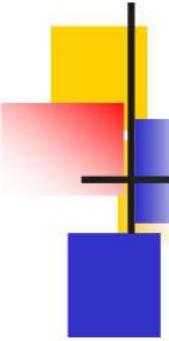


## 4.3 One-Dimensional Array

- The collection of data items can be stored under a one variable name using only one subscript, such a variable is the one-dimensional array.

//Add ten numbers and find sum and average

```
#include<stdio.h>
void main()
{
int i,num[10],sum=0;
float avg=0.0;
printf("enter ten numbers:\n");
for(i=0;i<10;i++)
{
scanf("%d",&num[i]);
sum+=num[i];
}
avg=(float)sum/10.0;
printf("\n sum of 10 number is:%7.2d",sum);
printf("\n average of 10 number is :%5.3f",avg);
getch();
}
```



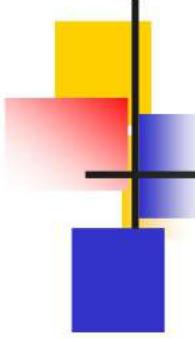
## 4.4 Two-Dimensional Array

- Two dimensional arrays are used in situation where a table of values need to be stored in an array.
- These can be defined in the same fashion as in one dimensional arrays, except a separate pair of square brackets are required for each subscript.
- Two pairs of square brackets required for two dimensional array and three pairs required for three dimensional arrays and so on.

### ***Syntax:***

**`data_type array_name[row size] [column size];`**

- **data\_type** - specifies the type of the data that will be contained in the array.
- **array\_name** - specifies the name of the array
- **[row size]** specifies the size of the row
- **[column size]** specifies the size of the column



## 4.4 Two-Dimensional Array

---

Example:

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

- Two dimensional arrays are stored in a row-column matrix, where the left index indicates the row the right indicates the column.
- Where 'a' is the array name and it reserves 3 row and 3 columns of memory as shown below

## 4.4 Two-Dimensional Array

```
***** MATRIX ADDITION ****/
#include <stdio.h>
int main() {
    int r, c, a[100][100], b[100][100], sum[100][100], i, j;
    printf("Enter the number of rows (between 1 and 100): ");
    scanf("%d", &r);
    printf("Enter the number of columns (between 1 and 100): ");
    scanf("%d", &c);

    printf("\nEnter elements of 1st matrix:\n");
    for (i = 0; i < r; ++i)
        for (j = 0; j < c; ++j) {
            printf("Enter element a%d%d: ", i + 1, j + 1);
            scanf("%d", &a[i][j]);
        }
    printf("Enter elements of 2nd matrix:\n");
    for (i = 0; i < r; ++i)
        for (j = 0; j < c; ++j) {
            printf("Enter element a%d%d: ", i + 1, j + 1);
            scanf("%d", &b[i][j]);
        }

    // adding two matrices
    for (i = 0; i < r; ++i)
        for (j = 0; j < c; ++j) {
            sum[i][j] = a[i][j] + b[i][j];
        }

    // printing the result
    printf("\nSum of two matrices: \n");
    for (i = 0; i < r; ++i)
        for (j = 0; j < c; ++j) {
            printf("%d ", sum[i][j]);
            if (j == c - 1) {
                printf("\n\n");
            }
        }
    return 0;
}
```



## 4.4 Two-Dimensional Array

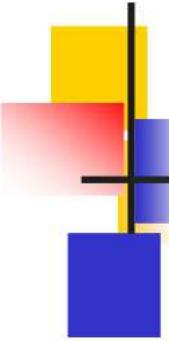
### Initializing an Two-Dimensional Array.

- A Two-Dimensional Array can be also initialized. For that the array values are specified within a Compound statement. (i.e.,) { and }.

#### General form :-

**Storage-class data-type array-name[r][c] = { value1, value2, , value n};**

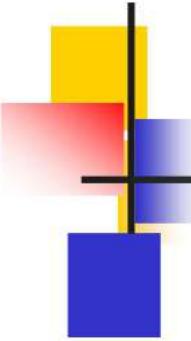
- Here storage class may be static or extern by default. Storage-class is optional.
- Data-type refers to a valid C data type.
- Array-name refers to a valid array name.
- Here r stands for number of rows and c stands for number of columns.
- Each values in an array are separated by commas and terminated by semicolon.



## 4.4 Two-Dimensional Array

---

```
#include<stdio.h>
void main()
{
int i,j;
float a[4][2]={{12.3,34.5},{23.4,45.6},{34.5,56.7},{45.6,67.8}};
printf("Element value Address");
for(i=0;i<4;i++)
{
    for(j=0;j<2;j++)
    {
        printf("\n a[%d] [%d] %0.2f %p",i,j,a[i][j],&a[i][j]);
    }
}
getch();
}
```



## 4.5 Multi-Dimensional Array

- Multi-dimensional Array consists of (or) requires more than two square brackets and it may contain any number of values specified within square brackets. It may be three, four, five, six and so on.
- A Multi dimensional Array in general takes the following form.

### General form :-

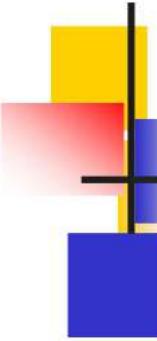
Storage-class data-type array-name[s1][s2] [sn];

- Here storage class may be static or extern by default. Here Storage-class is optional. Data-type refers to a valid C data type. Array-name refers to a valid array name. s1,s2,s3, ..... are sub scripts.



