LECTURE NOTES

on

PROGRAMMING & DATA STRUCTURE

Course Code : BCS101

By

Prof. Dr. Amiya Kumar Rath

Asst. Prof Sumitra Kisan

Asst. Prof Gargi Bhattacharjee

SYLLABUS

Module 1: (10 Lectures)
C Language Fundamentals, Arrays and Strings
Character set, Identifiers, Keywords, Data Types, Constant and Variables, Statements, Expressions, Operators, Precedence of operators, Input – output Assignments, Control structures, Decision making and Branching, Decision making & looping. Declarations.

Module 2: (10 Lectures)
Monolithic vs Modular programs, User defined vs standard functions, formal vs Actual arguments, Functions category, function prototypes, parameter passing, Recursion, Storage Classes: Auto, Extern, Global, Static.Character handling in C. String handling functions. Pointers, Structures, Union & File handling

Module 3: (10 Lectures)
Pointer variable and its importance, Pointer Arithmetic passing parameters, Declaration of structures, pointer to pointer, pointer to structure, pointer to function, unions dynamic memory allocations, unions, file handling in C.

Module 4: (10 Lectures)
Development of Algorithms: Notations and Analysis, Storage structures for arrays-sparse matrices, Stacks and Queues: Applications of Stack: Prefix, Postfix and Infix expressions. Circular queue, Double ended queue.

CONTENTS

Module: 1

Module: 2

Module: 3

Module: 4

References

MODULE 1

LECTURE NOTE-1

A BEGINNER'S GUIDE

INTRODUCTION TO COMPUTERS

Any programming language is implemented on a computer. Right form its inception, to the present day, all computer system (irrespective of their shape & size) perform the following 5 basic operations. It converts the raw input data into information, which is useful to the users.

- *Inputting*: It is the process of entering data & instructions to the computer system.
- *Storing*: The data & instructions are stored for either initial or additional processing, as & when required.
- *Processing*: It requires performing arithmetic or logical operation on the saved data to convert it into useful information.
- *Outputting*: It is the process of producing the output data to the end user.
- *Controlling*: The above operations have to be directed in a particular sequence to be completed.

Based on these 5 operations, we can sketch the block diagram of a computer.

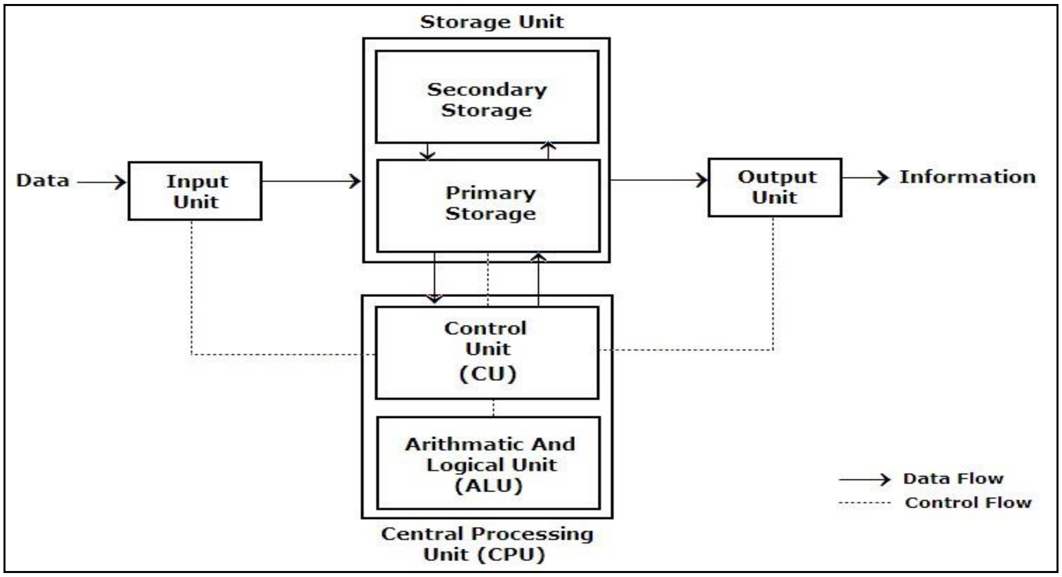


Fig 1: Block Diagram of a Computer

- *Input Unit*: We need to first enter the data & instruction in the computer system, before any computation begins. This task is accomplished by the input devices. (Eg: keyboard, mouse, scanner, digital camera etc). This device is responsible for linking the system with the external environment. The data accepted is in a human readable form. The input device converts it into a computer readable form.

- *Storage Unit*: The data & instruction that are entered have to be stored in the computer. Similarly, the end results & the intermediate results also have to be stored somewhere before being passed to the output unit. The storage unit provides solution to all these issues. This storage unit is designed to save the initial data, the intermediate result & the final result. This storage unit has 2 units: *Primary storage & Secondary storage.*

  Primary Storage: The primary storage, also called as the *main memory*, holds the data when the computer is currently on. As soon as the system is switched off or restarted, the information held in primary storage disappears (i.e. it is volatile in nature). Moreover, the primary storage normally has a limited storage capacity, because it is very expensive as it is made up of semiconductor devices.

  Secondary Storage: The secondary storage, also called as the *auxiliary storage,* handles the storage limitation & the volatile nature of the primary memory. It can retain information even when the system is off. It is basically used for holding the program instructions & data on which the computer is not working on currently, but needs to process them later.

- *Central Processing Unit*: Together the Control Unit & the Arithmetic Logic Unit are called as the Central Processing Unit (CPU). The CPU is the brain of the computer. Like in humans, the major decisions are taken by the brain itself & other body parts function as directed by the brain. Similarly in a computer system, all the major calculations & comparisons are made inside the CPU. The CPU is responsible for activating & controlling the operation of other units of the computer system.

  Arithmetic Logic Unit: The actual execution of the instructions (arithmetic or logical operations) takes place over here. The data & instructions stored in the primary storage are transferred as & when required. No processing is done in the primary storage. Intermediate results that are generated in ALU are temporarily transferred back to the primary storage, until needed later. Hence, data may move from the primary storage to ALU & back again to storage, many times, before the processing is done.

  Control Unit: This unit controls the operations of all parts of the computer but does not carry out any actual data processing.It is responsible for the transfer of data and instructions among other units of the computer.It manages and coordinates all the units of the system.It also communicates with Input/Output devices for transfer of data or results from the storage units.

- *Output Unit*: The job of an output unit is just the opposite of an input unit. It accepts the results produced by the computer in coded form. It converts these coded results to human readable form. Finally, it displays the converted results to the outside world with the help of output devices (Eg :monitors, printers, projectors etc..).

So when we talk about a computer, we actually mean 2 things:

- *Hardware*- This hardware is responsible for all the physical work of the computer.
- *Software*- This software commands the hardware what to do & how to do it.

Together, the hardware & software form the computer system.

This software is further classified as system software & application software.

System Software- System software are a set of programs, responsible for running the computer, controlling various operations of computer systems and management of computer resources. They act as an interface between the hardware of the computer & the application software. E.g.: Operating System

Application Software- Application software is a set of programs designed to solve a particular problem for users. It allows the end user to do something besides simply running the hardware. E.g.: Web Browser, Gaming Software, etc.

LECTURE NOTE-2

INTRODUCTION TO PROGRAMMING

A language that is acceptable to a computer system is called a ***computer language*** or ***programming language*** and the process of creating a sequence of instructions in such a language is called ***programming*** or ***coding***. A program is a set of instructions, written to perform a specific task by the computer. A set of large program is called ***software***. To develop software, one must have knowledge of a programming language.

Before moving on to any programming language, it is important to know about the various types of languages used by the computer. Let us first know what the basic requirements of the programmers were & what difficulties they faced while programming in that language.

COMPUTER LANGUAGES

Languages are a means of communication. Normally people interact with each other through a language. On the same pattern, communication with computers is carried out through a language. This language is understood both by the user and the machine. Just as every language like English, Hindi has its own grammatical rules; every computer language is also bounded by rules known as *syntax* of that language. The user is bound by that syntax while communicating with the computer system.

Computer languages are broadly classified as:

- ***Low Level Language***: The term low level highlights the fact that it is closer to a language which the machine understands.

  The low level languages are classified as:

  - ***Machine Language***: This is the language (in the form of 0's and 1's, called binary numbers) understood directly by the computer. It is machine dependent. It is difficult to learn and even more difficult to write programs.

  - ***Assembly Language***: This is the language where the machine codes comprising of 0'sand 1's are substituted by symbolic codes (called mnemonics) to improve their understanding. It is the first step to improve programming structure. Assembly language programming is simpler and less time consuming than machine level programming, it is easier to locate and correct errors in assembly language than in machine language programs. It is also machine dependent. Programmers must have knowledge of the machine on which the program will run.

- ***High Level Language:*** Low level language requires extensive knowledge of the hardware since it is machine dependent. To overcome this limitation, high level language has been evolved which uses normal English, which is easy to understand to solve any problem. High level languages are computer independent and programming becomes quite easy and simple. Various high level languages are given below:

  - BASIC (Beginners All Purpose Symbolic Instruction Code): It is widely used, easy to learn general purpose language. Mainly used in microcomputers in earlier days.
  - COBOL (Common Business Oriented language): A standardized language used for commercial applications.
  - FORTRAN (Formula Translation): Developed for solving mathematical and scientific problems. One of the most popular languages among scientific community.
  - C: Structured Programming Language used for all purpose such as scientific application, commercial application, developing games etc.
  - C++: Popular object oriented programming language, used for general purpose.

PROGRAMMING LANGUAGE TRANSLATORS

As you know that high level language is machine independent and assembly language though it is machine dependent yet mnemonics that are being used to represent instructions are not directly understandable by the machine. Hence to make the machine understand the instructions provided by both the languages, programming language instructors are used. They transform the instruction prepared by programmers into a form which can be interpreted & executed by the computer. Flowing are the various tools to achieve this purpose:

- ***Compiler***: The software that reads a program written in high level language and translates it into an equivalent program in machine language is called as compiler. The program written by the programmer in high level language is called source program and the program generated by the compiler after translation is called as object program.

- ***Interpreter***: it also executes instructions written in a high level language. Both complier & interpreter have the same goal i.e. to convert high level language into binary instructions, but their method of execution is different. The complier converts the entire source code into machine level program, while the interpreter takes 1 statement, translates it, executes it & then again takes the next statement.

- ***Assembler***: The software that reads a program written in assembly language and translates it into an equivalent program in machine language is called as assembler.

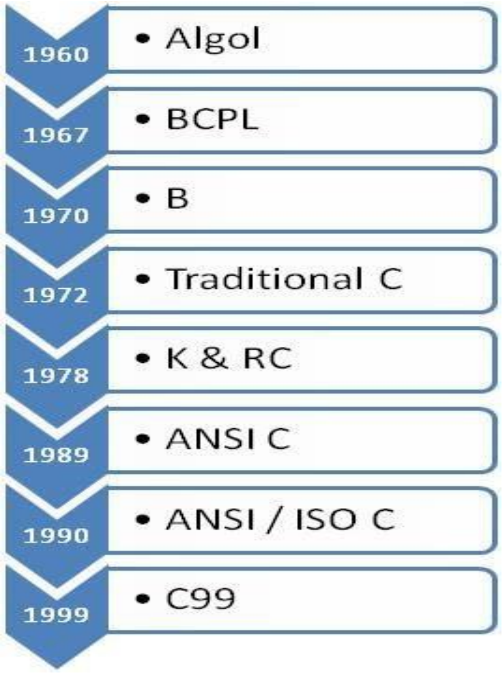
- ***Linker***: A linker or link editor is a computer program that takes one or more object files generated by a compiler and combines them into a single executable file, library file, or another object file.

