CHAPTER1
INTRODUCTION TO LANGUAGES

1. COMPUTER SYSTEM:

What is computer? Explain parts of computer?
Computer is a device capable of performing computations and making logical decisions at speed of millions and even billions of times faster than human being can.

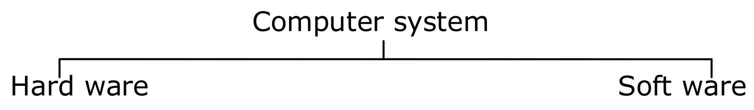
Or

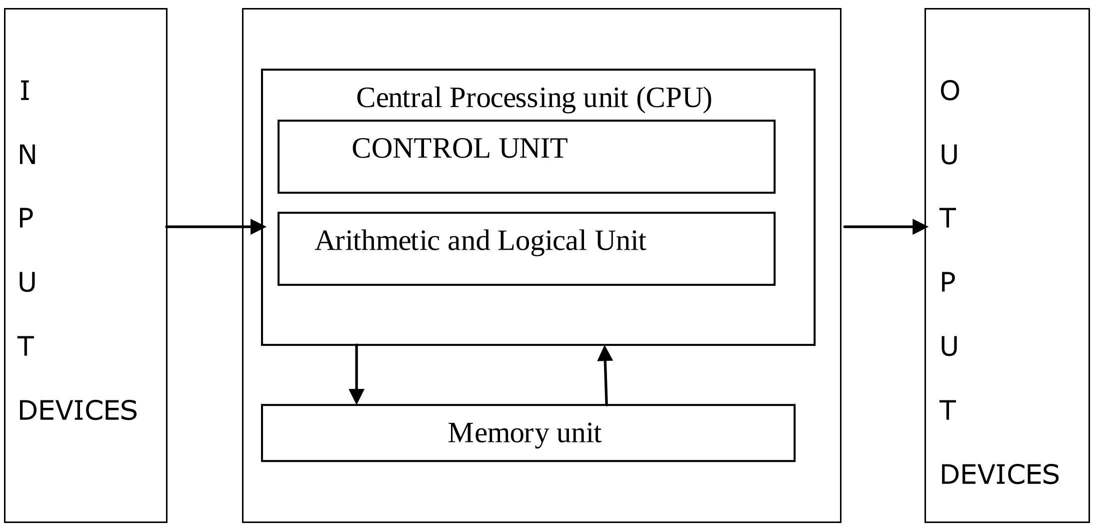
Computer is an electronics device which performs arithmetic and logical operations

A computer system made of two major components. They are:

1. Software,
2. Hardware.

The block diagram of computer system as given below:





Q) What is the need of software?

The need of software is to implement any specific task in a computer i.e. if you want to solve your problem through computer, then you have to develop software according to your problem and you have to install that software in your system. Then your system is ready to serve your requirement.

Note: Generally softwares are developed by using computer programming languages like C, C++, JAVA, .NET etc.

Q) What is SOFTWARE? Explain types of software?

- The software is the collection of programs that allow the hardware to do its job.
  Computer software is divided into two broad categories.
  1. System Software
  2. Application software.

SYSTEM SOFTWARE:

System software consists of programs that manage the hardware resources from computer and perform required information processing tasks. These programs are divided into 3 classes.

1. The operating system,
2. System support,
3. System development.

APPLICATION SOFTWARE:

Application software is broken into two classes.

1. General purpose software,
2. Application specific software.

General purpose software is purchased from a software developer and can be used for more than one word application processors. Data base management system etc.

Application specific software can be used only for its intended purpose.

A general ledger system used by accountant and a material requirement. Planning system used by a manufacturing organization are the examples of application specific software.

COMPUTER LANGUAGES:

Q) What is a programming language?

A programming language is an artificial language designed to communicate instructions to a machine, particularly a computer. Programming languages can be used to create programs that control the behavior of a machine and/or to express algorithms precisely.

A programming language is usually split into the two components of **syntax** (form) and **semantics** (meaning). Some languages are defined by a specification document (for example, the C programming language is specified by an ISO Standard).

To write a program for a computer we must use a computer language. A summary of computer languages as shown below

Machine language	symbolic language	high level language
1940S	1950S	1960S

MACHINE LEVEL LANGUAGE:

- The only language understood by computer is machine level language.
- Each computer has its own machine language, which is made streams of Os and 1's because the internal circuits of a computer are made of switches, transistors and other electronic devices that can be in one of two states off and on.
- In this the off state is represented by "O" and the on state is represented by 1.

ADVANTAGES:

1. Machine level language easily understood by the CPU.
2. The time taken for executing the program is very less.
3. The execution speed is very high because of machine instructions are directly understood by the CPU and no translation of the program is required.

DISADVANTAGES:

1. It is machine dependent language.
2. Length of the program is very high so difficult to modify.
3. Machine language is difficult to program. It is necessary for the programmer either to memorize the dozens of code numbers.
4. Difficult to check the errors

SYMBOLIC LEVEL LANGUAGE:

- Symbolic language uses symbols or mnemonics to represent the various machine language instructions. It is developed by admiral grace happen in the early 1950s.
- It is also called as **Assembly level language**.
- Computer does not understand symbolic language. So it must be translated to the machine language.
- A special program called as assembler translates symbolic code into machine language.

ADVANTAGES:

1. It is easier to understand and use.
2. It reduces the cost of associated ROM chips because of reduced code size.
3. It is easy to modify the program.
4. It is easy to rectify the errors.

DISADVANTAGES:

1. It is Machine dependent language.
2. To develop the assembly code, knowledge of hardware compulsory required.
3. Execution time is more because of translating of assembly code into machine level code.

HIGH LEVEL LANGUAGE:

- The languages which are developed in general language (English) are called as high level languages.
- High level having greater efficiency.
- These are portable languages.
- FORTAN, COBOL, C, C++, JAVA etc are the examples of high level languages.
- Machine does not understand high level languages directly. So to convert high level languages into its equivalent machine level languages translators are used.
- The translators are interpreter and compiler.

ADVANTAGES:

1. High level languages are always Machine independent.

2. These languages are easy to learn and use.
3. Preparation cost is less due to it takes less time and effort to develop the application.
4. Easy to maintain compare to machine level and assembly level languages.

DISADVANTAGES:

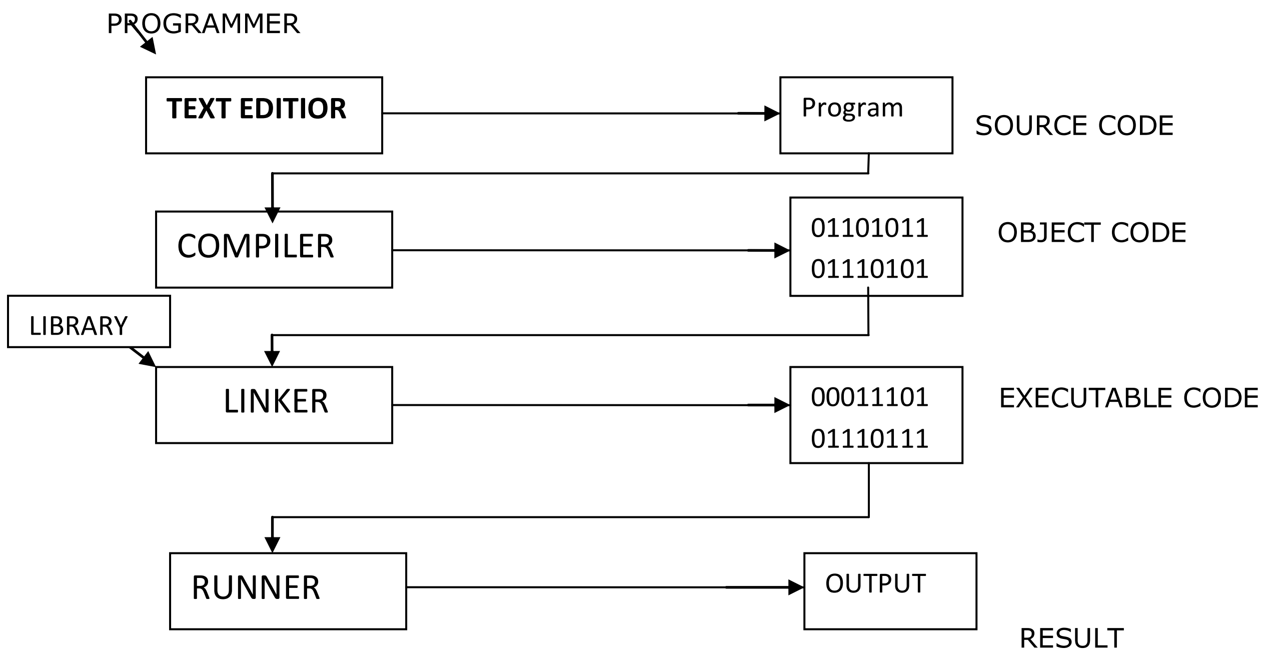
1. The efficiency of the program less compared with machine and assembly level languages.
2. Execution time is more because of translation i.e. high level to machine level languages.
3. This are having less flexible than assembly language.

4. CREATIES AND RUNNING PROGRAMMS:

It is the job of the programmer to write test the program. There are four steps in this process.

1. Writing and editing the program,
2. Competing the program,
3. Linking the program with equaled library modules,
4. Executing the program.

BLOCK DIAGRAM:



- The software used to write programs is as a text editor.
- The file created from a text editor is known as a source file.
- The code in a source file must be translated into machine languages using the 'c' compiler, which is made of two separate programs: the preprocessor and the translator.
- The file created from the compiler is known as an object module.

- An object module is linked to the standard functions necessary for running the program by the linker.
- Linker generates new code called executable code i.e the code which is having the functionality of the application that code is called as executable code.
- Executable code given to the runner or loader which will give the final result of our application.

PROGRAM DEVELOPMENT STEPS:

TO DEVELOP A PROGRAM, A PROGRAMMER MUST COMPLETE THE FOLLOWING STEPS

1. Understand the problem
2. Develop a solution using structure charts and either flow charts or Pseudo code.
3. Write the program.
4. Test the program.

Q) What is the difference between language and software?

A language is a specification which is used to develop the softwares. Software is a designed thing which is having a specified functionality.

Q) What are the different approaches we have to design the languages?

Basically we have three types approaches to design the languages

1. Monolithic

   EX: machine level language, Assembly level language

2. Procedure or structure oriented

   Ex: C language

3. Object oriented

   Ex: java, c++

Q) What is source code and Executable code?

Source code: Group of instructions which are present in high level language is known as source code. Generally source code visible in English

Executable code: Group of instructions which are present in low level language is known as Executable code. Generally Executable code visible in machine language format

Q) What is Object code?

Object code is a translated code of C compiler i.e. C compiler takes the source code and generates the object code. Source code is visible in the form of high level language but whereas object code is visible in the form of machine level language.

CHAPTER2
INTRODUCTION TO C LANGUAGE

INTRODUCTION TO "C" LANGUAGE

What is the need of c language:

- ✓ The ability to organize and process information is the key to success in the modern age.

- ✓ Computers are designed to handle and process large amounts of information quickly and efficiently, but they can't do anything until someone tells them what to do.
- ✓ Communicating with computers is not easy. They require instructions that are exact and detailed.
- ✓ Those set of instructions are called as programs.
- ✓ These programs are developed by using Programming languages.
- ✓ It was in 1957 that a high level language called FORTRAN was developed by IBM which was specially developed for scientist and engineers.
- ✓ Similarly one more language called COBOL which is widely used for business data processing task.
- ✓ 'Basic' language which is developed for the beginners in general purpose programming language.
- ✓ So above all languages are having a specific purpose oriented languages so we need a concrete programming language (that means the language must support multiple fields or regions).
- ✓ From above intension one more high level language introduces called 'C' language.
- ✓ C is currently the premier language for software developers.

The History of the C Language:

The C programming language was developed in the early 1970s by Dennis M. Ritchie an employee from Bell Labs (AT&T).

In the 1960s Ritchie worked, with several other employees of Bell Labs (AT&T), on a project called Multics. The goal of the project was to develop an operating system for a large computer that could be used by a thousand users. In 1969 AT&T (Bell Labs) withdrew from the project, because the project could not produce an economically useful system. So the employees of Bell Labs (AT&T) had to search for another project to work on (mainly Dennis M. Ritchie and Ken Thompson).

Ken Thompson began to work on the development of a new file system. He wrote, a version of the new file system for the DEC PDP-7, in assembler. (The new file system was also used for the game Space Travel). Soon they began to make improvements and add expansions. (They used their knowledge from the Multics project to add improvements). After a while a complete system was born. Brian W. Kernighan called the system UNIX, a sarcastic reference to Multics. The whole system was still written in assembly code.

Besides assembler and FORTRAN, UNIX also had an interpreter for the programming language B. (The B language is derived directly from Martin Richards BCPL). The language B was developed in 1969-70 by Ken Thompson. In the early days computer code was written in assembly code. To perform a specific task, you had to write many pages of code. A high-level language like B made it possible to write the same task in just a few lines of code. The language B was used for further development of the UNIX system. Because of the high-level of the B language, code could be produced much faster, then in assembly.

A drawback of the B language was that it did not know data-types. (Everything was expressed in machine words). Another functionality that the B language did not provide was the use of "structures". The lag of these things formed the reason for Dennis M. Ritchie to develop the programming language C. So in 1971-73 Dennis M. Ritchie turned the B language into the C language, keeping most of the language B syntax while adding data-types and many other changes. The C language had a powerful mix of high-level functionality and the detailed features required to program an operating system. Therefore many of the UNIX components were eventually rewritten in C (the Unix kernel itself was rewritten in 1973 on a DEC PDP-11).

The programming language C was written down, by Kernighan and Ritchie, in a now classic book called "The C Programming Language, 1st edition". (Kernighan has said that he had no part in the design of the C language: "It's entirely Dennis Ritchie's work". But he is the author of the famous "Hello, World" program and many other UNIX programs).

For years the book "The C Programming Language, 1st edition" was the standard on the language C. In 1983 a committee was formed by the American National Standards Institute (ANSI) to develop a modern definition for the programming language C (ANSI X3J11). In 1988 they delivered the final standard definition ANSI C. (The standard was based on the book from K&R 1st ed.).

The standard ANSI C made little changes on the original design of the C language. (They had to make sure that old programs still worked with the new standard). Later IN 1990, the ANSI C standard was adopted by the International Standards Organization (ISO). The correct term should therefore be ISO C, but everybody still calls it ANSI C. Again In 1995, minor changes were made to the standard. This version is known as C95. Much more significant update was made in 1999, known as C99. ANSI C was followed by a lot of other standards the latest being the "CIX" in the year 2007.

C popularity was due to two major factors:

1) The language didn't get in the way of the programmer. He could do just about anything by using the proper C construct.

2) C is popular is that a portable C compiler was widely available.

Characteristics of "C" Language:

We briefly list some of C's characteristics that define the language and also have lead to its popularity as a programming language.

1. General purpose language
2. Robust language.
3. Case sensitive language.
4. Structure oriented language. (or) POP language (procedure orientated programming).
5. compiler
6. Programs written in C are efficient and fast.
7. Many time faster than BASIC.
8. Small size
9. C is platform dependent
10. Extensive use of function calls
11. Low level (Bitwise) programming readily available
12. Pointer implementation - extensive use of pointers for memory, array, structures and functions.

C has now become a widely used professional language for various reasons.

- It has high-level constructs.
- It can handle low-level activities.
- It produces efficient programs.
- It can be compiled on a variety of computers.

APPLICATIONS of "C" LANGUAGE:-

C language mainly used for

- Language compilers and interpreters
- Device drivers
- Telecom applications
- Network programming
- Digital Signal processing applications
- Database applications
- Text editors

Procedure oriented programming:-

- Prime focus is on functions and procedures that operate on data.
- Large programs are divided into smaller units called functions.
- Data move freely around the systems from one function to another.
- Data and the functions that act up to it are treated as separate entities.
- Program design follows "top down approach".

Q) What is current version of C language?

Ans: CIX(C99)

Q) What are the advantages of C language?

Advantages of C:

C is Portable:

This means a program written for one computer may run successfully on other computer also.

C is fast:

This means that the executable program obtained after compiling and linking runs very fast.

C is compact:

The statements in C Language are generally short but very powerful.

Simple / Easy:

The C Language has both the simplicity of High Level Language and speed ofLow Level Language. So it is also known as Middle Level Language

C has only 32 Keywords

C Compiler is easily available

C has ability to extend itself. Users can add their own functions

Q) What do mean by Algorithm? Give its properties?

<u>Algorithm:-</u>

1. The word algorithm is derived from the name of the Persian mathematician Al Khwarizmi.

2. An algorithm can be defined as a step-by-step procedure for solving a problem.

<u>Properties:</u>

An algorithm has five important properties

Finiteness: - an algorithm terminates after a finite number of steps

Definiteness: - Each step in an algorithm is unambiguous. This means that the action specified by the step cannot be interpreted in multiple ways and can be performed without any confusion.

Input: -An algorithm accepts zero or more inputs

Output: -An algorithm produces at least one output.

Effectiveness: - An algorithm consists of basic instructions that are realizable. This means that the instructions can be performed by using the inputs in a finite amount of time.

Q) What Pseudo Code?

Pseudo Code:

A pseudo code, as its name suggests is just a false code that intends to explain the functionality or execution sequence of the Program. It is generally written using programming constructs like IF, WHILE etc... It's not actually the code but a set of normal statements explaining the execution of the program. It is written before developing the actual program. It is like developing the skeleton of the program before actually starting programming.

There can be ways to design programs automatically from its pseudo code given that there's always a fixed format for writing pseudo codes. Since pseudo codes does not have any particular syntax for writing, they cannot be interpreted to develop a corresponding code for it automatically.

Pseudo codes makes easier for a non - technical person to understand what the program does, rather than looking through all that technical code.

So you need to understand every step of pseudo code and convert it into corresponding code in whatever programming language you are using.

For example, Following is the pseudo code for finding whether a given number is even or odd.

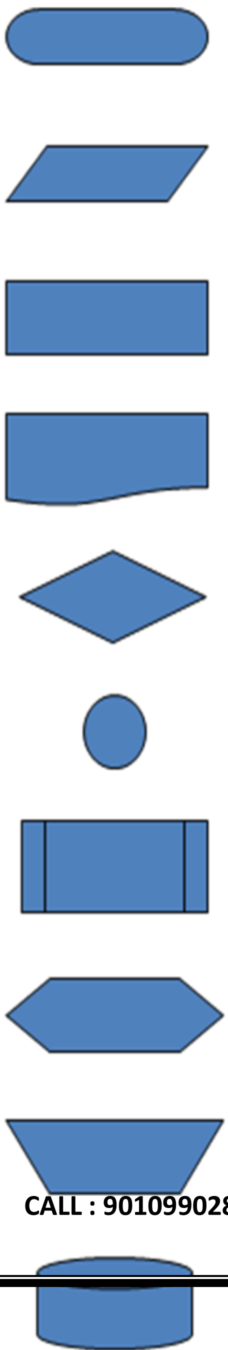
```
1) Input number 'X' from user

2) Divide the X by 2 and store its remainder As 'R'

3) if R is 0 then print 'X' is an even number

4) if R is not 0 then print 'X' is an odd number.

5) Exit
```

Now you can understand from the above program that if a given number is divisible by 2 then it is an even number else it is an odd number. So you can translate above pseudo code steps into a proper code as below :

C Program to find out whether a given number is even or odd.

Program:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int x,r;
x = 0;
r = 0;
clrscr();
printf("Enter a number ");
scanf("%d", x);
r = x/2;
if (r = 0)
printf("Given number is Even");
else
printf("Given number is Odd.");
getch();
}
```



Q) What is Flowchart? Explain its symbols?

<u>Flow Chart:-</u>

- By representing the various steps in the form of a diagram called as "flow chart"
- In Algorithm and flow chart each step can be called as an "instructions".
- In flow chart each symbol have unique operation.
- Flow charts avoid confusion whenever designing the application.

Standard symbols in flow chart:

Oval	**Terminal**
Parallegram	**Input/output**

Rectangle	**Process**
Document	**Hard copy**
Diamond	**Decision**
Circle	**Connector**
Double sided Rectangle	**Sub program**
Hexagon	**Iteration**
Trapezoid	**Manual Operation**
Cylinder	**Magnetic Disk Storage**

Q) What are the differences between algorithm and pseudocode?

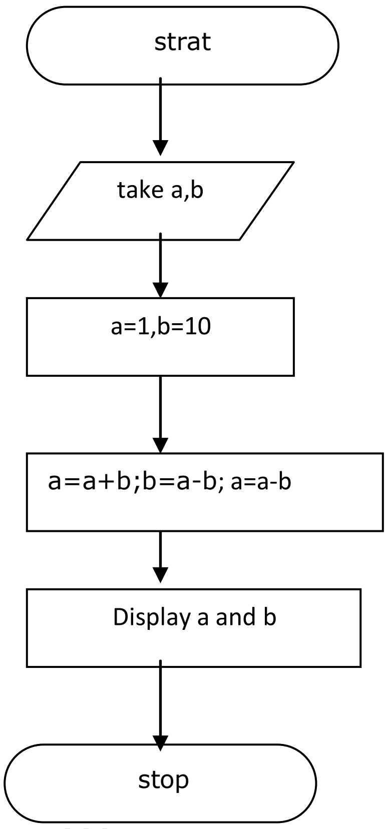
An algorithm is a well defined sequence of steps that provides a solution for a given problem, while a pseudo code is one of the methods that can be used to represent an algorithm. While algorithms can be written in natural language, pseudo code is written in a format that is closely related to high level programming language structures. But pseudo code does not use specific programming language syntax and therefore could be understood by programmers who are familiar with different programming languages. Additionally, transforming an algorithm presented in pseudo code to programming code could be much easier than converting an algorithm written in natural language.

Q) Write a algorithm and flow chart for swap two numbers

Algorithm:

Step1:	start
Step2:	input a,b
Step3:	a=a+b
Step4:	b=a-b
Step 5:	a=a-b
Step6:	Result a, b
Step7:	stop

Flow Chart:



<u>C is A Middle level language</u>

Q) How C is a middle level language?

All the programming language can be divided into two categories.

Problem oriented language (or) High level language:

These languages have been designed to give better program developments. According to programmer point of view applications are easily developed by using this kind of languages. And this are always exist in the form of general English.

Ex: FORTRAN, BASIC, PASCAL. Etc.

Machine oriented or low level language:-

These languages have been designed to give a better machine efficient is faster programs execution.
By using this languages developing the application is somewhat difficult and complex. But this programs having good accuracy and flexibility. And this are also having good execution time.

Ex: Assembly languages and machine language.

"C" stands in between these two categories. Because it is a general purpose language. That's why it is often called a **"Middle level language"** Since it was designed to have both kind of application development. Means a relatively good programming efficiency for low level languages programs and as well as high level programs..

C Tokens

Q) What is token? Explain types of tokens in c language?

Ans: The smallest individual units of a C- program are known as Tokens**.** Basically c tokens are classified into four types. They are

1. Key word
2. Identifiers
3. Constants
4. Operators

Key words:

Key words are the pre defined words meaning has already been explained to the "C" Compilers. These key words are also called as "Reserved words". There are 32 keywords available in standard C language. Following are the list of keywords in C

auto	double	if	static
break	else	int	struct
case	enum	long	switch
char	extern	near	type def
const	float	register	union
continue	far	short	voids
default	for	return	unsigned
do	goto	signed	while

- ⇨ All keywords must and should write in all small caps only.
- ⇨ We can't add anything to the keywords because this all are predefined.
- ⇨ We can't change the operation of any key words.
- ⇨ We can't combine two keywords.

Identifiers:

- ⇨ Identifiers are user defined words or names which doesn't have specific meaning with in language. The meaning of these words understood by programmer/developer.
- ⇨ Identifiers are used to identify the data instruction or user defined types.

Rules for Identifiers:

1. It consists of sequence of letters and digits.
2. Identifier name always start with alphabet.
3. The only special characters allowed is under score (-)
4. The length of the Identifier is up to 32 characters.
5. Identifiers should not be a keyword.
6. No two successive underscores are not allowed
7. It can be defined in uppercase or lowercase letters.
8. Identifiers are case sensitive in programming.
9. Duplicate names are not allowed.
10. Don't give any white spaces whenever we are writing identifier names.

Constants:

A constant is a value which never changes throughout the program. These values are represented in different formats.

Types of Constants:

Constants mainly classified in the two types.

1. Numeric Constants
2. Alpha - numeric Constants

Numeric constants are again classified into two types

1. Integer Constant:

⇨ integer constants again classified into three types

Decimal integer:

⇨ An integer value with base 10 is called decimal integer.
⇨ This integer value consists of digits from 0 to 9.
⇨ This integer is prefix with 1-9 sign(+/-)
⇨ Valid decimal integers are 100, 456, -345, 23456, -19202…. Etc

Octal integers:

⇨ An integer value with base 8 is called octal integer.
⇨ This integer value consists of digits from 0 to 7.
⇨ This integer is prefix with '0'

Ex: 0123, 0725 … etc are the valid octal integers.

Ex: 0128, 4647……etc are the invalid octal integers.

Hexa decimal integer:

⇨ An integer value with base 16 is called Hexa decimal integer.
⇨ This integer value consists of digits from 0 to9 and A-F or a-f.
⇨ This integer value is prefix with "ox"

Ex: 1ab - invalid
Ox1ab - valid

2. Real constants:

⇨ Real constants are also called as floating point constants.
⇨ Real constants are having both decimal and fractional parts.

Alpha numeric constants:

Character constants again classified into two types.

1. **Single character constant:**

⇨ A non numeric value represented with in ' ' is called single character value or constant.

⇨ Every character occupies 1 byte of memory.

2. **String constant:**

⇨ A non numeric value represented with in " " is called string or constant.

⇨ A string is collection of characters.

Examples:

"rama" – Alphabetic string

"A101" - Alphabetic string

"a**" - Alphabetic string

⇨ Every string is terminated with a special character called NULL which occupies 1-byte.

Note: In C program constants are declared by using " const" keyword.

Example:

```
const int a=10;

const float f=1.5;

const char ='f';
```

BASIC INPUT AND OUTPUT FUNCTIONS:

⇨ Input and output operations are performed by using predefined library functions.

printf():

⇨ printf() is a pre defined output function in C language. Basically printf() is used to for three purposes.

⇨ 1st purpose is to display some message on the consle screen

Syntax:

printf("Some text message");

Ex: printf ("My name is C");

⇨ 2nd purpose is to display some values on the consle screen

Syntax:

printf(“format specifier”, variable_names);

Ex: printf(“%d”,a);

```
printf(“%d%d”,a,b);
printf(“%d%f%c”,a,b,c);
printf(“value of a =%d”,a);
printf(“addition =%d”,c);
```

⇨ If you want to print address of the variable then go for printf()

Syntax:

printf(“control string”,&variable_name);

Ex: printf(“%d”,&a);

```
printf(“address of a =%d”,&a);
Printf(“address of a and b=%d%f”,a,b);
```

Note:

1. **The return type of the printf() function is integer type.**
2. **printf() defined in stdio.h(standard input output) header file.**

scanf ():-

⇨ Scanf function is a predefined function by using scanf() function we can read the data at runtime.
⇨ Scanf function returning an integer value i.e. total number of scanned values.
⇨ The complete behavior of scanf() function will depends on format specifying only but not on argument list.
⇨ When we working with scanf() function if you not provide ampersand to the argument then scanf() will read the values but those not stored in memory.

Syntax:

scanf (‘’format string ‘’, & (variables));

```
Ex: scanf (‘’%d %d ‘”, &a,& b);
      scanf(“%d%f”, &a, &b);
```

}

CHAPTER3
STRUCTURE OF A C PROGRAM

Structure of a 'C' program:

```
<Preprocessor commands>

<Global declarations>

void main ()

{

     <Local declarations>;

     <Statements>;

  }

Function (arguments)

{

     <Local declarations>;

     <Statements>;

 }
```

- ⇨ The 'C' program has a free formatted structure.

- ⇨ Every statement must be terminated with a semicolon (;)
- ⇨ A program is a collection of functions. There may be a lot of functions or not but at least one function must be there that is main (), where the execution starts.
- ⇨ C has case sensitivity. All the keywords are defined in lower case.
- ⇨ So better to write entire program in lower case.

Preprocessor commands:

The commands which start with a hash (#) symbol are called Preprocessor commands. These commands are used to place library information in our program. Generally in C programs library information placed in the form of header files.

Ex:

```
# include <stdio.h>

# include “conio.h”

# define PI 3.14159
```

Global declarations:

To use any variable it must be declared with its data type before the first executable statement. The variables which declared inside a block are available in that block only. Such variables are called as local variable.

To use the variable in entire program with same effect it must be declared as global. These global variables are declared outside the functions.

main ():

- ⇨ It is starting point for program execution. This function has no arguments and no parameters.
- ⇨ C language does not allow more than one main () function in a single program.
- ⇨ main () function always write in small caps only.
- ⇨ Every function must and should call through main () only (directly or indirectly).
- ⇨ This function consists of declarations and executable statements.
- ⇨ C compiler does not consider anything out of the main.

EXAMPLES 1:

```
/* Write a program to print your name on the console screen */
#include <stdio. h>
#include <conio. h>
void main()
{
clrscr();
printf(“ my name is suninfotech”);
}
```

Output:

my name is suninfotech

Save this program (F2) as FIRST.C

After compilation (Alt-F9) it creates an object file, and an executable file

File which can be executed at MS-DOS prompt.

By saving a modified file it creates a backup file.

FIRST.C

FIRST.BAK

FIRST.OBJ

FIRST.EXE

TURBO C EDITOR

Q) What is IDE?

Turbo c features as integrated Development environment, or IDE,. It is also referred to as the programmer's platform.) in IDE you can able to write/save/open your programs or code, compile using short cut keys, and also perform code debugging very easily.

Q) What is turbo c editor?

It is a compiler of C program and it can be also used as a general editor. To enter into editor first change into the directory this contains the software and enters the command TC at the command prompt.