# Array Example

```
/* Sort number in ascending order*/
#include<stdio.h>
void main()
{
int i,j,k,num[5],temp;
clrscr();
printf("Enter five numbers:\n");
for(i=0;i<5;i++)
{
scanf("%d",&num[i]);
}
printf("\n THE ORIGINAL LIST IS:\n");
for(i=0;i<5;i++)
{
printf("%d\t",num[i]);
}
for(i=0;i<5;i++)
{
for(j=i+1;j<5;j++)
{
if num[i]>num[j]
{
temp=num[i]; num[i]=num[j]; num[j]=temp;
}
}
printf("\n the sorted numbers are:\n");
for(i=0;i<5;i++)
printf("%d\t",num[i]); getch();
}
```

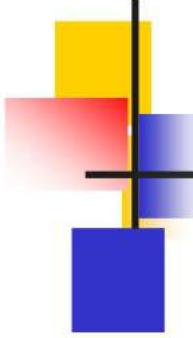


# Features of Array

---

## Features of Arrays :

- An array is a derived data type. It is used to represent a collection of elements of the same data type.
- The elements can be accessed with base address (index) and the subscripts define the position of the element.
- In array the elements are stored in continuous memory location. The starting memory location is represented by the array name and it is known as the base address of the array.
- It is easier to refer the array elements by simply incrementing the value of the subscript.



## 4.6 Arrays and Strings

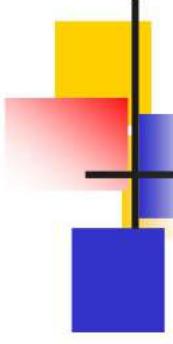
---

### String Manipulation

- In 'C' language the group of character, digits and symbols enclosed within quotation marks are called as string otherwise string are array of characters. Null character('0') is used to mark the end of the string.

**Example: char name[]={'S','T','R','I','N','G','/0'}**

- Each character is stored in one byte of memory and successive characters of the string are stored in successive byte



## 4.6 Arrays and Strings

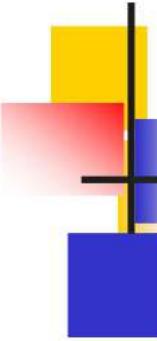
### Reading and Writing String

- The '%s' control string can be used in scanf() statement to read a string from the terminal and the same may be used to write string to the terminal in printf() statement.

**Example:**

```
char name[10];
scanf("%s",name);
printf("%s",name);
```

**\*\* there is no address (&) operator used in scanf() statement**



## 4.6 Arrays and Strings

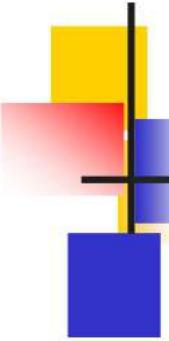
---

### Character Array

- Character Array is specified within single quotes and ended with semicolon. It is used to define a single character for the array.

*// Input 10 characters and print 10 characters*

```
#include<stdio.h>
void main()
{
    char character[10];
    int cnt;
    clrscr();
    /*read character one by one*/
    printf("enter 10 character\n");
    for(cnt=0;cnt<10;cnt++)
        character[cnt]=getchar();
    /*display character one by one*/
    printf("\nentered characters are :\n");
    for(cnt=0;cnt<10;cnt++)
        putchar(character[cnt]);
}
```



# 4.6 Arrays and Strings

## Strings Standard Function

- The commonly used string manipulation functions are follows

### 1. The **strlen()** function

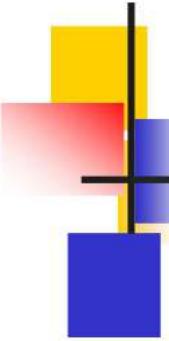
- This function is used to count and return the number of character present in a string

**Syntax: var =strlen(string);**

- where, var - Is the integer variable, which accepts the length of the string
- string - Is the string constant or string variable in which the length is going to be found.

### Example: Program using **strlen()** function

```
#include<stdio.h>
#include<string.h>
main()
{
    char name[100]; int length;
    printf("Enter the string");
    gets(name);
    length=strlen(name);
    printf("\nNumber of characters in the string is=%d",length);
}
```



## 4.6 Arrays and Strings

### 2. The strcpy() function

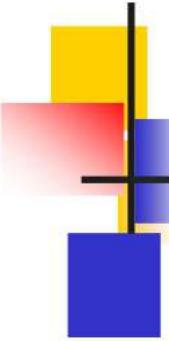
- This function is used to copy the contents of one string to another and it almost works like string assignment operator.

**Syntax: strcpy(string1,string2);**

- string1 is the destination string string2 is the source string
- i.e., The contents of string2 is assigned to the contents of string1. where string2 may be character array variable or string constant.

### Example: Program using strcpy() function

```
#include<stdio.h>
#include<string.h>
main()
{
    char source = "Welcome";
    char target[10];
    strcpy(target,source);
    printf("\n Source string is %s",source);
    printf("\n Target string is %s",target);
}
```



## 4.6 Arrays and Strings

### 3. The strcat() function

- The strcat() function is used to concatenate or combine, two strings together and forms a new concatenated string.

**Syntax : strcat(string1,string2);**

- where, string1 and string2 are character type arrays or string constants. When the above strcat function is executed, string2 is combined with string1 and it removes the null character(/0) of string1 and places string2.

#### **Example: Program using strcat() function**

```
#include<stdio.h>
#include<string.h>
main()
{
    char source[10] = "Ramesh";
    char target[10] = "Babu";
    strcat(source,target);
    printf("\n Source string is %s",source);
    printf("\n Target string is %s",target);
}
```

#### **OUTPUT**

Source string is RameshBabu

Target string is Babu

# 4.6 Arrays and Strings

## 4. The strcmp() function

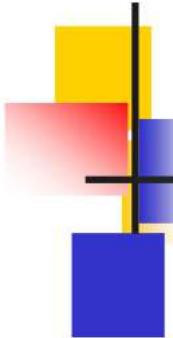
- This is a function which compares two strings to find out whether they are same or different. The two strings are compared, character by character until the end of one of the string is reached. If the two strings are identical strcmp() returns a value zero.
- If they are not equal, it returns the numeric difference between the first non-matching characters