LECTURE NOTE -3

INTRODUCTION TO C

Brief History of C

- The C programming language is a structure oriented programming language, developed at Bell Laboratories in 1972 by Dennis Ritchie.
- C programming language features were derived from an earlier language called "B" (Basic Combined Programming Language – BCPL)
- C language was invented for implementing UNIX operating system.
- In 1978, Dennis Ritchie and Brian Kernighan published the first edition "The C Programming Language" and is commonly known as K&R C.
- In 1983, the American National Standards Institute (ANSI) established a committee to provide a modern, comprehensive definition of C. The resulting definition, the ANSI standard, or "ANSI C", was completed late 1988.
- Many of C's ideas & principles were derived from the earlier language B, thereby naming this new language "C".



Taxonomy of C Language

WHY IS C POPULAR

- It is reliable, simple and easy to use.
- C is a small, block-structured programming language.
- C is a portable language, which means that C programs written on one system can be run on other systems with little or no modification.
- C has one of the largest assortments of operators, such as those used for calculations and data comparisons.
- Although the programmer has more freedom with data storage, the languages do not check data type accuracy for the programmer.

WHY TO STUDY C

- By the early 1980s, C was already a dominant language in the minicomputer world of Unix systems. Since then, it has spread to personal computers (microcomputers) and to mainframes.
- Many software houses use C as the preferred language for producing word processing programs, spreadsheets, compilers, and other products.
- C is an extremely flexible language—particularly if it is to be used to write operating systems.
- Unlike most other languages that have only four or five levels of precedence, C has 15.

CHARECTERESTICS OF A C PROGRAM

- Middle level language.

High Level	*Middle Level*	*Low Level*
High level languages provide almost everything that the programmer might need to do as already built into the language	Middle level languages don't provide all the built-in functions found in high level languages, but provides all building blocks that we need to produce the result we want	Low level languages provides nothing other than access to the machines basic instruction set
Examples: Java, Python	C, C++	Assembler

- Small size – has only 32 keywords
- Extensive use of function calls- enables the end user to add their own functions to the C library.
- Supports loose typing – a character can be treated as an integer & vice versa.
- Structured language

Structure oriented	*Object oriented*	*Non structure*
In this type of language, large programs are divided into small programs called functions	In this type of language, programs are divided into objects	There is no specific structure for programming this language
Prime focus is on functions and procedures that operate on the data	Prime focus is in the data that is being operated and not on the functions or procedures	N/A
Data moves freely around the systems from one function to another	Data is hidden and cannot be accessed by external functions	N/A
Program structure follows "Top Down Approach"	Program structure follows "Bottom UP Approach"	N/A
Examples: C, Pascal, ALGOL and Modula-2	C++, JAVA and C# (C sharp)	BASIC, COBOL, FORTRAN

- Low level (Bit Wise) programming readily available
- Pointer implementation - extensive use of pointers for memory, array, structures and functions.
- It has high-level constructs.
- It can handle low-level activities.
- It produces efficient programs.
- It can be compiled on a variety of computers.

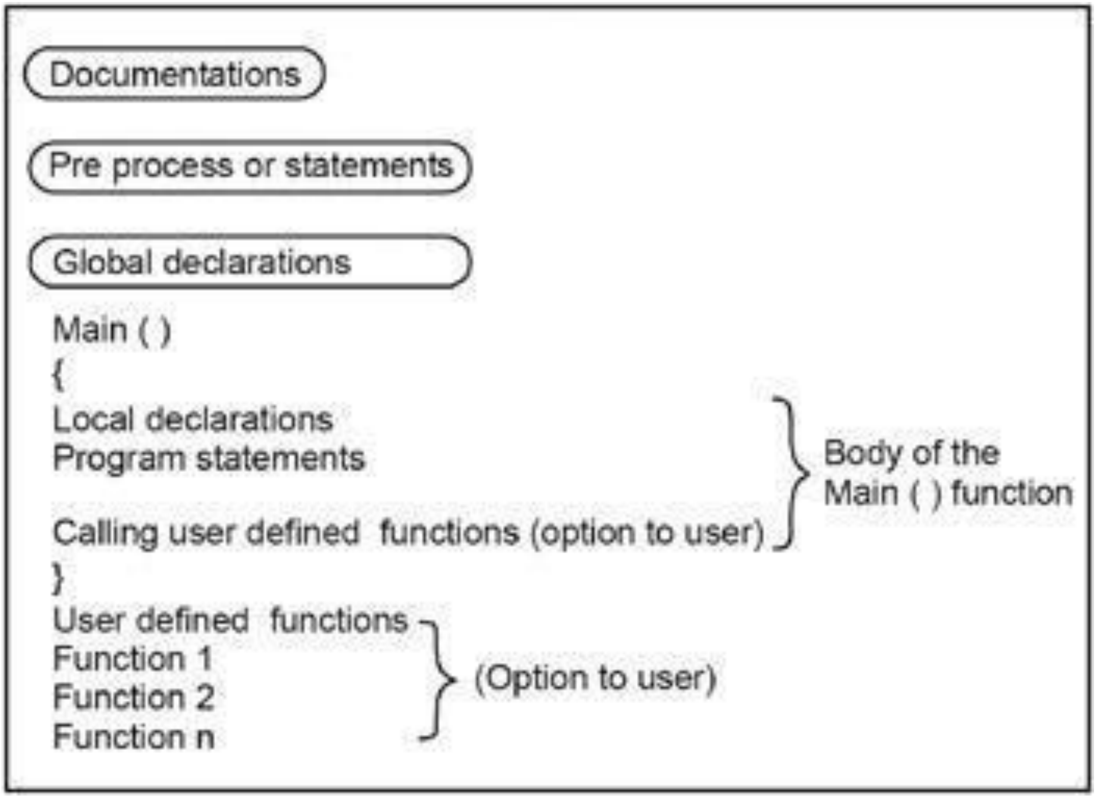
USES

The C programming language is used for developing system applications that forms a major portion of operating systems such as Windows, UNIX and Linux. Below are some examples of C being used:

- Database systems
- Graphics packages
- Word processors
- Spreadsheets
- Operating system development
- Compilers and Assemblers
- Network drivers
- Interpreters

STRUCTURE OF A C PROGRAM

The structure of a C program is a protocol (rules) to the programmer, which he has to follow while writing a C program. The general basic structure of C program is shown in the figure below.



Based on this structure, we can sketch a C program.

Example:

```
/* This program accepts a number & displays it to the user*/

#include <stdio.h>
void main(void)
{ int number;
printf( "Please enter a number: " );
scanf( "%d", &number );
printf( "You entered %d", number );
return 0;}
```

Stepwise explanation:

#include

- The part of the compiler which actually gets your program from the source file is called the preprocessor.
    - *#include <stdio.h>*
- #include is a pre-processor directive. It is not really part of our program, but instead it is an instruction to the compiler to make it do something. It tells the C compiler to include the contents of a file (in this case the system file called stdio.h).
- The compiler knows it is a system file, and therefore must be looked for in a special place, by the fact that the filename is enclosed in <> characters

<stdio.h>

- *stdio.h* is the name of the standard library definition file for all STanDard Input and Output functions.
- Your program will almost certainly want to send information to the screen and read things from the keyboard, and stdio.h is the name of the file in which the functions that we want to use are defined.
- The function we want to use is called printf. The actual code of printf will be tied in later by the linker.
- The ".h" portion of the filename is the language extension, which denotes an include file.

void

- This literally means that this means nothing. In this case, it is referring to the function whose name follows.
- Void tells to C compiler that a given entity has no meaning, and produces no error.

main

- In this particular example, the only function in the program is called *main*.
- A C program is typically made up of large number of functions. Each of these is given a name by the programmer and they refer to each other as the program runs.
- C regards the name main as a special case and will run this function first i.e. the program execution starts from *main*.

(void)

- This is a pair of brackets enclosing the keyword *void*.
- It tells the compiler that the function main has no parameters.
- A parameter to a function gives the function something to work on.

{ (Brace)

- This is a brace (or curly bracket). As the name implies, braces come in packs of two - for every open brace there must be a matching close one.
- Braces allow us to group pieces of program together, often called a block.
- A block can contain the declaration of variable used within it, followed by a sequence of program statements.
- In this case the braces enclose the working parts of the function main.

; (semicolon)

- The semicolon marks the end of the list of variable names, and also the end of that declaration statement.
- All statements in C programs are separated by ";" (semicolon) characters.
- The ";" character is actually very important. It tells the compiler where a given statement ends.
- If the compiler does not find one of these characters where it expects to see one, then it will produce an error.

scanf

- In other programming languages, the printing and reading functions are a part of the language.
- In C this is not the case; instead they are defined as standard functions which are part of the language specification, but are not a part of the language itself.
- The standard input/output library contains a number of functions for formatted data transfer; the two we are going to use are scanf (scan formatted) and printf (print formatted).

printf

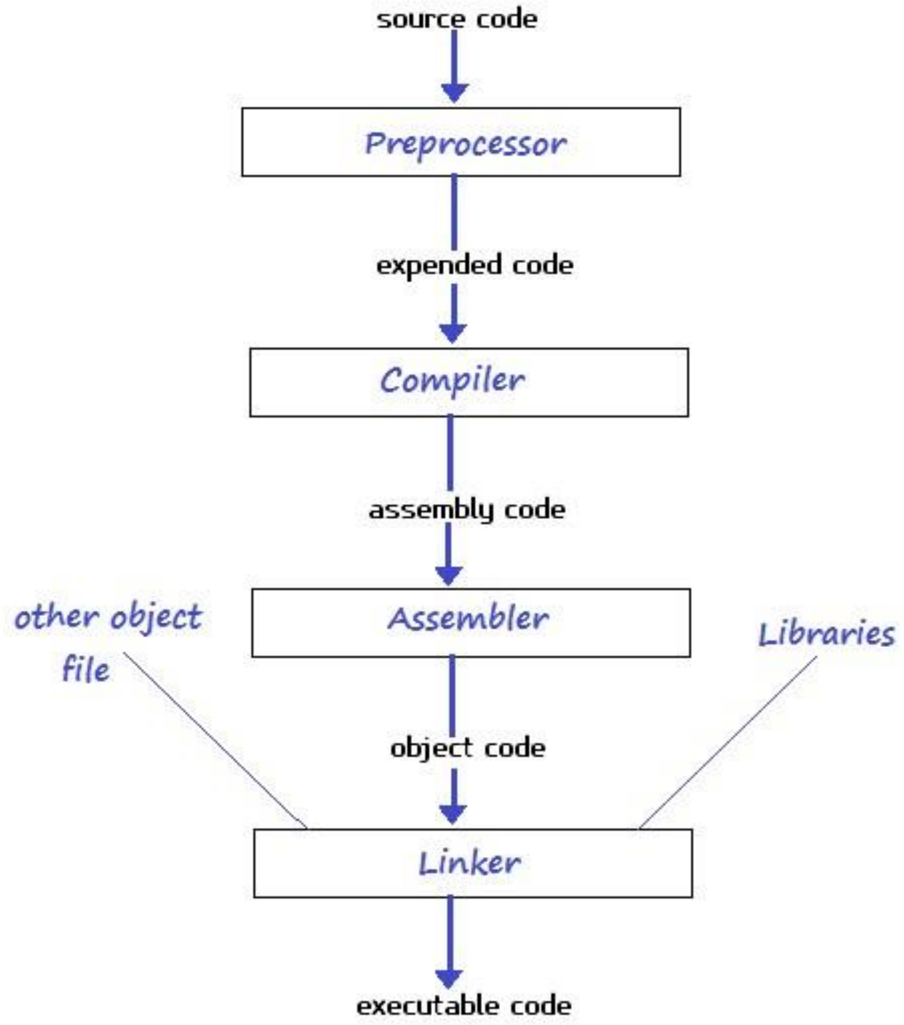
- The printf function is the opposite of scanf.
- It takes text and values from within the program and sends it out onto the screen.
- Just like scanf, it is common to all versions of C and just like scanf, it is described in the system file stdio.h.
- The first parameter to a printf is the format string, which contains text, value descriptions and formatting instructions.