```
C:\> CD TC
C:\TC> tc
```

Then it opens the editor which contains a menu bar at the top, a status bar at the bottom and a main window to write the programming statements and a sub window which shows the messages.

The menu bar contains some menu pads and they can be selected by pressing ALT and the highlighted character in the required menu pad.

Then it shows the submenu which contains some bars and they can be selected using arrow keys.

The status bar shows online help and the keys information.

1) To write a new program selects ‘New’ command from “File” menu.
2) To save the working program select ‘Save’ command from “File” menu

   Or press F2 and enter a name.
3) To compile the program select ‘Compile to OBJ’ command from “compile” menu or press Alt + F9. Then it shows the list of

Errors or warnings. If the program is error-free then the compiler

Creates an object file (.OBJ) and an executable file (.EXE).

4) To execute the program select ‘Run’ command from “Run” menu or press Ctrl + F9.
5) To see the output of the execution select ‘User Screen’ command from “Run” menu or press Alt + F5.
6) To close the editor select ‘Quit’ command from “File” menu or press Alt + X.

Common Short cut Keys Description:

F2 press to Save current work

F3 press to open an existing file

ALT-F3 press to close current

ALT-F9 press to compile only

ALT-F5 press to view the desired output of the program.

CTRL-F9 press to compile+run

ALT-X or ALT-F-X press to exit from TC IDE

clrscr() :-

This function is used to clear the screen. This function’s prototype has defined in the header file conio.h (CONIO ==> Console Input Output)

Syntax:

clrscr();

getch() :

This function is used to accept a single character for the variable while executing the program. But this function does not display the entered character. This function's prototype has defined in the header file CONIO.H

Note: To see the entered character the function getche() can be Used.

Syntax:

```
(Variable) = getch() ;
```

Ex :

```
char c;
c = getch();
```

CHAPTER4
DATA TYPES

Q) What do mean by data type?

- ⇨ Data types reserves space for data.
- ⇨ A data type defines for which type of data how much memory to be allocated.

1. Type
2. Size

Q) What are the different data types?

There are five data types

- ➢ Primary data types
- ➢ Extended data types
- ➢ Derived data types
- ➢ User defined data type
- ➢ Null data types

Primary Data types:

There are three types

1. Integer data types:

Key word	: int
Memory size	: 2 byte (16 bits)
Range: -	
Signed	: -2^{15} to 2^{15}-1
	: -32,768 to 32,767
Control string	: %d
Unsigned	: 0 to 2^{16}-1
	: 0 to 65, 535

Control string	: % u

Floating point data type:

Key word	: float
Memory range	: 4 bytes (32) bits
Range	: - 32, 769, 99 to 32,769, 98
Control string	: %f

Character data type:

Key word	: char
Memory size	: 1 byte
Range	: - 128 to 127 (ASCII code)
Control string	: %C

Note: every character should denote between in single quotations (‘a’)

ASCII:

The **American Standard Code for Information Interchange (ASCII)** is a character-encoding scheme based on the ordering of the English alphabet. ASCII codes represent text in computers, communications equipment, and other devices that use text. Most modern character-encoding schemes are based on ASCII, though they support many more characters than ASCII does.

Example:

ASCII code	Character
65	A
90	z
97	a
122	Z

Extended data type:

1. **Long integer data type :**

Key word	: long int
Memory size	: 4 bytes (32) bits
Range	:-2^{31} to 28-1
Control string	: % ld

2. **Double data type :**

Key word	: double
Memory string	: 8 bytes
Control string	: % lf

3. **Derived data types:**

1) Arrays ,
2) structures,
3) pointers ,
4) Unions.

4. **User defined data type:**
   1. Functions
   2. Typedef
   3. Enum

5. **Null data type:**

   Using the key word: void

Q) Give the all pre defined data types ranges?

<u>**Pre defined data type's ranges**</u>

Type	**Typical Size in Bits**	**Minimal Range**
char	**8**	**−128 to 127**
unsigned char	**8**	**0 to 255**
signed char	**8**	**−128 to 127**
int	**16 or 32**	**−32,768 to 32,767**
unsigned int	**16 or 32**	**0 to 65,535**
signed int	**16 or 32**	**Same as int**
short int	**16**	**−32,768 to 32,767**
unsigned short int	**16**	**0 to 65,535**
signed short int	**16**	**Same as short int**
long int	**32**	**−2,147,483,647 to 2,147,483,647**
long long int	**64**	**−(2^{63}) to 2^{63} − 1 (Added by C99)**
signed long int	**32**	**Same as long int**
unsigned long int	**32**	**0 to 4,294,967,295**
unsigned long long int	**64**	**2^{64} − 1 (Added by C99)**

float	**32**	**3.4e-38 to 3.4e+38**
double	**64**	**1.7e-308 to 1.7e+308**
long double	**80**	**3.4e-4932 to 1.1e+4932**
void	**--**	**data type that not return any value**

CHAPTER5

VARIABLES AND OPERATORS

Variable:

- ⇨ While solving a problem we need to remember some data values temporarily. Such values are placed in the named locations in the memory. These named locations in the memory are called variables.
- ⇨ A variable is an identifier. It can be consider as a name given to the location in memory where values are stored.
- ⇨ The quantities which can be changed during the execution of program are called Variables.
- ⇨ Variable is a container which contains data or Information.
- ⇨ Before using any variable in a program it must be declared first.

SYNTAX:

<Data-type> <variable-name>;

```
Ex: int count;
    float X;
    char name;
```

Rules:

1) Variable names can consist of alphabets, Digits, underscore, dollar character etc.
2) A variable name should not begin with a digit or any special symbol except #.
3) White spaces are not allowed in variable names.
4) Upper and lower case distinct I.e.,"A" Is differed from 'a'.
5) The maximum length of variable length is 32 characters.

Important points to remember:

1. In c programming language variable declaration must be exist top of the program i.e. after opening the curly braces and before writing the first instruction.
2. In variable declaration existing keyword, operators, separators, and constants values are not allowed.
3. At least once space should be required in between data-type and variable-name.

4. All data-types must write in small caps only.
5. Variable names are case sensitive.

Note: one variable contain only one value. If you change the value of the variable then old value will be erased and new value will be stored.

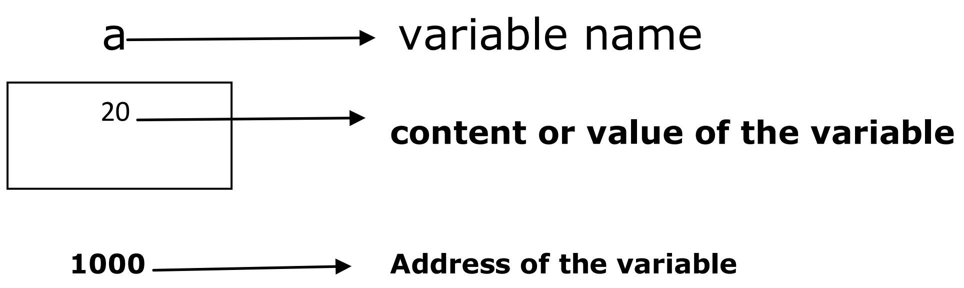
Q) What variable contain?

Variable contain mainly three things

1. Name of the variable
2. Value of the variable
3. Address of the variable

Example:

int a=10;



Types of variables:

⇨ Variables are broadly classified into four types. They are

1. Local variables
2. Global variables
3. Block level variables
4. Constant variable

- Local variables means, the variables which are defined inside a function. These variables are work within that function only. The scope of the local variables is inside a function in which variables are defined.
- Global variables are nothing but the variables which are defined outside of any function (generally main). The scope of these variables is throughout the program i.e. in all functions we can use these variables.
- The variables which are defined inside a block is called as block level variables. The scope of the block level variables is within the block.

- The variables which are declared with helf of const keyword, those variables are called as constant variables. Constant variables data we can't change it at any cost. Constant variable may be local, global and block level.

  ⇨ Block variables again classified into 2 types. They are

    1. Conditional block(generally conditional statements)

    Ex: while, for, do-while

    2. Iteration block(generally loops)

    Ex: if, switch

How to initialize a variable:

⇨ A variable can be initialized in two ways. They are

1. Static variable initialization
2. Dynamic variable initialization

Static variable initialization:

⇨ Declaring a variable by assigning constant value is called Static variable initialization.

Or

⇨ The value assigned to variable is known before compiling program is called Static variable initialization.

Ex:

int x=10;

float y=200;

char c="ngc";

Dynamic Variable Initialization:

⇨ Declaring a variable by assigning expression or variable is called Dynamic variable initialization.

Or

⇨ The value assigned to variable is not known during compiling time is called Dynamic variable initialization.

⇨ Dynamic initialization can be done by using pre defined functions which available in c library.

Example:

int c;

scanf("%d",&c); // dynamic initialization

c= 10; //static initialization

OPERATORS

- Operator is a special symbol which performs a particular operation.
- All operators are in C language are pre-defined.
- In c programming language total number of operators are 44
- Depending upon the number of operands operators are classified into three types
  1. **Unary operators**: It takes only one argument
  2. **Binary operator**: it takes two arguments.
  3. **Ternary operators**: it takes three arguments.

Arithmetic operators:

The operators which are used for Arithmetic operations (general mathematical calculations)

Operators	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	modulo division

Priorities of Arithmetic operators:

I	* / %
II	+ -
III	=

Modulus operator:

- ⇨ Modulus operator returns remainder value. Output sign depends on numerators value sign only.
- ⇨ When a numerator value is less than of denominator value then it returns numerator value as return value.

  Ex: a=8%5; o/p: 3   a=18%9; o/p: 0   a= 3.0*5/2*2; o/p: error
- ⇨ Modulus operator required two arguments and both arguments should be an integer type only.
- ⇨ In implementation when we need to find the floating data modulus value then go for fmod(), or fmodl() functions.
- ⇨ Fmod() and fmodl() functions are available in <stdlib.h> which is used to find the modulus of floating data values.

Example:

/* W.A.P. to calculate Addition of two numbers */

```
# include<stdio h >
# includ <conio h >
main ()
 {
 int  a, b, c;
 clrscr ();
 printf (“ enter a, b, values = “);
 scanf (“  %d %d”,  &a , &b);
 c=a+b;
 printf (“Addition =% d”, C);
 getch();
 }
```

Output:

```
enter a, b values= 10  20

Addition=30
```

Relational Operators:

- ⇨ In c all relational operators returns one or zero
- ⇨ If the expression is true then returns with ’1’ and if the expression is false then returns with zero.
- ⇨ Every non zero is called true when value become zero it is called false.

Relational operators	Action

<	is less than
< =	Is less than or equal
>	Is greater than
>=	is greater than equal
= =	is equal to
! =	is not equal

I	()	
II	! , +, -	unary
III	*, /, %	
iV	+, -	
V	<, >, <=, >=	
Vi	==, !=	
VII	&&	
VIII	||	
iX	?:	ternary
X	=	

Example:

/* write a program to verify relational operators */

```
#include<stdio.h>

#include<conio.h>

void main()

{

int a,b,c;

clrscr();

printf("\n Enter a,b values:");

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

c=a>b;

printf("\n value of a>b:%d",c);

c=a<b;

printf("\n value of a<b:%d",c);

c=a==b;
printf("\n value of a==b:%d",c);

c=a>=b;

printf("\n value of a>=b:%d",c);

c=a<=b;

printf("\n value of a<=b:%d",c);

c=a!=b;

printf("\n value of a!=b:%d",c);

getch();

}
```

Logical Operators:

- ⇨ Logical expressions combine two (or) more relational expressions. And also used to test more than a one condition one and make decision.

Logical Operators	**Action**
&&	Logical AND
||	Logical OR
!	Logical NOT

- ⇨ In c all logical operators returns one or zero.
- ⇨ If the expression is true then returns with '1' and if the expression is false then returns with zero.
- ⇨ Every non zero is called true when value become zero it is called false.

Example:

/* write a program to verify logical operators */

```
#include<stdio.h>

#include<conio.h>

void main()

{

int a,b,c,d;

clrscr();

printf("\n enter a,b,c values:");

scanf("%d%d%d",&a,&b,&c);
d=((a>b)&&(a>c));

printf("\n value of AND operator:%d",d);

d=((a>b)||(a>c));

printf("\n value of OR operator:%d",d);

d=((a>b)!=(a>c));

printf("\n value of NOT operator:%d",d);

getch();

}
```

Assignment Operators:

- ⇨ Assignment operator is a binary category operator.
- ⇨ Binary category means it required two operands that is left and right argument.
- ⇨ Among those to arguments any one missing it gives an error. And left side argument must be variable type only.

Syntax:

V op = exp;

Where V is a variable
Exp is an expression.
Op is a "C" binary arithmetic operator

These operators also called as short hand operators.

Simple assignment Operator	**Shorthand operator**
A=a+1	a+=1
A=a-1	a-=1
A=a*(n+1)	a*= n+1
A=a/(n+1)	a/= n+1
A=a%b	a%=b

Increment and Decrement Operators:

These operators are represent as '++' and '- -'increments opened by 1 and -- decrements operands by 1 they are unary operators.

Operator	**Action**
A ++	Post – Increments
++a	Pre – decrements
A --	Post– decrements
--A	Pre - decrement

Priorities of operators:

I	- →	( )
II	→	+, -, !, ++, --(pre)
III	→	* / %
iV	→	+,-
V	→	<, >, <=, >=,
VI	→	==, !=
VII	→	&&
VIII	→	||
iX	→	?:
X	→	=
Xi	→	++,--(post)

Example:

/* write a program to perform increment and decrement */

```
        #include<stdio.h>
        #include<conio.h>
        void main()
        {
        int a;
        clrscr();
        printf("\n Enter a,b values:");
        scanf("%d",&a);
        printf("\n Value Of a++:%d",a++);
        printf("\n Value Of a++:%d",++a);
        printf("\n Value Of a++:%d",a--);
        printf("\n Value Of a++:%d",--a);
        getch();
}
```

Conditional Operators:

- ⇨ Conditional operator are ternary category operators
- ⇨ Ternary category means it required three arguments i.e, left, middle, and right side arguments.
- ⇨ In ternary category operators if the expression is true then returns with middle arguments. If expression is false then returns with right side argument and left side wxpression is treated as condition.

Syntax:

Expression = left term? Middle term: right term;

Notes:

1. Number of question marks and colon marks should be equal
2. Every colon should match with just before the question mark
3. Every colon should followed by question mark only.

Example:

/* TO FIND GREATEST OF TWO NUMBERS USING CONDITIONAL OPERATOR. */

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

void main()
{
int a,b,c;
clrscr();
        printf("\n Enter value of 'a': ");
        scanf("%d",&a);
        printf("\n Enter value of 'b': ");
        scanf("%d",&b);

        c = (a>b) ? a:b;
        printf("\n The number %d is greater",c);
getch();
}
```

Bitwise Operators:

Bitwise operators are similar to that of logical operator's expert that they work on binary bits. When bitwise operators are used with various they are internally was in the converted to binary numbers and then bitwise operators are applied on individual bits. These operators work with char and int type data. They cannot by used with floating point.

Bitwise operators	Action
&	bitwise AND

!	bitwise OR
^	bitwise
< <	Shift left
>>	shift right

A	**B**	**A & B**	**A | B**	**A ^ B**	**~A**
1	**1**	**1**	**1**	**0**	**0**
1	**0**	**0**	**1**	**1**	**0**
0	**1**	**0**	**1**	**1**	**1**
0	**0**	**0**	**0**	**0**	**1**

Precedence and Associativity of Operators:

Precdence Group **Operators** **Associativity**

(Highest to Lowest)

(param) subscript etc.,	**( ) [ ] ->.**	**L → R**
Unary operators	**- + ! ~ ++ – – (type) * & sizeof**	**R → L**
Multiplicative	*** / %**	**L → R**
Additive	**+ –**	**L → R**
Bitwise shift	**<< >>**	**L → R**
Relational	**< <= > >=**	**L → R**
Equality	**= = !=**	**L → R**
Bitwise AND	**&**	**L → R**
Bitwise exclusive OR	**^**	**L → R**
Bitwise OR	**|**	**L → R**
Logical AND	**&&**	**L → R**
Logical OR	**| |**	**L → R**
Conditional	**? :**	**R → L**
Assignment	**= += –= *= /= %= &= ^= |= <<= >>=**	**R → L**
Comma	**,**	**L → R**

- ⇨ Precedence is used to determine the order in which different operators in a complex expression are evaluated.
- ⇨ Associativity is used to determine the order in which operators with the same precedence are evaluated in a complex expression

EXAMPLE:

```
#include<stdio.h>

#include<conio.h>

void main()

{

int a=10;
int b=20;

int c=30;

clrscr();

printf("a* b+c is:%d\n", a * b+c);

printf("a*(b+c) is:%d\n",a*(b+c));

getch();

}
```

Output:

```
a* b+c is:230

a*(b+c) is:500
```

SOME IMPORTANT QUESTIONS:

Q) What is language?

ANS: A language is a communication between two different things.

Q) What is a programming language?

ANS: A **programming language** is an artificial language designed to communicate instructions to a machine, particularly a computer. Programming languages can be used to create programs that control the behavior of a machine and/or to express algorithms precisely.

Q) What is programming?

ANS: programming is a process of organizing data and instructions according to given problem.

Q) What is programming concept?

ANS: programming concept define set of rules and regulations to organize data and instruction.

Q) What is syntax?

- ⇨ Grammar of specific programming language is called syntax.
- ⇨ The basic syntax of C programming language is every statement should end with semicolon.
- ⇨ And one more is in C language each and every declaration must do before using; it may be function, array, structure and variable etc...

Q) What is format specifier?

=> Format specifier decides that which type of data need to be printed on console.

%d → integer data

%f → floating data

%c → character data

Q) How many types of errors present in C language?

A: Three types of errors

1) **Compile time errors:**

   The errors which are occurring at the time of complication those errors are called as compile time errors.
   EX: Syntax Errors.

2) **Runtime Errors :**

   The errors which occur while exciting a program those errors are called runtime errors.
   Ex: num/o; //runtime error

3) **Logical Error:**

   Logical errors are the errors which will not raise that compiler time (or) Runtime but they give a wrong output.

Q) What is translator and Explain different types of translator?

=>Translator is pre-defined software which converts high level instructions to machine understandable instructions. These translators are classified into three type.

1. Assembler
2. Interpreter
3. Compiler

ASSEMBLER:

⇨ The assembler converts the assembly language code into machine understandable code (binary instructions) which is understood by the computer. The assembler are directly used under system programming or under microprocessor programming on direct interface.

Ex: TASM, MASM etc.

INTERPRETER:

- ⇨ These translaters translate the source code step by step into machine language until any error. If there is any error it stops and shows some message. After correction it can continue.

<u>Ex:</u> BASIC, DBase III+, ...

COMPILER:

- ⇨ These translaters translate the entire source code into machine language when it is error-free and creates an object file in machine language. If there is any error it shows the list of error. After debugging it creates the object file.

<u>Ex:</u> COBOL, C, C++, ...

Q) Explain different flavours of c language ?

- ⇨ C compiler Can be available in different flavors since the portability was extended Under various operating systems duo to the siginificance of system level programming.
- ⇨ Based on operating system the C compiler is energized much more stronger since all the system resourses are under the controll of operating system.

1. Turbo	C/C++	dos
2. ANSI	C/C++	Unix
3. Microsoft	C/C++	Windows
4. Borland	C/C++	Windows
5. BerKley	C/C++	Unix-Bsd
6. Pro	C/C++	Oracle-unix/ windows
7. Dynamic	C/C++	Linux
8. Embedded	C/C++	linux, RTOS

Q) Explain sizeof() operator ?

sizeof is a keyword, not a function whose value is determined at runtime but rather an operator whose value is determined by compiler, soi t Can be known as compile time unary operator. sizeof (object) is the amount of memory in bytes required to store object. A cell of type char can hold one character, therefore on any C system the value of sizeof(char) is 1. The values of sizeof for other data types vary from systems to system.

Syntax :

sizeof(type-name)

Ex :

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a;
float b;
char c;
double s;
clrscr();
printf("%d\n",sizeof(a));
printf("%d\n",sizeof(b));
printf("%d\n",sizeof(c));
printf("%ld\n",sizeof(s));
getch();
}
```

Output :

```
2
4
1
8
```

Q) What is statement? Explain different types of statement.

Statements:

A statement is a complete direction instructing the computer to carry out some task. In C, statements are usually written one per line, although some statements span multiple lines. C statements always end with a semicolon. You've already been introduced to some of C's statement types. For example:

```
x = 2 + 3;
```

is an assignment statement. It instructs the computer to add 2 and 3 and to assign the result to the variable x.

Statements and White Space:

The term white space refers to spaces, tabs, and blank lines in your source code. The C compiler isn't sensitive to white space. When the compiler reads a statement in your source code, it looks for the characters in the statement and for the terminating semicolon, but it ignores white space. Thus, the statement

```
x=2+3;
```

is equivalent to this statement:

```
x = 2 + 3;
```

It is also equivalent to this:

```
x =
2
+
3;
```

This gives you a great deal of flexibility in formatting your source code. You shouldn't use formatting like the previous example, however. Statements should be entered one per line with a standardized scheme for spacing around variables and operators. If you follow the formatting conventions you should be in good shape. As you become more experienced, you might discover that you prefer slight variations. The point is to keep your source code readable. However, the rule that C doesn't care about white space has one exception: Within literal string constants, tabs and spaces aren't ignored; they are considered part of the string. A string is a series of characters. Literal string constants are strings that are enclosed within quotes and interpreted literally by the compiler, space for space. Although it's extremely bad form, the following is legal:

```
printf(
"Hello, world!"
);
```

This, however, is not legal:

```
printf("Hello,
World!");
```

To break a literal string constant line, you must use the backslash character (\) just before the break. Thus, the following is legal:

```
printf("Hello,\
world!");
```

Null Statements:

If you place a semicolon by itself on a line, you create a null statement. A statement that doesn't perform any action. This is perfectly legal in C.

Compound Statements:

A compound statement, also called a block, is a group of two or more C statements enclosed in braces. Here's an example of a block:

```
{
printf("Hello, ");
printf("world!");
}
```

In C, a block can be used anywhere a single statement can be used.
Note that the enclosing braces can be positioned in different ways. The following is equivalent to the preceding example:

```
{printf("Hello, ");
printf("world!");}
```

It's a good idea to place braces on their own lines, making the beginning and end of blocks clearly visible. Placing braces on their own lines also makes it easier to see whether you've left one out.

Q) Give the list of Convesion specifiers?

Convesion specifiers:

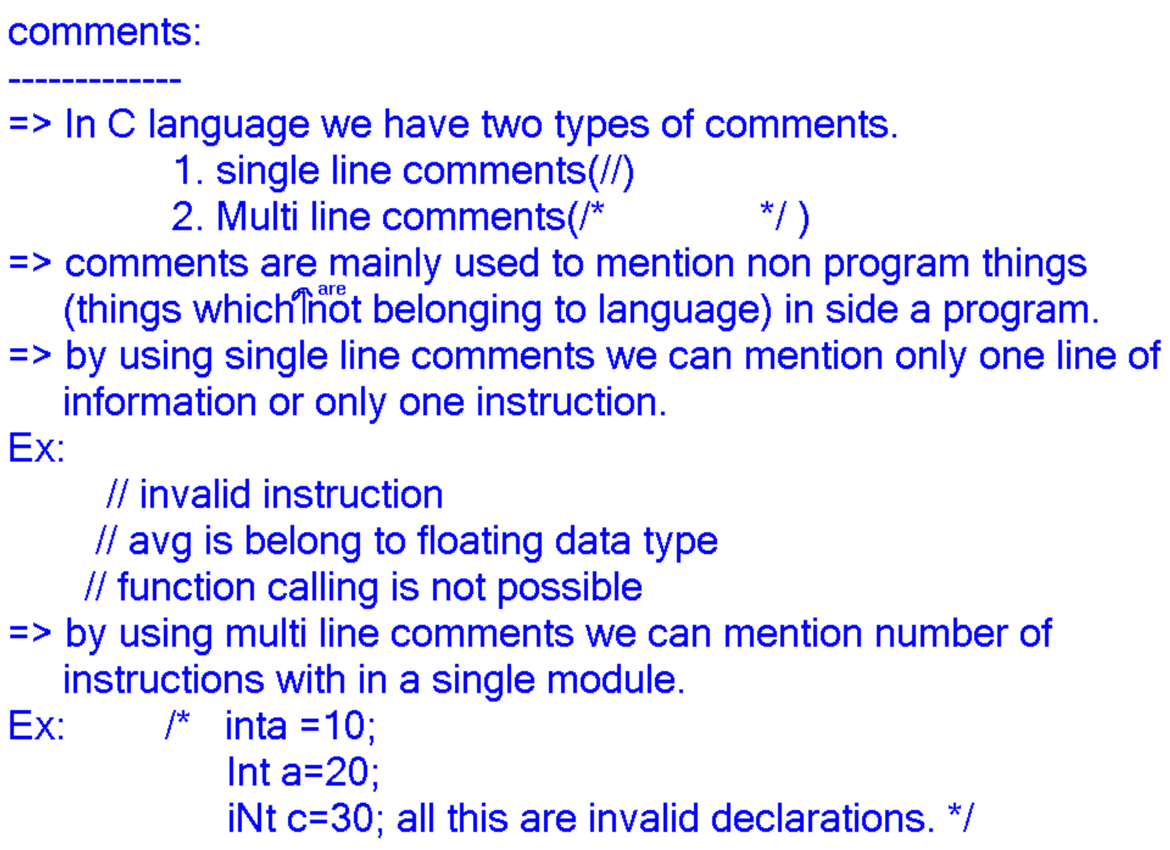
Code	**Format**
%a	**Hexa decimal output in the form of 0xh.hhhhp+d(C99 only)**
%s	**String of characters (until null zero is reached)**
%c	**Character**

%d	Decimal integer
%f	Floating-point numbers
%e	Exponential notation floating-point numbers
%g	Use the shorter of %f or %e
%u	Unsigned integer
%o	Octal integer
%x	Hexadecimal integer
%i	Signed decimal integer
%p	Display a pointer
%n	The associated argument must be a pointer to integer, This sepecifier causes the number of characters written in to be stored in that integer.
%hd	short integer
%ld	long integer
%lf	long double
%%	Prints a percent sign (%)

Back Slash (Escape Sequence) Characters

Code	Meaning
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\"	Double quote
\'	Single quote
\ \	Backslash
\v	Vertical tab
\a	Alert
\?	Question mark
\N	Octal constant (N is an octal constant)
\xN	Hexadecimal constant (N is a hexadecimal constant)

Comments in c language:



comments:

=> In C language we have two types of comments.
1. single line comments(//)
2. Multi line comments(/*   */)
=> comments are mainly used to mention non program things (things which $\overset{\text{are}}{\wedge}$not belonging to language) in side a program.
=> by using single line comments we can mention only one line of information or only one instruction.

Ex:

```
    // invalid instruction
   // avg is belong to floating data type
  // function calling is not possible
```

=> by using multi line comments we can mention number of instructions with in a single module.