**Syntax: strcmp(string1,string2);**

- string1 and string2 are character type arrays or string constants

### Example: Program using strcmp() function

```
#include<stdio.h>
#include<string.h>
main()
{
char s1[20],s2[20];
int x;
printf("Enter the strings");
scanf("%s%s",s1,s2);
x=strcmp(s1,s2);
if(x!=0)
{
printf("\nStrings are not equal\n"); else
printf("\nStrings are equal");
}
```



## 4.6 Arrays and Strings

### 5. The strrev() function

- The strrev function is used to reverse a string. This function takes only one argument and return one argument. The general form of strrev() function is

**Syntax: strrev(string);**

String are characters type arrays of\r string constants

#### **Example: Program using strrev() function**

```
#include<stdio.h>
#include<string.h>
main()
{
char a[30];
printf("Enter the string:");
gets(a);
printf("The string reversed is : %s", strrev(a));
}
```

#### **OUTPUT**

- Enter the string : array
- The string reversed is : yarra

# 4.6 Arrays and Strings

## STRING LIBRARY FUNCTIONS

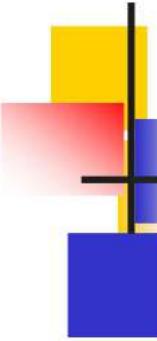
- The header file related to string functions is <string.h>

FUNCTIONS	MEANING
strcmp(s1,s2)	Compares two strings. Return negative value if s1<s2. 0 if s1 and s2 are identical. Return positive value if s1>s2.
strcpy(s1,s2)	Copies a string s2 to another string s1.
strcat(s1,s2)	Combines the string s1 at the end of string s2.
strchr(s1,c)	Finds first occurrence of a given character in a string. Compare characters of string s1 with character starting from head of string s1.
strlen(s)	Find length of a string s.
strlwr(s)	Converts string s to lowercase
strrev(s)	Reverse the string s.
strupr(s)	Converts the string s to uppercase
strdup(s)	Duplicate string s.
strncpy(s1,s2,n)	Copies portion of string s2 to string s1 upto position n.
strncmp(s1,s2,n)	Compares portion of string s2 to string s1 upto position n.
strrchr()	Find last occurrence of a given character in a string.
strstr()	Find first occurrence of a given string in another string.
strcmpi()	Compares two strings without regard to case ("i" denotes that this function ignores case ).



## 4.7 Structures

- C supports a constructed data type known as structures, a mechanism for packing data of different types.
- "*A structure is a convenient tool for handling a group of logically related data items*".
  - Structure is a type of data structure in which each individual elements differ in type.
  - The elements of a structure are called members.
  - The structure elements contain integer, floating point numbers, character arrays, pointers as their members.
  - Structure act as a tool for handling logically related data items.
  - Fields of structure are called structure elements or members. Each member may be of different type.



## 4.7 Structures

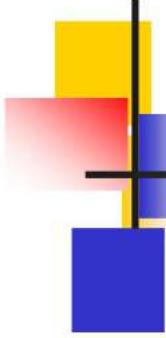
---

- **Syntax:**

```
struct structure_name
{
    data_type member1;
    data_type member2;
    .
    .
    data_type memebn;
} [one or more structure variables];
```

- **Example:**

```
struct Books
{
    char title[50];
    char author[50];
    char subject[100];
    int book_id;
} book;
```



## 4.7 Structures

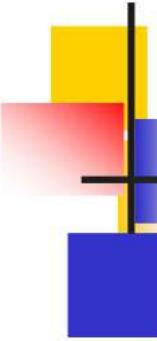
---

### ■ Structure Initialization

1. When we declare a structure, memory is not allocated for uninitialized variable.
2. We can initialize structure variable in different ways.

#### i. **Declare and Initialize**

```
struct student
{
    char name[20];
    int roll;
    float marks;
}std1 = { "Pritesh",67,78.3 };
```



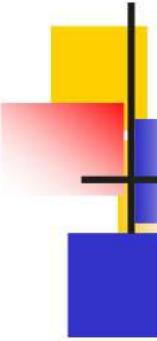
## 4.7 Structures

### ii. Declaring and Initializing Multiple Variables

```
struct student
{
    char name[20];
    int roll;
    float marks;
}
std1 = {"Pritesh",67,78.3};
std2 = {"Don",62,71.3};
```

### iii. Initializing Single Member

```
struct student
{
    int mark1;
    int mark2;
    int mark3;
} sub1={67};
```



## 4.7 Structures

---

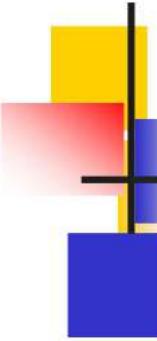
### iv. Initializing Inside Main

```
struct student
{
    int mark1; int mark2; int mark3;
};

void main()
{
    struct student s1 = {89,54,65};
    -----
    -----
}
```

### iv. Initializing Single Member

```
struct student
{
    int mark1;
    int mark2;
    int mark3;
} sub1={67};
```



## 4.7 Structures

---

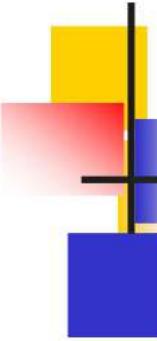
### ■ **Uses of Structures**

- Used to store huge data. Structures act as a database.
- Used to send data to the printer.
- Structures can interact with keyboard and mouse to store the data.
- Used in drawing and floppy formatting.
- Used to clear output screen contents.
- Used to check computer's memory size etc.