FILES USED IN A C PROGRAM

- ***Source File-*** This file contains the source code of the program. The file extension of any c file is **.c**. The file contains C source code that defines the *main* function & maybe other functions.

- ***Header File-*** A header file is a file with extension **.h** which contains the C function declarations and macro definitions and to be shared between several source files.
- ***Object File-*** An object file is a file containing object code, with an extension **.o**, meaning relocatable format machine code that is usually not directly executable. Object files are produced by an assembler, compiler, or other language translator, and used as input to the linker, which in turn typically generates an executable or library by combining parts of object files.
- ***Executable File-*** The binary executable file is generated by the linker. The linker links the various object files to produce a binary file that can be directly executed.

COMPLIATION & EXECUTION OF A C PROGRAM



ELEMENTS OF C

Every language has some basic elements & grammatical rules. Before starting with programming, we should be acquainted with the basic elements that build the language.

Character Set

Communicating with a computer involves speaking the language the computer understands. In C, various characters have been given to communicate.

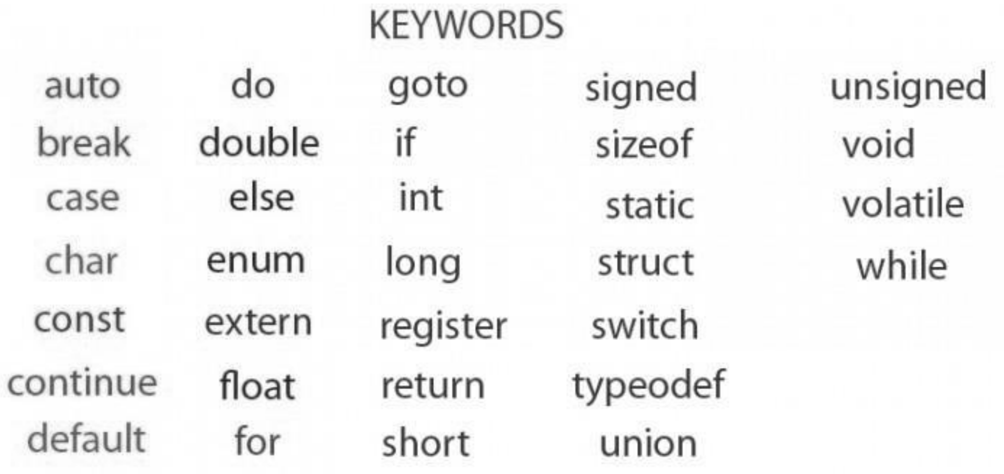
Character set in C consists of;

Types	**Character Set**
Lower case	a-z
Upper case	A-Z
Digits	0-9
Special Character	!@#$%^&*
White space	Tab or new lines or space

Keywords

Keywords are the words whose meaning has already been explained to the C compiler. The keywords cannot be used as variable names because if we do so we are trying to assign a new meaning to the keyword, which is not allowed by the computer.

There are only 32 keywords available in C. Below figure gives a list of these keywords for your ready reference.



KEYWORDS

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

Identifier

In the programming language C, an identifier is a combination of alphanumeric characters, the first being a letter of the alphabet or an underline, and the remaining being any letter of the alphabet, any numeric digit, or the underline.

Two rules must be kept in mind when naming identifiers.

1. The case of alphabetic characters is significant. Using "INDEX" for a variable is not the same as using "index" and neither of them is the same as using "InDeX" for a variable. All three refer to different variables.
2. As C is defined, up to 32 significant characters can be used and will be considered significant by most compilers. If more than 32 are used, they will be ignored by the compiler.

Data Type

In the C programming language, data types refer to a domain of allowed values & the operations that can be performed on those values. The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted. There are 4 fundamental data types in C, which are- *char, int, float &, double. Char* is used to store any single character; *int* is used to store any integer value, *float* is used to store any single precision floating point number & *double* is used to store any double precision floating point number. We can use 2 qualifiers with these basic types to get more types.

There are 2 types of qualifiers-

Sign qualifier- signed & unsigned
Size qualifier- short & long

The data types in C can be classified as follows:

Type	**Storage size**	**Value range**
char	1 byte	-128 to 127
unsigned char	1 byte	0 to 255
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535

long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

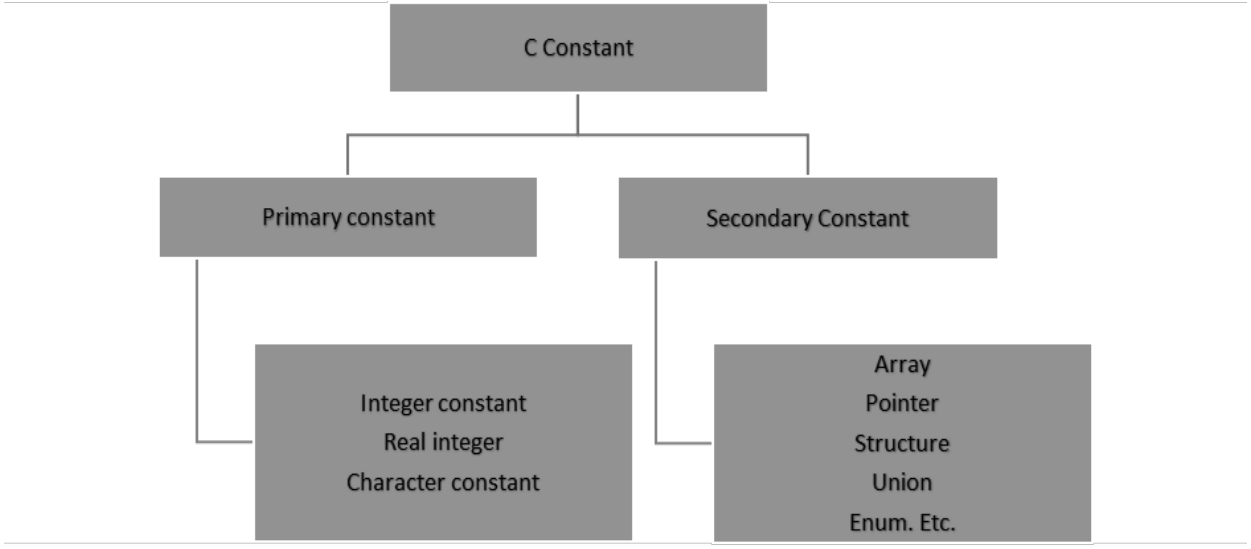
Type	**Storage size**	**Value range**	**Precision**
float	4 bytes	1.2E-38 to 3.4E+38	6 decimal places
double	8 bytes	2.3E-308 to 1.7E+308	15 decimal places
long double	10 bytes	3.4E-4932 to 1.1E+4932	19 decimal places

Constants

A constant is an entity that doesn’t change whereas a variable is an entity that may change.

C constants can be divided into two major categories:

- Primary Constants
- Secondary Constants



Here our only focus is on primary constant. For constructing these different types of constants certain rules have been laid down.

Rules for Constructing Integer Constants:

An integer constant must have at least one digit.

a) It must not have a decimal point.
b) It can be either positive or negative.

c) If no sign precedes an integer constant it is assumed to be positive.

d) No commas or blanks are allowed within an integer constant.

e) The allowable range for integer constants is -32768to 32767.

Ex.: 426, +782,-8000, -7605

Rules for Constructing Real Constants:

Real constants are often called Floating Point constants. The real constants could be written in two forms—Fractional form and Exponential form.

Rules for constructing real constants expressed in fractional form:

a) A real constant must have at least one digit.

b) It must have a decimal point.

c) It could be either positive or negative.

d) Default sign is positive.

e) No commas or blanks are allowed within a real constant.

Ex. +325.34, 426.0, -32.76, -48.5792

Rules for constructing real constants expressed in exponential form:

a) The mantissa part and the exponential part should be separated by a letter e.

b) The mantissa part may have a positive or negative sign.

c) Default sign of mantissa part is positive.

d) The exponent must have at least one digit, which must be a positive or negative integer. Default sign is positive.

e) Range of real constants expressed in exponential form is -3.4e38 to 3.4e38.

Ex. +3.2e-5, 4.1e8, -0.2e+3, -3.2e-5

Rules for Constructing Character Constants:

a) A character constant is a single alphabet, a single digit or a single special symbol enclosed within single inverted commas.

b) The maximum length of a character constant can be 1 character.

Ex.: ‘M’, ‘6’, ‘+’

LECTURE NOTE-5

VARIABLES

Variables are names that are used to store values. It can take different values but one at a time. A data type is associated with each variable & it decides what values the variable can take. When you decide your program needs another variable, you simply declare (or define) a new variable and C makes sure you get it. You declare all C variables at the top of whatever blocks of code need them. Variable declaration requires that you inform C of the variable's name and data type. Syntax – datatype variablename;

Eg:

```
int page_no;

char grade;

float salary;

long y;
```

- **Declaring Variables:**

There are two places where you can declare a variable:

- After the opening brace of a block of code (usually at the top of a function)
- Before a function name (such as before main() in the program) Consider various examples:

  Suppose you had to keep track of a person's first, middle, and last initials. Because an initial is obviously a character, it would be prudent to declare three character variables to hold the three initials. In C, you could do that with the following statement:

1.

```
main()

{

char first, middle, last;

 // Rest of program follows

}
```

2. *main()*

```
{ char first;
 char middle;
char last;
        // Rest of program follows
}
```

- **Initialization of Variables**

  When a variable is declared, it contains undefined value commonly known as garbage value. If we want we can assign some initial value to the variables during the declaration itself. This is called *initialization of the variable.*

  Eg- int pageno=10;

  char grade='A';

  float salary= 20000.50;

Expressions

An expression consists of a combination of operators, operands, variables & function calls. An expression can be arithmetic, logical or relational. Here are some expressions:

a+b – arithmetic operation

a>b- relational operation a

== b - logical operation

func (a,b) – function call

4+21

a*(b + c/d)/20

q = 5*2 x =

++q % 3

q > 3

As you can see, the operands can be constants, variables, or combinations of the two. Some expressions are combinations of smaller expressions, called subexpressions. For example, c/d is a subexpression of the sixth example.

An important property of C is that every C expression has a value. To find the value, you perform the operations in the order dictated by operator precedence.

Statements

Statements are the primary building blocks of a program. A program is a series of statements with some necessary punctuation. A statement is a complete instruction to the computer. In C, statements are indicated by a semicolon at the end. Therefore

legs = 4

is just an expression (which could be part of a larger expression), but

legs = 4;

is a statement.