```
Ex:      /*   inta =10;
              Int a=20;
              iNt c=30; all this are invalid declarations. */
```

Programs:

/* Using Escape Sequences   */

```
# include <stdio.h>

# include <conio.h>

main()

{

  clrscr() ;

  printf("Hello \t ") ;

  printf("ngc \n") ;

  printf("how are you ") ;

 }
```

Output :

```
Hello ngc
how are you
```

EXAMPLE 3:

```
/* W A P to calculate multiplication of two numbers */
#include<stdio.h >
#include<conio.h >
main ()
{
float a,b,c;
clrscr ();
printf ("enter a,b, values = ");
scanf ("%f%f" , &a, &b);
c= a*b;
printf (" multiplication = %f " , c);
getch ();
}
```

Output:

```
enter a,b, values = 10
5
multiplication = 50.000000
```

```
/* W A P to swap two numbers */
#include<stdio.h>
#include<conio.h>
main ()
{
int a,b,temp;
clrscr();
printf("enter a,b values = ");
scanf("%d%d",&a,&b);
temp = a;
a = b;
b = temp;
printf("\n given values after swaping a=%d \n b=%d",a ,b);
getch();
}
```

Output:

```
enter a,b values = 10
20
 given values after swaping a=20
 b=10
```

/* WAP to calculate months and years*/

```
#include<stdio.h>
#include<conio.h>
main ()
{
int months,days,years;
clrscr();
printf("enter the no of days = ");
scanf("%d",&days);
months = days/30 ;
years  = months/12;
printf("in months = %d, years = %d",months, years);
getch();
}
```

Output: -

```
enter the no of days = 365
in months = 12, years = 1
```

/* find the output of following program */

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i=-1,j=-1,k=0,l=2,m;
clrscr();
m=i++&&j++&&k++||l++;
printf("%d %d %d %d %d",i,j,k,l,m);
getch();
}
```

Output:

```
    0   0  1  3  1
```

/* find the value of a and b */

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b;
clrscr();
a = 200*200/200;
b = 200/200*200;
printf("\n *** values of a and b is ***\n ");
printf("\n a = %d \t b = %d",a,b);
getch();
}
```

Output: -

```
 *** values of a and b is ***
 a = -127        b = 200
```

/* find the output of the following program */

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i=10;
clrscr();
printf("%d  %d  %d  %d",i++,++i,i--,--i);
printf("\n value of i after increment=%d",i);
getch();
}
```

Output:

```
9  9  9  9
 value of i after increment=10
```

/* find the output of the following program*/

```
#include<stdio.h>
#include<conio.h>
void main()
```

```
{
int a,b;
clrscr();
a=1;
b=a++*++a*a++;
printf("%d \t %d",a,b);
getch();
}
```

Output:

```
4      8
```

/* TO CALCULATE THE AREA OF A CIRCLE. */

```
#include<stdio.h>

#include<conio.h>

#define PI 3.14

void main()

{

int r;

float a;

clrscr();

printf("\n ENTER THE RADIUS: ");

scanf("%d",&r);

a=PI*r*r;

printf("\n AREA COMES TO BE: %f",a);

getch();

}
```

Output:

```
ENTER THE RADIUS: 10

AREA COMES TO BE: 314.000000
```

getchar() function:

By using getchar() function we can store only one character. If you give multiple characters also system will take only one character i.e. first character.

/* PRINT OUT THE RESULTS USING getchar(). */

```
#include<stdio.h>

#include<conio.h>

void main()

{

char ch;

clrscr();

printf("\n Enter a character...\n ");

ch=getchar();

printf("\n Result is...\n %c",ch);

getch();

}
```

Output:

```
Enter a character...

 R

 Result is...

 R
```

putchar() function:

By using putchar() function we can print single character value on the console screen. If you give multiple characters also putchar() print only single character (First character).

/* TO PRINT THE OUTPUT USING putchar(). */

```
#include<stdio.h>

#include<conio.h>

void main()

{

char ch='s';

clrscr();

printf("\n Result is as...\n ");
putchar(ch);
getch();
}
```

Output:

```
Result is as...
 S
```

/* TO PRINT THE VALUE OBTAINED BY VARIOUS OPERATORS. */

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b,c,d,e,f,g;
clrscr();
printf("\n Enter value of 'a': ");
scanf("%d",&a);
printf("\n Enter value of 'b': ");
scanf("%d",&b);
c = a + b;
d = a - b;
e = a * b;
f = a / b;
g = a % b;
printf("\n Result is as...");
printf("\n %d \t %d \t %d \t %d \t %d",c,d,e,f,g);
getch();
}
```

Output:

```
Enter value of 'a': 10
Enter value of 'b': 3
```

Result is as...

```
 13     7      30     3      1
```

/* TO FIND GREATEST OF THREE NUMBERS USING CONDITIONAL OPERATOR. */

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

void main()
{
int a,b,c,d;
clrscr();
        printf("\n Enter value of 'a': ");
        scanf("%d",&a);
        printf("\n Enter value of 'b': ");
        scanf("%d",&b);
        printf("\n Enter value of 'c': ");
        scanf("%d",&c);

        d = (a>b) ? ((a>c) ? a:c) : ((b>c) ? b:c);
        printf("\n The number %d is greater",d);
getch();
}
```

CHAPTER6

CONTROL STATEMENTS (LOOPS)

Q) Why Iterative Statements?

=> Iteration means repeated execution of statements. In most situations, we need to execute a set of statements repeatedly for a given number of times.

=> Writing the same set of statements repeatedly will have the following drawbacks.

1) Wastage of time.

2) Wastage of memory.

3) Chances of creating errors.

=> To overcome this problem, we go to the concept of loops. Using loops, we will write the statements only once, and execute them as many no. of times as required.

Q) What do meant by conditional control statement?

=> The statements which can be executed for specified set of times until the given condition satisfies are known as conditional controlled statements. These statements are supported under C language with following key words such as

1. while

2. do-while

3. for

Q) What is a loop?

=> Loop is a group of instructions compiler executes repeatedly while some condition remains true.

Q) Explain types of loops?

⇨ Loops are classified into two types. They are

1. Entry control loop
2. Exit control loop

Entry control loop:

- ⇨ In this control loop compiler will check condition first. If condition is true then compiler move into the body. If condition fail compiler doesn't execute body atleast once.
- ⇨ Entry control loop is also called as Pre-checking control loop.

Ex: while, for

Exit control loop:

- ⇨ In this kind of control loop compiler first executes the body then it go for the condition. If condition is true again it goes to the body.
- ⇨ If condition fails it goes out of the body.
- ⇨ So in exit control loop without knowing condition compiler execute body minimum one time.
- ⇨ Exit control loop is also as post control loop.

Q) Explain steps how to implements loops in a program?

=> Generally every loop require three steps

1. Initialization

2. Condition

3. Iteration

Q) Explain the need of initialization, condition, iteration in a loop?

=> Initialization indicates the starting point of the loop

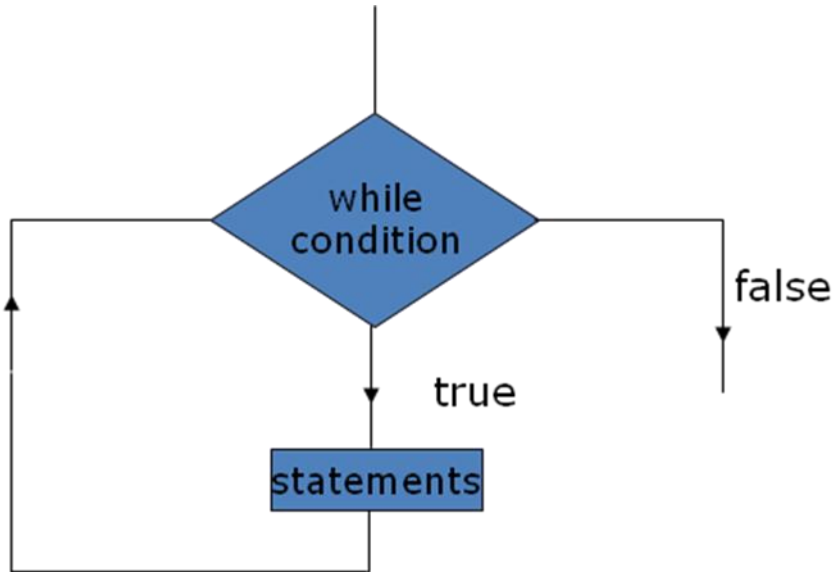
=> Condition indicates ending point of loop (i.e. .when condition fail automatically loop terminated)

=> To rotate the body of the loop either in clock or anti clock wise direction iteration required in a loop.

while: -

syntax:

```
while(condition)

{
statements1;

Statement2;

.

.

.

Increment/ decrement;

}
```



- ⇨ If condition is true it execute statements which are present in side block. If condition is fail it goes out of body.
- ⇨ When we are working with while loop always pre checking process will be occur.
- ⇨ While loop will be repeats in clock direction.

EXAMPLES1:

/* W A P to print one to given values*/

```
#include<stdio.h>

#include<conio.h>

void main()

{

int n,i;

clrscr();

printf("enter n values=");

scanf("%d",&n);

i=1;

while (i<=n)

{

printf("\t%d",i);

i++;

}

getch();

}
```

Output:

```
enter n values=5

        1       2       3       4       5
```

NOTE:

-> Because the condition is placed at the entry of the loop, do-while is also called "entry-restricted loop".

-> In while loop, without executing the statements for at least one time, we can terminate the loop.

EXAMPLE2:

/* write a program to print given number table */

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i,n;
clrscr();
printf("enter n value=");
scanf("%d",&n);
i=1;
while(i<=10)
{
printf("\n %d * %2d = %2d",n,i,n*i);
i=i+1;
}
getch();
}
```

Output:

```
enter n value=2
 2 *  1 =  2
 2 *  2 =  4
 2 *  3 =  6
 2 *  4 =  8
 2 *  5 = 10
 2 *  6 = 12
 2 *  7 = 14
 2 *  8 = 16
 2 *  9 = 18
 2 * 10 = 20
```

EXAMPLE3:

/* write a program to find the following result for a given number

1) number 2)sum of digits 3)number of digits*/

```
#include<stdio.h>

#include<conio.h>

void main()

{

int n,rn,sd,nd;

/* n->number, rn->reverse number, sd-> sum of digits

   nd-> number of digits */

clrscr();

printf("enter n value=");

scanf("%d",&n);

rn=sd=nd=0;

while(n>0)

{

rn=rn*10+n%10;

sd=sd+n%10;

nd=nd+1;

n=n/10;

}

printf("\n reverse number   : %d", rn);

printf("\n sum of digits    : %d", sd);

printf("\n number of digits : %d", nd);

getch();

}
```

Ouput: -

```
enter n value=4567

 reverse number   : 7654
 sum of digits    : 22

 number of digits : 4
```

EXAMPLE4:

//finonacci service

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i,j,n,f;
clrscr();
printf("enter a value=");
scanf("%d",&n);
i=0;
j=1;
printf("%d\t%d",i,j);
f=i+j;
while(f<=n)
{
printf("\t %d",f);
i=j;
j=f;
f=i+j;
}
getch();
}
```

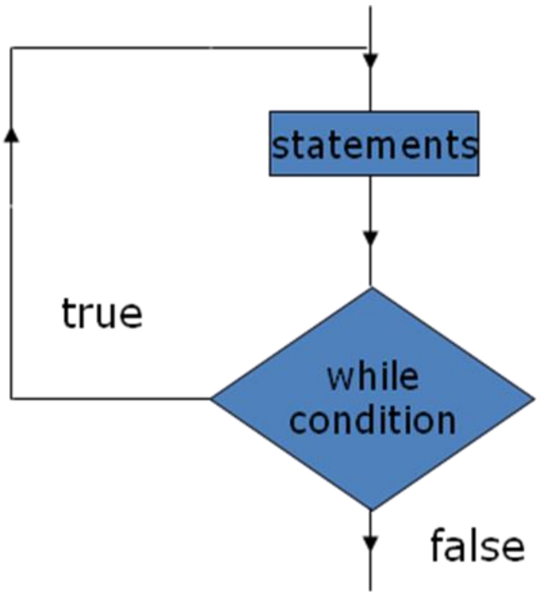
Output:

```
enter a value=125
0     1      1     2     3     5     8     13     21     34
 55     89
```

do – while:

syntax:

```
do

{
statements 1;

Statement2:

.

.

Increment/decrement;

}

While(condition);
```



- ⇨ If condition is true it execute statements which are present in side a block. If condition is false it goes out of body.
- ⇨ When we are working with do while loop always post checking process will be occur.
- ⇨ do While loop will be repeats in anti clock direction.
- ⇨ In do while loop first execute the statement block then only it check the condition
- ⇨ In implementation when we need to execute statement block at least once then go for do while loop.

EXAMPLE5:

/* W A P to print one to given values*/

```
#include<stdio.h>

#include<conio.h>

void main()

{

int n,i;

clrscr();

printf("enter n values=");

scanf("%d",&n);

i=1;
do
{
printf("\t%d",i);
i++;
}
while (i<=n);
getch();
}
```

Output:

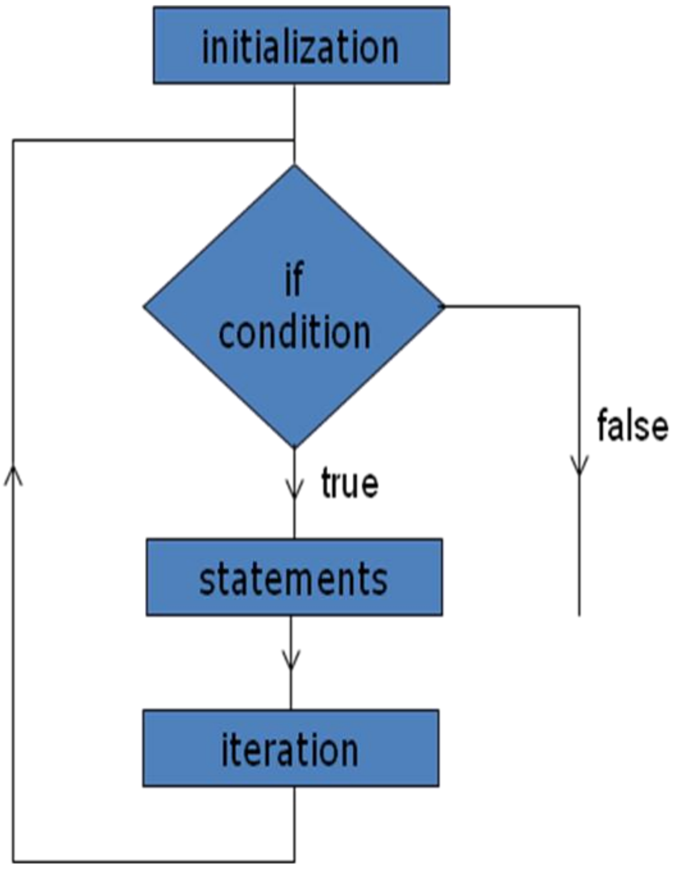
```
enter b values= 5
1       2    3    4    5
```

NOTE:

-> Because the condition is placed at the end of the loop, do-while is also called "exit-restricted loop".

-> In do-while loop, irrespective of the value of the condition, the statements are executed for at least one time.

for:



Syntax:

for (initialization; condition; iteration)

{

Body of the loop;

}

⇨ The execution process of the for loop will starts from initialization block.

⇨ Initialization block will be executed only once when we are entering into the loop first time

⇨ After the execution of initialization block control will pass to condition block. If the condition is evaluated as true then control will pass to statement block.

⇨ After executing of the statement block once again the control will pass to condition block after execution of iteration block.

⇨ Always the reputation will happen between conditional statement block and iteration only.

⇨ When we are working with for loop everything is optional in the for loop but it should be required two semicolons.

⇨ Always the execution process is faster in for loop. Because in for loop is no any bouncing process will be takes place.

⇨ When we are working with the for loop always pre checking process will be occur and it repeats in anti clock wise direction.

EXAMPLE6:

```
/* program on for loop*/

#include<stdio.h>

#include<conio.h>

void main()

{

int i,n;

clrscr();

printf("enter n values=");

scanf("%d",&n);

for(i=1;i<=n;i++)

{

printf("\t %d",i);

}

getch();

}
```

Output:

```
enter n values=10

        1       2       3       4       5       6       7       8       9 10
```

EXAMPLE7:

/*WAP to find out factorial of a given number*/

```
#include<stdio.h>
#include<conio.h>
void main()
{
int n,i,fact=1;
clrscr();
printf("enter the number=");
scanf("%d",&n);
for(i=n; i>0; i--)
{
fact=fact*i;
}
printf("factorial=%d",fact);
getch();
}
```

Output:

```
enter the number=5
factorial=120
```

EXAMPLE8:

/*calculate average of n numbers */

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i,n, no, sum=0;
float average;
clrscr();
printf("enter total no.of number=");
scanf ("%d",&n);
for (i=1;i<=n;i++)

{

printf("enter the number=");

scanf ("%d",&no);

sum=sum+no;

}

average=sum/n;

printf ("\n average=%f", average);

getch();

}
```

Output:

```
enter total no.of number=5

enter the number=1

enter the number=2

enter the number=3

enter the number=4

enter the number=5

 average=3.000000
```

EXAMPLE9:

/* write a program to implement follwing format

1

1 2

1 2 3

1 2 3 4

1 2 3 4 5

(assume in this case n value is 5) */

```
# include <stdio.h>

# include <conio.h>

void main()
```

```
{
int n, k, j ;
clrscr() ;
printf("Enter any number ") ;
scanf("%d", &n) ;
for(k=1; k<=n; k++)
{
printf("\n ") ;
for(j=1; j<=k; j++)
printf("%d ", j);
}
getch();
}
```

OUTPUT:

```
Enter any number 5
 1
 1 2
 1 2 3
 1 2 3 4
 1 2 3 4 5
```

EXAMPLE10:

/* write a program for following format

1

2 2

3 3 3

4 4 4 4

5 5 5 5 5

(assume here n value is 5) */

```
#include<stdio.h>

#include <conio.h>

void main()

{

int n, k, j ;

clrscr() ;

printf("Enter any number ") ;

scanf("%d", &n) ;

for(k=1; k<=n; k++)

{

printf("\n ") ;

for(j=1; j<=k; j++)

printf("%d ", k) ;

}

getch() ;

}
```

Output:

```
Enter any number 5

 1

 2 2

 3 3 3

 4 4 4 4

 5 5 5 5 5
```

Q) Explain about nested loops? Give one Example?

⇨ A loop can be executed with in another loop is called as nested loop. By using nested loops we can easily solve complex problems.

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i,j,x;
printf("\n enter value of x:");
scanf("%d",&x);
clrscr();
for(j=1;j<=x;j++)
{
for(i=1;i<=j;i++)
printf("%3d",i);
printf("\n");
}
printf("\n");
for(j=x;j>=1;j--)
{
for(i=j;i>=1;i--)
printf("%3d",i);
printf("\n");
}
getch();
}
```

Output:

```
  1
  1  2
  1  2  3
  1  2  3  4
```

```
1 2 3 4 5


5 4 3 2 1

4 3 2 1

3 2 1

2 1

1
```

break:

- break is a keyword by using break we can terminate the loop body or switch body.
- Using break is always optional but it should be exist with in the loop body or switch body only.
- In implementation where we knows the maximum number of reputations but certain condition is their where we need to terminate loop body then go for break keyword.

continue:

- continue is a keyword by using continue we can skip the statements with in the loop body.
- Using continue is always optional but it should be exist with in the loop body only.
- In implementation where we knows the maximum number of reputations but certain condition is their where we need to skip some statements from loop body then go for continue.

goto:

- goto is keyword by using goto we can pass the control any place in the program with in the local scopr.
- goto always refers on identifier called label.
- Any valid identifier followed by colon is called label.
- goto statement makes the program in unstructured manner.

EXAMPLES:

/* WRITE A PROGRAM ON GOTO KEYWORD */

```
# include <stdio.h>
# include <conio.h>
void main()
{
clrscr();
printf("\n Hello ");
printf("\n Friends ");
goto abc;
printf("\n Go out ");
xyz:
printf("\n To NGC ");
goto end;
abc:
printf("\n Welcome ");
goto xyz;
end:
getch();
}
```

OUTPUT:

```

 Hello
 Friends
 Welcome
 To NGC
```

/* write a program on break keyword */

```
#include<stdio.h>
#include<conio.h>
main()
```

```
{
int i;
clrscr();
for(i=0;i<10;i++)
{
if(i==5)
break;
printf("%2d",i);
}
getch();
}
```

Output:

```
1     2     3     4
```

/* write a program on continue keyword */

```
#include<stdio.h>
#include<conio.h>
main()
{
int i;
clrscr();
for(i=0;i<10;i++)
{
if(i==5)
continue;
printf("%2d",i);
}
getch();
}
```

Output:

```
1       2       3       4       6       7       8       9       10
```

CHAPTER7
CONDITIONAL STATEMENTS

CONDITIONAL STATEMENTS:

Selection statements: These statements are also called 'Branching Statements or conditional statements'. Because they are used to transfer control from one statement to another

Conditional statements are mainly classified into two types they are

1. If statements,
2. Switch statement.

If statements are again classified into three types

if()

"C" uses the keyword "if" to implement the decision control statement or instructions. The general form of its statement looks like is

Syntax:

```
If (condition)

{

Statements 1;

Statements n;

}
```

- If the condition is true then executes the if block statements otherwise it does not execute the statements.
- In implementation when we need to create decision making block then go for if.
- Constructing the body is always optional.
- For a single statements body need not be constructed.
- If the body is not specified then automatically condition path terminated with next semi colon.
-

if else statements:

The if else statement is an extension of the simple if statement. The general form is or syntax is

Syntax;-

```
if (condition)
{
Block of statements; //true block
}
else
 {
Block of statements; //false block
  }
```

If the condition is true then it executes the if block statements otherwise if executes the else block statements.

If else if statements:-

It is also an extension of if else statement to implement multiple conditions. General form is

Syntax:

```
if (condition)

{

Block of statements;
}
else if (condition)
{
Block of statements;
}
.
.
else
{
Block of statements;
}
```

If the condition is true then it executes the block statements otherwise it executes the else block statements .if, 1st if condition is not true then compiler moves into second if condition if it is true then compiler enter into body of that particular block. If suppose above all mentioned condition are fail then finally compiler moves to else block

SWITCH STATEMENT:

- ⇨ Switch is keyword, by using switch we can constructs multiple blocks of a selection statements.
- ⇨ Multiple blocks can be constructed with "case" keyword.
- ⇨ It provides an easy way to dispatch execution to different parts of your code based on the value of an expression. As such it often provides a better alternative than a large series of if-else statements.

Syntax:

```
switch (expression)
{
case value1:
//statement sequence
break;
case value 2;
//statement  sequence
break;
.
.
.
case value n;
//statement sequence
Break;
default:
//default statement sequence
}
```

- ⇨ The switch statement works like this: the value of the expression is compared with each of the constant values in the case statement. If a match is found, the code sequence following that case statement is executed. If none of the constants matches the value of the expression then the default statement is executed.
- ⇨ Default is a special kind of case which will be executed automatically when the matching case is not found.
- ⇨ Using default is always optional. It is recommended to use when we are not handling all provisions of switch block.

EXAMPLES:

/*WAP to print big value of given two numbers by using if statement k*/

```
#include<stdio.h>

#include<conio.h>

void main()

{

int a,b;

clrscr();

printf ("enter a,b, values =");

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

if(a>b)

{

printf("a is greater than b", a, b);

}

if(b>a)
{
printf("b is greater than a", b, a);
}
getch();
}
```

Output:

```
enter a,b, values =10
20
b is greater than a
```

<u>Example2:</u>

/*WAP to print the given value is even or odd */

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a;
clrscr();
printf("enter a value=");
scanf("%d", &a);
if(a%2==0)
{
printf("a is even ", a );
}
else
{
printf("a is odd", a);
}
getch();
}
```

Output:

```
enter a value=50
a is even
```

EXAMPLE3:

/*WAP to check the given character is vowel or not */

```
#include<stdio.h>
#include<conio.h>
void main()
{
char ngc;
clrscr();
printf("enter a character=");
scanf("%c", &ngc);
if(ngc=='a')
{
printf("a is vowel", ngc);
}
else if(ngc=='e')
{
printf("it is vowel", ngc);
}
else if (ngc =='o')
{
printf("it is vowel", ngc);
}
else if(ngc=='i')
{
printf("it is a vowel",ngc);
}
else if ( ngc =='u')
{
printf("u is vowel", ngc);
}
else
{
printf("%c is a not vowel", ngc);
}
getch();
}
```

Output:

```
enter a character=o
it is vowel
```

EXAMPLE4:

/*WAP to check the given number is positive or negative */

```
#include<stdio.h>
#include<conio.h>
void main()
{
int k;
clrscr();
printf("enter any number=");
scanf("%d", &k);
if(k==0)
{
```

```
printf("the number is zero");
}
else if(k>0)
{
printf("the number is positive");
}
else
{
printf("the number is negative");
}
getch();
}
```

Output:

```
enter any number=25
the number is positive
```

EXAMPLE5:

/* WAP to print the given year value is leap year or not */

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a;
clrscr();
printf("enter a value=");
scanf("%d", &a);
if(a%4==0)
{
printf("%d is leap year", a);
}
else
{
printf("%d is not a leap year",a);
}
getch();
}
```

Output:

```
enter a value=2012
2012 is leap year
```

EXAMPLE6:

/*Determine the division obtained by a student */

```
#include<stdio.h>
#include<conio.h>
void main()
{
int m1,m2,m3;
float per;
clrscr();
printf("enter marks in three subjects=");
scanf("%d", &m1);
scanf("%d", &m2);
scanf("%d", &m3);
per=(m1+m2+m3)/3;
if(per>=60)
{
printf("first division");
}
else if(per>=50)
{
printf("second division");
}
else if (per>=40)
{
printf("third division");
}
else
{
printf("fail");
}
getch();
}
```

Output:

```
enter marks in three subjects=45
65
89
first division
```

EXAMPLE7:

/* w a p to find the given number is armstrong number or not*/

```
#include<stdio.h>
#include<conio.h>
void main()
{
int n,temp,i,sum=0;
clrscr();
printf("enter n value=");
scanf("%d",&n);
for(temp=n;temp!=0;)
{
i=temp%10;
sum=sum+(i*i*i);
temp=temp/10;
}
if(sum==n && n!=1)
{
printf("\n %d is armstrong number ",n);
}
else
{
printf("%d is not a armstrong number",n);
}
getch();
}
```

Output:

```
enter n value=153
153 is armstrong number
```

EXAMPLE8:

/* w a p find the given number is perfect number or not */

```
#include<stdio.h>
#include<conio.h>
void main()
{
int n,i,sum=0;
clrscr();
printf("enter n value=");
scanf("%d",&n);
for(i=1;i<n;i++)
{
if(n%i==0)
{
sum=sum+i;
}
}
if(n==sum)
{
printf("\n %d is perfect number ",n);
}
else
{
printf("%d is not a perfect number",n);
}
getch();
}
```

Output:

```
enter n value=6
6 is perfect number:
```

EXAMPLE9:

/* Write a C program to generate all the prime numbers between 1 and n, where n is a value supplied by the user. */

```
#include <stdio.h>
void main()
{
int no,counter,counter1,check;
clrscr();
printf("\nINPUT THE VALUE OF N:");
scanf("%d",&no);

printf("\n\nTHE PRIME NO. SERIES B/W 1 TO %d : \n\n",no);


for(counter = 1; counter <= no; counter++)

{

check = 0;

for(counter1 = counter-1; counter1 > 1 ; counter1--)

if(counter%counter1 == 0)

{

check++;

break;

}

if(check == 0)

printf("%d\t",counter);

}

getch();

}
```