### ■ **Accessing structure members:**

- Members in a structure are accessed using the Period Operator '.'.
- Period Operator establishes a link between member and variable name.
- Structure members are processed by using a variable name with period '.' and member name.
- Period Operator is also known as member operator or dot operator.
- The syntax for accessing structure members is

**structure\_variable.member\_name**



## 4.7 Structures

---

### ■ Rules of Initializing Structures

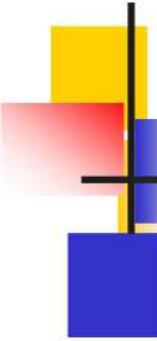
1. We cannot initialize individual members inside the structure template.
2. The order of values enclosed in braces must match the order of members in the structure definition.
3. It is permitted to have a partial initialization. We can initialize only the first few members and leave the remaining blank. The uninitialized members should be only at the end of the list.
4. The uninitialized members will be assigned default values as follows
  - Zero for integer and floating point number.
  - '0' for characters and strings.

# 4.7 Structures

## Structure Operations

- The period operator '.' has high precedence over unary operator, arithmetic operator, relational operator, logical operator and assignment operator.

EXPRESSION	MEANING
<code>++(variable.member)</code>	preincrement
<code>(variable.member)++</code>	post increment
<code>&amp;variable</code>	Beginning address of the variable.
<code>&amp;variable.member</code>	Access address of variable.member
<code>variable.member.submember</code>	Refers member of structure within a structure.
<code>variable.member[expression]</code>	Refer individual element of array in a structure



## 4.7 Structures

---

### 1. Example Program using structure:

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

struct Books
{
    char title[50];
    char author[50]; char subject[100]; int book_id;
};

int main( )
{
    struct Books Book1; /* Declare Book1 of type Book */
    struct Books Book2; /* Declare Book2 of type Book */

    /* book 1 specification */
    strcpy( Book1.title, "C Programming");
    strcpy( Book1.author, "Nuha Ali");
    strcpy( Book1.subject, "C Programming Tutorial");
    Book1.book_id = 6495407;
```

# 4.7 Structures

## Example Program using structure:

```
/* book 2 specification */
strcpy( Book2.title, "Telecom Billing");
strcpy( Book2.author, "Zara Ali");
strcpy( Book2.subject, "Telecom Billing Tutorial");
Book2.book_id = 6495700;

/* print Book1 info */
printf( "Book 1 title : %s\n", Book1.title);
printf( "Book 1 author : %s\n", Book1.author);
printf( "Book 1 subject : %s\n", Book1.subject);
printf( "Book 1 book_id : %d\n", Book1.book_id);

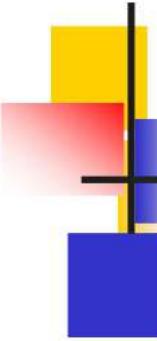
/* print Book2 info */
printf( "Book 2 title : %s\n", Book2.title);
printf( "Book 2 author : %s\n", Book2.author);
printf( "Book 2 subject : %s\n", Book2.subject);
printf( "Book 2 book_id : %d\n", Book2.book_id);

return 0;
}
```

### Output:

Book 1 title : C Programming  
Book 1 author : Nuha Ali  
Book 1 subject : C Programming Tutorial  
Book 1 book\_id : 6495407

Book 2 title : Telecom Billing  
Book 2 author : Zara Ali  
Book 2 subject : Telecom Billing Tutorial  
Book 2 book\_id : 6495700



## 4.7.1 ARRAYS AND STRUCTURES

### 2. Example Program using ARRAYS AND STRUCTURES:

- C Structure is collection of different datatypes ( variables ) which are grouped together.
- Whereas, array of structures is nothing but collection of structures.
- This is also called as structure array in C.

```
#include <stdio.h>
#include <string.h>
struct student
{
    int id;
    char name[30]; float percentage;
};

int main()
{
    int i;
    struct student record[2];

    // 1st student's record
    record[0].id=1;
    strcpy(record[0].name, "Raju");
    record[0].percentage = 86.5;
```



## 4.7.1 ARRAYS AND STRUCTURES

### Example Program using ARRAYS AND STRUCTURES:

```
// 2nd student's record
record[1].id=2;
strcpy(record[1].name, "Surendren");
record[1].percentage = 90.5;

// 3rd student's record
record[2].id=3;
strcpy(record[2].name, "Thiyagu");
record[2].percentage = 81.5;

//Printing Student's record
for(i=0; i<3; i++)
{
    printf("    Records of STUDENT : %d \n", i+1);
    printf(" Id is: %d \n", record[i].id);
    printf(" Name is: %s \n", record[i].name);
    printf(" Percentage is: %f\n",record[i].percentage);
}
return 0;
}
```

#### Output:

```
Records of STUDENT : 1
Id is: 1
Name is: Raju
Percentage is: 86.500000 Records of
STUDENT : 2
Id is: 2
Name is: Surendren
Percentage is: 90.500000
Records of STUDENT : 3
Id is: 3
Name is: Thiyagu
Percentage is: 81.500000
```



## 4.7.2 NESTED STRUCTURES

- Nested structure in C is nothing but structure within structure.
- One structure can be declared inside other structure as we declare structure members inside a structure.
- The structure variables can be a normal structure variable or a pointer variable to access the data.

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

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

```
struct student_college_detail
{
    int college_id;
    char college_name[50];
};
```

```
struct student_detail
{
    int id;
    char name[20];
    float percentage;
    // structure within structure
    struct student_college_detail clg_data;
}stu_data;
```



## 4.7.2 NESTED STRUCTURES

---

```
int main()
{
    struct student_detail stu_data = {1, "Raju", 90.5, 71145,"Anna University"};
    printf(" Id is: %d \n", stu_data.id);
    printf(" Name is: %s \n", stu_data.name);
    printf(" Percentage is: %f \n", stu_data.percentage);
    printf(" College Id is: %d \n", stu_data.clg_data.college_id);
    printf(" College Name is: %s \n", stu_data.clg_data.college_name);
    return 0;
}
```

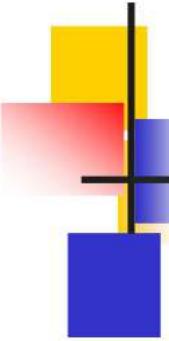
### OUTPUT