What makes a complete instruction? First, C considers any expression to be a statement if you append a semicolon. (These are called expression statements.) Therefore, C won't object to lines such as the following:

8;

3 + 4;

However, these statements do nothing for your program and can't really be considered sensible statements. More typically, statements change values and call functions:

x = 25;

++x;

y = sqrt(x);

Although a statement (or, at least, a sensible statement) is a complete instruction, not all complete instructions are statements. Consider the following statement:

x = 6 + (y = 5);

In it, the subexpression y = 5 is a complete instruction, but it is only part of the statement. Because a complete instruction is not necessarily a statement, a semicolon is needed to identify instructions that truly are statements.

Compound Statements (Blocks)

A compound statement is two or more statements grouped together by enclosing them in braces; it is also called a block. The following while statement contains an example:

```
while (years < 100)

{

   wisdom = wisdom * 1.05;

printf("%d %d\n", years, wisdom);

years = years + 1;


}
```

If any variable is declared inside the block then it can be declared only at the beginning of the block. The variables that are declared inside a block can be used only within the block.

LECTURE NOTE -6

INPUT-OUTPUT IN C

When we are saying **Input** that means we feed some data into program. This can be given in the form of file or from command line. C programming language provides a set of built-in functions to read given input and feed it to the program as per requirement.

When we are saying **Output** that means to display some data on screen, printer or in any file. C programming language provides a set of built-in functions to output the data on the computer screen.

Functions *printf()* and *scanf()* are the most commonly used to display out and take input respectively. Let us consider an example:

```
#include <stdio.h>      //This is needed to run printf() function.
int main()
{
   printf("C Programming");  //displays the content inside quotation
return 0;
}
```

Output:

```
C Programming
```

Explanation:

- Every program starts from *main()* function.
- *printf()* is a library function to display output which only works if *#include<stdio.h>*is included at the beginning.
- Here, *stdio.h* is a header file (standard input output header file) and *#include* is command to paste the code from the header file when necessary. When compiler encounters *printf()*function and doesn't find *stdio.h* header file, compiler shows error.
- *return 0*; indicates the successful execution of the program.

Input- Output of integers in C

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

Output

```
Number=5
```

Inside quotation of *printf()* there, is a conversion format string "%d" (for integer). If this conversion format string matches with remaining argument, i.e, *c* in this case, value of *c* is displayed.

```
#include<stdio.h>
int main()
{  int c;
   printf("Enter a number\n");
   scanf("%d",&c);
   printf("Number=%d",c);
   return 0;
}
```

Output

```
Enter a number
4
Number=4
```

The *scanf()* function is used to take input from user. In this program, the user is asked an input and value is stored in variable *c*. Note the '&' sign before *c*. &c denotes the address of *c* and value is stored in that address.

Input- Output of floats in C

```
#include <stdio.h>
int main()
{
        float a;
        printf("Enter value: ");
        scanf("%f",&a);
        printf("Value=%f",a);    //%f is used for floats instead of %d
        return 0;
}
```

Output

```
Enter value: 23.45
Value=23.450000
```

Conversion format string "%f" is used for floats to take input and to display floating value of a variable.

Input - Output of characters and ASCII code

```
#include <stdio.h>
int main()
{
   char var1;
   printf("Enter character: ");
   scanf("%c",&var1);
   printf("You entered %c.",var1);
   return 0;
}
```

Output

```
Enter character: g
You entered g.
```

Conversion format string "%c" is used in case of characters.

ASCII code

When character is typed in the above program, the character itself is not recorded a numeric value (ASCII value) is stored. And when we displayed that value by using "%c", that character is displayed.

```
#include <stdio.h>
int main()
{
char var1;
printf("Enter character: ");
scanf("%c",&var1);
printf("You entered %c.\n",var1);
/* \n prints the next line(performs work of enter). */
printf("ASCII value of %d",var1);
return 0;
}
```

Output:

```
Enter character:
g
103
```

When, *'g'* is entered, ASCII value 103 is stored instead of *g*.

You can display character if you know ASCII code only. This is shown by following example.

```
#include <stdio.h>
int main()
{
int var1=69;
printf("Character of ASCII value 69: %c",var1);
return 0;
}
```

Output

```
Character of ASCII value 69: E
```

The ASCII value of 'A' is 65, 'B' is 66 and so on to 'Z' is 90. Similarly ASCII value of 'a' is 97, 'b' is 98 and so on to 'z' is 122.

LECTURE NOTE-7

FORMATTED INPUT-OUTPUT

Data can be entered & displayed in a particular format. Through format specifications, better presentation of results can be obtained.

Variations in Output for integer & floats:

```
#include<stdio.h>
int main()
{
   printf("Case 1:%6d\n",9876);
/*  Prints the number right justified within 6 columns  */
printf("Case 2:%3d\n",9876);
/* Prints the number to be right justified to 3 columns but, there are 4 digits so number is not
right justified  */
   printf("Case 3:%.2f\n",987.6543);
/* Prints the number rounded to two decimal places */
printf("Case 4:%.f\n",987.6543);
/* Prints the number rounded to 0 decimal place, i.e, rounded to integer */
printf("Case 5:%e\n",987.6543);
/* Prints the number in exponential notation (scientific notation) */
return 0;
}
```

Output

```
Case 1:  9876
Case 2:9876
Case 3:987.65
Case 4:988
Case 5:9.876543e+002
```

Variations in Input for integer and floats:

```
#include <stdio.h>
int main()
{
int a,b;
float c,d;
printf("Enter two intgers: ");
/*Two integers can be taken from user at once as below*/
```

```
 scanf("%d%d",&a,&b);
 printf("Enter intger and floating point numbers: ");
/*Integer and floating point number can be taken at once from user as below*/
scanf("%d%f",&a,&c);
return 0;
}
```

Similarly, any number of inputs can be taken at once from user.

EXERCISE:

1. To print out a and b given below, which of the following printf() statement will you use?

```
#include<stdio.h>
float a=3.14;
double b=3.14;
```

   A. printf("%f %lf", a, b);
   B. printf("%Lf %f", a, b);
   C. printf("%Lf %Lf", a, b);
   D. printf("%f %Lf", a, b);

2. To scan a and b given below, which of the following scanf() statement will you use?

```
#include<stdio.h>
float a;
double b;
```

   A. scanf("%f %f", &a, &b);
   B. scanf("%Lf %Lf", &a, &b);
   C. scanf("%f %Lf", &a, &b);
   D. scanf("%f %lf", &a, &b);

3. For a typical program, the input is taken using.
   A. scanf
   B. Files
   C. Command-line
   D. None of the mentioned

4. What is the output of this C code?

```
#include <stdio.h>
int main()
{   int i = 10, j = 2;
    printf("%d\n", printf("%d %d ", i, j));
}
```

A. Compile time error
B. 10 2 4
C. 10 2 2
D. 10 2 5

5. What is the output of this C code?

```
#include <stdio.h>
int main()
{
        int i = 10, j = 3;
        printf("%d %d %d", i, j);
}
```

A. Compile time error
B. 10 3
C. 10 3 some garbage value
D. Undefined behavior

6. What is the output of this C code?

```
#include <stdio.h>
int main()
{     int i = 10, j = 3, k = 3;
     printf("%d %d ", i, j, k);
}
```

A. Compile time error
B. 10 3 3
C. 10 3
D. 10 3 somegarbage value

7. The syntax to print a % using printf statement can be done by.
   A. %
   B. %
   C. ‘%’
   D. %%

8. What is the output of this C code?

```
#include <stdio.h>
int main()
{  int n;
   scanf("%d", n);
   printf("%d\n", n);
   return 0;
}
```

   A. Compilation error
   B. Undefined behavior
   C. Whatever user types
   D. Depends on the standard

9. What is the output of this C code?

```
#include <stdio.h>
int main()
{  short int i;
   scanf("%hd", &i);
   printf("%hd", i);
  return 0;
}
```

   A. Compilation error
   B. Undefined behavior
   C. Whatever user types
   D. None of the mentioned

10. In a call to printf() function the format specifier %b can be used to print binary equivalent of an integer.
    A. True
    B. False

11. Point out the error in the program?

```
#include<stdio.h>
int main()
{
    char ch;
    int i;
   scanf("%c", &i);
  scanf("%d", &ch);
  printf("%c %d", ch, i);
 return 0;
}
```

A. Error: suspicious char to in conversion in scanf()
B. Error: we may not get input for second scanf() statement
C. No error
D. None of above

12. Which of the following is NOT a delimiter for an input in scanf?

A. Enter
B. Space
C. Tab
D. None of the mentioned

OPERATORS

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C language is rich in built-in operators and provides the following types of operators:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Increment and decrement operators
- Conditional operators
- Misc Operators

Arithmetic operator:

These are used to perform mathematical calculations like addition, subtraction, multiplication, division and modulus.

Following table shows all the arithmetic operators supported by C language. Assume variable A holds 10 and variable B holds 20 then:

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A – B will give -10
*	Multiplies both operands	A * B will give 200
/	Divides numerator by de-numerator	B / A will give 2

%	Modulus Operator and remainder of after an integer division	B % A will give 0
++	Increments operator increases integer value by one	A++ will give 11
--	Decrements operator decreases integer value by one	A–will give 9

Relational Operators:

These operators are used to compare the value of two variables.

Following table shows all the relational operators supported by C language. Assume variable A holds 10 and variable B holds 20, then:

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

Logical Operators:

These operators are used to perform logical operations on the given two variables.

Following table shows all the logical operators supported by C language. Assume variable A holds 1 and variable B holds 0, then:

Operator	**Description**	**Example**
&&	Called Logical AND operator. If both the operands are nonzero, then condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true.

Bitwise Operators

Bitwise operator works on bits and performs bit-by-bit operation. Bitwise operators are used in bit level programming. These operators can operate upon *int* and *char* but not on *float* and *double*.

Showbits( ) function can be used to display the binary representation of any integer or character value.

Bit wise operators in C language are; & (bitwise AND), | (bitwise OR), ~ (bitwise OR), ^ (XOR), << (left shift) and >> (right shift).

The truth tables for &, |, and ^ are as follows:

p	***q***	**p & q**	**p | q**	**p ^ q**
0	*0*	0	0	0
0	*1*	0	1	1
1	*1*	1	1	0
1	*0*	0	1	1

The Bitwise operators supported by C language are explained in the following table. Assume variable A holds 60 (00111100) and variable B holds 13 (00001101), then:

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12, which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61, which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49, which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61, which is 1100 0011 in 2's complement form.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 0000 1111

Assignment Operators:

In C programs, values for the variables are assigned using assignment operators.

There are following assignment operators supported by C language:

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A

-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C – A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator	C |= 2 is same as C = C | 2

LECTURE NOTE-9

INCREMENT AND DECREMENT OPERATOR

In C, ++ and – are called increment and decrement operators respectively. Both of these operators are unary operators, i.e, used on single operand. ++ adds 1 to operand and – subtracts 1 to operand respectively. For example:

```
Let a=5 and b=10

a++;    //a becomes 6

a--;    //a becomes 5

++a;    //a becomes 6

--a;    //a becomes 5
```

When i++ is used as prefix(like: ++var), ++var will increment the value of var and then return it but, if ++ is used as postfix(like: var++), operator will return the value of operand first and then only increment it. This can be demonstrated by an example:

```
#include <stdio.h>

int main()

{

 int c=2,d=2;

printf("%d\n",c++); //this statement displays 2 then, only c incremented by 1 to 3.

Printf("%d",++c);   //this statement increments 1 to c then, only c is displayed.

Return 0;

}
```

Output

```
   2

   4
```

Conditional Operators (? :)

Conditional operators are used in decision making in C programming, i.e, executes different statements according to test condition whether it is either true or false.