Output:

```
INPUT THE VALUE OF N:100

THE PRIME NO. SERIES B/W 1 TO 100 :

1       2       3       5       7       11      13      17      19      23   29      31      37      41
43      47      53      59   61      67      71      73      79      83      89      97
```

EXAMPLE10:

/* Write a C program to find the roots of a quadratic equation. */

```
#include<stdio.h>

#include<conio.h>

#include<math.h>

void main()

{

float a,b,c,root1,root2;
clrscr();
printf("\n Enter values of a,b,c for finding roots of a quadratic eq:\n");
scanf("%f%f%f",&a,&b,&c);
/*checking condition*/
if(b*b>4*a*c)
{
root1=-b+sqrt(b*b-4*a*c)/2*a;
root2=-b-sqrt(b*b-4*a*c)/2*a;
printf("\n*****ROOTS ARE*****\n");
printf("\n root1=%f\n root2=%f",root1,root2);
}
else
printf("\n Imaginary Roots.");
getch();
}
```

Output:

```
Enter values of a,b,c for finding roots of a quadratic eq:
12
14
16
 Imaginary Roots.
```

EXAMPLE11:

/*WAP to implement Arithmetic operation in a switch */

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b,k;
clrscr();
printf("enter two numbers=");
scanf("%d%d", &a, &b);
printf("\n 1.addition:");
printf("\n 2. Substraction:");
printf("\n 3.multiplication:");
printf("\n 4.division");
printf("\n select your choice:");
scanf("%d", &k);
printf("\n");
switch(k)
{
case 1:printf("the addition=%d", a+b);
break;
case 2:printf("the subtraction =%d",a-b);
break;
case 3:printf("the multiplication=%d", a*b);
break;
case 4:printf("the division=%d", a/b);
break;
default:printf("in valid choice");
}
getch();
}
```

Output:

```
enter two numbers=20
50
 1. addition:
 2. Substraction:
 3 .multiplication:
```

```
 4. division

 select your choice:1

the addition=70
```

EXAMPLE12:

/* write a program to find given number is less than or greater than five by using switch*/

```
# include <stdio.h>

# include <conio.h>

void main()

{

int k ;

clrscr() ;

printf("Enter any number " );

scanf("%d", &k) ;

switch(k)

{

case  0 : printf("\n Number is zero " );

case  1 :

case  2 :

case  3 :

case  4 :  printf("\n Number is less than five ") ;

break ;

case  5 : printf("\n Number is five " );

break ;

default : printf("\n Number is greater than five ") ;

}

getch() ;

}
```

Output:

```
Enter any number 4
 Number is less than five
```

EXAMPLE13:

```
# include <stdio.h>
# include <conio.h>
main()
{
char  k;
clrscr() ;
printf("Enter any one of the alphabets " );
k = getche() ;
switch(k)
{
case  'a' :
case  'A' :  printf("\n A for Active ") ;
break ;
case  'b' :
case  'B' :  printf("\n B for Brave ") ;
break ;
case  'c' :
case  'C' : printf("\n C for Courage ");
break ;
case  'd' :
case  'D' : printf("\n D for Dare ") ;
break ;
default  : printf("\n You are timid ") ;
}
getch() ;
}
```

Output:

```
Enter any one of the alphabets A
 A for Active
```

gotoxy() :

This function locates the cursor position to the given place on the screen. This function's prototype has defined in the header file CONIO.H

Syntax:

gotoxy(column, row) ;

Generally in MS-DOS mode the screen contains 80 columns and 25 rows.

EXAMPLE:

```
# include <stdio.h>

# include <conio.h>

void main()

{

clrscr() ;

gotoxy(10, 5) ;

printf("Hello ");

gotoxy(15, 10);

printf("Bhanodaya ") ;

gotoxy(20,15);

printf("Welcome ");

gotoxy(25,20);

printf("To smile ");

getch() ;

}
```

Output:

```
Hello

          Bhanodaya

                    Welcome

                               To smile
```

Self writing programs:

1. Write a program to implement your bio-data.
2. Write a program to explain about India.
3. Write a program to find area of circle
4. Write a program to find perimeter of circle.
5. Write a program to print ASCII codes
6. Write a program to print odd numbers up to given number.
7. Write a program to print even number up to given number.
8. Write a program to find the student result.
9. Write a program to find length of the string using prinyf() function.
10. Write a program to swap the values of two variables without using third variable.
11. Write a program to perform all mathematical (+,-,*,/) operations without third variable.
12. Write a program to check person is eligible for voter id or not.
13. Write a program to print squares of first five number(1,2,3,4,5).
14. Write a program to count number between 1 to 100 not divisible by 2,3 and 5.
15. Write a C program to calculate the following Sum:
    Sum=1-x2/2! +x4/4!-x6/6!+x8/8!-x10/10!

CHAPTER8 ARRAYS

ARRAYS

Q) What is the need of arrays?

- A variable having only one memory location so we can store only one value at a time. But my requirement is i want to store multiple data values into one variable at a time (the total data belonging to similar data only). In c language by using arrays we can achieve this requirement (i.e. storing set of data values into single variable). This is the need of arrays.

Q) What is array? Explain its properties?

- An array is a collection of data storage locations, each having the same data type and the same name.
- Each storage location in an array is called an array element or array index.
- Array is derived data type.
- Array is a composite data type.

Properties:

1. Array index values always start from zero(0).
2. The variable name of array contains the base address of the memory block.
3. The array variable is created at the time of compilation.
4. The size of the array can't be alerted at runtime.
5. The address held by the array variable is static.
6. The array size must be constant.

Q) What are the advantages and disadvantages of array?

Advantages:

- Array supporting grouping of similar data elements.
- Referring more than one value with the same name which decreases complexity in reading and writing values.

Disadvantages:

- The disadvantage of array is wastage of memory because of its static behavior.

Q) Explain types of arrays?

Arrays are broadly classified in two types. They are

1. Single Dimensional arrays.
2. Multi Dimensional arrays.

Q) What is single dimensional array? Explain with examples?

- One dimensional array is a list of values of the same data type. A one dimensional array contains only one subscripts
- A subscript is a number inside a bracket that follows an array's name. This number can identify the number of individual elements in the array.
- Array elements are stored in sequential memory locations

Syntax:

<data-type> <variable> [size] ;

Ex:

int rno[100];

float sal[200];

chat name[50];

- Before going to use any array we have to declare first.
- The data-type can be int, float, char, double, long etc.
- The size must be integer value only.
- Size must be constant.
- We can’t store different data type values into single array.

Initialization:

Syntax:

<Data-type> <arrayname> [size]={ele1,ele2,ele3,..,.,.,.,.,.elen};

Ex:

int eno[5] = { 56, 67, 45, 89, 45};

char ena[] = { ‘a’, ‘b’, ‘c’, ‘d’ } ;

float bas[] = { 56.67, 900.45, 567 } ;

NOTE:

- Remember that array elements start with 0, not 1. Also remember that the last element is one less than the number of elements in the array. For example, an array with 10 elements contains elements 0 through 9.

EXAMPLE1:

/* write a program to store 10 elements into a single variable*/

```
#include<stdio.h>
#include<conio.h>
void main()
{
int num[10],i;
clrscr();
for(i=0;i<10;i++)
{
printf("enter number value=");
scanf("%d",&num[i]);
}
for(i=0;i<10;i++)
{
printf("\nentered numbers are=%d",num[i]);
}
getch();
}
```

OUTPUT:

```
enter number value=10
enter number value=20
enter number value=30
enter number value=40
enter number value=50
enter number value=60
enter number value=70
enter number value=80
enter number value=90
enter number value=100
entered numbers are=10
entered numbers are=20
entered numbers are=30
entered numbers are=40
entered numbers are=50
entered numbers are=60
entered numbers are=70
entered numbers are=80
entered numbers are=90
entered numbers are=100
```

EXAMPLE2:

/* Write a C program to find both the larges and smallest number in a list of integers.*/

```
#include<stdio.h>
#include<conio.h>
void main()
{
char name[25];
int sub[100];
int n,min,max,i;
clrscr();
printf("enter student name=");
scanf("%s",&name);
printf("\n how many subjects=");
scanf("%d",&n);
printf("\n enter subject marks=");
for(i=0;i<n;i++)
scanf("%d",&sub[i]);
min=max=sub[0];
for(i=0;i<n;i++)
{
if(sub[i]>max)
max=sub[i];
else
if(sub[i]<min)
min=sub[i];
}
printf("\n student name=%s",name);
printf("\n minimum marks=%d",min);
printf("\n maximum marks=%d",max);
getch();
}
```

Output:

```
enter student name=ravi
 how many subjects=6
enter subject marks=89
```

```
65
45
84
98
72
 student name=ravi
 minimum marks=45
 maximum marks=98
```

Q) What is multidimensional array? Explain with examples?

Multidimensional Arrays:

- ⇨ A multidimensional array has more than one subscript. I.e. A two-dimensional array has two subscripts; a three-dimensional array has three subscripts, and so on. There is no limit to the number of dimensions C array can have.
- ⇨ A two-dimensional array has a row-and-column structure.

Initializing Multidimensional Arrays:

- ⇨ Multidimensional arrays can also be initialized. The list of initialization values is assigned to array elements in order, with the last array subscript changing first. For example:

EXAMPLE:

```
int array[4][3]={ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
```

Results in the following assignments:

```
array[0][0] is equal to 1
array[0][1] is equal to 2
array[0][2] is equal to 3
array[1][0] is equal to 4
array[1][1] is equal to 5
array[1][2] is equal to 6
...
…
…
array[3][1] is equal to 11
array[3][2] is equal to 12
```

- ⇨ When you initialize multidimensional arrays, you can make your source code clearer by using extra braces to group the initialization values and also by spreading them over several lines. The following initialization is equivalent to the one just given:

EXAMPLE:

```
int array[4][3] = { { 1, 2, 3 } , { 4, 5, 6 } ,{ 7, 8, 9 } , { 10, 11, 12 } };
```

⇨ Remember, initialization values must be separated by a comma--even when there is a brace between them. Also, be sure to use braces in pairs--a closing brace for every opening brace--or the compiler becomes confused.

SOME MORE EXAMPLES ON MULTIDIMENSTIONAL ARRAYS:

```
int   a[10][5][3];
float  sqr[3][3][3][3];
float   area[5][5][5][5]…….[n];
```

PROGRAMS:

EXAMPLE1:

```
/* Write a program to perform addition of two matrices*/

#include<stdio.h>
#include<conio.h>
void main()
{
int i,j, a[2][3],b[2][3],c[2][3];
clrscr();
printf("\n enter the array a elements");
for(i=0; i<2; i++)
{
for(j=0;j<3;j++)
{
scanf("%d", &a[i][j]);
}
}
printf("\n enter the array  B elements");
for(i=0; i<2; i++)
{
for(j=0; j<3; j++)
{
scanf("%d", &b[i][j]);
}
}
printf("\n sum of a and b is=");
for (i=0; i<2; i++)
{
for(j=0; j<3; j++)
{
c[i][j] = a[i][j]+b[i][j];
printf("\t%d", c[i] [j]);
}
}
getch();
}
```

output:

```
 enter the array a elements10
20
30
40
50
60

 enter the array  B elements10
20
30
40
50
60

 sum of a and b is=     20      40      60      80      100     120
```

EXAMPLE2:

```
/*write a to print multiplication of two matrixs*/
#include<stdio.h>
#include<conio.h>
void main()
{
int a[3][3],b[3][3],c[3][3];
int i,j,k;
clrscr();
printf("enter a array elements:");
for(i=0;i<3;i++)
{
for(j=0;j<3;j++)
scanf("%d", &a[i][j]);
}
printf("in enter b array element=");
for (i=0;i<3;i++)
{
for(j=0;j<3;j++)
{
scanf("%d", &b[i][j]);
}
}
printf("\n the product of a and b is in");
for(i=0;i<3;i++)
{
for(j=0;j<3;j++)
{
c[i][j]=0;
for(k=0;k<3;k++)
{
c[i][j]=c[i][j]+(a[i][k]*b[k][j]);
}
}
}
printf("\n\n");
for(i=0; i<3; i++)
{
for(j=0; j<3; j++)
printf("\t%d", c[i][j]);
printf("\t\n");
}
getch();
}
```

Output:

```
enter a array elements:1
1
1
1
1
1
1
1
1
in enter b array element=1
1
1
1
1
1
1
1
1
 the product of a and b is in

        3       3       3
        3       3       3
        3       3       3
```

EXAMPLE3:

/*wap to find out entered matrix is unit or not */

```
 #include<stdio.h>
 #include<conio.h>
 main()
 {
 int a[3][3],i,j;
 int t=0;
 printf("enter array elements=");
 for(i=0;i<3;i++)
 {
 for(j=0;j<3;j++)
 {
 scanf("%d", &a[i] [j]);
 }
 }
 for (i=0; i<3; i++)
 {
 for (j=0; j<3; j++)
 {
 if(a[i][j]!=1)
 t=t+1;
 }
 }
 if (t==0)
 printf("\n unit matrix");
 else
 printf(" not a unit matrix");
 getch();
 }
```

Output:

```
enter array elements=1 1 1
1 1 1
1 1 1
 unit matrix
```

EXAMPLE4:
/* wap to find sum of diagonals of an square matrix*/

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i,j, a[3][3], sum=0;
clrscr();
printf("enter array element=");
for (i=0; i<3; i++)
{
for (j=0; j<3; j++)
{
scanf ("%d", &a[i][j]);
}
}
for (i=0; i<3; i++)
{
for (j=0; j<3; j++)
{
if (i==j)
sum=sum+a[i][j];
}
}
printf("sum of diagonals=%d", sum);
getch();
}
```

Output:

```
enter array element=1 2 3
2 1 3
3 2 1
sum of diagonals=3
```

CHAPTER9 STRINGS

<u>STRINGS</u>

Introduction

A string is a sequence of characters. The characters include alphabets, digits, special symbols.

Example:

ngc

College

Ramana143

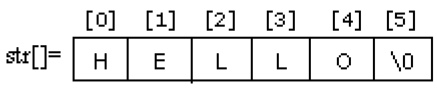
Civilengineering

C- Strings:

In C Language, strings are sequence of characters enclose in double quotes (" "). They are also called as character constants.

In C, a string is also defined as array of characters. Similarly, there is no separate data type available to represent string. Such arrays of strings are terminated by a Null characters('\0'), that is slash followed by zero.

For example, the following array represents a string,



Example,

"ngc"

"College"

"Ramana143"

"Civilengineering" are all treated as strings in C- Language.

Similarly, the C declarations,

char *string1 = "Hello";
char string2[] = "Hello";
char string3[6] = "Hello"; are string constants.

The C–Program to demonstrate how to assign and print strings.

```
#include <stdio.h>
 void main()
 {
  int a = 'H';   /* character constants */
  char b = 'm';

  char str1[6] = {'H','E','L','L','O'};    /* a character array */

  char str2[7] = {'C','o','f','f','e','e','\0'}; /* extra element for a null character! */

  char str3[] = "Hello world";
  char *str4;
  str4 = "Hyderabad";

  clrscr();
  printf("a : %c\n", a);
  printf("b : %c\n", b);
  printf("str1[] : %s\n", str1);
  printf("str2[] : %s\n", str2);
  printf("str3[] : %s\n", str3);
  printf("str4     : %s\n", str4);
  getch();
}
```

Output:

```
a : H

b : m

str1[] : HELLO

str2[] : Coffee

str3[] : Hello world

str4     : Hyderabad
```

Note that you can only use the string format specifier, %s, for string constants.

In the above, the strings assigned in from program only. Those strings holder are fixed for the string assigned to them, so they are constants.

String Input/Output Functions

The scanf(), gets() are predefined functions for reading strings or array of characters thru standard input called keyboard, the printf(), puts() predefined functions are used to for printing strings or array of characters on standard output called monitor.

There is a difference between scanf and gets while reading string. The scanf considers space and newline in input as end of string, where as gets considers newline input as end of string.

The printf() and scanf() for input/output and, the gets() and puts() for string input/output are defined in header file stdio.h, so, include it to program which uses these functions.

There are some more input/output function in C, they will be discussed in coming chapters.

Example:

Program that uses the scanf() and the printf() functions for reading and printing the array of characters or strings.

```
#include <stdio.h>
void main()
{
  char str1[15];
  char *str2="";
  clrscr();

  printf("Enter str1[]:");
  scanf("%s", str1);
  printf("str1[] : %s\n", str1);

  printf("Enter str2:");
  scanf("%s", str2);
  printf("str2 : %s\n", str2);
  getch();
}
```

Output:

```
Enter str1[]:Hello world

str1[] : Hello

Enter str2:

str2 : world

Enter str1[]:JNTU      <press enter key>

str1[] : JNTU

Enter str2: University,Hyd

str2 : University,Hyd
```

Program using gets() and puts() for reading and printing respectively the array of characters or strings.

```
#include <stdio.h>
 void main()
 {
  char str1[15];
  char *str2="";

  clrscr();

  puts("Enter str1[]:");
  gets(str1);
  puts(str1);

  puts("Enter str2:");
  gets(str2);
  puts(str2);
}
```

Output:

```
Enter str1[]:Hello world
```

```
Hello world

Enter str2: ramesh

ramesh
```

Arrays of Strings:

In C, the array of string is defined as two-dimensional array of characters. This is defined as follows,

```
char names[4][20];
```

this a array, for storing 4 names, each name can has at most 20 characters, but last character reserved for null characters('\0').

Similarly,

```
char  *names[4];
```

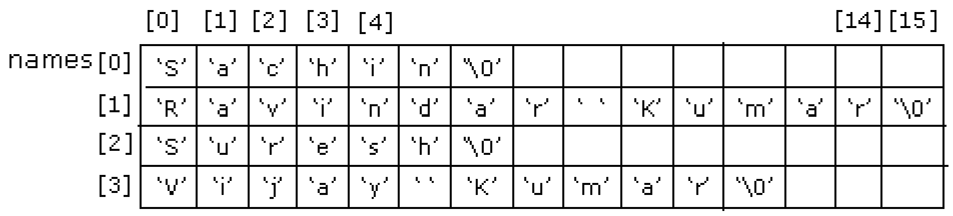
this is array of 4 names, each name can has at most specified number of characters,

certainly, the last character would be null character ('\0').

For example,

```
char names[4][15] ={"Sachin", "Ravindar Kumar", "Suresh", "Vijay Kumar"};
```

stores the strings into array of strings as shown below.

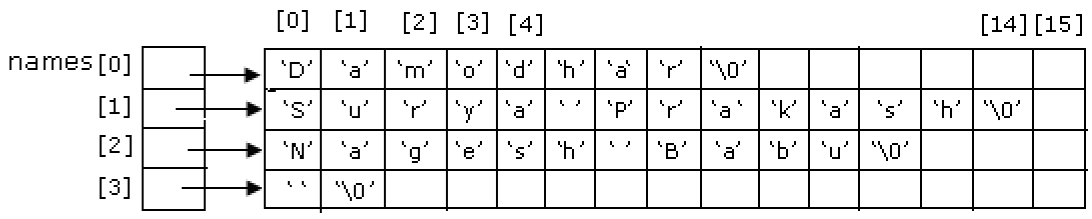


[0] [1] [2] [3] [4] [14] [15]
names [0] 'S' 'a' 'c' 'h' 'i' 'n' '\0'
[1] 'R' 'a' 'v' 'i' 'n' 'd' 'a' 'r' ' ' 'K' 'u' 'm' 'a' 'r' '\0'
[2] 'S' 'u' 'r' 'e' 's' 'h' '\0'
[3] 'V' 'i' 'j' 'a' 'y' ' ' 'K' 'u' 'm' 'a' 'r' '\0'

Similarly, with the statement,

```
char  *names[4] ={ "Damodhar", "Surya Prakash", "Nagesh Babu", " "};
```

stores the strings into array of strings as shown below.



[0] [1] [2] [3] [4] [14] [15]
names [0] 'D' 'a' 'm' 'o' 'd' 'h' 'a' 'r' '\0'
[1] 'S' 'u' 'r' 'y' 'a' ' ' 'P' 'r' 'a' 'k' 'a' 's' 'h' '\0'
[2] 'N' 'a' 'g' 'e' 's' 'h' ' ' 'B' 'a' 'b' 'u' '\0'
[3] ' ' '\0'

We can read and print the strings using program.

Program to demonstrate reading/printing strings from C-program

```
#include<stdio.h>
void main()
{
char name[4][15];
int i;

clrscr();

printf("Enter 4 names:\n");
for(i=0;i<4;i++)
scanf("%s", name[i]);

printf("\nThe names[][] are:\n");
i=0;
while(i<4)
{
printf("name[%d] : %s\n", i, name[i]);
i = i + 1;
}

getch();
}
```

Output:

```
Enter 4 names:

Sachine

Suresh

Vijay

rahul

The names[][] are:

name[0] : Sachine

name[1] : Suresh

name[2] : Vijay

name[3] : rahul
```

Another program to demonstrate reading/printing strings from C-program

```
#include<stdio.h>
void main()
```

```
{
char *name[4];
int i;

clrscr();

printf("Enter 4 names:\n");
for(i=0;i<4;i++)
scanf("%s", name[i]);
printf("\nThe names[][] are:\n");
i=0;
while(i<4)
{
printf("name[%d] : %s\n", i, name[i]);
i = i + 1;
}

getch();
}
```

```
Enter 4 names:

Damodhar

Ragul

Nagesh

Vijay



The names[][] are:

name[0] : Damodhar

name[1] : Ragul

name[2] : Nagesh

name[3] : Vijay
```

The gets, puts can also be used in above program. The scanf, printf, gets, and puts function can be in any combination in any C-program.

String Manipulation Functions

The string manipulation functions are predefined functions, they are used to perform operations on strings. They are defined in header file called “string.h”. Therefore, the program using such functions should include this header file(#include<string.h>).

Most frequently used string functions from “string.h” are,

1. strcat
2. strlen
3. strncpy
4. stricmp
5. strcmpi
6. strupr
7. strlwr
8. strstr
9. tolower
10. toupper

and many more, each of these are discussed in following explanation.

strcpy():

This function copies a string into another string.

The general structure of function is,

```
char *strcpy(char *dest, char *source);
```

Example:

```
/* wap  to copy a string to another */

#include<stdio.h>

#include<conio.h>

void main()

{

char a[100],b[100];

clrscr();

puts("enter a string=");

gets(a);

puts("enter b string=");

gets(b);

strcpy(a,b);

printf("\n copied(a) string=%s",a);

printf(" \n original(b) string=%s", b);

getch();

}
```

Output:

```
enter a string=
ngc
enter b string=
ramesh
 copied(a) string=ramesh
 original(b) string=ramesh
```

strcmp():

This function compares two strings with case sensitivity.

Syntax:

```
#include<string.h>
int strcmp( char *str1,  char *str2 );
```

The function strcmp() compares str1 and str2, then returns,

Return value	*Explanation*
less than 0(-ve)	str1 is less than str2
equal to 0	str1 is equal to str2
greater than 0(+ve)	str1 is greater than str2

Example:

/* write a program to compare two strings */

```
#include<stdio.h>
#include<conio.h>
void main()
{
char a[100],b[100];
int i;
clrscr();
puts("enter a string=");
gets(a);
puts("enter b string=");
gets(b);
i=strcmp(a,b);
printf("\n comparing string=%d",i);
getch();
}
```

Output:

```
enter a string=
ravi
enter b string=
ramesh
comparing string=9
```

strcat()

This function concatenates two strings, and returns the resultant string.

Syntax:

```
#include <string.h>

char *strcat( char *str1, char *str2 );
```

The strcat() function concatenates str2 onto the end of str1, and returns str1. Example*:*

/* write a program to combine two string */

```
#include<stdio.h>
#include<conio.h>
void main()
{
char a[100],b[100];
int i;
clrscr();
puts("enter a string=");
gets(a);
puts("enter b string=");
gets(b);
strcat(a,b);
printf("\n concated string=%s",a);
getch();
}
```

Output:enter a string=

ramesh

enter b string=reddy

concated string=rameshreddy

strlen()

This function finds the length of a given string and returns integer value.

Syntax:

```
#include <string.h>
int strlen( char *str );
```

The strlen() function returns the length of str (determined by the number of characters excluding null character which indicate termination of string).

Example:

```
        char *str = “Hello world”;

        int length;

        length = strlen(str);
```

strncpy()

This function copies the specified number of characters from a string into another string. Syntax:

```
#include <string.h>

char *strncpy( char *str1,  char *str2, int n );
```

The strncpy() function copies at most n characters of str2 to the string str1. If str2 has less than n characters, the remaining cells is padded with '\0' characters. The return value is the resulting string in str1.

Example:

```
/* wap  to copy required number of characters into anotherstring */
#include<stdio.h>
#include<conio.h>
void main()
{
char a[100],b[100];
int i,n;
clrscr();
puts("enter a string=");
gets(a);
puts("enter b string=");
gets(b);
puts("enter how many characters=");
scanf("%d",&n);
strncpy(a,b,n);
printf("\n copied string=%s",a);
getch();
}
```

Output:

```
enter a string=

ngc

enter b string=

ramesh

enter how many characters=
2

copied string=ratwin
```

stricmp(), strcmpi():

These functions compare two strings without case sensitivity.

Syntax:

```
#include<string.h>
int stricmp( char *str1,  char *str2 );
int strcmpi( char *str1,  char *str2 );
```

The function stricmp() or strcmpi() compares str1 and str2, then returns,

Return value	*Explanation*
less than 0	str1 is less than str2
equal to 0	str1 is equal to str2
greater than 0	str1 is greater than str2

strupr(), strlwr():

The strupr() function converts all lower case letters in a given string into upper case.

The strlwr() function converts all upper case letters in a given string into lower case.

Syntax:

```
char *strupr( char *str );

char *strlwr( char *str );
```

strstr():

This strstr() function finds the first occurrence of a given substring in a given string.

Syntax:

```
        char* strstr (char *str1, char *str2);
```

This function searches for first occurrence of str2 in str1. If success, it returns pointer to the element or cell of memory where str2 begins in str1. Otherwise returns null pointer.

tolower(), toupper()

The tolower() function converts a upper case character to lower case.

The toupper() function convert a lower case character to upper case.

Syntax:

```
        char  tolower(char c);

        char  toupper(char c);
```

CHAPTER10

FUNCTIONS

Functions:

DEFINITION:

A function is a named, independent section of C code that performs a specific task and optionally returns a value to the calling program. Now let's look at the parts of this definition:

A function is named: Each function has a unique name. By using that name in another part of the program, you can execute the statements contained in the function. This is known as calling the function. A function can be called from within another function.

A function is independent: A function can perform its task without interference from or interfering with other parts of the program.

A function performs a specific task: This is the easy part of the definition. A task is a discrete job that your program must perform as part of its overall operation, such as sending a line of text to a printer, sorting an array into numerical order, or calculating a cube root.

A function can return a value to the calling program. When your program calls a function, the statements it contains are executed. If you want them to, these statements can pass information back to the calling program.

The Advantages of Structured Programming:

Why is structured programming so great? There are two important reasons:

1. It's easier to write a structured program, because complex programming problems are broken into a number of smaller, simpler tasks. Each task is performed by a function in which code and variables are isolated from the rest of the program. You can progress more quickly by dealing with these relatively simple tasks one at a time.

2. It's easier to debug a structured program. If your program has a bug (something that causes it to work improperly), a structured design makes it easy to isolate the problem to a specific section of code (a specific function).

Q) What is modular programming?

If a program is large, it is subdivided into a number of smaller programs that are called modules or subprograms. If a complex problem is solved using more modules, this approach are known as modular programming

Q) How to write a function? Give it's description?