```
Id is: 1
Name is: Raju
Percentage is: 90.500000
College Id is: 71145
College Name is: Anna University
```



## 4.7.3 PASSING STRUCTURES TO FUNCTIONS

- A structure can be passed to any function from main function or from any sub function.
- Structure definition will be available within the function only.
- It won't be available to other functions unless it is passed to those functions by value or by address(reference).
- Else, we have to declare structure variable as global variable. That means, structure variable should be declared outside the main function. So, this structure will be visible to all the functions in a C program.
- Passing structure to function can be done in three ways
  - Passing structure to a function by value
  - Passing structure to a function by address(reference)
  - No need to pass a structure – Declare structure variable as global



## 4.7.3 PASSING STRUCTURES TO FUNCTIONS

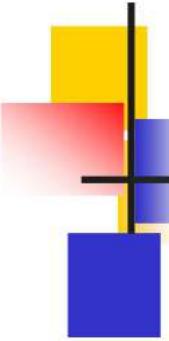
- **Passing structure to a function by value**

- In this program, the whole structure is passed to another function by value. It means the whole structure is passed to another function with all members and their values. So, this structure can be accessed from called function.

```
#include <stdio.h>
#include <string.h>
struct student
{
    int id;
    char name[20];
    float percentage;
};
void func(struct student record);

int main()
{
    struct student record;
    record.id=1;
    strcpy(record.name, "Raju");
    record.percentage = 86.5;
    func(record);
    return 0;
}
```

```
void func(struct student record)
{
    printf(" Id is: %d \n", record.id);
    printf(" Name is: %s \n", record.name);
    printf(" Percentage is: %f \n", record.percentage);
}
```



## 4.7.3 PASSING STRUCTURES TO FUNCTIONS

### ■ Passing structure to a function by address

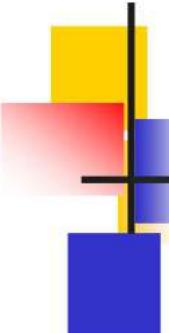
- In this program, the whole structure is passed to another function by address. It means only the address of the structure is passed to another function. The whole structure is not passed to another function with all members and their values. So, this structure can be accessed from called function by its address.

```
#include <stdio.h>
#include <string.h>
struct student
{
    int id;
    char name[20];
    float percentage;
};
void func(struct student *record)

int main()
{
    struct student record;

    record.id=1;
    strcpy(record.name, "Raju");
    record.percentage = 86.5;
    func(&record);
    return 0;
}
```

```
void func(struct student *record)
{
    printf(" Id is: %d \n", record->id);
    printf(" Name is: %s \n", record->name);
    printf(" Percentage is: %f \n", record->percentage);
}
```



## 4.7.3 PASSING STRUCTURES TO FUNCTIONS

- **To declare structure variable as global**

- Structure variables also can be declared as global variables as we declare other variables in C. So, When a structure variable is declared as global, then it is visible to all the functions in a program. In this scenario, we don't need to pass the structure to any function separately.

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

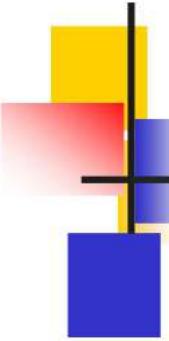
struct student
{
    int id;
    char name[20];
    float percentage;
};

struct student record; // Global declaration of structure
void structure_demo();

int main()
{
    record.id=1; strcpy(record.name, "Raju");
    record.percentage = 86.5;

    structure_demo(); return 0;
}

void structure_demo()
{
    printf(" Id is: %d \n", record.id);
    printf(" Name is: %s \n", record.name);
    printf(" Percentage is: %f \n", record.percentage);
}
```

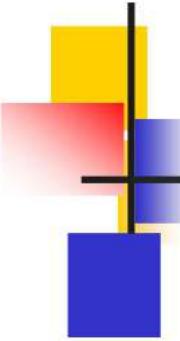


## 4.8 UNION

- A **union** is a special data type available in C that allows to store different data types in the same memory location. Unions provide an efficient way of using the same memory location for multiple-purpose.

- **Concept/Advantages of Union**

- The compiler allocates sufficient space to hold the **largest data item** in the union and not for all members.
- They are used to conserve memory.



## 4.8 UNION

---

- Defining Union

```
union [union tag]
```

```
{
```

```
    member definition;
```

```
    member definition;
```

```
    ...
```

```
    member definition;
```

```
}
```

 [one or more union variables];

- Example

```
union Data
```

```
{
```

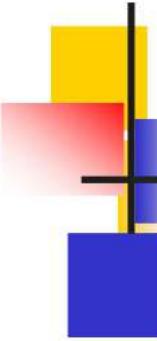
```
    int i;
```

```
    float f;
```

```
    char str[20];
```

```
}
```

 data;



## 4.8 UNION

- Example Program to Find Total Memory Occupied by Union

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

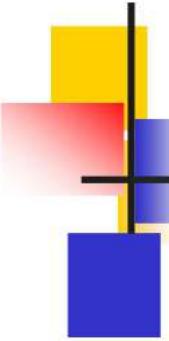
```
union Data
{
    int i; float f;
    char str[20];
};
```

```
int main( )
{
    union Data data;
    printf( "Memory size occupied by data : %d\n", sizeof(data));
    return 0;
}
```

**OUTPUT:**

Memory size occupied by data : 20

- In the above example, Data type will occupy 20 bytes of memory space because this is the maximum space which can be occupied by a character string.