Syntax of conditional operators;

conditional_expression?expression1:expression2

If the test condition is true (that is, if its value is non-zero), expression1 is returned and if false expression2 is returned.

Let us understand this with the help of a few examples:

```
        int x, y ;

        scanf ( "%d", &x ) ;

        y = ( x> 5 ? 3 : 4 ) ;
```

This statement will store 3 in y if x is greater than 5, otherwise it will store 4 in y.

The equivalent if statement will be,

```
if ( x > 5 )

        y = 3 ;

else

        y = 4 ;
```

Misc Operators:

There are few other operators supported by c language.

Operator	**Description**	**Example**
sizeof()	It is a unary operator which is used in finding the size of data type, constant, arrays, structure etc.	sizeof(a), where a is integer, will return 4.

&	Returns the address of a variable.	&a; will give actual address of the variable.
*	Pointer to a variable.	*a; will pointer to a variable.

Operators Precedence in C

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator.

For example x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

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

Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

LECTURE NOTE -10

CONTROL STATEMENTS

In C, programs are executed sequentially in the order of which they appear. This condition does not hold true always. Sometimes a situation may arise where we need to execute a certain part of the program. Also it may happen that we may want to execute the same part more than once. Control statements enable us to specify the order in which the various instructions in the program are to be executed. They define how the control is transferred to other parts of the program. Control statements are classified in the following ways:

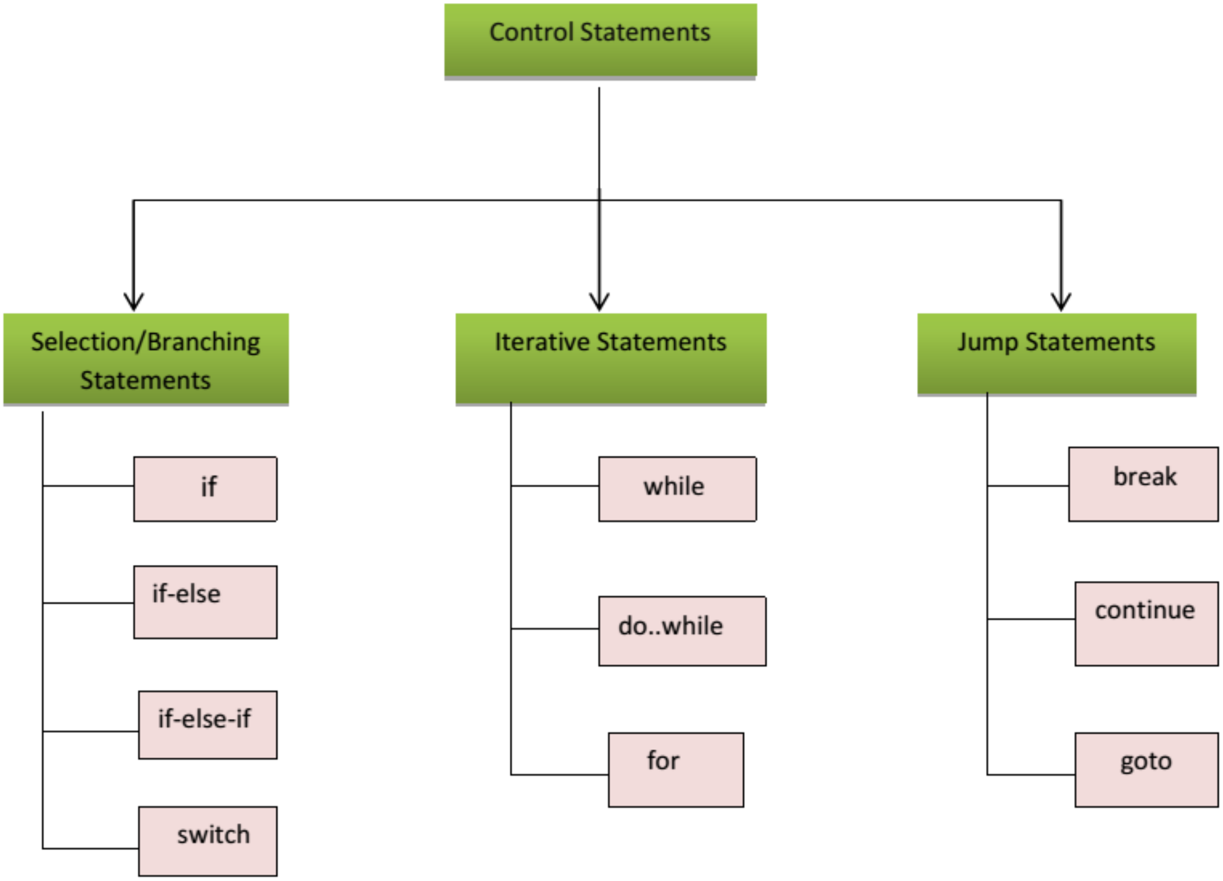


Fig: 1 Classification of control statements

SELECTION STATEMENTS

The selection statements are also known as *Branching* or *Decision Control Statements.*

Introduction to Decision Control Statements

Sometime we come across situations where we have to make a decision. E.g. *If the weather is sunny, I will go out & play, else I will be at home.* Here my course of action is governed by the kind of weather. If it's sunny, I can go out & play, else I have to stay indoors. I choose an option out of 2 alternate options. Likewise, we can find ourselves in situations where we have to select among several alternatives. We have decision control statements to implement this logic in computer programming.

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

if Statement

The keyword *if* tells the compiler that what follows is a decision control instruction. The *if* statement allows us to put some decision -making into our programs. The general form of the *if* statement is shown Fig 2:

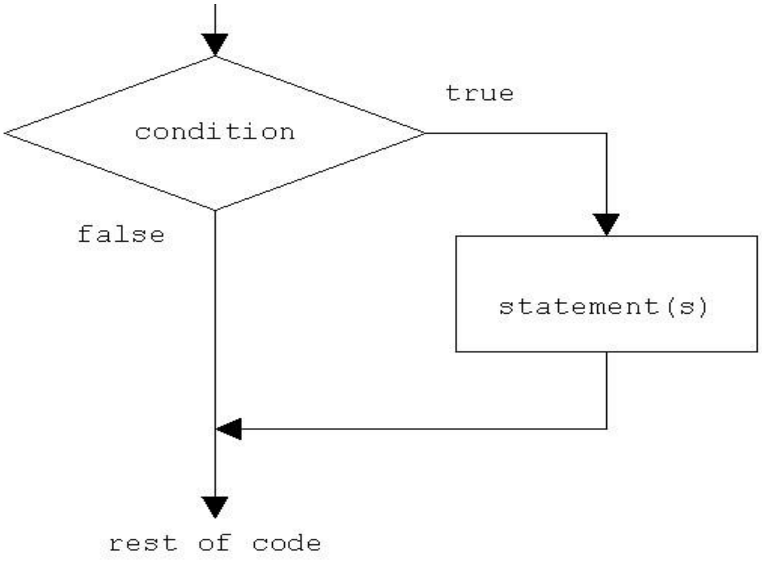


Fig 2: *if* statement construct

Syntax of if statement:

```
        if (condition )
        {
                Statement 1;
                …………..
                Statement n;
         }
                //Rest of the code
```

If the condition is true(nonzero), the statement will be executed. If the condition is false(0), the statement will not be executed. For example, suppose we are writing a billing program.

```
 if (total_purchase >=1000)

        printf("You are gifted a pen drive.\n");
```

Multiple statements may be grouped by putting them inside curly braces {}. For example:

```
         if (total_purchase>=1000)

        {

                gift_count++;

                printf("You are gifted a pen drive.\n");

        }
```

For readability, the statements enclosed in {} are usually indented. This allows the programmer to quickly tell which statements are to be conditionally executed. As we will see later, mistakes in indentation can result in programs that are misleading and hard to read.

Programs:

1. Write a program to print a message if negative no is entered.

```
        #include<stdio.h>
        int main()
        {
                int no;
                printf("Enter a no : ");
                scanf("%d", &no);
                if(no<0)
                 {
                        printf("no entered is negative");
                        no = -no;
                  }
                printf("value of no is %d \n",no);
                return 0;
        }
```

Output:

```
Enter a no: 6

value of no is 6
```

Output:

```
Enter a no: -2

value of no is 2
```

2. Write a program to perform division of 2 nos

```
#include<stdio.h>
int main()
{
        int a,b;
        float c;
    printf("Enter 2 nos : ");
   scanf("%d %d", &a, &b);
   if(b == 0)
         {
                 printf("Division is not possible");
          }
        c = a/b;
        printf("quotient is %f \n",c);
        return 0;
}
```

```
Output:
Enter 2 nos:  6 2
quotient is 3
```

```
Output:
Enter 2 nos:  6 0
Division is not possible
```

if-else Statement

The *if* statement by itself will execute a single statement, or a group of statements, when the expression following *if* evaluates to true. By using *else* we execute another group of statements if the expression evaluates to false.

```
if (a > b)
        { z = a;
         printf(“value of z is :%d”,z);
        }
else
        { z = b;
         printf(“value of z is :%d”,z);
        }
```

The group of statements after the *if* is called an ‘if block’. Similarly, the statements after the else form the ‘else block’.

Programs:

3. Write a program to check whether the given no is even or odd

```
#include<stdio.h>
int main()
 {
         int n;
         printf("Enter an integer\n");
         scanf("%d",&n);
        if ( n%2 == 0 )
                printf("Even\n");
        else
```

```
                        printf("Odd\n");
            return 0;
}
```

Output:

```
Enter an integer 3
Odd
```

Output:

```
Enter an integer 4
Even
```

4. Write a program to check whether a given year is leap year or not

```
#include <stdio.h>
int main()
{
  int year;
  printf("Enter a year to check if it is a leap year\n");
 scanf("%d", &year);
if ( (year%4 == 0) && (( year%100 != 0) || ( year%400 == 0 ))
           printf("%d is a leap year.\n", year);
 else
           printf("%d is not a leap year.\n", year);

 return 0;
}
```

Output:

```
Enter a year to check if it is a leap year 1996
1996 is a leap year
```

Output:

```
Enter a year to check if it is a leap year 2015
2015 is not a leap year
```

Nested *if-else*

An entire *if-else* construct can be written within either the body of the *if* statement or the body of an *else* statement. This is called 'nesting' of *if*s. This is shown in the following structure.

```
 if (n > 0)
{
          if (a > b)
                  z = a;
}
else
        z = b;
```

The second *if* construct is nested in the first *if* statement. If the condition in the first *if* statement is true, then the condition in the second *if* statement is checked. If it is false, then the *else* statement is executed.

Program:

5. Write a program to check for the relation between 2 nos

```
#include <stdio.h>
int main()
{
  int m=40,n=20;
  if ((m >0 ) && (n>0))
  {
          printf("nos are positive");
          if (m>n)
         {
                 printf("m is greater than n");
          }
         else
         {
                 printf("m is less than n");
         }
}
else
{
        printf("nos are negative");
 }
return 0;
}
```

Output

```
40 is greater than 20
```

else-if Statement:

This sequence of if statements is the most general way of writing a multi−way decision. The expressions are evaluated in order; if an expression is true, the statement associated with it is executed, and this terminates the whole chain. As always, the code for each statement is either a single statement, or a group of them in braces.

```
If (expression)
        statement
else if (expression)
        statement
else if (expression)
        statement
else if (expression)
        statement
else
        statement
```

The last *else* part handles the ``none of the above'' or default case where none of the other conditions is satisfied. Sometimes there is no explicit action for the default; in that case the trailing can be omitted, or it may be used for error checking to catch an “impossible” condition.