Syntax:

```
(Return type)  (function name (argument list))
{
Local variable declarations,
Executable statement1;
Executable statement2'
..
Return (expression);
}
```

- ⇨ The first line of every function is the function header, which has three components, each serving a specific function.
- ⇨ The function return type specifies the data type that the function returns to the calling program. The return type can be any of C's data types: char, int, long, float, or double. You can also define a function that doesn't return a value by using a return type of **void**.
- ⇨ You can name a function anything you like, as long as you follow the rules for C variable names.
- ⇨ Many functions use arguments, which are values passed to the function when it's called. A function needs to know what kinds of arguments to expect i.e the data type of each argument. You can pass a function any of C's data types. Argument type information is provided in the function header by the parameter list. For each argument that is passed to the function, the parameter list must contain one entry. This entry specifies the data type and the name of the parameter.

Function Body:

⇨ The function body is enclosed in braces, and it immediately follows the function header. It's here that the real work is done. When a function is called, execution begins at the start of the function body and terminates (returns to the calling program) when a return statement is encountered or when execution reaches the closing brace.

Function Statements:

⇨ There is essentially no limitation on the statements that can be included within a function. The only thing you can't do inside a function is define another function. You can, however, use all other C statements, including loops, if statements, and assignment statements. You can call library functions and other user-defined functions.

Returning a Value:

⇨ To return a value from a function, you use the return keyword, followed by a C expression. When execution reaches a return statement, the expression is evaluated, and execution passes the value back to the calling program.

Q) What are the advantages of functions?

Reusability: write the code once and use many times which decreases the size of the program.

Modularity: dividing problem into small parts (modules) in which each and every module having separate operation. Due to this we can easily develop the programs and we can easily debug the program.

Efficiency: Avoid redundant instructions which decreases size of the program and increases efficiency.

Some more advantages

1. Debugging is easier
2. It is easier to understand the logic involved in the program
3. Testing is easier
4. Recursive call is possible

5. Irrelevant details in the user point of view are hidden in functions
6. Functions are helpful in generalizing the program

Q) Explain types of functions?

Basically we have two types of functions.

1. Pre defined,
2. User defined.

PREDEFINED:

A function which has own definition and comes along with the software is called as predefined functions. Or library functions are called as predefined.

Ex: printf, scanf, get char() , put char() etc.

- ⇨ We can't change the predefine function operation.
- ⇨ We can't modify predefined function.

USER DEFINED:

A function which is defined by user for a particular purpose is called as user defined functions.

- ⇨ User defined functions follow the c language functions pre- defined rules.
- ⇨ Pre defined rules user can't change.
- ⇨ In user defined functions user can implement their own operation.
- ⇨ User defined functions names and the operations can change when user want.

Example:

```
void main()
int abc(int a, int b);
float addition(float a, float b);
```

Q) What is a function and built-in function?

A large program is subdivided into a number of smaller programs or subprograms. Each subprogram specifies one or more actions to be performed for large program. Such subprograms are functions. The function supports only static and extern storage classes. By default, function assumes extern storage class. Functions have global scope. Only register or auto storage class is allowed in the function parameters. Built-in functions that predefined and supplied along with the compiler are known as built-in functions. They are also known as library functions.

Q) What are the Parameters? Explain types of parameters?

In c language parameters are classified into two types. They are

1. Actual parameter
2. Formal parameter

Actual parameter:

⇨ The values which are send/pass from calling function to called function are called actual parameters are actual arguments

Formal parameter:

⇨ Function parameters are called formal parameters. These are local variable which receive values from calling function.

Q) Explain about Local Variables?

⇨ You can declare variables within the body of a function. Variables declared in a function are called local variables.

⇨ The term local means that the variables are private to that particular function and are distinct from other variables of the same name declared elsewhere in the program.

For now, you should learn how to declare local variables.

A local variable is declared like any other variable, using the same variable types and rules f Local variables can also be initialized when they are declared.
You can declare any of C's variable types in a function.
Here is an example of four local variables being declared within a function:

```
int func1(int y)
{
int a, b = 10;
float rate;
double cost = 12.55;
/* function code goes here... */
}
```

The preceding declarations create the local variables a, b, rate, and cost, which can be used by the code in the function. Note that the function parameters are considered to be variable declarations, so the variables, if any, in the function's parameter list also are available.

When you declare and use a variable in a function, it is totally separate and distinct from any other variables that are declared elsewhere in the program. This is true even if the variables have the same name.

Note: scope of the local variable is always with in block only.

Q) Explain about Global Variables?

- ⇨ You can declare variables outside of the body of a function(main). Variables declared outside of the function are called as global variables.
- ⇨ The term global means that the variables are public.

For now, you should learn how to declare global variables.

A global variable is declared like any other variable, using the same variable types and rules of normal variables can also be initialized when they are declared.
You can declare any of C's variable types in a function.

Note: sometimes global variables are called as extern variables.

<u>Some important questions:</u>

Q If global variables can be used anywhere in the program, why not make all variables global?

A: As your programs get bigger, you will begin to declare more and more variables. There are limits on the amount of memory available. Variables declared as global take up memory for the entire time the program is running; however, local variables don't. For the most part, a local variable takes up memory only while the function to which it is local is active. Additionally, global variables are subject to unintentional alteration by other functions. If this occurs, the variables might not contain the values you expect them to when they're used in the functions for which they were created.

Q How does the computer knows the difference between a global variable and a local variable that have the same name?

A: The important thing to know is that when a local variable is declared with the same name as a global variable, the program temporarily ignores the global variable. It continues to ignore the global variable until the local variable goes out of scope.

Q Can I declare a local variable and a global variable that have the same name, as long as they have different variable types?

A: Yes. When you declare a local variable with the same name as a global variable, it is a

Completely different variable. This means that you can make it whatever type you want. You should be careful, however, when declaring global and local variables that have the same name.

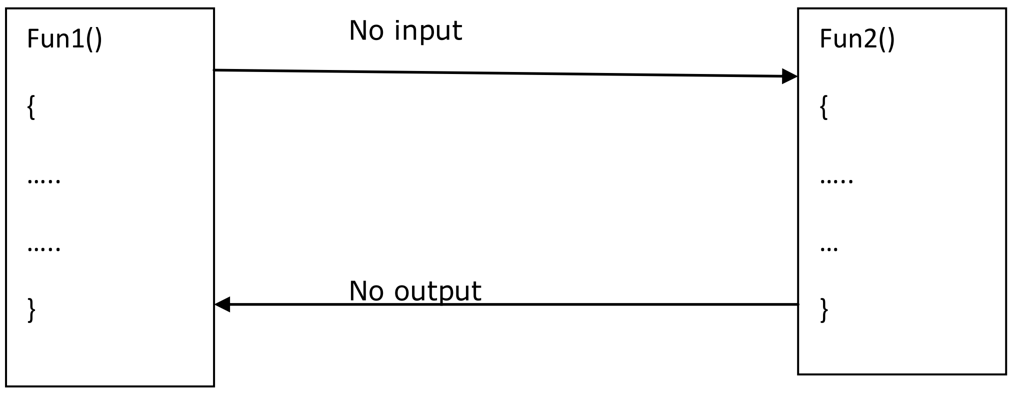
CATEGORY OF FUNCTIONS:

Functions are categories depending on whether arguments are present or not and whether a value is returned or not. In C language functions May belong to one of the following categories

1. Functions with no arguments and no return values.
2. Functions with arguments and no return values.
3. Functions with arguments and returns values.

NO ARGUMENTS AND NO RETURN VALUES:

When a function has no arguments it does not receive, any data from the calling function. Similarly, when it does not return a value the calling function does not receive any data from the called function. It shows in the figure. The dotted lines indicate that there is only a transfer of control but not data..

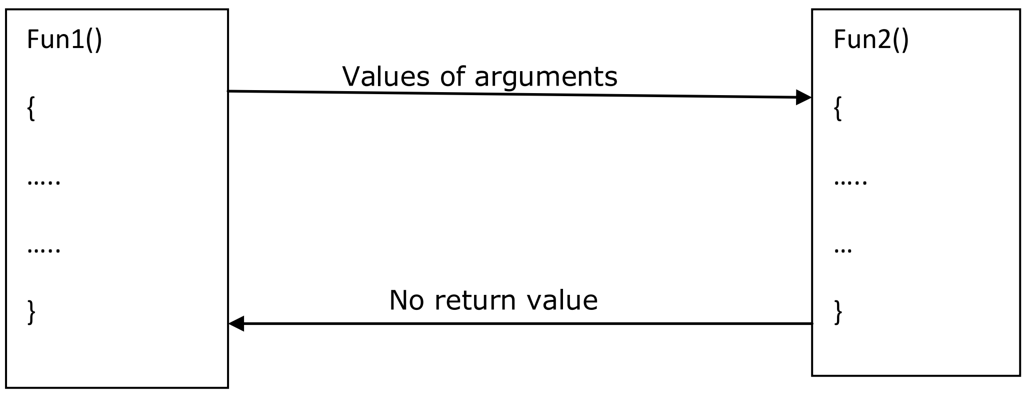


No data communication between functions.

ARGUMENTS BUT NO RETURN VALUES:

When a function has arguments it receive data from the calling function. Similarly when it does not return a value the calling function does not receive any data from the called function.

The nature of data communication between the calling function and the called function with arguments but no return values as shown below.

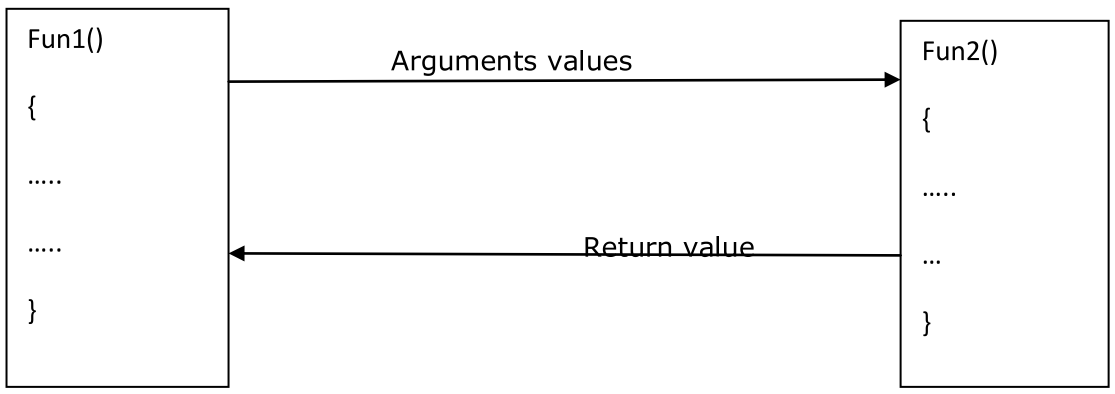


One -way data communication.

ARGUMENTS WITH RETURN VALUES:

When a function have arguments it receive data from the calling function similarly when function having return a value the calling function receive data from the called function. In effect the data transfer between the calling function and called function happened.

Two -way data communication between the called function and calling function as shown below.



two -way data communication

EXAMPLES:

/* no arguments and no return type*/

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

```
#include<conio.h>
void first (void);
void second (void);
void third (void)
{
printf("\n this is in third function");
}
void fourth (void)
{
printf("\n this is in fourth  function");
}
void main()
{
clrscr();
printf("this is in main");
first();
second();
third();
fourth();
printf("\n this is in main again");
getch();
}
void first (void)
{
printf("\n this is in first function");
}
void second (void)
{
printf("\n this is in second function");
}
```

Output:

```
this is in main
 this is in first function
 this is in second function
 this is in third function
 this is in fourth  function
 this is in main again
```

EXAMPLE2:

/*arguments with no return value functions*/

```
#include<stdio.h>
#include<conio.h>
void replicate (int n, int ch)
{
int k;
for (k=1; k<=n; k++)
printf("%c", ch);
}
void main()
{
clrscr();
replicate (10, 'x');
printf("\n hello \n");
replicate (5, '#');
printf("\n bhano daya  \n");
replicate (10, '%');
printf("\n  welcome \n");
replicate (5, '@');
getch();
}
```

Output:

```
xxxxxxxxxx
      hello
     #####
      bhano daya
     %%%%%%%%%%
       welcome
     @@@@@
```

EXAMPLE3:

/*arguments with return value functions*/

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

```
int add (int x, int y)
{
int z;
z=x+y;
return z;
}
int subtr(int p, int q)
{
return (p-q);
}
int mult (int, int);
int div (int, int);
int main()
{
int a,b,c;
clrscr();
printf("enter any two numbers \n");
scanf("%d%d", &a, &b);
c=add(a, b);
printf("\n the addition is %d", c);
printf("\n the subtraction %d", subtr(a, b));
c=mult(a, b);
printf("\n the multiplication %d", mult(a,b));
printf("\n the division %d", div(a,b));
getch();
return(0);
}
int mult (int a, int b)
{
return a*b;
}
int div(int m, int n)
{
int p=m/n;
return p;
}
```

Output:

```
enter any two numbers
    10
    3
     the addition is 13
     the subtraction 7
     the multiplication 30
     the division 3
```

EXAMPLE4:

/* write a program to find given string is palindrome or not*/

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

void palindrome(char*);
void main()
{
char string[40];
clrscr();
printf("Enter a string:");
scanf("%s",string);
palindrome(string);
getch();
}
void palindrome(char *str)
{
int left, right;
left =0;
right = strlen(str);
right = right - 1;
while(left <= right)
{
if(str[left] == str[right])
{
left++;
right--;
}
else
{
printf("\nString is not palindrome.");
break;
}
}
if(left>right)
printf("\nString is palindrome.");
}
```

Output:

```
Enter a string:mom

String is palindrome.
```

Parameter passing techniques:

- ⇨ In c language two types of techniques are available for passing parameters to the functions. they are
  1. Call by value.
  2. Call by reference.
  3.

Call by value:

- ⇨ In this technique the values of actual arguments in the calling function are copied into the formal arguments in the called function.
- ⇨ Here the data transfer done in only one direction i.e. from calling function to called function, so any changes happen in formal arguments does not effect on the actual arguments.
- ⇨ But if any changes will happen in the actual arguments, formal arguments automatically effected.

EXAMPLE:

//write a program to send value by call by calue

```
#include<stdio.h>
#include<conio.h>
void change();
void main()
{
int x,y;
clrscr();
printf("\n enter values of xand y=");
scanf("%d%d",&x,&y);
change(x,y);
printf("\n in main () x=%d, y=%d",x,y);
getch();
}
void change(int a,int b)
{
int k;
k=a;
a=b;
b=k;
printf("\n in  change() a=%d,b=%d",a,b);
}
```

Output:

```
 enter values of xand y=10
20

 in  change() a=20,b=10
 in main () x=10, y=20
```

Call by reference:

- ⇨ In this technique the addresses of actual arguments in the calling function are copied into the formal arguments in the called function.
- ⇨ Any changes happen in formal arguments does effect on the actual arguments.
- ⇨ But if any changes will happen in the actual arguments, formal arguments automatically effected.

EXAMPLE:

/* write a program swaping of two numbers by using call by reference technique*/

```
#include<stdio.h>
#include<conio.h>
void main()
{
int x,y;
clrscr();
printf("\n enter value of x,y:");
scanf("%d%d",&x,&y);
change(&x,&y);
printf("\n in main() x=%d,y=%d",x,y);
getch();
}
change(int *a,int *b)
{
int *k;
*k=*a;
*a=*b;
*b=*k;
printf("\n in change() x=%d,y=%d",*a,*b);
}
```

Output:

```
 enter value of x,y:10 20

 in change() x=20,y=10
 in main() x=20,y=10
```

Recursion:

The term recursion refers to a situation in which a function calls itself either directly or indirectly. Indirect recursion occurs when one function calls another function that then calls the first function. C allows recursive functions, and they can be useful in some situations.
For example, recursion can be used to calculate the factorial of a number. The factorial of a number x is written x! and is calculated as follows:

x! = x * (x-1) * (x-2) * (x-3) * ... * (2) * 1

However, you can also calculate x! like this:

x! = x * (x-1)!

Going one step further, you can calculate (x-1)! using the same procedure:

(x-1)! = (x-1) * (x-2)!

You can continue calculating recursively until you're down to a value of 1, in which case you're finished.

EXAMPLE:
/* write a find factorial of number*/

```
#include <stdio.h>
#include <conio.h>
unsigned int f, x;
unsigned int factorial(unsigned int a);
main()
{
clrscr();
puts("Enter an integer value between 1 and 8: ");
scanf("%d", &x);
if( x > 8 || x < 1)
{
printf("Only values from 1 to 8 are acceptable!");
}
else
{
f = factorial(x);
printf("%u factorial equals %u\n", x, f);
}
return 0;
getch();
}
unsigned int factorial(unsigned int a)
```

```
        {
        if (a == 1)
        return 1;
        else
        {
        a *= factorial(a-1);
        return a;
        }
}
```

Output:

```
Enter an integer value between 1 and 8:
        5
5 factorial equals 120
```

Note: recursion eliminates the looping structures.

Passing array as arguments to functions:

- An array can be passed to a function as on argument in a manner similar to that used to passing variables.
- To pass an array to a called function, it is sufficient to list the names of the array, without any subscripts and the size of the array as argument.
- The called function with array as parameters contains minimum two parameters, the array and the size of the array.

Syntax:

<function-name> (<array-name>[array-size])

- When we pass an array as an argument to a function we need not specify the array size in the arguments declaration part of the function.
- The array-name is followed by empty square brackets is an array.

Things to note:

- ⇨ Remember one main distinction from passing ordinary variables, if a function changes the value of an array elements, the changes will be made to the original array that was passed to the function.

EXAMPLE:

//display elements using arrays and function

```
        #include<stdio.h>
        #include<conio.h>
        void main()
        {
        int a[10],num;
        void getarray(int a[], int);
        void printarray(int a[],int);
        printf("\n size of array not>10");
        scanf("%d",&num);
        getarray(a,num);
        printarray(a,num);
        }
        void getarray(int a[], int n)
        {
        int k;
        for(k=0;k<n;k++)
        {
        printf("\n enter a[%d] element:",k);
        scanf("%d",&a[k]);
        }
        return 0;
        }
        void printarray(int a[], int n)
        {
        int k;
        for(k=0;k<n;k++)
        return;
}
```

Output:

```
         enter size of array not>10=5
```

```
 enter a[0] element:10

 enter a[1] element:20

 enter a[2] element:30

 enter a[3] element:40

enter a[4] element:50
```

CHAPTER11
POINTERS

POINTERS:

Introduction

A pointer is a variable that stores the memory address of other variables. Then, pointer variable will start referring the variable whose address is stored in it.

The pointer is a variable, so it is declared just like any other variable. The only difference is that pointer variables must have the dereferencing operator, *, before its name.

The pointer variable of one data type will only hold the address of variable of same data type.

Generally, the pointer variable are declared as,

```
datatype  *ptr_varname;
```

where "ptr_varname" name of pointer variable of type "datatype".

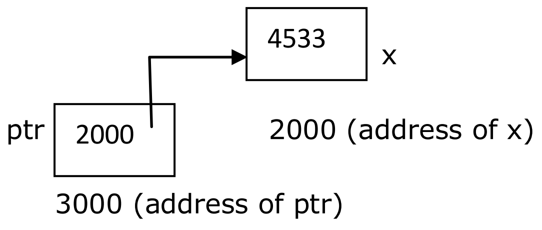
Example,

```
int *x;  /* read as, x is a pointer to int */
```

Example, declaring pointer and storing address,

```
int x, *ptr;

ptr = &x;  /*read as, address is assigned to ptr.*/
```

the &x gives "the address of x", and is stored into pointer variable "ptr". This is as viewed below,



i) The number in the box labeled x is value of x, which is a garbage because, we did not assign any value.

ii) The number in the box labeled ptr is address of memory of x.

Accessing values of variables by using pointers:

The value from memory location of variable is accessed by using its pointer variable.

This is explained by following statements,

```
int  a, *ptr, b;  /* The ptr is a pointer, a, b are variables */

a = 50;     /* a is assigned with value 50 */
ptr = &a;  /* the pointer ptr  points the memory location of a */
```

```
/*The *ptr  retrieves value from memory location whose address is stored in it*/
b = *ptr;   /* b is stored with *ptr value, which is value of a  */

/* value of a is modified, now it would be 105.*/
*ptr = *ptr + 55;
```

This is also illustrated by small program given below,

```
#include <stdio.h>

void main()
{
  int x = 12;
  int *ptr = &x;

  printf("Address of x: %u\n", ptr);
  printf("Address of x: %u\n", &x);
  printf("Address of ptr: %u\n", &ptr);
  printf("Value of x: %d\n", *ptr);

  return 0;
 }
```

Output:

```
Address          of          x:          1000
Address          of          x:          1000
Address          of          ptr:        2000
Value of x: 12
```

The first, printf("Address of x: %u\n", ptr); in program,

prints the address of memory location of x, which is stored in the pointer variable, ptr.

The second, printf("Address of x: %u\n", &x);

prints the address of memory location of x.

The third, printf("Address of ptr: %u\n", &ptr);

prints the address of memory location variable, ptr.

The finally, printf("Value of x: %d\n", *ptr);

uses the dereference operator to extract the value pointed to by ptr, like this: *ptr to print values stored in variable, x.

Q) What are Pointers used for?

Pointers are used in situations when passing actual values is difficult or not desired. Some of the situations where pointers can be used are –

1. To return more than one value from a function.
2. To pass arrays and strings more conveniently from one function to another
3. To manipulate arrays easily by moving pointers to them instead of moving the arrays itself
4. To allocate memory and access it (Direct Memory Allocation).
5. To create complex data structures, such as linked lists

Q) What is pointer?

- ⇨ Pointer is a derived data type.
- ⇨ A variable of type pointer is called pointer variable.
- ⇨ A pointer variable holds an address of memory location.

Q) What is address?

- ⇨ The address is integer value given to each location or byte within memory.
- ⇨ These addresses are generated by operating system.
- ⇨ Addressing method is depends on compiler , if compiler is 16 bit , it uses 16 bit addressing method and if compiler is 32 bit it uses 32 bit addressing methods.

Q) What are the advantages of pointer?

1. Dynamic memory allocation
2. Avoid wastage of memory
3. Pointers increases efficiency of program
4. Complex data types
5. Dynamic data structures

Q) What are the disadvantages of pointer?

1. Pointers are not secured.
2. Memory leak.

Q) What is Reference operator (&)?

The address that locates a variable within memory is what we call a reference to that variable. This reference to a variable can be obtained by preceding the identifier of a variable with an ampersand sign (&), known as reference operator, and which can be literally translated as "address of".

Q) What is Dereference operator (*)?

Using a pointer we can directly access the value stored in the variable which it points to. To do this, we simply have to precede the pointer's identifier with an asterisk (*), which acts as dereference operator and that can be literally translated to "value pointed by".

Q) What is a pointer value and address?

A pointer value is a data object that refers to a memory location. Each memory locaion is numbered in the memory. The number attached to a memory location is called the address of the location.

Pointers for Inter Function Communication:

The pointer are used for inter function communication, which means, communication between called and caller function is can established. Base on the this,

we call function with,

i) call by value
ii) call by reference or address.

These are explained in the following topics.

Functions with Call-by-Value:

The function with call by value is named, by looking the way the arguments are passed. In this, the values of actual parameters are copied into formal parameters. The changes in values of these parameters will not reflect into actual parameters.

This is illustrated with the program given below,

```
#include<stdio.h>

#include<conio.h>

void swap( int, int );

void main()

{

int a, b;

printf("\nEnter value of a, b: ");

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

printf("\n Before swap function called, a= %d, b= %d", a, b);

swap(a, b);

printf("\n After swap function called, a= %d, b= %d", a, b);

}

/* Try to Exchange a, b value */

void swap(int m, int n)
```

```
{
    int  temp;
    temp = m;
    m = n;
    n = temp;
}
```

Output:

```
Enter value of a, b: 12  25
Before swap function called, a= 12,  b= 25
After swap function called, a= 12, b= 25
```

The output of above program is observed, that no change in the values of a, b.

Functions with Call-by-Reference or Address:

The function with call by reference or address is named, by looking the way the arguments are passed. In this, the addresses of actual parameters are copied into formal parameters. So, the formal parameters have to be pointers to data type of actual parameters. Then changes in values of formal parameters will reflect into actual parameters.

This is illustrated with the program given below,

```
#include<stdio.h>
#include<conio.h>
void swap( int* , int* );
void main()
{
int a, b;
printf("\nEnter value of a, b: ");
scanf("%d%d", &a, &b);
printf("\n Before swap function called, a= %d, b= %d", a, b);
swap( &a,  &b);
printf("\n After swap function called, a= %d, b= %d", a, b);
}
```

```
    /* Try to Exchange a, b value */
    void swap(int *m,  int *n)
    {
        int  temp;
        temp = *m;
        *m = *n;
        *n = temp;
    }
```

Output:

```
Enter value of a, b: 12  25
Before swap function called, a= 12,  b= 25
After swap function called, a= 25, b= 12
```

The output of above program is observed that, changes take place in the values of a, b.

Pointers to Pointers:

A pointer variable can also be pointed by another pointer variable. This, means, a pointer variable stores address of another pointer variable.

In general it is declared as,

```
        datatype  **ptr;
```

this is read as, "ptr is pointer to pointer to datatype."

Example,

```
                int  **x;   /*x is a pointer to pointer to int*/
```

Example, this time we take float data type.

```
float  *p, **ptp, n, m;
        n = 45.25;
        p = &n;
        ptp = &p
        m = **ptp;
```

another example,

```
                        int ***Ptr      /*ptr is pointer to pointer to pointer to int */
```

```
                float ****pf   /*pf is pointer to pointer to pointer to pointer to float
*/
```

Q) How many levels of pointers can you have?

The answer depends on what you mean by "levels of pointers." If you mean "How many levels of indirection can you have in a single declaration?" the answer is "At least 12."