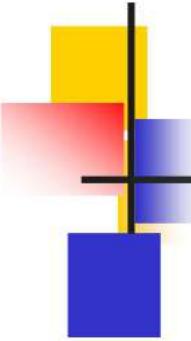
## 4.8 UNION

### ■ Accessing Union Members

- To access any member of a union, we use the member access operator (.) .
- The member access operator is coded as a period between the union variable name and the union member that we wish to access.
- The keyword union is used to define variables of union type.
- The following example shows how to use unions in a program –

```
#include <stdio.h>
#include <string.h>
union Data
{
    int i; float f;
    char str[20];
};

int main( )
{
    union Data data;
    data.i = 10;
    printf( "data.i : %d\n", data.i);
```



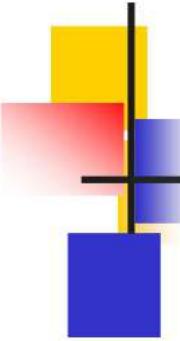
## 4.8 UNION

---

```
data.f = 220.5;  
printf( "data.f : %f\n", data.f);  
strcpy( data.str, "C Programming");  
printf( "data.str : %s\n", data.str);  
  
return 0;  
}
```

**OUTPUT**

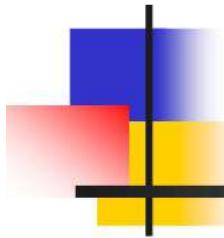
```
data.i : 10  
data.f : 220.500000  
data.str : C Programming
```



## 4.8 UNION

### ■ Difference between Union and Structure

- A **union** allocates the memory equal to the maximum memory required by the member of the union. However, a **structure** allocates the memory equal to the **total memory** required by the members.
- In a **union**, **one block** is used by all the member of the union. However, in case of a **structure**, each member has its **own memory space**.



# **PROBLEM SOLVING USING C PROGRAMMING**

**Bachelor of Computer Application  
SEMESTER - I**

**GURU NANAK COLLEGE(Autonomous)  
VELACHERY ROAD, CHENNAI – 600042  
(Re-Accredited 'A' grade by NAAC)**



# Syllabus – UNIT 5

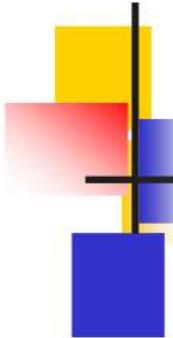
---

## ■ Pointers

- Declarations
- Passing pointers to Functions
- Operation in Pointers
- Pointer and Arrays
- Arrays of Pointers

## ■ Files :

- Creating ,
- Processing,
- Opening and Closing a data file.



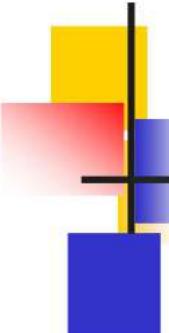
## 5.1 Pointers

---

- A **pointer** is a variable which contains the address in memory of another variable. We can have a **pointer** to any variable type.
- The unary or monadic operator **&** gives the “address of a variable”.
- The indirection or dereference operator **\*** gives the “contents of an object pointed to by a **pointer**”.

**General form:** datatype \*variablename;

**Example:** int \*ptr;



## 5.1 Pointers

---

### ■ Uses of a Pointer

- Allows to keep track of address of memory locations.
- Allows to easily manipulate data in different memory locations.
- Allows dynamic allocation of memory
- Allows accessing array elements, passing arrays and strings to functions, creating data structures such as linked lists, trees, graphs and so on.

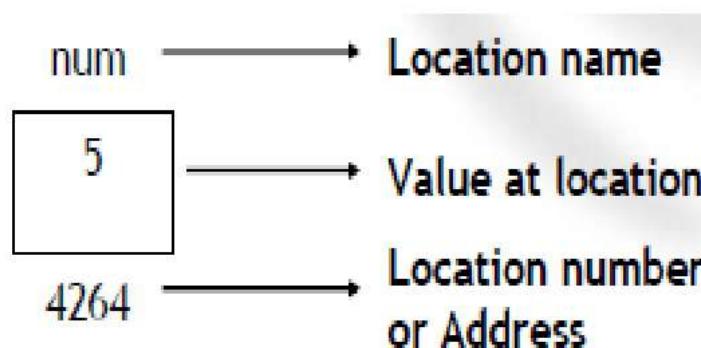
## 5.1 Pointers

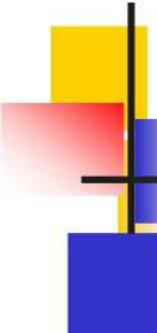
### Example of a Pointer

- Consider the following declaration

```
int num=5;
```

- The compiler will automatically assign memory for this data item.
- The data item can be accessed if we know the location(that is the address) of the first memory cell.





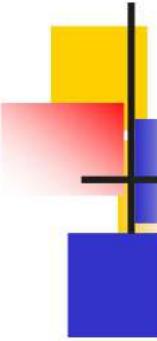
## 5.1 Pointers

### ■ Example Program for Usage of Pointers

```
#include <stdio.h>
int main ()
{
    int var = 20; /* actual variable declaration */
    int *ip;      /* pointer variable declaration */
    ip = &var; /* store address of var in pointer variable*/
    printf("Address of var variable: %x\n", &var );
    /* address stored in pointer variable */
    printf("Address stored in ip variable: %x\n", ip );
    /* access the value using the pointer */
    printf("Value of *ip variable: %d\n", *ip );
    return 0;
}
```

#### OUTPUT

Address of var variable : bffd8b3c  
Address stored in ip variable : bffd8b3c  
Value of \*ip variable : 20



## 5.1 Pointers

### ■ **NULL Pointer**

- It is always a good practice to assign a NULL value to a pointer variable in case we do not have an exact address to be assigned.
- This is done at the time of variable declaration.
- A pointer that is assigned NULL is called a **null** pointer.
- The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program –

```
#include <stdio.h>
int main ()
{
    int *ptr = NULL;
    printf("The value of ptr is : %x\n", ptr );
    return 0;
}
```