Program:

6. The above program can be used as an eg here.

```
#include <stdio.h>
int main()
{
 int m=40,n=20;
if (m>n)
    {
              printf("m is greater than n");
    }
 else if(m<n)
    {
              printf("m is less than n");
    }
 else
    {
              printf("m is equal to n");
    }
}
```

Output:

```
m is greater than n
```

switch case:

This structure helps to make a decision from the number of choices. The *switch* statement is a multi−way decision that tests whether an expression matches one of a number of constant integer values, and branches accordingly [3].

```
switch( integer expression)
{
        case constant 1 :
                do this;
        case constant 2 :
                do this ;
        case constant 3 :
                do this ;
        default :
                do this ;
}
```

The integer expression following the keyword switch is any C expression that will yield an integer value. It could be an integer constant like 1, 2 or 3, or an expression that evaluates to an integer. If a case matches the expression value, execution starts at that case. All case expressions must be different. The case labelled *default* is executed if none of the other cases are satisfied. A default is optional; if it isn't there and if none of the cases match, no action at all takes place. Cases and the default clause can occur in any order.

Consider the following program:

```
main( )
{ int i = 2; switch
         ( i )
         {
         case 1:
           printf ( "I am in case 1 \n" ) ;
         case 2:
           printf ( "I am in case 2 \n" ) ;
         case 3:
           printf ( "I am in case 3 \n" ) ;
         default :
           printf ( "I am in default \n" ) ;  }
}
```

The output of this program would be:

```
I am in case 2
I am in case 3
I am in default
```

Here the program prints case 2 and 3 and the default case. If you want that only case 2 should get executed, it is up to you to get out of the switch then and there by using a *break* statement.

```
main( )
{
int i = 2 ;
switch ( i )
{
 case 1:
    printf ( "I am in case 1 \n" ) ;
```

```
     break ;
  case 2:
    printf ( "I am in case 2 \n" ) ;
     break ;
  case 3:
    printf ( "I am in case 3 \n" ) ;
     break ;
  default:
    printf ( "I am in default \n" ) ;
  }
}
```

The output of this program would be:

```
I am in case 2
```

Program

7. WAP to enter a grade & check its corresponding remarks.

```
#include <stdio.h>
int main ()
{
     char grade;
     printf(“Enter the grade”);
    scanf(“%c”, &grade);
   switch(grade)
  {
          case 'A' :printf("Outstanding!\n" );
                      break;
         case 'B' : printf("Excellent!\n" );
                       break;
         case 'C' :printf("Well done\n" );
                    break;
         case 'D' : printf("You passed\n" );
                   break;
         case 'F' : printf("Better try again\n" );
                   break;
         default :  printf("Invalid grade\n" );
  }
  printf("Your grade is  %c\n", grade );
return 0;
}
```

Output

```
Enter the grade
B
Excellent
Your grade is B
```

LECTURE NOTE -11

ITERATIVE STATEMENTS

while statement

The *while* statement is used when the program needs to perform repetitive tasks. The general form of a *while* statement is:

```
        while ( condition)
                statement ;
```

The program will repeatedly execute the statement inside the *while* until the condition becomes false(0). (If the condition is initially false, the statement will not be executed.) Consider the following program:

```
main( )
{  int p, n, count;
float r, si;
count = 1;
while ( count <= 3 )
        {
                printf ( "\nEnter values of p, n and r " ) ;
                scanf(“%d %d %f", &p, &n, &r ) ;
                si=p * n * r / 100 ;
                printf ( "Simple interest = Rs. %f", si ) ;
                count = count+1;
        }
}
```

Some outputs of this program:

```
Enter values of p, n and r 1000 5 13.5
Simple Interest = Rs. 675.000000
Enter values of p, n and r 2000 5 13.5
Simple Interest = Rs. 1350.000000
Enter values of p, n and r 3500 5 13.5
Simple Interest = Rs. 612.000000
```

The program executes all statements after the *while* 3 times. These statements form what is called the ‘body’ of the *while* loop. The parentheses after the *while* contain a condition. As long as this condition remains true all statements within the body of the *while* loop keep getting executed repeatedly.

Consider the following program;

```
 /* This program checks whether a given number is a palindrome or not */

#include <stdio.h>
int main()
{
   int n, reverse = 0, temp;
   printf("Enter a number to check if it is a palindrome or not\n");
   scanf("%d",&n);
    temp = n;
   while( temp != 0 )
   {
         reverse = reverse * 10;
        reverse = reverse +temp%10;
       temp = temp/10;
   }
   if ( n == reverse )
      printf("%d is a palindrome number.\n", n);
else
      printf("%d is not a palindrome number.\n", n);

   return 0;
}
```

Output:

```
Enter a number to check if it is a palindrome or not
12321
12321 is a palindrome

Enter a number to check if it is a palindrome or not
12000
12000 is not a palindrome
```

do-while Loop

The body of the *do-while* executes at least once. The *do-while* structure is similar to the *while* loop except the relational test occurs at the bottom (rather than top) of the loop. This ensures that the body of the loop executes at least once. The *do-while* tests for a positive relational test; that is, as long as the test is True, the body of the loop continues to execute. The format of the do-while is

```
do
        { block of one or more C statements; }
while (test expression)
```

The test expression must be enclosed within parentheses, just as it does with a while statement.

Consider the following program

// C program to add all the numbers entered by a user until user enters 0.

```
#include <stdio.h>
int main()
{    int sum=0,num;
   do            /* Codes inside the body of do...while loops are at least executed once. */
   {
        printf("Enter a number\n");
       scanf("%d",&num);
        sum+=num;
   }
   while(num!=0);
   printf("sum=%d",sum);
return 0;
}
```

Output:

```
Enter a number
3
Enter a number
-2
Enter a number
```

```
0
sum=1
```

Consider the following program:

```
#include <stdio.h>
main()
{
    int i = 10;
   do
  {
       printf("Hello %d\n", i );
       i = i -1;
  }while ( i > 0 );
}
```

Output

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

Program

8. Program to count the no of digits in a number

```
#include <stdio.h>
int main()
{
  int n,count=0;
  printf("Enter an integer: ");
  scanf("%d", &n);
```

```
    do
      {
            n/=10;            /* n=n/10 */
            count++;
     } while(n!=0);

     printf("Number of digits: %d",count);
    }
```

Output

```
Enter an integer: 34523
Number of digits: 5
```

for Loop

The *for* is the most popular looping instruction. The general form of *for* statement is as under:

```
for ( initialise counter ; test counter ; Updating counter )
{
        do this;
        and this;
        and this;
}
```

The *for* allows us to specify three things about a loop in a single line:

(a) Setting a loop counter to an initial value.
(b) Testing the loop counter to determine whether its value has reached the number of repetitions desired.
(c) Updating the value of loop counter either increment or decrement.

Consider the following program

```
int main(void)
{
int num;
 printf("    n    n cubed\n");
for (num = 1; num <= 6; num++)
```

```
        printf("%5d %5d\n", num, num*num*num);
return 0;
}
```

The program prints the integers 1 through 6 and their cubes.

```
n   n cubed
1     1
2     8
3     27
4     64
5     125
6     216
```

The first line of the *for* loop tells us immediately all the information about the loop parameters: the starting value of num, the final value of num, and the amount that num increases on each looping [5].

Grammatically, the three components of a *for* loop are expressions. Any of the three parts can be omitted, although the semicolons must remain.

Consider the following program:

```
main( )
{
        int i ;
        for ( i = 1 ; i <= 10 ; )
        {
         printf ( "%d\n", i ) ;
         i = i + 1 ;
        }
}
```

Here, the increment is done within the body of the *for* loop and not in the *for* statement. Note that in spite of this the semicolon after the condition is necessary.

Programs:

9. Program to print the sum of 1st N natural numbers.

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

```
{
 int n,i,sum=0;
 printf("Enter the limit: ");
scanf("%d", &n);
for(i=1;i<=n;i++)
 {
         sum = sum +i;
 }
printf("Sum of N natural numbers is: %d",sum);
}
```

Output

```
Enter the limit: 5
Sum of N natural numbers is 15.
```

10. Program to find the reverse of a number

```
#include<stdio.h>
int main()
{
int num,r,reverse=0;
printf("Enter any number: ");
scanf("%d",&num);
for(;num!=0;num=num/10)
{
    r=num%10;
    reverse=reverse*10+r;
}
 printf("Reversed of number: %d",reverse);
return 0;
}
```

Output:

```
Enter any number: 123
Reversed of number: 321
```

NESTING OF LOOPS

C programming language allows using one loop inside another loop. Following section shows few examples to illustrate the concept.

Syntax:

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

```
for ( init; condition; increment )
{

 for ( init; condition; increment)

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

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

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

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

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

A final note on loop nesting is that you can put any type of loop inside of any other type of loop. For example, a for loop can be inside a while loop or vice versa.

Programs:

11. program using a nested for loop to find the prime numbers from 2 to 20:

```
#include <stdio.h>
int main ()
{
  /* local variable definition */
int i, j;
  for(i=2; i<20; i++)
 {
    for(j=2; j <= (i/j); j++)
      if(!(i%j))
          break; // if factor found, not prime
     if(j > (i/j)) printf("%d is prime\n", i);
  }

  return 0;
}
```

Output

```
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
```

```
12.     *
       ***
      *****
     *******
    *********
```

```
#include <stdio.h>
int main()
{
  int row, c, n,I, temp;
 printf("Enter the number of rows in pyramid of stars you wish to see ");
 scanf("%d",&n);
 temp = n;
 for ( row = 1 ; row <= n ; row++ )
{
    for ( i= 1 ; i < temp ; i++ )
    {
          printf(" ");
          temp--;
          for ( c = 1 ; c <= 2*row - 1 ; c++ )
          {
                    printf("*");
                    printf("\n");
          }
    }
 }
   return 0;
}
```

13. Program to print series from 10 to 1 using nested loops.

```
#include<stdio.h>
void main ()
{
          int a;
          a=10;
        for (k=1;k=10;k++)
       {
          while (a>=1)
        {
                printf ("%d",a);
                a--;
        }
        printf("\n");
        a= 10;
        }
}
```

Output:

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

LECTURE NOTE-12

JUMP STATEMENTS

The break Statement

The *break* statement provides an early exit from *for*, *while*, and *do*, just as from *switch*. A *break* causes the innermost enclosing loop or switch to be exited immediately. When *break* is encountered inside any loop, control automatically passes to the first statement after the loop.

Consider the following example;

```
main( )
{
        int i = 1 , j = 1 ;
        while ( i++ <= 100 )
        {
                while ( j++ <= 200 )
                {
                if ( j == 150 )
                  break ;
                 else
                  printf ( "%d %d\n", i, j );
                }
        }
}
```

In this program when j equals 150, break takes the control outside the inner while only, since it is placed inside the inner *while*.

The *continue* Statement

The *continue* statement is related to *break*, but less often used; it causes the next iteration of the enclosing *for*, *while*, or *do* loop to begin. In the *while* and *do*, this means that the test part is executed immediately; in the *for*, control passes to the increment step. The *continue* statement applies only to loops, not to switch.

Consider the following program:

```
main( )
{
```

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

The output of the above program would be...