```
int i = 0;
int *ip01 = & i;
int **ip02 = & ip01;
int ***ip03 = & ip02;
int ****ip04 = & ip03;
int *****ip05 = & ip04;
int ******ip06 = & ip05;
int *******ip07 = & ip06;
int ********ip08 = & ip07;
int *********ip09 = & ip08;
int **********ip10 = & ip09;
int ***********ip11 = & ip10;
int ************ip12 = & ip11;
************ip12 = 1; /* i = 1 */
```

The ANSI C standard says all compilers must handle at least 12 levels. Your compiler might support more.

Pointer to void:

In C, the void is a keyword. This is used to create variables of functions of type void. We might have observed that, void main(), void swap() etc

The void main() means, the function main is defined to returns no data or nothing.

We can also declare pointer to void. Then, such pointer is generally called as generic pointer, which means, we can convert or type cast into other data types.

Its declaration is similar to pointers of other data type. This is,

```
    void *vptr;    /*the vptr is a pointer to void.*/
```

Example,

```
        int *ptr;

        float *pf;

        void  *vptr;

        ptr = (int *)vptr;    /*( int* ) is type casting to data type of ptr  */

        ………………

        ………………
```

```
pf = (float*)vptr;    /*( float* ) is type casting to data type of pf  */
```

In C, some predefined functions return data of pointer to void type, which data may used by program by converting or type casting to required data type.

Q) What is a dangling pointer?

A dangling pointer arises when you use the address of an object after its lifetime is over. This may occur in situations like returning addresses of the automatic variables from a function or using the address of the memory block after it is freed.

We can also define like as it is a pointer variable which holds an address of unreserved memory location is called dangling pointer.

To avoid dangling pointer it has to be initiated with null.

This pointer variable leads to logical error.

Q) What is null pointer?

A pointer variable which holds null address is called null pointer. This pointer variable cannot point to any memory location.

Pointer Arithmetic:

As pointers are variables, there should be operators those can be used with pointers.

If a pointer variable is made to point a certain variable then, we can either reassign it another variable's address, or performing arithmetic operations to point other memory location. The operator permitted to use with pointers are,

If we have,

```
int *ptr, *py, x, y;

ptr = &x;

py = &y;
```

then,

Addition:

```
ptr = ptr +1, ptr +=1, ++ptr,  ptr++  are same.
```

These are calculated internally by computer as, if ptr contains address of x which 1000 then,

```
ptr      = (1000+ sizeof(int))

         =(1000+2) = (1002);
```

Which means, moving the pointer forward by 2 bytes (i.e. the size of its data type).

ii) ptr = ptr + 2;

ptr = (1000 + 2*sizeof(int))

=(1004)

iii) ptr + py is not allowed. Adding of two pointer variable is not allowed.

In general,

ptr = (ptr + n * sizeof (datatype)), where n =0, 1, 2, 3… and datatype is char int, float, double, long int, etc.

Subtraction:

Similarly, ptr = ptr -1 , ptr--, --ptr and ptr -=1 will move the pointer "back" 2 bytes, assuming ptr is an int pointer. The calculation is as,

i) ptr = (1000 - sizeof(int)) is allowed

=(1000 - 2) = (998)

ii) ptr – py and py – ptr are allowed. If ptr is holding 1000, and py is holding 1024, then, these expression are internally calculated as follows, so, it can

be used to know how number of values stored,

(py – ptr)/sizeof(int) = (1024 – 1000)/2 = 24/2 = 12 locations or values.

In general,

ptr = (ptr - n * sizeof (datatype)), where n =0, 1, 2, 3… and datatype is char int, float, double, long int, etc.

Cannot be used:

The operators multiplication and division cannot be used, because those operation on pointer are meaningless.

Pointers to Arrays

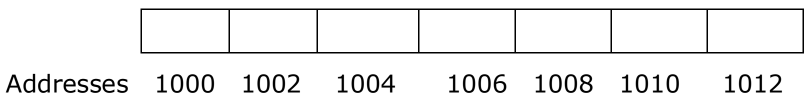
The pointer variables are also used to point arrays. If a pointer is made to point an array of some data type then, such pointer can be used to access cells of that array and values stored in those cells.

For example, with the statement,

int x[7];

an array of 7 integers is created, and it may be viewed as,

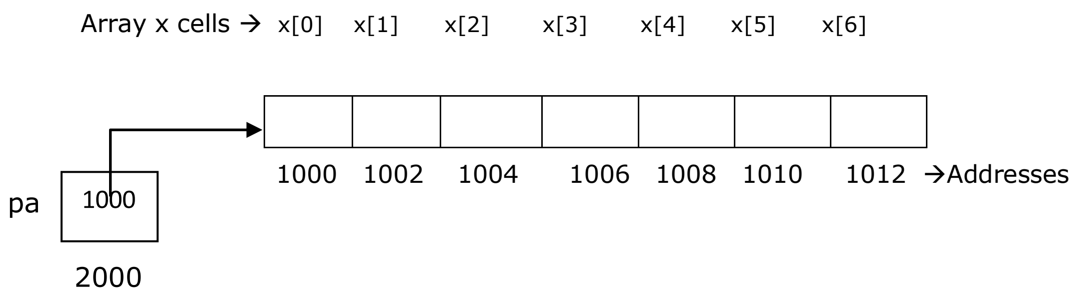
Array x cells: x[0] x[1] x[2] x[3] x[4] x[5] x[6]



Now, &x[0] or simply x, indicates 1000, which is the base address of array x[] is stored into pointer variable, then it points the array x[].

```
int *pa;

pa = x;
```

this would be as shown below,


Array x cells → x[0] x[1] x[2] x[3] x[4] x[5] x[6]
1000 1002 1004 1006 1008 1010 1012 →Addresses
pa
1000
2000

Now, pa can be is used for accessing array x[]. This can be understood as

x[0] = *(p+0) and &x[0] = (pa+0)

x[1] = *(p+1) and &x[1] = (pa+1)

x[2] = *(p+2) and &x[2] = (pa+2) and in general,

x[i] = *(p+i) and &x[i] = (pa+i).

In general, ith cell address is caluculated as, if i=5, address of x[5],

= (baseaddress + (i * sizeof(datatype))

= (1000 + 5 * sizeof(int))

= (1000 + 5 * 2)

= (1010).

The address calculations shown above are also applicable for other data type like char, float, double, etc. In this case, sizeof(data type) will be equivalent those sizes.

Example:

// A Program to illustrate the concept of pointers to single dimensional arrays.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int x[5] = {10, 11, 12, 13, 14};
int *pa, i;

pa = x;

printf(" Value printed using pa pointer .\n");
for( i= 0; i<5; i++)
{
     printf("%3d", *(pa+i));
}

printf(" Value printed using x[].\n");
for( i= 0; i<5; i++)
{
     printf("%3d", x[i] );
}
return 0;
}
```

Output:

```
Value printed using pa pointer

10 11 12 13 14

Value printed using x[] pointer

10 11 12 13 14
```

Example:

//A Program to illustrate the concept of pointers to single dimensional arrays.

```
#include <stdio.h>
#include <conio.h>
int my_array[] = {1,23,17,4,-5,100};
int *ptr;

int main(void)
{
    int i;
    ptr = &my_array[0];     /* point our pointer to the first
                                    element of the array */
    clrscr();
    for (i = 0; i < 6; i++)
    {
      printf("\nmy_array[%d] = %d",i,my_array[i]);
      printf("\nptr + %d = %d",i, *(ptr + i));
    }
    return 0;
}
```

Output:

```
my_array[0] = 1

ptr + 0 = 1

my_array[1] = 23

ptr + 1 = 23

my_array[2] = 17

ptr + 2 = 17

my_array[3] = 4

ptr + 3 = 4

my_array[4] = -5

ptr + 4 = -5

my_array[5] = 100

ptr + 5 = 100
```

Pointers to 2-Dimensional arrays:

The pointers are also used to point the multi-dimensional arrays.

For example,

```
#define ROWS 4
#define COLS 3

int a[ROWS][COLS];

int (*pa)[COLS];
```

where, pa is a pointer to group of single dimensional arrays each of size COLS. This pointer can be used to access cells indexed i, j as given below,

```
pa = a;  /* base address of a[][] is copied into pa.*/

*(*(pa + i) + j)  is equivalent to a[ i ][ j ].
```

This can be clarified from a program.

Example:

//Program to illustrate concept of pointer to 2-Dimensional arrays.

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

void main(void)
{
   int i, j;
int a[3][4]={ {4, 6,-19, 25}, {7, 5, 30, 20}, {23,8, 12, 16} };
int (*pa)[4];
pa = a;
clrscr();
printf("\n Array values are:\n");
for(i=0; i<4; i++)
{
for(j=0; j<4; j++)
printf("%3d", *(*(pa+i)+j));
printf("\n");
}
getch();
}
```

Output:

```
Array values are:

  4   6 -19  25

  7   5  30  20

  2   8  12  16
```

Explanation, We can access individual elements of the array a[][] using

a[row][col]

where row, col are indices of row and column of 2-Dimensional array respectively.

We can also access values of two-dimensional array as,

((a + row) + col)

where a is the name of the array, which also gives base address of array a[][].
To understand more fully what is going on, assume pa is a pointer to (a + row), Then,

(pa + 0) points the row indexed 0,

(pa + 1) points the row indexed 1, and

(Pa + i) points the row indexed i.

Then, *(pa + i) means the ith row. The (*(pa +row)+ col) will be pointing element at a[row][col]. Then, to retrieve value *(*(pa + row) + col) has to be used.

Example:

//Passing single dimensional arrays as arguments to functions

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

int add_array(int*, int); /* declare function */
void main()
{
int array[20];
int i, sum,n;

clrscr();
printf("Enter size(less than 20) of array: ");
scanf("%d",&n);

printf("Enter %d array values:\n", n);
for(i=0 ; i<n ; i++)
scanf("%d", &array[i]);

sum = add_array(array, n);
printf("\nSum of array numbers= %d\n", sum);
getch();
}

int add_array(int *pa, int m)
{
int sum=0;
int i;

for(i=0 ; i<m ; i++)
sum += *(pa+i);

return sum;
}
```

Output:

```
Enter size(less than 20) of array: 5
Enter 5 array values:
10 11 12 13 14

Sum of array numbers= 60
```

Exapmle: Passing 2-Dimensional arrays as arguments to function.

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

void display_array(int (*)[], int); /* declare function */

void main()
{
int array[3][4] = {{0,1,2,3}, {4,5,6,7}, {8,9,10,11}};
clrscr();

display_array(array, 3);
getch();
}

/*Definitionof display_array function */
void display_array(int (*pa)[4], int rows)
{
int i, j;

for(i=0; i<rows; i++)
{
for(j=0 ; j<4 ; j++)
{
printf("%3d ", *(*(pa+i)+j));
}
printf("\n");
}
}
```

Output:

```
 Array Values are:
  0   1   2   3
  4   5   6   7
  8   9  10  11
```

Arrays of Pointers:

A pointer is a variable that stores the memory address of another variable. So it's possible to have an array of pointers. In other words, an array of memory addresses:

These pointers may pointing individual variables or arrays of some data type.

In general it is declared as,

```
            datatype  *array_ptrs[n];
```

In the above declaration, the array_ptrs[] is a single dimensional array of n pointers to datatype.

Example,

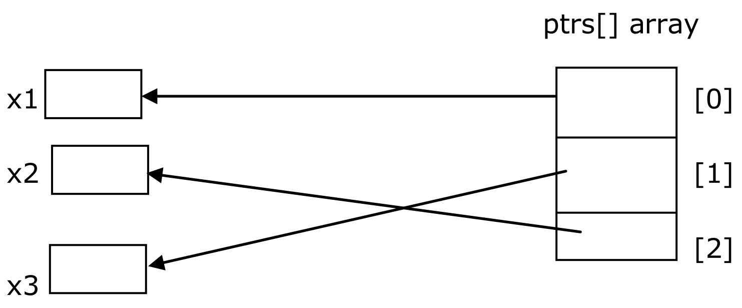
```
                         int  *ptrs[5];
```

where ptrs[] is a array of 5 pointers to int data type.

Example,

```
                         float  x1, x2, x3;
                         float *ptrs[3];
```

```
prts[0] = &x1;
prts[1] = &x3;
prts[2] = &x2;
```



Example:

A Program to illustrate the concept of array of pointers.

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

void main()
{
int *ptrs[3], *p1, *p2;
int x1,x2,x3;

x1=10;
x2=20;
x3=30;

p1 = &x1;
p2 = &x3;

ptrs[0] = p1;
ptrs[1] = p2;
ptrs[2] = &x2;
clrscr();

printf("\nValues of x1,x2, x3 are:\n");
printf("\nx1 = %d", *p1);
printf("\nx2 = %d", *ptrs[2]);
printf("\nx3 = %d", *ptrs[1]);
getch();
}
```

Output:

```
Values of x1,x2, x3 are:
x1 = 10
x2 = 20
x3 = 30
```

Example:

//Program on concept of arrays of pointers

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

void main()
{
  int array[3] = {1,2,3};

  int *ptr1 = &array[0];
  int *ptr2 = &array[1];

  int *ptrs[3];

  ptrs[0] = ptr1;
  ptrs[1] = ptr2;
  ptrs[2] = &array[2];

  clrscr();

  printf("The value of array[0] at %u is %d\n", ptrs[0], *ptrs[0]);
  printf("The value of array[1] at %u is %d\n", ptrs[1], *ptrs[1]);
  printf("The value of array[2] at %u is %d\n", ptrs[2], *ptrs[2]);
 getch();
}
```

Output:

```
The value of array[0] at 65520 is 1

The value of array[1] at 65522 is 2

The value of array[2] at 65524 is 3
```

The above example will be modified to work also for floats and doubles.

Pointers to Functions:

The pointer variable is also used to point functions. Such pointer can be used to call or invoke that function. Extending this concept, we can also pass function name as an argument to another function. These, can be understood with explanation given below, In general a pointer to function is declared as,

datatype (*ptr_function)(datatype, datatype, ….datatype);

where "datatype" is the data type allowed in C,
"and ptr_function" is pointer name which points to function.

Example,

```
int  (*ptrfun)(int, float);
```

defines a pointer name "ptrfun" which points the function with two parameters, and return type int, where first parameter is of type int, and latter is of type float.

Similarly, Example,

```
float * (*pfun)(float [], int *);
```

defines a pointer "pfun" , which points the function with two parameters, and return type float*, where first parameter is array of type float and later is point of type int.

Example,
int add(int, int);

```
int (*ptr)(int, int);

ptr = add;      /*ptr point add() function*/
```

The ptr is a point which holds address of a function. The name of function itself is a address, so, this assignments makes to ptr to point the function add().

Note: int (*ptrfun)(int, float) is different from int *ptrfun(int, float). So, parentheses around the pointer name to function are mandatory.

Example:

//Program to demonstrate how to use pointer to function.

```
#include<stdio.h>

int add(int, int); /*prototype of add()*/
void main()
{
int x, y, sum;
int (*ptr)(int, int);

/* assigning add() to pointer ptr.*/
ptr = add;
clrscr();

printf("Enter x and y values:");
scanf("%d%d", &x, &y);

/* invoking add() using its pointer ptr */
sum = (*ptr)(x, y);

printf("%d + %d = %d\n", x, y, sum);
getch();
}

/* definition of add() function */
int add(int x, int y)
{
return(x+y);
}
```

Output:

```
Enter x and y values:15 48
15 + 48 = 63
```

Just remember to enclose the function pointer with brackets when declaring it. Notice that I only included the arguments' types in the parameters in the pointer declaration. I think adding variable names in there are optional, so this would still compile successfully,

```
int (*ptr)(int x, int y);
```

or

```
int (*ptr)(int a, int b);
```

Also note that, the type of the pointer has to match that of the functions return type it pointing to.

Example: Program to illustrate concept of pointer to function.

```
#include<stdio.h>

float average(float [], int*);
void main()
{
float avg, marks[20];
int n, i;
float (*pf)(float [], int*);

/* assigning average() to pointer pf.*/
pf = average;

clrscr();
printf("\nEnter size of array:");
scanf("%d", &n);

printf("\nEnter %d float values into array:\n", n);
for(i=0; i<n;i++)
scanf("%f", &marks[i]);

/* invoking average() using its pointer pf */

avg = (*pf)(marks, &n);

printf("Average of marks=%.2f",avg);
getch();
}

/* definition of add() function */
float average(float *m, int *size)

{

int i;
float sum=0;
for(i=0; i < (*size); i++)
sum += *(m + i);

return(sum/(*size));


}
```

Output:

```
Enter size of array:5

Enter 5 float values into array:

55.25

65.5

79.0

45.95

45.25


Average of marks=58.19
```

CHAPTER12
STORAGE CLASSES

STORAGE CLASSES:

INTRODUCTION:

⇒ All variables have a data type and they also have a “storage class”.

⇒ We have written several programs in ‘c’, but we have not mentioned storage classes yet, because storage classes have default.

⇒ If we do not specify the storage class of a variable in its declaration, the compiler will assume a storage class depending on the context in which the variable is used.

⇒ Thus, variable have certain default storage classes.

⇒ Basically two kinds of locations in a computer where a variable value may be kept. They are “MEMORY AND CPU REGISTER”

A VARIABLE’S STORAGE CLASS TELLS US:-

⇒ Where the variable would be stored.

⇒ What will be the initial value of the variable, if the initial value is not specifically assigned, (i.e., the default initial value)

⇒ What is the scope of the variable, i.e., in which functions the value of the variable would be available.

⇒ What is the life of the variable i.e., how long would the variable exist.

TYPES OF STORAGE CLASSES:-

There are four storage classes in c, which is supported with respect to ‘os’.

1. Auto storage class
2. Static storage class
3. External storage class

4. Register storage class

AUTO STORAGE CLASS:

The features of a variable defined to have an automatic storage class are as under:

Storage	- Stack of memory
Default value	- An unpredictable value, which is often called a garbage value. It refers to Base address of the variable.
Scope	- local to the block in which the variable defined or within the function.
Life	- Till the control remains within the blocks in which the variable is defined.

Note: the keyword for this storage class is auto and not automatic.

STATIC STORAGE CLASS:-

The features of a variable defined to have an automatic storage class are as under:

Storage	- PMA of memory
Default value	- zero
Scope	- local to the block in which the variable defined or with in the function
Life	- value of the variable persists between different function calls.

Note: static variable are initialized only once at the declaration.

EXTERNAL STORAGE CLASS:

The features of a variable defined to have an automatic storage class are as under:

Storage	- PMA of memory
Default value	- zero
Scope	- as long as the program's execution
Life	- doesn't come to an end.

Note:

1. External variables are declared outside all functions.
2. The keyword for this storage class is 'extern'.

Register storage class:

Storage	- CPU registers
Default value	- garbage value

Scope	- local to the block in which the variable is defined
Life	- Till the control remains within the block in which the variable is defined.

Note:

1. A value stored in a cpu register can always be accessed faster than the one which is stored in Memory.

2. A good example of frequently used variables is loop counters.

3. We cannot use register storage class for all types of variables. Because the CPU registers in a Microcomputer are usually 16 bit registers and therefore cannot hold a **float** value or a **double** Value, which require 4 and 8 bytes respectively for storing a value.

Some important questions:

Q) What Is mean by Scope of the variable?

The *scope* of a variable refers to the extent to which different parts of a program have access to the variable. In other words, where the variable is *visible*. When referring to C variables, the terms **accessibility** and **visibility** are used interchangeably. When speaking about scope, the term *variable* refers to all C data types: simple variables, arrays, structures, pointers, and so forth. It also refers to symbolic constants defined with the const keyword. Scope also affects a variable's *lifetime:* how long the variable persists in memory, or when the variable's storage is allocated and de allocated.

Q) Compare all c's storage classes with its characteristics?

Storage Lifetime	Where It's Defined	Scope	Class	Keyword
Automatic	None1	Temporary	In a function	Local
Static	static	Temporary	In a function	Local
Register	register	Temporary	In a function	Local
External	None2	Permanent	Outside a function	Global
External	static	Permanent	Outside a function	Global

1. The auto keyword is optional.

2. The extern keyword is used in functions to declare a static external variable that is defined elsewhere.

1. What are advantages and disadvantages of external storage class?

Advantages of external storage class:

1) Persistent storage of a variable retains the latest value

2) The value is globally available

Disadvantages of external storage class:

1) The storage for an external variable exists even when the variable is not needed

2)The side effect may produce surprising output

3)Modification of the program is difficult

4)Generality of a program is affected

2. Differentiate between an internal static and external static variable?

An internal static variable is declared inside a block with static storage class whereas an external static variable is declared outside all the blocks in a file. An internal static variable has persistent storage, block scope and no linkage. An external static variable has permanent storage, file scope and internal linkage.

3. What are the advantages of auto variables?

1)The same auto variable name can be used in different blocks

2)There is no side effect by changing the values in the blocks

3)The memory is economically used

4)Auto variables have inherent protection because of local scope.

4. What is storage class and what are storage variable?

A storage class is an attribute that changes the behavior of a variable. It controls the lifetime, scope and linkage.

There are five types of storage classes

1) auto

2) static

3) extern

4) register

5) typedef

5. What is a static function?

A static function is a function whose scope is limited to the current source file. Scope refers to the visibility of a function or variable. If the function or variable is visible outside of the current source file, it is said to have global, or external, scope. If the function or variable is not visible outside of the current source file, it is said to have local, or static, scope.

EXAMPLES:

1. /* w a p to implement auto keyword */

```
#include<stdio.h>

#include<conio.h>

void main()

{

clrscr();

increment();

increment();

increment();

}

increment()

{

auto int i=1;

printf("%d\n",i);

i=i+1;

}
```

2. /* w a p to implement static keyword */

```
#include<stdio.h>

#include<conio.h>

void main()

{

clrscr();

increment();

increment();

increment();

}
```

```
increment()
{
Static  int i=1;
printf("%d\n",i);i=i+1;
}
```

3. /* Write a program to display the Fibonacci Series upto the given number

<u>Fibonacci Series:</u> 0, 1, 1, 2, 3, 5, 8, 13, 21,35,55…

In this series every element is the sum of its previous two numbers   */

```
#include<stdio.h>
#include<conio.h>
void main()
{
int count, n;
long int fibonacci(int count);
printf("how many fibonacci number :");
scanf("%d",&n);
for(count=1;count<=n;count++)
printf("%ld\n",fibonacci(count));
getch();
}
long int  fibonacci(int count)
{
static long  int f1=1,f2=1;
long int f;
f=(count<3)?1:f1+f2;
f2=f1;
f1=f;
return f;
}
```

4. /* write a program to use register keyword*/

```
    # include <stdio.h>

    # include <conio.h>

     main()
{
    register int k ;

    clrscr() ;

    printf("%d \n", k );
    for(k=1; k<=100; k++)
    printf("%d  ", k) ;
    getch() ;
}
```

5. /* w a p to implement extern keyword*/

```
    # include <stdio.h>

    # include <conio.h>


    int k ;
    main()
{
    clrscr() ;

    printf("\n %d", k) ;

    k = 5 ;
    disp1() ;

    disp2() ;

    printf("\n %d", k) ;

    getch() ;

}
    disp1()
{
    printf("\n %d", k );
    k += 3 ;
}


    disp2()
{
    int  k = 20 ;
    printf("\n %d", k );
    disp3() ;
}


    disp3()
    {
    printf("\n %d", k );
}
```

CHAPTER13 STRUCTURES

STRUCTURES:

Q) What is the need of structures?

Ordinary variable can hold one piece of information and array can hold a number of pieces of information of the same data type. These two data types can handle a great variety of situations. However if we want to represent a collection of data items of different types using a single name then we cannot use an array and ordinary variable.

Fortunately C supports a constructed data type known as a "structures" which is a method for packing data of different types. Structures help to organize complex data in a more meaningful way.

Q) What is a structure? Explain it's syntax?

A structure is a collection of one or more variables grouped under a single name for easy manipulation. The variables in a structure, unlike those in an array, can be of different variable types. A structure can contain any of C's data types, including arrays and other structures. Each variable within a structure is called a member of the structure.

Syntax:

```
struct  structure_ name
{
Data_type member1;
Data_type member2;
.
.
data _type  membern;
};
```

Note:

The struct keyword is used to declare structures; the keyword struct identifies the beginning of a structure definition. It's followed by a tag that is the name given to the structure.

Defining and Declaring Structures:

If you're writing a graphics program, your code needs to deal with the coordinates of points on the screen. Screen coordinates are written as an x value, giving the horizontal position, and a y value, giving the vertical position. You can define a structure named coord that contains both the x and y values of a screen location as follows:

```
struct coord {

int x;
float y;

};
```

The struct keyword, which identifies the beginning of a structure definition, must be followed immediately by the structure name, or tag (which follows the same rules as other C variable names). Within the braces following the structure name is a list of the structure's member variables. You must give a variable type and name for each member. The preceding statements define a structure type named coord that contains one integer variables x and one float variable y. They do not, however, actually create any instances of the structure coord. In other words, they don't declare (set aside storage for) any structures. There are two ways to declare structures. One is to follow the structure definition with a list of one or more variable names, as is done here.

```
struct coord {
int x;
float y;
} first, second;
```

These statements define the structure type coord and declare two structures, first and second, of type coord. first and second are each instances of type coord; first contains one integer member named x and other floating member y, and so does second. This method of declaring structures combines the declaration with the definition. The second method is to declare structure variables at a different location in your source code from the definition. The following statements also declare two instances of type coord

```
struct coord {
int x;
float y;
};

/* Additional code may go here */

struct coord first, second;
```

Accessing Structure Members:

Individual structure members can be used like other variables of the same type. Structure members are accessed using the structure member operator (.), also called the dot operator, between the structure name and the member name. Thus, to have the structure named first refer to a screen location that has

coordinates x=50, y=100, you could write

```
first.x = 50;
first.y = 100;
```

To display the screen locations stored in the structure second, you could write

```
printf("%d%f", second.x, second.y);
```

At this point, you might be wondering what the advantage is of using structures rather than individual variables. One major advantage is that you can copy information between structures of the same type with a simple equation statement. Continuing with the preceding example, the statement

```
first = second;
```

is equivalent to this statement:

```
first.x = second.x;

first.y = second.y;
```

When your program uses complex structures with many members, this notation can be a great time saver. Other advantages of structures will become apparent as you learn some advanced techniques. In general, you'll find structures to be useful whenever information of different variable types needs to be treated as a group. For example, in a mailing list database, each entry could be a structure, and each piece of information (name, address, city, and so on) could be a structure member.

Example 1

/* declare a structure template called SSN */

```
struct student {

int rollno;
char sname[30];
int sum;
char grade;
float avg;
}

/* Use the structure template */

struct student s1;
```

Example 2

/* declare a structure and instance together */

```
struct date {

char month[2];
char day[2];
char year[4];

} current_date;
```

Example 3

/* Declare and initialize a structure */

```
struct time {

int hours;
int minutes;
int seconds;
} time_of_birth = { 8, 45, 0 };
```

Program:

/* wap to print the student details using structure */

```
#include<stdio.h>
#include<conio.h>
struct student
{
int rno,m1, m2, m3;
float total, avg;
char name[50];
};
main()
{
struct student b;
clrscr();
printf("enter name=");
scanf("%s", &b.name);
printf("enter rno, m1, m2, m3=");
scanf("%d%d%d%d", &b.rno, &b.m1, &b.m2, &b.m3);
b.total=b.m1+b.m2+b.m3;
b.avg=b.total/3;
printf("\n*** STUDENT DETAILS***\n");
printf("\n name is %s", b.name);
printf("\n rno  \t %d", b.rno);
printf("\n sub1 \t %d", b.m1);
printf("\n sub2 \t %d", b.m2);
printf("\n sub3 \t %d", b.m3);
printf("\n total\t %f", b.total);
printf("\n avg  \t %f", b.avg );
getch();
}
```

Output:

```
enter name=Praveen

enter rno, m1, m2, m3=100 98 85 94

*** STUDENT DETAILS***

 name is Praveen

 rno     100

 sub1    98

 sub2    85

 sub3    94

 total   277.000000

 avg     92.333336
```

Arrays of Structures:

As we know array is a collection of similar data types. In the same way we can also define array of structures. In such type of array every element is of structure type. Array of structure can be declared as follows.

```
struct cricket

{
```

```
int balls;
float srate;
char name[25];

}match[10];
```

Example:

/* Demonstrates using arrays of structures. */

```
#include<stdio.h>
#include<conio.h>
struct entry {
char fname[20];
char lname[20];
char phone[10];
};
struct entry list[4];
int i;
void main()
{
clrscr();
for (i = 0; i < 4; i++)
{
printf("\nEnter first name: ");
scanf("%s", list[i].fname);
printf("Enter last name: ");
scanf("%s", list[i].lname);
printf("Enter phone in 123-4567 format: ");
scanf("%s", list[i].phone);
}
printf("\n\n");
for (i = 0; i < 4; i++)
{
printf("Name: %s %s", list[i].fname, list[i].lname);
printf("\tPhone: %s\n", list[i].phone);
}
return 0;
}
```

Output:

```
Enter first name: praveen
Enter last name: kumar
Enter phone in 123-4567 format: 040-1234

Enter first name: ravi
Enter last name: kiran
Enter phone in 123-4567 format: 040-1235

Enter first name: manoj
Enter last name: kumar
Enter phone in 123-4567 format: 040-1236

Enter first name: laxmi
Enter last name: rani
Enter phone in 123-4567 format: 040-1237

Name: praveen kumar     Phone: 040-1234
Name: ravi kiran        Phone: 040-1235
Name: manoj kumar       Phone: 040-1236
Name: laxmi rani        Phone: 040-1237
```

Pointers to structure:

We know that pointer is a variable that holds the address of another data variable. The variable may be of any data type i.e. int, float or double. In the same way we can also define pointer to structure. Here, starting address of the member variable can be accessed. Thus, such pointers are called structure pointers.

Example:

```
struct book
{
char name[25];
char author[25];
int pages;
};

struct book *ptr;
```

in the above example *ptr is pointer to structure book. The syntax for using pointer with member is as given below.