### **OUTPUT**

The value of ptr is : 0

## 5.2 Pointers and Function

### ■ NULL Pointer

- A function like a variable has a type and an address location in the memory. It is therefore possible to declare a pointer to a function, which can be used as an argument in another function.
- Pointer to a function contains the address of function in memory. Pointers are passed to a function as arguments.

```
#include <stdio.h>
/* function declaration */
double getAverage(int *arr, int size);
int main ()
{
    /* an int array with 5 elements */
    int balance[5] = {1000, 2, 3, 17, 50};
    double avg;
    /* pass pointer to the array as an argument */
    avg = getAverage( balance, 5 ) ;
    /* output the returned value */
    printf("Average value is: %f\n", avg );
    return 0;
}
```

```
double getAverage(int *arr, int size)
{
    int i, sum = 0;
    double avg;
    for (i = 0; i < size; ++i)
    {
        sum += arr[i];
    }
    avg = (double)sum / size;
    return avg;
}
```

### OUTPUT

Average value is: 214.40000

## 5.3 Operation in Pointers

### POINTERS AND STRINGS

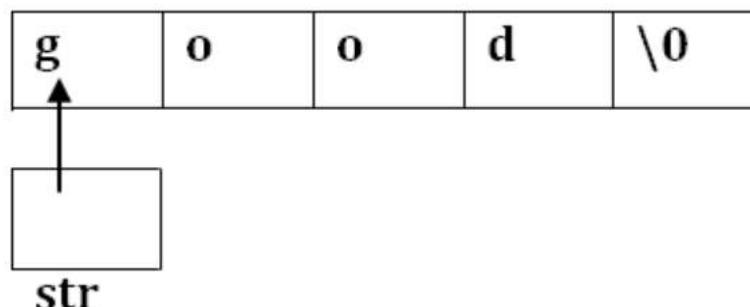
- Strings are treated like character arrays and therefore, they are declared and initialized as follows.

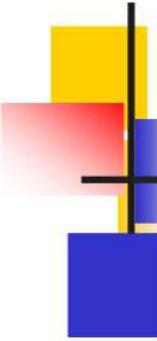
```
char str[5] = "good";
```

- The compiler automatically inserts **the null character '\0'** at the end of the string. C supports an alternative method to create strings using pointer variables of type **char**. For example

```
char *str = "good";
```

- This creates a string for the literal and then stores its address in the pointer variable **str**. The pointer str now points to the first character of the string “good” as:





## 5.3 Operation in Pointers

### POINTERS AND STRINGS

- We can also use the run-time assignment for giving values to a string pointer. For example

```
char * string1;  
string1="good";
```

- Note that the assignment

```
string1="good";
```

- is not a string copy, because the variable **string1**
- is a pointer, not a string.
- C does not support copying one string to another through assignment operation.
- We can print the content of the string **string1** using either **printf** or **puts** function as follows

```
printf("%s", string1);  
puts(string1);
```

# 5.3 Operation in Pointers

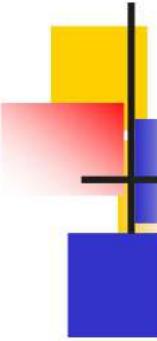
- **Example Program to determine the length of a character string using pointers**

```
#include<stdio.h>
#include<conio.h>
void main()
{
    char *name;
    int length;
    char *cptr=name;
    name="India";
    printf("%s\n",name);
    while(*cptr !='\0')
    {
        printf("%c is stored at address %u\n", *cptr, cptr);
        cptr++;
    }
    length =cptr-name;
    printf("\n Length of the string=%d\n", length);
}
```

## OUTPUT

INDIA

I is stored at address :	24
N is stored at address :	25
D is stored at address :	26
I is stored at address :	27
A is stored at address :	28



## 5.4 POINTERS AND ARRAYS

---

- Array is a collection of similar data type elements stored under common name. When we declare an array the consecutive memory location are located to the array of elements.
- The elements of an array can be efficiently accessed by using pointers.
- Array elements are always stored in consecutive memory location according to the size of the array.
- The size of the data type with the pointer variables refers to, depends on the data type pointed by the pointer.
- A pointer when incremented, always points to a location after skipped the number of bytes required for the data type pointed to by it.

## 5.4 POINTERS AND ARRAYS

```
#include <stdio.h>
int main ()
{
/* an array with 5 elements */
double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
double *p;
int i;
p = balance;
/* output each array element's value */
printf("Array values using pointer\n");
for ( i = 0; i < 5; i++ )
{
printf(*(p + %d) : %f\n", i, *(p + i) );
}
printf( "Array values using balance as address\n");
for ( i = 0; i < 5; i++ )
{
printf(*(balance + %d) : %f\n", i, *(balance + i) );
}
return 0;
}
```

### OUTPUT

Array values using pointer

\*(p + 0) : 1000.000000

\*(p + 1) : 2.000000

\*(p + 2) : 3.400000

\*(p + 3) : 17.000000

\*(p + 4) : 50.000000

Array values using balance as address

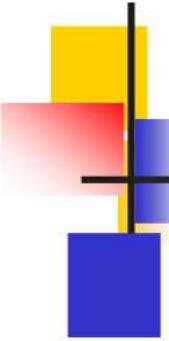
\*(balance + 0) : 1000.000000

\*(balance + 1) : 2.000000

\*(balance + 2) : 3.400000

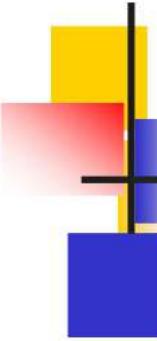
\*(balance + 3) : 17.000000

\*(balance + 4) : 50.000000



## 5.5 FILES

- "**A file is a collection of related information that is permanently stored on the disk and allows us to access and alter the information whenever necessary.**"
- Group of related record form a FILE. FILE is a place on the disk where group of related data is stored. DATA FILE is used to store information in floppy disk, hard disk. Field is a data item used to store a single unit. Group of related field constitute a RECORD. FILE is classified into three types. They are
  - **Numeric file**
    - It contains numeric data.
  - **Text file**
    - It consists of text data.
  - **Data files**
    - It consists of data grouped in the form of structure. It may have text, numeric data or both.