```
1 2
2 1
```

Note that when the value of I equals that of j, the *continue* statement takes the control to the *for* loop (inner) by passing rest of the statements pending execution in the *for* loop (inner).

The *goto* statement

Kernighan and Ritchie refer to the *goto* statement as "infinitely abusable" and suggest that it "be used sparingly, if at all.

The *goto* statement causes your program to jump to a different location, rather than execute the next statement in sequence. The format of the *goto* statement is;

goto statement label;

Consider the following program fragment

```
if (size > 12)
          goto a;
goto b;

a: cost = cost * 1.05;
flag = 2;
```

*b: bill = cost * flag;*

Here, if the *if* conditions satisfies the program jumps to block labelled as *a:* if not then it jumps to block labelled as *b:*.

Exercise questions:

1. WAP to input the 3 sides of a triangle & print its corresponding type.
2. WAP to input the name of salesman & total sales made by him. Calculate & print the commission earned.

TOTAL SALES	RATE OF COMMISSION
1-1000	3 %
1001-4000	8 %
6001-6000	12 %
6001 and above	15 %

3. WAP to calculate the wages of a labor.

TIME	WAGE
First 10 hrs.	Rs 60
Next 6 hrs.	Rs 15
Next 4 hrs.	Rs 18
Above 10 hrs.	Rs 25

4. WAP to calculate the area of a triangle, circle, square or rectangle based on the user's choice.
5. WAP that will print various formulae & do calculations:
   i. Vol of a cube
   ii. Vol of a cuboid
   iii. Vol of a cyclinder
   iv. Vol of sphere
6. WAP to print the following series
   i. S = 1 + 1/2 + 1/3 ……..1/10
   ii. P= (1*2) + (2 *3) + (3* 4)+…….(8 *9) +(9 *10)
   iii. Q= ½ + ¾ +5/6 +….13/14
   iv. S = 2/5 + 5/9 + 8/13….n
   v. $S = x + x^2 + x^3 + x^4 ......+ x^9 + x^{10}$
   vi. $P= x + x^3/3 + x^5/5 + x^7/7$……………n terms
   vii. S= (13 *1) + (12 * 2)……(1 *13)
   viii. $S = 1 + 1/(2^2) + 1/ (3^3) + 1/(4^4) + 1/(5^5)$

ix. S = 1/1! + 1/2! + 1/3! ……………+1/n!

x. S = 1 + 1/3! + 1/5!+……..n terms

xi. S = 1 + (1+2) +(1+2+3) + (1+2+3+4)……………+(1+2+3…….20)

xii. S= x + x^2/2! + x^3/3! + x^4/4!.....+x^{10}/10!

xiii. P = x/2! + x^2/3! +…….x^9/10!

xiv. S = 1 – 2 + 3 - 4………. + 9 – 10

xv. S = 1 -2^2 + 3^2 - 4^2………. +9^2 - 10^2

xvi. S = 1/(1 + 2) + 3/(3 + 5)……15/(15 + 16)

xvii. S = 1 +x^2/2! – x^4/4! + x^6/6!....n

xviii. S = 1 + (1 + 2) + (1+2+3)……..(1+2+3+4…..20)

xix. S = 1 + x + x^2/2 + x^3/3…….+x^n/n

xx. S = 1 * 3/ 2 * 4 * 5 + 2 * 4 / 3 * 5 * 6 + 3 * 5/ 4 * 6 * 7……..n * (n+2)/ (n+1) * (n+3) * (n+4)

7. WAP to input a no & print its corresponding table.
8. WAP to print the table from 1 to 10 till 10 terms.
9. WAP to input a no & print its factorial.
10. WAP to input a no & check whether it is prime or not.
11. WAP to input a no & print all the prime nos upto it.
12. WAP to input a no & print if the no is perfect or not.
13. WAP to find the HCF of 2 nos.
14. WAP to print the Pythagoras triplets within 100. (A Pythagorean triplet consists of three positive integers *a*, *b*, and *c*, such that $a^2 + b^2 = c^2$).
15. WAP to input a no & check whether its automorphic or not. (An automorphic number is a number whose square "ends" in the same digits as the number itself. For example, 5^2 = 2**5**, 6^2 = 3**6**, 76^2 = 5776, and 890625^2 = 793212**890625**, so 5, 6, 76 and 890625 are all automorphic numbers).
16. WAP to convert a given no of days into years, weeks & days.
17. WAP to input a no & check whether it's an Armstrong no or not. (An Armstrong no is an integer such that the sum of the cubes of its digits is equal to the number itself. For example, 371 is an Armstrong number since $3^3 + 7^3 + 1^3 = 371$).
18. A cricket kit supplier in Jalandhar sells bats, wickets & balls. WAP to generate sales bill. Input form the console the date of purchase, name of the buyer, price of each item & quantity of each item. Calculate the total sale amount & add 17.5 % sales tax if the total sales amount >300000 & add 12.5 % if the total sales amount is >150000 & 7 % otherwise. Display the total sales amount, the sales tax & the grand total.
19. WAP to check whether a given number is magic number or not.

(What is a magic number? Example: 1729

- Find the sum of digits of the given number.(1 + 7 + 2 + 9 => 19)
- Reverse of digit sum output. Reverse of 19 is 91

- Find the product of digit sum and the reverse of digit sum.(19 X 91 = 1729)
- If the product value and the given input are same, then the given number is a magic number.(19 X 91 <=> 1729)

20. Write a C program to calculate generic root of the given number. (To find the generic root of a no we first find the sum of digits of the no until we get a single digit output. That resultant no is called the generic no. Eg: 456791: 4+5+6+7+9+1=32. 3 +2 =5. So, 5 becomes the generic root of the given no)

MODULE 2

LECTURE NOTE-13

FUNCTION

MONOLITHIC VS MODULAR PROGRAMMING:

1. Monolithic Programming indicates the program which contains a single function for the large program.
2. Modular programming help the programmer to divide the whole program into different modules and each module is separately developed and tested. Then the linker will link all these modules to form the complete program.
3. On the other hand monolithic programming will not divide the program and it is a single thread of execution. When the program size increases it leads inconvenience and difficult to maintain.

Disadvantages of monolithic programming: 1. Difficult to check error on large programs. 2. Difficult to maintain. 3. Code can be specific to a particular problem. i.e. it can not be reused.

Advantage of modular programming: 1. Modular program are easier to code and debug. 2. Reduces the programming size. 3. Code can be reused in other programs. 4. Problem can be isolated to specific module so easier to find the error and correct it.

FUNCTION:

A function is a group of statements that together perform a task. Every C program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.

Function Declaration OR Function Prototype:

1. It is also known as function prototype .

2. It inform the computer about the three things

a) Name of the function

b) Number and type of arguments received by the function.

c) Type of value return by the function

Syntax :

return_type function_name (type1 arg1 , type2 arg2);

OR

return_type function_name (type1 type2);

3. Calling function need information about called function .If called function is place before calling function then the declaration is not needed.

Function Definition:

1. It consists of code description and code of a function .

   It consists of two parts

   a) Function header

   b) Function coding

Function definition tells what are the I/O function and what is going to do.

Syntax:

```
return_type  function_name (type1  arg1 , type2 arg2)
 {
local variable;

statements ;

return (expression);

  }
```

2. Function definition can be placed any where in the program but generally placed after the main function .
3. Local variable declared inside the function is local to that function. It cannot be used anywhere in the program and its existence is only within the function.
4. Function definition cannot be nested.
5. Return type denote the type of value that function will return and return type is optional if omitted it is assumed to be integer by default.

USER DEFINE FUNCTIONS VS STANDARD FUNCTION:

User Define Fuction:

A function that is declare, calling and define by the user is called user define function. Every user define function has three parts as:

1. Prototype or Declaration
2. Calling
3. Definition

Standard Function:

The **C standard library** is a standardized collection of header files and library routines used to implement common operations, such as input/output and character string handling. Unlike other languages (such as COBOL, FORTRAN, and PL/I) C does not include built in keywords for these tasks, so nearly all C programs rely on the standard library to function.

LECTURE NOTE 14

FUNCTION CATAGORIES

There are four main categories of the functions these are as follows:

1. Function with no arguments and no return values.
2. Function with no arguments and a return value.
3. Function with arguments and no return values.
4. Function with arguments and return values.

Function with no arguments and no return values:

syntax:

```
void funct (void);
main ( )
{
funct (  );
}
void funct ( void );
{
}
```

NOTE: There is no communication between calling and called function. Functions are executed independently, they read data & print result in same block.

Example:

```
void  link (void) ;
int  main ()
{
link ();
}
void link ( void );
{
printf (" link the file ")
}
```

Function with no arguments and a return value: This type of functions has no arguments but a return value

example:

```
 int msg (void) ;
int main ( )
{
int s = msg ( );
printf( “summation = %d” , s);
}
 int msg ( void )
{
int a, b, sum ;
sum = a+b ;
return (sum) ;
}
```

NOTE: Here called function is independent, it read the value from the keyboard, initialize and return a value .Both calling and called function are partly communicated with each other.

Function with arguments and no return values:

Here functions have arguments so, calling function send data to called function but called function does no return value. such functions are partly dependent on calling function and result obtained is utilized by called function .

Example:

```
 void  msg ( int , int );

int main ( )

{

int a,b;
a= 2; b=3;

msg( a, b);

}

void msg ( int a , int b)

{

int s ;

sum = a+b;

printf (“sum = %d” , s ) ;

}
```

Function with arguments and return value:

Here calling function of arguments that passed to the called function and called function return value to calling function.

example:

```
 int  msg ( int , int ) ;

int main ( )

{

int a, b;

a= 2; b=3;
int s = msg (a, b);

printf (“sum = %d” , s ) ;

}

int msg( int a , int b)

{

int sum ;

sum =a+b ;
return (sum);

}
```

LECTURE NOTE 15

ACTUAL ARGUMENTS AND FORMAL ARGUMENTS

Actual Arguments:

1. Arguments which are mentioned in the function in the function call are known as calling function.
2. These are the values which are actual arguments called to the function.

It can be written as constant , function expression on any function call which return a value .

ex: funct (6,9) , funct (a,b)

Formal Arguments:

1. Arguments which are mentioned in function definition are called dummy or formal argument.
2. These arguments are used to just hold the value that is sent by calling function.
3. Formal arguments are like other local variables of the function which are created when function call starts and destroyed when end function.

Basic difference between formal and local argument are:

a) Formal arguments are declared within the ( ) where as local variables are declared at beginning.

b) Formal arguments are automatically initialized when a value of actual argument is passed.

c) Where other local variables are assigned variable through the statement inside the function body.

Note: Order, number and type of actual argument in the function call should be matched with the order , number and type of formal arguments in the function definition .

PARAMETER PASSING TECHNIQUES:

1. call by value
2. call by reference

Call by value:

Here value of actual arguments is passed to the formal arguments and operation is done in the formal argument.

Since formal arguments are photo copy of actual argument, any change of the formal arguments does not affect the actual arguments

Changes made to the formal argument t are local to block of called function, so when control back to calling function changes made vanish.

Example:

```
  void swap (int a , int b)      /* called function */

      {

      int t;

      t = a;

      a=b;

      b = t;


    }

  main( )

   {

      int k = 50,m= 25;
      swap( k, m) ;   / * calling function */ print
      (k, m);     / * calling function */
      }

      Output:
              50, 25
```

Explanation:

int k= 50, m=25 ;

Means first two memory space are created k and m , store the values 50 and 25 respectively.

swap (k,m);

When this function is calling the control goes to the called function.

void swap (int a , int b),
k and m values are assigned to the 'a' and 'b'.
then a= 50 and b= 25 ,

After that control enters into the function a temporary memory space 't' is created when int t is executed.

t=a; Means the value of a is assigned to the t , then t= 50.

a=b; Here value of b is assigned to the a , then a= 25;

b=t; Again t value is assigned to the b , then b= 50;

after this control again enters into the main function and execute the print function print (k,m). it returns the value 50 , 25.

NOTE:

Whatever change made in called function not affects the values in calling function.

Call by reference:

Here instead of passing value address or reference are passed. Function operators or address rather than values .Here formal arguments are the pointers to the actual arguments.

Example:

```
 #include<stdio.h>
void add(int *n);
int main()
        {
         int num=2;
         printf("\n The value of num before calling the function=%d", num);
         add(&num);
         printf("\n The value of num after calling the function = %d", num);
         return 0;
         }
          void add(int *n)
         {
```

```
   *n=*n+10;
printf("\n The value of num in the called function = %d", n);
}
```

Output:

The value of num before calling the function=2

The value of num in the called function=20 The

value of num after calling the function=20

NOTE:

In call by address mechanism whatever change made in called function affect the values in calling function.

EXAMPLES:

1: Write a function to return larger number between two numbers:

```
int fun(int p, int q)
{
int large;
if(p>q)
 {
large = p;
 }
 else
  {
large = q;
}
return large;
}
```

2: Write a program using function to find factorial of a number.