```
ptr->name
ptr->author
ptr->pages
```

Example:

/* A program to display book details using pointers and structures*/

```
#include<stdio.h>
#include<conio.h>
struct book
{
char name[20];
char author[20];
int pages;
};
void main()
{
struct book b1={"c language material","Ramesh",150};
struct book *p;
p=&b1;
clrscr();
printf("\n%s by %s of %d pages",b1.name,b1.author,b1.pages);
printf("\n%s by %s of %d pages",p->name,p->author,p->pages);
getch();
}
```

Output:

```
c language material by Ramesh of 150 pages
c language material by Ramesh of 150 pages
```

Passing Structures as Arguments to Functions:

Like other data types, a structure can be passed as an argument to a function.

Example:

/* Demonstrates passing a structure to a function. */

```
#include <stdio.h>
struct data{
float amount;
char fname[30];
char lname[30];
} rec;
void print_rec(struct data x);
void main()
{
clrscr();
printf("Enter the donor's first and last names,\n");
printf("separated by a space: ");
scanf("%s %s", rec.fname, rec.lname);
printf("\nEnter the donation amount: ");
scanf("%f", &rec.amount);
print_rec( rec );
return 0;
}
void print_rec(struct data x)
{
printf("\nDonor %s %s gave $%.2f.\n", x.fname, x.lname,
x.amount);
}
```

Nested structure:

We can take any data type for declaring structure members like int, float, char etc. in the same way we can also take object of one structure as member in another structure. Thus, structure can be used to create complex data application.

Example:

```
struct coord {

int x;
int y;

};

struct rectangle {

struct coord topleft;
struct coord bottomrt;

}mybox;
```

To access the actual data locations (the type int members), you must apply the member operator (.) twice. Thus, the expression

```
mybox.topleft.x;
```

Refers to the x member of the topleft member of the type rectangle structure named mybox. To define a rectangle with coordinates (0,10),(100,200), you would write.

```
mybox.topleft.x = 0;
mybox.topleft.y = 10;
mybox.bottomrt.x = 100;
mybox.bottomrt.y = 200;
```

Example:

/* Demonstrates structures that contain other structures. */

/* Receives input for corner coordinates of a rectangle and calculates the area. Assumes that the y coordinate of the upper-left corner is greater than the y coordinate of the lower-right corner, that the x coordinate of the lower- right corner is greater than the x coordinate of the upper- left corner, and that all coordinates are positive. */

```
#include <stdio.h>
int length, width;
long area;
struct coord{
int x;
int y;
};
struct rectangle{
struct coord topleft;
struct coord bottomrt;
} mybox;
void main()
{
clrscr();
printf("\nEnter the top left x coordinate: ");
scanf("%d", &mybox.topleft.x);
printf("\nEnter the top left y coordinate: ");
scanf("%d", &mybox.topleft.y);
printf("\nEnter the bottom right x coordinate: ");
scanf("%d", &mybox.bottomrt.x);
printf("\nEnter the bottom right y coordinate: ");
scanf("%d", &mybox.bottomrt.y);
width = mybox.bottomrt.x - mybox.topleft.x;
length = mybox.bottomrt.y - mybox.topleft.y;
area = width * length;
printf("\nThe area is %ld units.\n", area);
return 0;
getch();
}
```

Output:

```
Enter the top left x coordinate: 10
Enter the top left y coordinate: 5
Enter the bottom right x coordinate: 5
Enter the bottom right y coordinate: 2

The area is 15 units.
```

Unions

Unions are similar to structures. A union is declared and used in the same ways that a structure is. A union differs from a structure in that only one of its members can be used at a time. The reason for this is simple. All the members of a union occupy the same area of memory. They are laid on top of each other.

Defining, Declaring, and Initializing Unions:

Definition:

The union keyword is used for declaring unions. A union is a collection of one or more variables (union_members) that have been grouped under a single name. In addition, each of these union members occupies the same area of memory. The keyword union identifies the beginning of a union definition. It's followed by a tag that is the name given to the union.

Unions are defined and declared in the same fashion as structures. The only difference in the declarations is that the keyword union is used instead of struct. To define a simple union of a char variable and an integer variable, you would write the following:

```
union shared {
char c;
int i;
};
```

This union, shared, can be used to create instances of a union that can hold either a character value c or an integer value i. This is an OR condition. Unlike a structure that would hold both values, the union can hold only one value at a time. A union can be initialized on its declaration. Because only one member can be used at a time, only one can be initialized. To avoid confusion, only the first member of the union can be initialized. The
following code shows an instance of the shared union being declared and initialized:

```
union shared generic_variable = {`@'};
```

Notice that the generic_variable union was initialized just as the first member of a structure would be initialized.

Accessing Union Members:

Individual union members can be used in the same way that structure members can be used by using the member operator (.). However, there is an important difference in accessing union members. Only one union member should be accessed at a time. Because a union stores its members on top of each other, it's important to access only one member at a time.

Example:

/* Example of using more than one union member at a time */

```
#include <stdio.h>
void main()
{
union shared_tag {
char c;
int i;
long l;
float f;
double d;
} shared;
shared.c = '$';
clrscr();
printf("\n char c   = %c", shared.c);
printf("\n int i    = %d", shared.i);
printf("\n long l   = %ld", shared.l);
printf("\n float f  = %f", shared.f);
printf("\n double d = %f", shared.d);
shared.d = 123456789.8765;
printf("\n\nchar c  = %c", shared.c);
printf("\n int i    = %d", shared.i);
printf("\n long l   = %ld", shared.l);
printf("\n float f  = %f", shared.f);
printf("\n double d = %f\n", shared.d);
return 0;
}
```

Output:

```
 char c   = $
 int i    = 4900
 long l   = -1888873692
 float f  = -0.000000
 double d = -0.000000

char c  = 7
 int i    = -30409
 long l   = 1468107063
 float f  = 284852666499072.000000
 double d = 123456789.876500
```

Typedef:

By using typedef we can create new data type. The statement typedef is to be used while defining the new data type. The syntax is as given under.

Syntax:

typedef type dataname;

here, type is the data type and dataname is the user defined name for that type.

typedef int rollno;

Here rollno is the another name for int and we can use rollno instead of int in the program.

Example:

/* write a program to create userdefined data type rollno on int data type and use it in the program*/

```
#include<stdio.h>

#include<conio.h>

void main()

{

typedef int rollno;

rollno sno;

clrscr();

printf("enter student serial number=");

scanf("%d",&sno);

printf("\n student serial number=%d",sno);

getch();

}
```

Output:

```
enter student serial number=100

student serial number=100
```

BitFields:

- The Bit-field are used to reserve the memory space in terms of bits.
- we cant use the pointers to bit fields and hence we cant supply the value at runtime through scanf()function.
- Unlike some other computer languages, C has a built–in feature called a bit-field that allows you to access a single bit.
- Bit –fields a can be useful for a number of reasons such as
- It storage is limited, you can store several Boolean (true/ false), variables in one byte.
- the general from of a bit –field definition is

<datatype> name**:** length;

Here , type is the type of the bit-fields and length is the number of bits the field the type of bitfield must be int, signed or unsigned.

Example:

/* write a program to reserve memory in the form of bits */

```
#include<stdio.h>

#include<conio.h>

main()

{

struct dare

{

unsigned int dd:5;

unsigned int mm:4;

unsigned int yy:7;

};

struct  dare d={26,5,11};

clrscr();

printf("\n date=%u %u %u\n",d.dd,d.mm,d.yy);

}
```

Output:

```
date=26 5 11
```

Enumerated Data Type:

The enumerated data type gives you an opportunity to invent your own data type and define what value the variable of this data type can take. The format of the enum-definition is similar to that of a structure. Here's how the example started above can be implemented.

syntax:

```
enum enumdata_type

{

variables

};
```

Ex:

```
enum c++

{
```

function,class,constructor,encapsulation

};

enum c++ f1,f2;

like structure,this declaration has two parts;

a) The first part declares the data types and specifies its possible values. These values are called "enumetators".

b) The Second part declares variables of this data type.

Now we can give values to these variables

class

person1=class;

person2=constructor;

Remember we can't use values that are not in the original declaration, thus, the following expression would causes an error.

person1=unknown;

Internally the complier treats the enumerators as integers. Each value on the list of permissible values corresponds to an integer, starting with 0. Thus in our example, function is stored as o, and class is stored as 1,constructor is stored at 2,encapaulation is stored as 3 This way of assigned numbers can be overridden by the programmer by initializing a the enumerators to different integers value as shown below

enum c++

{

function=100,class=200,constructor=300,encapsulation=400

}

Advantages:

1. This can helps in making the program listing more readable. Which can be advantages when a program gets complicated or when more than one programmer would be working on it.
2. Using this we can also reduce programming errors.

/* write a program to explain enumdata type */

#include<stdio.h>

#include<conio.h>

int main()

```
{
enum sample
{
x,y=10,z
};
enum sample s;
s=z;
clrscr();
printf("%d\n",x);
printf("%d\n",y);
printf("%d\n",z);
return(0);
getch(); }
```

CHAPTER14

FILES

FILES:

- scanf( ) and printf( ) functions read and write data which always uses the terminal (keyboard and screen) as the target.
- It becomes confusing and time consuming to use large volumes of data through terminals.
- The entire data is lost when either program terminates or computer is turned off.
- Sometimes it may be necessary to store data in a manner that can be later retrieved and processed.

This leads to employ the concept of FILES to store data permanently in the system.

DEFINATION:

File is a set of records that can be accessed through the set of library functions. **Record** is logical group of data fields that comprise a single row of information, which describes the characteristics of an object. A **File** is a place on disk where a group of related data (records) can be stored.

File Operations:

1. Creating a new file
2. Opening an existing file
3. Reading from a file
4. Writing to a file
5. Moving to a specific location in a file (seek)
6. Closing a file

Q) What Exactly Is Program Input/ Output?

C program keeps data in random access memory (RAM) while executing. This data is in the form of variables, structures, and arrays that have been declared by the program. Where did this data come from, and what can the program do with it? Data can come from some location external to the program. Data moved from an external location into RAM, where the program can access it, is called input. The keyboard and disk files are the most common sources of program input. Data can also be sent to a location external to the program; this is called output. The most common destinations for output are the screen, a printer, and disk files. Input sources and output destinations are collectively referred to as devices. The keyboard is a device, the screen is a device, and so on. Some devices (the keyboard) are for input only, others (the screen) are for output only, and still others (disk files) are for both input and output. Whatever the device, and whether it's performing input or output, C carries out all input and output operations by means of streams.

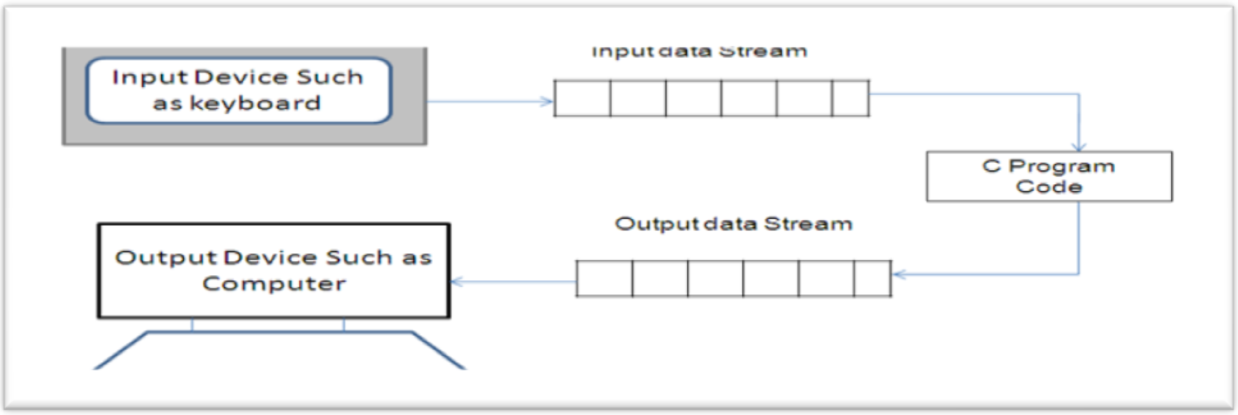
Q) What Is a Stream?

A stream is a sequence of characters. More exactly, it is a sequence of bytes of data. A sequence of bytes flowing into a program is an input stream; a sequence of bytes flowing out of a program is an output stream. By focusing on streams, you don't have to worry as much about where they're going or where they originated. The major advantage of streams, therefore, is that input/output programming is device independent. Programmers don't need to write special input/output functions for each device (keyboard, disk, and so on). The program sees input/output as a continuous stream of bytes no matter where the input is
coming from or going to.

Every C stream is connected to a file. In this context, the term file doesn't refer to a disk file. Rather, it is an intermediate step between the stream that your program deals with and the actual physical device being used for input or output. For the most part, the beginning C programmer doesn't need to be concerned with these files, because the

details of interactions between streams, files, and devices are taken care of automatically
by the C library functions and the operating system.

NOTE: Input and output can take place between your program and a variety of external devices.



A Stream acts as an interface between a program and an input/output Device.

Text modes:

Mode	**meaning**
r	Opens the file for reading. If the file doesn't exist, fopen() returns NULL.
W	Opens the file for writing. If a file of the specified name doesn't exist, it is created. If a file of the specified name does exist, it is deleted without warning, and a new, empty file is created.
a	Opens the file for appending. If a file of the specified name doesn't exist, it is created. If the file does exist, new data is appended to the end of the file.
r+	Opens the file for reading and writing. If a file of the specified name doesn't exist, it is created. If the file does exist, new data is added to the beginning of the file, overwriting existing
w+	Opens the file for reading and writing. If a file of the specified name doesn't

	exist, it is created. If the file does exist, it is overwritten.
a+	Opens a file for reading and appending. If a file of the specified name doesn't exist, it is created. If the file does exist, new data is appended to the end of the file.

File input and output functions:

File input/output functions	
fopen(fp, mode)	**Open existing file / Create new file**
fclose(fp)	**Closes a file associated with file pointer.**
closeall ( )	**Closes all opened files with fopen()**
fgetc(ch, fp)	**Reads character from current position and advances the pointer to next character.**
fprintf( )	**Writes all types of data values to the file.**
fscanf()	**Reads all types of data values from a file.**
gets()	**Reads string from a file.**
puts()	**Writes string to a file.**
getw()	**Reads integer from a file.**
putw()	**Writes integer to a file.**
fread()	**Reads structured data written by fwrite() function**
fwrite()	**Writes block of structured data to the file.**
fseek()	**Sets the pointer position anywhere in the file**
feof()	**Detects the end of file.**
rewind()	**Sets the record pointer at the beginning of the file.**
ferror()	**Reports error occurred while read/write operations**
fflush()	**Clears buffer of input stream and writes buffer of output stream.**

ftell()	**Returns the current pointer position.**

Opening a File:

The process of creating a stream linked to a disk file is called opening the file. When you open a file, it becomes available for reading (meaning that data is input from the file to the program), writing (meaning that data from the program is saved in the file), or both. When you're done using the file, you must close it. To open a file, you use the fopen() library function. The prototype of fopen() is located in STDIO.H and reads as follows:

FILE *fopen (const char *filename, const char *mode);

This prototype tells you that fopen() returns a pointer to type FILE, which is a structure declared in STDIO.H. The members of the FILE structure are used by the program in the various file access operations, but you don't need to be concerned about them. However, for each file that you want to open, you must declare a pointer to type FILE. When you call fopen(), that function creates an instance of the FILE structure and returns a pointer to that
structure. You use this pointer in all subsequent operations on the file. If fopen() fails, it returns NULL. Such a failure could be caused, for example, by a hardware error or by trying to open a file on a diskette that hasn't been formatted. The argument filename is the name of the file to be opened. As noted earlier, filename can and should contain a path specification. The filename argument can be a literal string enclosed in double quotation marks or a pointer to a string variable. The argument mode specifies the mode in which to open the file. In this context, mode controls whether the file is binary or text and whether it is for reading, writing, or both.

The default file mode is text. Remember that fopen() returns NULL if an error occurs. Error conditions that can cause a return value of NULL include the following:

1. Using an invalid filename.
2. Trying to open a file on a disk that isn't ready (the drive door isn't closed or the disk isn't formatted, for example).
3. Trying to open a file in a nonexistent directory or on a nonexistent disk drive.
4. Trying to open a nonexistent file in mode r.

Formatted File Output:

Formatted file output is done with the library function fprintf(). The prototype of fprintf() is in the header file STDIO.H, and it reads as follows:
int fprintf(FILE *fp, char *fmt, ...);

The first argument is a pointer to type FILE. To write data to a particular disk file, you pass the pointer that was returned when you opened the file with fopen().

The second argument is the format string. The format string used by fprintf() follows exactly the same rules as printf().The final argument is .... What does that mean? In a function prototype, ellipses represent a variable number of arguments. In other words, in addition to the file pointer and the format string arguments, fprintf() takes zero, one, or more additional arguments. This is just like printf(). These arguments are the names of the variables to be output to the specified stream.
Remember, fprintf() works just like printf(), except that it sends its output to the stream specified in the argument list. In fact, if you specify a stream argument of stdout, fprintf() is identical to printf().

Formatted File Input:

For formatted file input, use the fscanf() library function, which is used like scanf(), except that input comes from a specified stream instead of from stdin. The prototype for fscanf() is

int fscanf(FILE *fp, const char *fmt, ...);

The argument fp is the pointer to type FILE returned by fopen(), and fmt is a pointer to the format string that specifies how fscanf() is to read the input. The components of the format string are the same as for scanf(). Finally, the ellipses (...) indicate one or more additional arguments, the addresses of the variables where fscanf() is to assign the input.
Before getting started with fscanf(), you might want to review the section on scanf(). fscanf() works exactly the same as scanf(), except that characters are taken from the specified stream rather than from stdin.

Character Input and Output:

When used with disk files, the term character I/O refers to single characters as well as lines of characters. Remember, a line is a sequence of zero or more characters terminated by the newline character. Use character I/O with text-mode files. The following sections describe character input/output functions.

Character Input:

There are three character input functions: getc() and fgetc() for single characters, and fgets() for lines.

The getc() and fgetc() Functions:

The functions getc() and fgetc() are identical and can be used interchangeably. They input a single character from the specified stream. Here is the prototype of getc(), which is in STDIO.H:

int getc(FILE *fp);

The argument fp is the pointer returned by fopen() when the file is opened. The function returns the character that was input or EOF on error. If getc() and fgetc() return a single character, why are they prototyped to return a type int? The reason is that, when reading files, you need to be able to read in the end-of-file marker, which on some systems isn't a type char but a type int

The fgets() Function:

To read a line of characters from a file, use the fgets() library function. The prototype is

char *fgets(char *str, int n, FILE *fp);

The argument str is a pointer to a buffer in which the input is to be stored, n is the maximum number of characters to be input, and fp is the pointer to type FILE that was returned by fopen() when the file was opened. When called, fgets() reads characters from fp into memory, starting at the location pointed to by str. Characters are read until a newline is encountered or until n-1 characters have been read, whichever occurs first. By setting n
equal to the number of bytes allocated for the buffer str, you prevent input from overwriting memory beyond allocated space. (The n-1 is to allow space for the terminating \0 that fgets() adds to the end of the string.) If successful, fgets() returns str. Two types of errors can occur, as indicated by the return value of NULL:

1. If a read error or EOF is encountered before any characters have been assigned to str, NULL is returned, and the memory pointed to by str is unchanged.

2. If a read error or EOF is encountered after one or more characters have been assigned to str, NULL is returned, and the memory pointed to by str contains garbage.

Character Output:

You need to know about two character output functions: putc() and fputs().

The putc() Function:

The library function putc() writes a single character to a specified stream. Its prototype in STDIO.H reads.

int putc(int ch, FILE *fp);

The argument ch is the character to output. As with other character functions, it is formally called a type int, but only the lower-order byte is used. The argument fp is the pointer associated with the file (the pointer returned by fopen() when the file was opened). The function putc() returns the character just written if successful or EOF if an error occurs. The symbolic constant EOF is defined in STDIO.H, and it has the value -1. Because no "real"
character has that numeric value, EOF can be used as an error indicator (with text-mode files only).

The fputs() Function:

To write a line of characters to a stream, use the library function fputs(). This function works just like puts(),
The only difference is that with fputs() you can specify the output stream. Also, fputs() doesn't add a newline to the end of the string; if you want it, you must explicitly include it. Its prototype in STDIO.H is

char fputs(char *str, FILE *fp);

The argument str is a pointer to the null-terminated string to be written, and fp is the pointer to type FILE returned by fopen() when the file was opened. The string pointed to by str is written to the file, minus its terminating \0. The function fputs() returns a non negative value if successful or EOF on error.

Direct File Input and Output:

You use direct file I/O most often when you save data to be read later by the same or a different C program. Direct I/O is used only with binary-mode files. With direct output, blocks of data are written from memory to disk. Direct input reverses the process: A block of data is read from a disk file into memory. For example, a single direct-output function call can write an entire array of type double to disk, and a single direct-input function call can read the entire array from disk back into memory. The direct I/O functions are fread() and fwrite().

fwrite() Function:

The fwrite() library function writes a block of data from memory to a binary-mode file. Its prototype in STDIO.H is

int fwrite(void *buf, int size, int count, FILE *fp);

The argument buf is a pointer to the region of memory holding the data to be written to the file. The pointer type is void; it can be a pointer to anything. The argument size specifies the size, in bytes, of the individual data items, and count specifies the number of items to be written. For example, if you wanted to save a 100-element integer array, size would be 2 (because each int occupies 2 bytes) and count would be 100 (because the array contains 100 elements). To obtain the size argument, you can use the sizeof() operator. The argument fp is, of course, the pointer to type FILE, returned by fopen() when the file was opened. The fwrite() function returns the number of items written on success; if the value returned is less than count, it means that an error has occurred.

fread() Function:

The fread() library function reads a block of data from a binary-mode file into memory. Its prototype in STDIO.H is

int fread(void *buf, int size, int count, FILE *fp);

The argument buf is a pointer to the region of memory that receives the data read from the file. As with fwrite(), the pointer type is void. The argument size specifies the size, in bytes, of the individual data items being read, and count specifies the number of items to read. Note how these arguments parallel the arguments used by fwrite(). Again, the sizeof()
operator is typically used to provide the size argu-ment. The argument fp is (as always) the pointer to type FILE that was returned by fopen() when the file was opened. The fread() function returns the number of items read; this can be less than count if end-of-file was reached or an error occurred.

fseek():

It is used to set the position to a desired point in the file. fseek function is used to move the fi le position to a desired location within the fi le. It takes the following form:

fseek(fi le ptr, offset, position);

fi le ptr is a pointer to the fi le concerned, offset is a number or variable of type long, and position is an integer number. The offset specifi es the number of positions (bytes) to be moved from the location specified by position. The position can take one of the following three values:

Value	**Meaning**
0	Beginning of fi le
1	Current position
2	End of fi le

ftell():

It is used to determine the current position in the fi le. It returns the position in terms of bytes from the start. ftell takes a fi le pointer and returns a number of type long, that corresponds to the current position. This function is useful in saving the current position of a fi le, which can be used later in the program. It takes the following form:
n = ftell(fp);
n would give the relative offset (in bytes) of the current posi tion. This means that n bytes have already been read (or written).

rewind():

It is used to set the position of the fi le pointer at the beginning of the fi le. rewind() takes a fi le pointer and resets the position to the start of the fi le. For example, the statement
rewind(fp);
n = ftell(fp);
would assign 0 to n because the fi le position has been set to the start of the fi le by rewind. Remember, the fi rst byte in the fi le is numbered as 0, second as 1, and so on.

This function helps us in reading a file more than once, without having to close and open the fi le. Remember that whenever a fi le is opened for read ing or writing, a rewind is done implicitly.

Closing a file:

A file must be closed as soon as all operations on it have been completed. This ensures that all outstanding information associated with the fi le is fl used out from the buffers and all links to the file are broken. This would close the fi le associated with the FILE pointer file inter. Look at the Following segment of a program.

```
.....
FILE *p1, *p2;
p1 = fopen("INPUT", "w");
p2 = fopen("OUTPUT", "r");
.....
fclose(p1);
fclose(p2);
.....
```

This program opens two files and closes them after all operations on them are completed. Once a file is closed, its file pointer can be reused for another file.

Examples1:
/* write a program to open a file and store information*/

```
#include<stdio.h>

#include<conio.h>

void main()

{

char ch;

FILE *fp;

fp=fopen("ramesh","w");

printf("enter data\n");

ch=getchar();

while(ch!=EOF)

{

putc(ch,fp);

ch=getchar();

}

fclose(fp);
}
```

Example2:

```
/* write a program to open a file and  read the information*/
#include<stdio.h>
#include<conio.h>
void main()
{
char ch;
FILE *fp;
clrscr();
fp=fopen("ramesh","r");
ch=getc(fp);
while(ch!=EOF)
{
putchar(ch);
ch=getc(fp);
}
fclose(fp);
}
```

CHAPTER15

COMMAND LINE ARGUMENTS

AND

The C Preprocessor

AND

INTERVIEW QUESTIONS

COMMAND LINE ARGUMENTS:

In C it is possible to accept command line arguments. Command-line arguments are given after the name of a program in command-line operating systems like DOS or Linux, and are passed in to the program from the operating system.

To use command line arguments in your program, you must first understand the full declaration of the main function, which previously has accepted no arguments. In fact, main can actually accept two arguments:

i) one argument is number of command line arguments,

ii) second argument is a full list of all of the command line arguments.

The full declaration of main looks like this:

int main (int argc, char *argv[])

The integer, argc is the argument count. It is the number of arguments passed into the program from the command line, including the name of the program.

The array of character pointers is the listing of all the arguments. argv[0] is the name of the program, or an empty string if the name is not available. After that, every element number less than argc is a command line argument.

We can use each argv element just like a string, or use argv as a two dimensional array. The argv[argc] is a null pointer.

Almost any program that wants its parameters to be set when it is executed would use this. One common use is to write a function that takes the name of a file and outputs the entire text of it onto the screen.

For example to create a directory, we execute at command line or prompt as follows,

```
mkdir dir_name
```

where actually, mkdir is the name of the program written by somebody and is used to

create a directory with a given name. The usage of that program would be as shown above.

So, we can also write such programs using arguments called command-line arguments.

Example: Program to illustrate command line arguments in C -program.

/*the **cmdline.c** is the name of the program. */

```
#include<stdio.h>
void main (int argc, char *argv[])
{
int i;
if(argc != 2)
{
printf("Too few arguments.\n");
exit(0);
}

printf("\nThe command-line arguments %d are:\n", argc);
for(i=0;i<argc; i++)
printf("argv[%d]: %s\n", i, argv[i]);

getch();
}
```

Output:

```
D:\cpds>cmdline Suresh Rahul Harikrishna Govardhan

The command-line arguments 5 are:

argv[0]: D:\CPDS\CMDLINE.EXE

argv[1]: Suresh

argv[2]: Rahul

argv[3]: Harikrishna

argv[4]: Govardhan
```

Explanation: How to execute this program,

i) Type program in file : cmdline.c
ii) Compile cmdline.c
iii) Identify the location of cmdline.exe. in my case, cmdline.exe is in D:\cpds directory. Or copy cmdline.exe into require directory or drive. Then execute from there, as shown below.
iv) D:\cpds>cmdline Suresh Rahul Harikrishna Govardhan.

The C Preprocessor

Recall that preprocessing is the first step in the C program compilation stage this feature is unique to C compilers. The preprocessor more or less provides its own language which can be a very powerful tool to the programmer. Recall that all preprocessor directives or commands begin with #.

Use of the preprocessor is advantageous since it makes:

- programs easier to develop,
- easier to read,
- easier to modify
- C code more transportable between different machine architectures.

The preprocessor also lets us customize the language. For example to replace { ... } block statements delimiters by PASCAL like begin ... end we can do:

```
    #define begin {
                        #define end}
```

During compilation all occurrences of begin and end get replaced by corresponding { or } and so the subsequent C compilation stage does not know any difference!!!.

Lets look at #define in more detail

#define:

Use this to define constants or any macro substitution. Use as follows:

```
    #define<macro> <replacement name>
```

For Example

```
                        #define FALSE 0
                        #define TRUE !FALSE
```

We can also define small ``functions'' using #define. For example max. of two variables:

```
#define max(A,B) ( (A) > (B) ? (A):(B))
```

? is the ternary operator in C.

Note: that this does not define a proper function max.

All it means that wherever we place max(C⊥,D⊥) the text gets replaced by the appropriate definition. [⊥ = any variable names - not necessarily C and D]

So if in our C code we typed something like:

```
        x = max(q+r,s+t);
```

after preprocessing, if we were able to look at the code it would appear like this:

```
        x = ( (q+r) > (r+s) ? (q+r) : (s+t));
```

Other examples of #define could be:

```
#define Deg_to_Rad(X) (X*M_PI/180.0)
/* converts degrees to radians, M_PI is the value
of pi and is defined in math.h library */