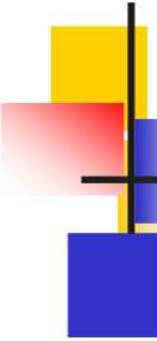


## 5.6 FILES data type

- FILE is a structure data type. It is used to establish file buffer area.
- To get buffer area the syntax is

*FILE \*fp;*

- where fp is a file pointer or stream pointer. fp contains all the information about file. It serves as link between system and program. FILE is a keyword. It refers to the file control structure for streams.

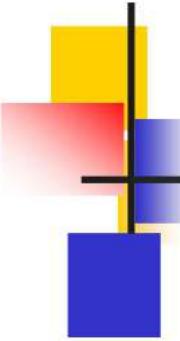


## 5.7 FILES Processing

---

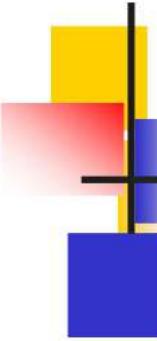
### FILE OPERATION LEVELS

- There are two FILE operation levels. They are
  - **Low level Input/Output operations**
    - It uses system calls.
  - **High level Input/Output operations**
    - It uses functions in standard C Input Output Library.
- To store data in a file Secondary Memory is used. When a file is retrieved from an Operating System OS specifies the filename, Data structures and Purpose.
- Here File name is a string of characters. It contains two parts. First one is Primary name and then the other is optional period with extension.



## 5.7 FILES

Function Name	Operations
fopen()	Creates a new file for use. Opens an existing file for use.
fclose()	Closes a file which has been opened for use.
fgetc()	Reads a character from a file.
fputc()	Writes a character to the file.
fprintf()	Writes set of data values to a file.
fscanf()	Reads a set of data values from a file.
getw()	Reads an integer from a file.
putw()	Writes an integer to a file.
fseek()	Sets the position to desired position in the file.
ftell()	Gives the current position in the file.
rewind()	Sets the position to the beginning of the file
fgets()	Reads a string from the file
fputs()	Writes a string to the file.
fread()	Reads unformatted data from the file
fwrite()	Writes unformatted data to the file

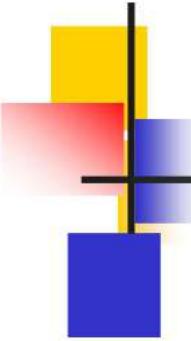


## 5.8 Opening a file

- “fopen” function is used to open a stream or data file. File should be opened before reading or writing from it. If a file is opened ‘fopen’ functions returns FILE pointer to the beginning of buffer area. If file is not opened NULL value is returned.
- The Syntax of fopen is

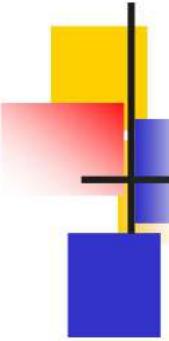
*FILE \*fopen("filename","mode");*

- fopen() function returns pointer to structure associated with the file. fopen() function needs two arguments of type string.
- First argument filename refers to the name of file which is to be opened. Second argument mode refers to file mode.



## 5.8 Opening a file

File Type	Modes of File Operations
"w"	Opens new file for writing. If the file exists contents of the file are overwritten.
"rb"	Opens a binary file for reading.
"a+"	Opens existing file for reading and writing.
"w+b" or "wb+"	Creates and opens binary file for read and write operations.
"a+b" or "ab+"	Opens a binary file for read and write operations.

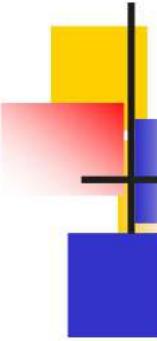


## 5.9 Closing a file

---

- File is closed when all input output operations are completed.
- `fclose()` function is used to close opened file.
- This function makes all buffers associated with the file empty and all links to the file broken.
- Whenever we want to reopen the same file we have to close all other file.
- The syntax for closing a file is

```
fclose(fp);
```



## 5.9 Closing a file

---

```
#include< stdio.h >
void main()
{
    file *f1;
    clrscr();
    printf("Data input output");
    f1=fopen("Input","w");
    /*Open the file Input*/
    while((c=getchar())!=EOF) /*get a character from key board*/
        putc(c,f1); /*write a character to input*/
    fclose(f1); /*close the file input*/
    printf("\nData output\n");
    f1=fopen("INPUT","r");
    /*Reopen the file input*/
    while((c=getc(f1))!=EOF)
        printf("%c",c);
    fclose(f1);
}
```



## 5.9 OPERATIONS ON FILES

---

- There are eight operations on files. They are
  - **putc()**
  - **getc()**
  - **getw()**
  - **putw()**
  - **fscanf()**
  - **fread()**
  - **fprintf()**
  - **fwrite()**

## 5.9 OPERATIONS ON FILES

```
#include< stdio.h >
void main()
{
FILE *f1,*f2,*f3;
int number i;
printf("Contents of the data file\n\n");
f1=fopen("DATA","W");
for(i=1;i< 30;i++)
{
scanf("%d",&number);
if(number== -1)
break;
putw(number,f1);
}
fclose(f1);
f1=fopen("DATA","r");
f2=fopen("ODD","w");
f3=fopen("EVEN","w");
while((number=getw(f1))!=EOF)
/* Read from data file*/
{
if(number%2==0)
putw(number,f3); /*Write to even file*/
else
putw(number,f2); /*write to odd file*/
}
fclose(f1); fclose(f2); fclose(f3);
f2=fopen("ODD","r");
f3=fopen("EVEN","r");
printf("\n\nContents of the odd file\n\n");
while(number=getw(f2))!=EOF)
printf("%d",number);
printf("\n\nContents of the even file");
while(number=getw(f3))!=EOF)
printf("%d",number);
fclose(f2);
fclose(f3);
getch();
}
```