#define LEFT_SHIFT_8 <<8
```

NOTE: The last macro LEFT_SHIFT_8 is only valid so long as replacement context is valid *I.e.* x = y LEFT_SHIFT_8.

This commands undefined a macro. A macro **must** be undefined before being redefined to a different value.

#include

This directive includes a file into code.

It has two possible forms:

#include <file>

or

```
        #include ``file''
```

<file> tells the compiler to look where system include files are held. Usually UNIX systems store files in **\usr\include\** directory.

``file'' looks for a file in the current directory (where program was run from)

Included files usually contain C prototypes and declarations from header files and not (algorithmic) C code (SEE next Chapter for reasons)

#if -- Conditional inclusion:

#if evaluates a constant integer expression. You always need a #endif to delimit end of statement.

We can have *else etc.* as well by using #else and #elif -- else if.

Another common use of #if is with:

#ifdef

-- >if defined and

#ifndef

-- >if not defined

These are useful for checking if macros are set. Perhaps from different program modules and header files.

For example, to set integer size for a portable C program between TurboC (on MSDOS) and Unix (or other) Operating systems. Recall that TurboC uses 16 bits/integer and UNIX 32 bits/integer.

Assume that if TurboC is running a macro TURBOC will be defined. So we just need to check for this:

```
 #ifdef TURBOC
              #define INT_SIZE 16
              #else
              #define INT_SIZE  32
              #endif
```

As another example if running program on MSDOS machine we want to include file msdos.h otherwise a default.h file. A macro SYSTEM is set (by OS) to type of system so check for this:

```
 #if SYSTEM == MSDOS
              #include <msdos.h>
             #else
                           #include ``default.h''
```

```
                        #endif
```

Examples:

/* Idenitify the output of the following program*/

```
#include<stdio.h>

#include<conio.h>

void main()

{

clrscr();

printf("\nWelcome");

#if 5!=5

printf("\nASR");

printf("\nReddy");

#endif

printf("\nRamesh");

getch();

}
```

/* Idenitify the output of the following program*/

```
#include<stdio.h>

#include<conio.h>

void main()

{

clrscr();

printf("\nWelcome");

#if 5>=8!=0

printf("\nASR");

printf("\nRAMESH");

#else

printf("\nC & C++");

printf("\nJAVA");
#endif

printf("\nOracle");

getch();

}
```

SOME IMPORTANT QUESTIONS:

1. What does static variable mean?

Ans: Static is a storage class in C. A variable of type static will persist through out the program. This persists the value of the variable through subsequent function calls.

2. What is a pointer?

Ans: A pointer is variable which stores the address of another variable or areas of dynamically allocated memory.

3. What is a structure?

Ans: A structure is a user defined data type. This stores the real data. For example store a student information we can declare a structure type student.

4. What are difference between structure and arrays?

Ans: Array is group of similar data type elements where as structure is group of dissimilar data type elements.

5. In header files whether functions are declared or defined?

Ans: The functions are only declared in the header file.

6. What are the differences between the malloc and calloc function.

Ans: The major difference between malloc and calloc function is malloc allocates the memory chunk and initializes to garbage value where as calloc function allocates the memory in block wise and initializes to 0. Malloc function takes one argument where as calloc takes two arguments.

7. What are macros? What are the advantages and disadvantages?

ANS: A *macro* is a fragment of code which has been given a name. Whenever the name is used in the program, it is replaced by the contents of the macro. The ways in which macros are handled by preprocessors vary greatly from compiler to compiler. Some problems are:

1. Spaces between parameters in the declaration are accepted in some preprocessors and misinterpreted in others.
2. Blanks after a slash in some compilers are simply ignored but in others they produce errors.

8. Difference between pass by reference and pass by value?

Ans: In call by value the actual argument will not be affected by the change of the formal parameter. Where as in Call by reference the actual argument will be affected by the formal parameters.

9. What is static Identifier?

Ans: In C , a variable declared as static is local to a particular function. It is initialized once and on leaving the function, the static variable retains the value. Next time when the function is called again.

10. Where are the auto variable stored?

Ans: The auto variable will be stored in the temporary memory location and by default it initializes to a variable to garbage.

11.Where does global, static, local, register variable free memory and C Program instructions get stored?

Ans: Global, local and static variables will be store in the temporary memory location where as register variable will be stored in the cpu register if it has free memory in CPU registers.

12. Difference between arrays and Linked List?

Ans: The major difference between array and linked list is that array is a static data structure and linked list is the dynamic data structure. In linked list we can add elements dynamically and also remove but where as in arrays it is not possible.

13.What are enumerations?

Ans: It is a data type similar to a structure. Enumeration types provide the facility to specify the possible values of a variable by meaningful symbolic means. This can help in making the program more readable.

14.Describe about storage allocation and scope of global, extern, static, local and register variables?

Ans: In C Storage classes specifies about the scope, life time, storage location and default value of a particular variable. Auto, extern and static variables are stored in temporary memory location. Register variables are stored in CPU registry. The scope of auto, static and register variable is with in the function but where as extern variables scope is through out the program. Auto and register variable by default initializes to garbage where as static and extern variables are by default initializes to 0.

15.What are register variables? What are the advantages of using register variables?

Ans: Register variables will be stored in the CPU register memory so accessing the value from a register will be faster compare to the normal temporary memory, So register variables are used as a loop counter.

16.What is the use of typedef?

Ans: typedef is a keyword in the C programming languages. It is used to give a data type a new name. The intent is to make it easier for programmers to comprehend source code.

17.Can we specify variable field width in a scanf() format string? If possible how?

Ans: We can use the variable field width in a scanf() function and the syntax to give the required length accepting example scanf(“%5s”,str) where str is a character array.

18.Out of fgets() and gets() which functions safe to use and why?

Ans: gets() function is very much danger to use because it takes input directly from standard input, and it has no protection against buffer overflow. A very easily exploited combination.

19.Difference between strdup() and strcpy()?

Ans: strcpy - copy a string to a location YOU created (you create the location, make sure that the source string will have enough room there and afterwards use strcpy to copy)

strdup - copy a string to a location that will be created by the function. The function will allocate space, make sure that your string will fit there and copy the string. Will return a pointer to the created area.

20.What is a recursion?

Ans: Recursion in computer programming defines a function in terms call itself. The great advantage of recursion is that an infinite set of possible statements executed by a finite computer program. It can be indirect or direct.

21.Differentiate between a for loop and a while loop ? What are it uses?

Ans: The main difference between a WHILE loop and a FOR loop is that a FOR loop is guaranteed to finish, but a WHILE loop is not. The FOR statement specifies the exact number of times the loop executes, unless a statement causes the routine to exit the loop. With while, it is possible to create an endless loop.

22.What are different storage classes in C.

There are four kind of storage classes are there that are auto, register, static and extern.

23.What is the difference between structure and union.

Ans: Both structure and union are user defined data type only the difference between them is the only between structure and union is the size of the union is equal to the size of the largest member of the union where as size of the structure is the sum of the size of al members of the structure. And one more thing is that we can use one member of the union at a time.

24.What are the advantages of using Union?

Ans: Union is a data structure that stores one of several types of data at a single location so memory can be minimized.

25.What are the advantages of using pointers in a program?

Ans: By using a pointer we can dynamically allocate the memory so we can decide at runtime how many elements we need to take input overcomes the limitation of Array by being a static memory allocation. And we can access the other variables from a function.

26.What is the difference between String and Arrays?

Ans: A normal character array is like a array as like integers which can store number of characters. But string is a character array which ends with '\0' null character.

27.What is a far pointer and where we use it?

Ans: Far pointers are the normalized pointers of four bytes which are used to access the main memory of the computer ...it can access both the data segment and code segment thus by modifying the offset u can modify refer two different addresses but refer to the same memory.

28.How will you declare an array of three function pointer where each function receives two ints and returns a float.

Ans: Declare an array of function pointers

```
float (*Func_Pointer[3])(int,int);
```

29. What is NULL pointer? Whether it is same as an uninitialized pointer ?

Ans: A null pointer has a reserved value, often but not necessarily the value zero, indicating that it refers to no object. Null pointers are used routinely, particularly in C and C++ where the compile-time constant NULL is used, to represent conditions such as the lack of a successor to the last element of a linked list. Null pointer means it is not referring to any object but Uninitialized pointer can contain any value (just like any other uninitialized variable), which we call garbage.

30. What is NULL Macro? What is the difference between a NULL pointer and a NULL Macro?

Ans: NULL can make it more clear in the code that you are expecting and dealing with pointers, but NULL is not part of the language proper, but part of the library (meaning that you must include a library before it is defined).

In a case Null Macro 0 requires no support library (i.e. it is part of the language proper), but it can lead some loss of code clarity when dealing w/ pointers (specifically integer pointers, is the intent to compare the pointer or the destination of the pointer ?).

31. What does the error "Null Pointer Assignment" mean and what causes this error?

Ans: A NULL pointer assignment is a runtime error It occurs due to various reasons one is that your program has tried to access an illegal memory location.

32. What is near, far and huge pointer? How many bytes are occupied by them?

Ans: Huge pointers are fully recognized and evaluated for their entire width. Far pointers only allow normal operations to be done to their offset amount and not the segment. Near pointers have a size of 2 bytes? They only store the offset of the address the pointer is referencing. A near pointer can be incremented and decremented using arithmetic operators (+, -, ++, and --) through the entire address range. Any attempt to increment a near pointer that has a value of 64K (0xffff) will result in a value of 0.

33. Are the expressions arr and *arr same for an array of integers?

Ans: Whenever we say arr that means the address of the arr[0] th location that is base index address of the array whereas * arr means the value which is stored in arr[0].

34. What are the similarities between structure, union and enumeration?

Ans: The major similarities that they all are user defined data types to store real data. In structure and Union their members can be objects of any type, including other structures and unions or arrays. A member can also consist of a bit field. The only operators valid for use with entire structures and unions are the simple assignment (=) and sizeof operators. A structure or a union can be passed by value to functions and returned by value by functions.

35. Can structure contain a Pointer to itself?

Ans: A structure can contain a Pointer to itself. It is called self – referential pointer which can store an address of another structure object by which we can create a list.

36. How can we check whether the contents of two structure variables are same or not?

Ans: By using memcmp() function we can compare two structure objects contents same or not.

37.What is the difference between an enumeration and set of Pre-Processor # defines?

Ans: a macro just replaces the code assigned to it. For example if a macro "PI" is defined with a value 3.14. Compiler just replaces PI with 3.14 where ever it finds PI in the program. Enum-declared constants are to be preferred because they obey scoping rules (e.g. they can be encapsulated within
a class or namespace).

38.What do the 'c ' and 'v' in argc and argv stand for?

Ans: Here c stands for the counter and v stands for the value.

39.Are the variables argc and argv are local to main?

Ans: Yes,These both variables are local to the main only.

40.What is the maximum combined length of command line arguments including the space between adjacent arguments?

Ans: There is 126 character limit is there in passing as command line argument.

41.If we want that any wild card characters in the command line arguments should be appropriately expanded, are we required to make any special provisions? If yes which?

Ans: If we have any special wild characters in the command line argument then precede with a '\' character to appropriately expanding. You can quote only some parts of the wildcard to protect only those parts from expansion; the unquoted parts will still be expanded. This allows to use wildcards with embedded whitespace and expand file names with special characters which need to be quoted, like in c:/Prog*' 'F* (which should expand into c:/Program Files) and *.c"'"v (which should expand into all files with the *.c'v extension).

42.What are bit fields ? What is the use of bit fields in a structure declaration?

Ans: Bit fields provide greater control over the structure's storage allocation and allow tighter packing of information in memory. By using bit fields, data can be densely packed into storage.

43.To which numbering system can the binary number 1101100100111100 be easily converted to?

Ans: To convert to decimal numbering system will be easy.

44.Which bit wise operator is suitable for checking whether a particular bit is on or off?

Ans: Bitwise operator not(!) is used for checking whether the bit is on or off.

45.Which bit wise operator is suitable for turning off a particular bit in a number?

Ans: Bit wise Not operator is used for turning off to a particular bit in a number.

46. What are the advantages of using typedef in a Program?

Ans: The typedef keyword allows the programmer to create new names for types such as int ,char or to a user defined data type like structure.

Syntax for creating Typedef

Typedef Data_Type New_Name;

Ex: Lets struct student{ int id,age;}

Typedef struct student stud; //struct student now can be represent as stud.

47. How would you dynamically allocate a one dimensional and two dimensional array of integers?

Ans: To allocate the one dimensional array dynamically we have to use a simple malloc function like we can say int *p=(int *) malloc(i* sizeof(int)) . So here I number of integers can be stored by the pointer. But when we are going to dynamically allocate the double dimensional array then you must first allocate memory sufficient to hold all the elements of the array (the data), then allocate memory for pointers to each row of the array. For arrays of more than two dimensions it gets more complicated.

48. How can you increase the size of a dynamically allocated array?

Ans: By using realloc() function we can allocate memory again to a dynamic allocated array.

50. Which function should be used to free the memory allocated by calloc()?

Ans: By using the free() method we can free the memory which allocated by calloc() function.

51. Which header file should you include if you are to develop a function which can accept variable number of arguments?

Ans: We can use <stdarg.h> header file for including the variable number of arguments in the c programming.

52. How can a called function determine the number of arguments that have been passed to it?

Ans: To determine number of arguments passed by a variable arguments manually we have to figure it out . There is no function have been defined where we can ask compiler for how many values have been passed to this function.

53. Can there be at least some solution to determine the number of arguments passed to a variable argument list function?

Ans: To determine number of arguments passed by a variable arguments manually we have to figure it out . There is no function have been defined where we can ask compiler for how many values have been passed to this function.

54. How do you declare the following :

i. An array of three Pointers to chars.

Ans: char *arr[3];

ii. An array of three char pointers.

Ans: char * arr[3];

iii. A Pointer to array of three chars.

Ans: char *p;

iv. A Pointer to function which receives an int pointer and returns a float pointer.

Ans: float *(*fp)(int *)

v. A Pointer to a function which receives nothing and returns nothing.

Ans: void (*fp)(void)

55. What do the function atoi() ,itoa() and gcvt() do?

Ans: int atoi(const char *) function converts a character string into specified integer type.

ii void itoa(int input, char *buffer, int radix) function constructs a string representation of an integer

iii char *gcvt(double value, int ndigit, char *buf) function converts a floating-point number to a string.

56. How would you use the functions sin(),pow() and sqrt()?

Ans: i. double sin(double arg) The function returns the sine of *arg*, where *arg* is given in radians.

ii. double pow(double base,double exp) The function returns *base* raised to the *exp* power. There's a domain error if *base* is zero and *exp* is less than or equal to zero.

iii. double sqrt(double num) The function returns the square root of *num*. If *num* is negative, a domain error occurs.

57. How would you use the function memcpy(),memset(),memmove() ?

i. The function memcpy() copies *count* characters from the array *from* to the array *to*. memcpy() returns *to*. The behavior of memcpy() is undefined if *to* and *from* overlap.

void *memcpy(void *to, const void *from, size_t count);

ii. The function memset() copies *ch* into the first *count* characters of *buffer*, and returns *buffer*. memset() is useful for intializing a section of memory to some value. For example, this command:

void *memset(void *buffer, int ch, size_t count);

iii. The memmove() function is identical to memcpy(), except that it works even if *to* and *from* overlap.

void *memmove(void *to, const void *from, size_t count);

58. How would you use the functions fseek(),fread(), fwrite() and ftell()?

i. int fseek(FILE *stream, long offset, int origin);

The function fseek() sets the file position data for the given stream. The *origin* value should have one of the following values (defined in stdio.h):

Name	Explanation
SEEK_SET	Seek from the start of the file
SEEK_CUR	Seek from the current location
SEEK_END	Seek from the end of the file

fseek() returns zero upon success, non-zero on failure. You can use fseek() to move beyond a file, but not before the beginning. Using fseek() clears the EOF flag associated with that stream.

ii. int fread(void *buffer, size_t size, size_t num, FILE *stream);

The function fread() reads *num* number of objects (where each object is *size* bytes) and places them into the array pointed to by *buffer*. The data comes from the given input stream. The return value of the function is the number of things read...use feof() or ferror() to figure out if an error occurs.

iii. int fwrite(const void *buffer, size_t size, size_t count, FILE *stream);

The fwrite() function writes, from the array *buffer*, *count* objects of size *size* to *stream*. The return value is the number of objects written.

iv. long ftell(FILE *stream);

The ftell() function returns the current file position for *stream*, or -1 if an error occurs.

59. How would you obtain the current time and difference between two times?

Ans: *To print a current time we can use the following code.*

```
#include<time.h>

#include<stdio.h>

main()

{

        time_t tt;

        struct tm *tod;

        time(&tt);

        tod=localtime(&tt);

        printf("%d:%d:%d",tod->tm_hour,tod->tm_min,tod->tm_sec);

        exit(0);

}
```

60. How would you use the functions randomize () and random ()?

Ans: int random (int n);

Which generates a random number in the range of 0 to n-1? For example;

```
        y = random (100);
```

Y will be in the range of 0 though 99

61 How would you implement a substr() function that extracts a sub string from a given string?

Ans : basic_string substr(size_type index, size_type num = npos);

The substr() function returns a substring of the current string, starting at index, and num characters long. If npos is omitted, it will default to string::npos, and the substr() function will simply return the remainder of the string starting at index.

62. What is the difference between the functions memmove() and memcpy()?

Ans: void *memcpy(void *to, const void *from, size_t count);

The function memcpy() copies *count* characters from the array *from* to the array *to*. memcpy() returns *to*. The behavior of memcpy() is undefined if *to* and *from* overlap.

void *memmove(void *to, const void *from, size_t count);

The memmove() function is identical to memcpy(), except that it works even if *to* and *from* overlap.

63. Can you use the function fprintf() to display the output on the screen?

Ans: Yes we can use fprintf() function for printing to the output screen

Let s example I want to write a String to output screen we can write

```
fprintf (stdout,”%s”,ch1);//where ch1 is a character String
```

65. How can we read/write structures from/to data files?

Ans: To write or read structure object to a file we can use fread() and fwrite() function. The Syntax of the fread() function is fread(Address of Buffer, Number of Object, object size, File Pointer) and fwrite(Address of Buffer, Number of Object, Object Size, File Pointer).

2nd Level Interview Questions:

Q) If I want to give someone a program I wrote, which files do I need to give him?

One of the nice things about C is that it is a compiled language. This means that after the source code is compiled, you have an executable program. This executable program is a stand-alone program. If you wanted to give HELLO to all your friends with computers, you could. All you need to give them is the executable program, HELLO.EXE. They don't need the source file, HELLO.C, or the object file, HELLO.OBJ. They don't need to own a C compiler, either.

Q Can I ignore warning messages?

Some warning messages don't affect how the program runs, and some do. If the compiler gives you a warning message, it's a signal that something isn't right. Most compilers let you set the warning level. By setting the warning level, you can get only the most serious warnings, or you can get all the warnings, including the most minute.

Some compilers even offer various level sin-between. In your programs, you should look at each warning and make a determination. It’s always best to try to write all your programs with absolutely no warnings or errors. (With an error, your compiler won't create the executable file.)

Q) What effect do comments have on a program?

Comments are for the programmer. When the compiler converts the source code to object code, it throws the comments and the white space away. This means that they have no effect on the executable program. Comments do make your source file bigger, but this is usually of little concern. To summarize, you should use comments and white space to make your source code as easy to understand and maintain as possible.

Q) What is the difference between a statement and a block?

A block is a group of statements enclosed in braces ({}). A block can be used in most places that a statement can be used.

Q) How can I find out what library functions are available?

Many compilers come with a manual dedicated specifically to documenting the library functions. They are usually in alphabetical order. Another way to find out what library functions are available is to buy a book that lists them. Appendix E, "Common C Functions," lists many of the available functions. After you begin to understand more of C, it would be a good idea to read these appendixes so that you don't rewrite a library function. (There's no use reinventing the wheel!)

Q) What effect do spaces and blank lines have on how a program runs?

White space (lines, spaces, tabs) makes the code listing more readable. When the program is compiled, white space is stripped and thus has no effect on the executable program. For this reason, you should use white space to make your program easier to read.

Q) How do I know what a good function name is?

A good function name describes as specifically as possible what the function does.

Q) Does main() have to be the first function in a program?

No. It is a standard in C that the main() function is the first function to execute; however, it can be placed anywhere in your source file. Most people place it first so that it's easy to locate.

Q How do I know which programming control statement to use--the for, the while, or the do...while?

If you look at the syntax boxes provided, you can see that any of the three can be used to solve a looping problem. Each has a small twist to what it can do, however. The for statement is best when you know that you need to initialize and increment in your loop. If you only have a condition that you want to meet, and you aren't dealing with a specific number of loops, while is a good choice. If you know that a set of statements needs to be executed at least once, a do...while might be best. Because all three can be used for most problems, the best course is to learn them all and then evaluate each programming situation to determine which is best.

Q) Why should I use puts() if printf() does everything puts() does and more?

Because printf() does more, it has additional overhead. When you're trying to write a small, efficient program, or when your programs get big and resources are valuable, you will want to take advantage of the smaller overhead of puts(). In general, you should use the simplest available resource.

Q) What happens if I leave the address-of operator (&) off a scanf() variable?

This is an easy mistake to make. Unpredictable results can occur if you forget the address-of operator. know that if you omit the address-of operator, scanf() doesn't place the entered information in your variable, but in some other place in memory. This could do anything from apparently having no effect to locking up your computer so that you must reboot.

Q) What happens if I use a subscript on an array that is larger than the number of elements in the array?

If you use a subscript that is out of bounds with the array declaration, the program will probably compile and even run. However, the results of such a mistake can be unpredictable. This can be a difficult error to find once it starts causing problems, so make sure you're careful when initializing and accessing array elements.

Q) What happens if I use an array without initializing it?

This mistake doesn't produce a compiler error. If you don't initialize an array, there can be any value in the array elements. You might get unpredictable results. You should always initialize variables and arrays so that you know exactly what's in them.

Q) Why are pointers so important in C?

Pointers give you more control over the computer and your data. When used with functions, pointers let you change the values of variables that were passed, regardless of where they originated.

Q) How does the compiler know the difference between * for multiplication, for dereferencing, and for declaring a pointer?

The compiler interprets the different uses of the asterisk based on the context in which it is used. If the statement being evaluated starts with a variable type, it can be assumed that the asterisk is for declaring a pointer. If the asterisk is used with a variable that has been declared as a pointer, but not in a variable declaration, the asterisk is assumed to dereference. If it is used in a mathematical expression, but not with a pointer variable, the asterisk can be assumed to be the multiplication operator.

Q) What happens if I use the address-of operator on a pointer?

You get the address of the pointer variable. Remember, a pointer is just another variable that holds the address of the variable to which it points.

Q) Are variables always stored in the same location?

No. Each time a program runs, its variables can be stored at different addresses within the computer. You should never assign a constant address value to a pointer.

Q) Do all computers support the extended ASCII character set?

No. Most PCs support the extended ASCII set. Some older PCs don't, but the number of older PC slacking this support is diminishing. Most programmers use the line and block characters of the extended set.

Q) Is it more common to use a type def or a structure tag?

Many programmers use typedefs to make their code easier to read, but it makes little practical difference. Many add-in libraries that contain functions are available for purchase. These add-ins usually have a lot of typedefs to make the product unique. This is especially true of data based-in products.

Q) Is it better to use a switch statement or a nested loop?

A If you're checking a variable that can take on more than two values, the switch statement is almost always better. The resulting code is easier to read, too. If you're checking a true/false condition, go with an if statement.

Q) Why don't all compilers have the same functions?

Certain C functions aren't available with all compilers or all computer systems. For example, sleep() is available with the Borland C compilers but not with the Microsoft compilers. Although there are standards that all ANSI compilers follow, these standards don't prohibit compiler manufacturers from adding additional functionality. They do this by creating and including new functions. Each compiler manufacturer usually adds a number of functions that they believe will be helpful to their users.

Q) Isn't C supposed to be a standardized language?

C is, in fact, highly standardized. The American National Standards Institute (ANSI) has developed the ANSI C Standard, which specifies almost all details of the C language, including the functions that are provided. Some compiler vendors have added more functions ones that aren't part of the ANSI standard to their C compilers in an effort to one-up the competition. In addition, you sometimes come across a compiler that doesn't claim to meet the ANSI standard. If you limit yourself to ANSI-standard compilers, however, you'll find that 99 percent of program syntax and functions are common among them.

Q) Is there any danger in using non-ANSI functions in a program?

Most compilers come with many useful functions that aren't ANSI-standard. If you plan on always using that compiler and not porting your code to other compilers or platforms, there won't be a problem. If you're going to use other compilers and platforms, you should be concerned with ANSI compatibility.

Q) Why shouldn't I always use fprintf() instead of printf()? Or fscanf() instead of scanf()?

If you're using the standard output or input streams, you should use printf() and scanf(). By using these simpler functions, you don't have to bother with any other streams.

Q) How many levels deep can I go with pointers to pointers?

You need to check your compiler manuals to determine whether there are any limitations. It's usually impractical to go more than three levels deep with pointers (pointers to pointers to pointers). Most programs rarely go over two levels.

Q) Is there a difference between a pointer to a string and a pointer to an array of characters?

No. A string can be considered an array of characters.

Q) Can I read beyond the end of a file?

Yes. You can also read before the beginning of a file. Results from such reads can be disastrous. Reading files is just like working with arrays. You're looking at offsets within memory. If you're using fseek(), you should check to make sure that you don't go beyond the end of the file.

Q) What happens if I don't close a file?

It's good programming practice to close any files you open. By default, the file should be closed when the program exits; however, you should never count on this. If the file isn't closed, you might not be able to access it later, because the operating system will think that the file is already in use.

Q) How many files can I open at once?

This question can't be answered with a simple number. The limitation on the number of files that can be opened is based on variables set within your operating system. On DOS systems, an environment variable called FILES determines the number of files that can be opened (this variable also includes programs that are running). Consult your operating system manuals for more information.

Q) Can I read a file sequentially with random-access functions?

When reading a file sequentially, there is no need to use such functions as fseek(). Because the file pointer is left at the last position it occupied, it is always where you want it for sequential reads. You can use fseek() to read a file sequentially; however, you gain nothing.

Q How do I know whether a function is ANSI-compatible?

Most compilers have a Library Function Reference manual or section. This manual or section of a manual lists all the compiler's library functions and how to use them. Usually the manual includes information on the compatibility of the function. Sometimes the descriptions state not only whether the function is ANSI-compatible, but also whether it is compatible with DOS,UNIX,Windows, C++, or OS/2. (Most compilers tell you only what is relevant to their compiler.)

Q) Is passing pointers as function arguments a common practice in C programming?

Definitely! In many instances, a function needs to change the value of multiple variables, and there are two ways this can be accomplished. The first is to declare and use global variables. The second is to pass pointers so that the function can modify the data directly. The first option is advisable only if nearly every function will use the variable; otherwise, you should avoid it.

Q) Is it better to modify a variable by assigning a function's return value to it or by passing a pointer to the variable to the function?

When you need to modify only one variable with a function, usually it's best to return the value from the function rather than pass a pointer to the function. The logic behind this is simple. By not passing a pointer, you don't run the risk of changing any data that you didn't intend to change, and you keep the function independent of the rest of the code.

Q) What's the advantage of dynamic memory allocation? Why can't I just declare the storage space I need in my source code?

If you declare all your data storage in your source code, the amount of memory available to your program is fixed. You have to know ahead of time, when you write the program, how much memory will be needed. Dynamic memory allocation lets your program control the amount of memory used to suit the current conditions and user input. The program can use as much memory as it needs, up to the limit of what's available in the computer.

Q) Why would I ever need to free memory?

When you're first learning to use C, your programs aren't very big. As your programs grow, their use of memory also grows. You should try to write your programs to use memory as efficiently as possible. When you're done with memory, you should release it. If you write programs that work in a multitasking environment, other applications might need memory that you aren't using.

Q) Do header files need to have an .H extension?

No. You can give a header file any name you want. It is standard practice to use the .H extension.

Q) When including header files, can I use an explicit path?

Yes. If you want to state the path where a file to be included is, you can. In such a case, you put the name of the include file between quotation marks.