```
#include <stdio.h> int
factorial (int n)
{
   int i, p;
   p = 1;
  for (i=n; i>1; i=i-1)
```

```
    {
            p = p * i;
    }

    return (p);
}

void main()
{
    int a, result;
    printf ("Enter an integer number: ");
    scanf ("%d", &a);
    result = factorial (a);
    printf ("The factorial of %d is %d.\n", a, result);
}
```

EXERCISE:

1. What do you mean by function?
2. Why function is used in a program?
3. What do you mean by call by value and call by address?
4. What is the difference between actual arguments and formal arguments?
5. How many types of functions are available in C?
6. How many arguments can be used in a function?
7. Add two numbers using
   a) with argument with return type
   b) with argument without return type
   c) without argument without return type
   d) without argument with return type
8. Write a program using function to calculate the factorial of a number entered through the keyboard.
9. Write a function power(n,m), to calculate the value of n raised to m.
10. A year is entered by the user through keyboard. Write a function to determine whether the year is a leap year or not.
11. Write a function that inputs a number and prints the multiplication table of that number.
12. Write a program to obtain prime factors of a number. For Example: prime factors of 24 are 2,2,2 and 3 whereas prime factors of 35 are 5 and 7.
13. Write a function which receives a float and a int from main(), finds the product of these two and returns the product which is printed through main().

14. Write a function that receives % integers and returns the sum, average and standard deviation of these numbers. Call this function from main() and print the result in main().
15. Write a function that receives marks obtained by a student in 3 subjects and returns the average and percentage of these marks. Call this function from main() and print the result in main().
16. Write function to calculate the sum of digits of a number entered by the user.
17. Write a program using function to calculate binary equivalent of a number.
18. Write a C function to evaluate the series
    $\sin(x) = x - (x^3/3!) + (x^5/5!) - (x^7/7!) + \ldots\ldots\ldots\ldots$ to five significant digit.
19. If three sides of a triangle are p, q and r respectively, then area of triangle is given by
    area $= (S(S-p)(S-q)(S-r))^{1/2}$, where $S = (p+q+r)/2$.
    Write a program using function to find the area of a triangle using the above formula.
20. Write a function to find GCD and LCM of two numbers.
21. Write a function that takes a number as input and returns product of digits of that number.
22. Write a single function to print both amicable pairs and perfect numbers.(Two different numbers are said to be amicable if the sum of proper divisors of each is equal to the other. 284 and 220 are amicable numbers.)
23. Write a function to find whether a character is alphanumeric.

LECTURE NOTE 16

RECURSION

Recursion is a process in which a problem is defined in terms of itself. In ‘C’ it is possible to call a function from itself. Functions that call themselves are known as **recursive** functions, i.e. a statement within the body of a function calls the same function. Recursion is often termed as

‘Circular Definition’. Thus recursion is the process of defining something in terms of itself. To implement recursion technique in programming, a function should be capable of calling itself.

Example:

```
void main()

{

………………………                 /* Some statements*/

fun1();


………………………                 /* Some statements */

} void

fun1()


{

………………………                  /* Some statements */ fun1();

/*RECURSIVE CALL*/


………………………                 /* Some statements */

}
```

Here the function **fun1()** is calling itself inside its own function body, so fun1() is a recursive function. When main() calls fun1(), the code of fun1() will be executed and since there is a call to fun1() insidefun1(), again fun1() will be executed. It looks like the above program will run up to infinite times but generally a terminating condition is written inside the recursive functions which end this recursion. The following program (which is used to print all the numbers starting from the given number to 1 with successive decrement by 1) illustrates this:

```
void main()
{
 int a;
 printf(“Enter a number”);
 scanf(“%d”,&a);
 fun2(a);
}
int fun2(int b)
{
 printf(“%d”,b);
 b--;
 if(b>=1)    /* Termination condition i.e. b is less than 1*/
{
 fun2(b);
}
}
```

How to write a Recursive Function?

Before writing a recursive function for a problem its necessary to define the solution of the problem in terms of a similar type of a smaller problem.

Two main steps in writing recursive function are as follows:

> (i). Identify the Non-Recursive part(base case) of the problem and its solution(Part of the problem whose solution can be achieved without recursion).
> (ii). Identify the Recursive part(general case) of the problem(Part of the problem where recursive call will be made).

Identification of Non-Recursive part of the problem is mandatory because without it the function will keep on calling itself resulting in infinite recursion.

How control flows in successive recursive calls?

Flow of control in successive recursive calls can be demonstrated in following example:

Consider the following program which uses recursive function to compute the factorial of a number.

```
void main()
{
int n,f;
printf(“Enter a number”);
scanf(“%d”,&n);
f=fact(a);
printf(“Factorial  of  %d  is %d”,n,f);
}
int fact(int m)
{
int a;
if (m==1)
  return (1);
else
  a=m*fact(m-1);
return (a);
}
```

In the above program if the value entered by the user is 1 i.e.**n**=1, then the value of **n** is copied into **m**. Since the value of **m** is 1 the condition ‘if(m==1)’ is satisfied and hence 1 is returned through return statement i.e. factorial of 1 is 1.

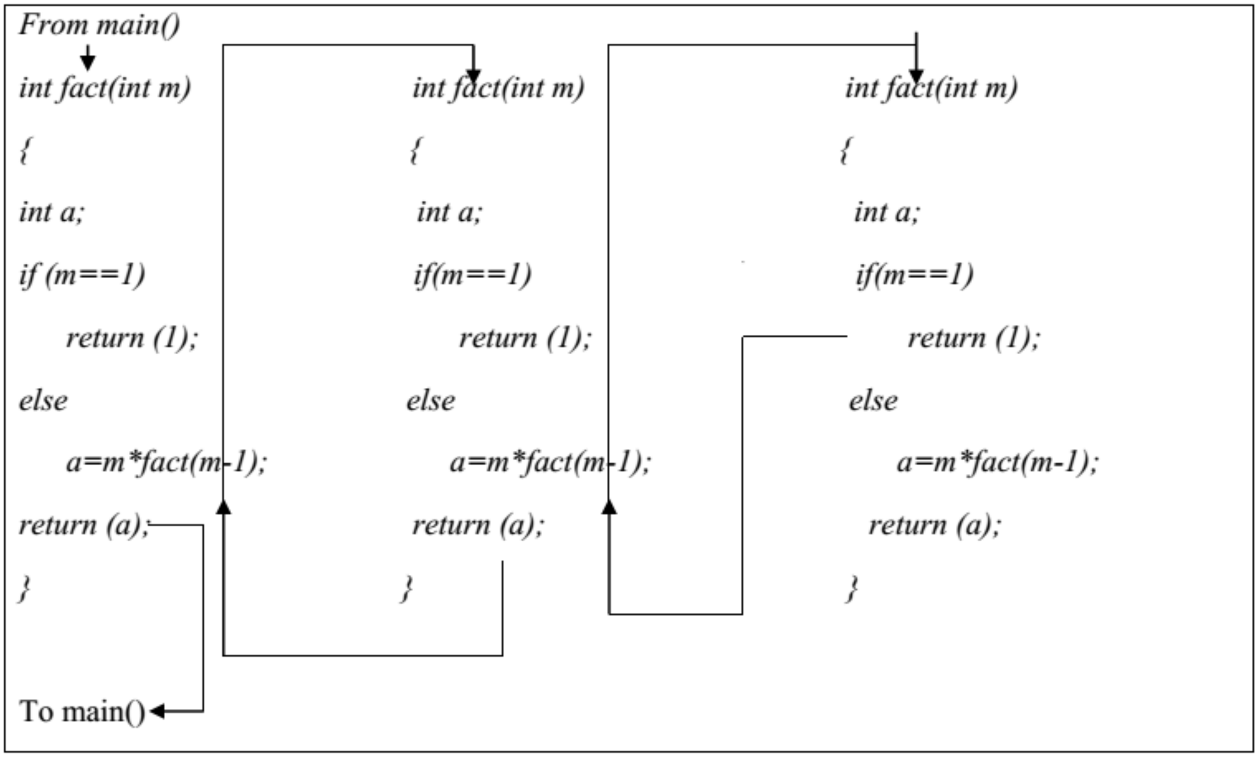
When the number entered is 2 i.e. n=2, the value of n is copied into m. Then in function fact(), the condition ‘if(m==1)’ fails so we encounter the statement a=m*fact(m-1); and here we meet recursion. Since the value of **m** is 2 the expression (m*fact(m-1)) is evaluated to (2*fact(1)) and the result is stored in **a**(factorial of a). Since value returned by fact(1) is 1 so the above expression reduced to (2*1) or simply 2. Thus the expression m*fact(m-1) is evaluated to 2 and stored in **a** and returned to main(). Where it will print ‘Factorial of 2 is 2’.

In the above program if **n**=4 then main() will call fact(4) and fact(4) will send back the computed value i.e. **f** to main(). But before sending back to main() fact(4) will call fact(4-1) i.e. fact(3) and wait for a value to be returned by fact(3). Similarly fact(3) will call fact(2) and so on. This series of calls continues until **m** becomes 1 and fact(1) is called, which returns a value which is so called as termination condition. So we can now say what happened for **n**=4 is as follows

```
fact(4) returns (4*fact(3) )

fact(3) returns (3*fact(2) )

fact(2) returns (2*fact(1) )

fact(1) returns (1)
```

So for **n**=4, the factorial of 4 is evaluated to 4*3*2*1=24.

For **n**=3, the control flow of the program is as follows:



```
int fact(int m)
{
int a;
if (m==1)
    return (1);
else
    a=m*fact(m-1);
return (a);
}
```

```
int fact(int m)
{
 int a;
if(m==1)
    return (1);
else
   a=m*fact(m-1);
return (a);
}
```

```
int fact(int m)
{
 int a;
 if(m==1)
      return (1);
else
     a=m*fact(m-1);
  return (a);
}
```

Winding and Unwinding phase

All recursive functions work in two phases- winding phase and unwinding phase.

Winding phase starts when the recursive function is called for the first time, and ends when the termination condition becomes true in a call i.e. no more recursive call is required. In this phase a function calls itself and no return statements are executed.

After winding phase unwinding phase starts and all the recursive function calls start returning in reverse order till the first instance of function returns. In this phase the control returns through each instance of the function.

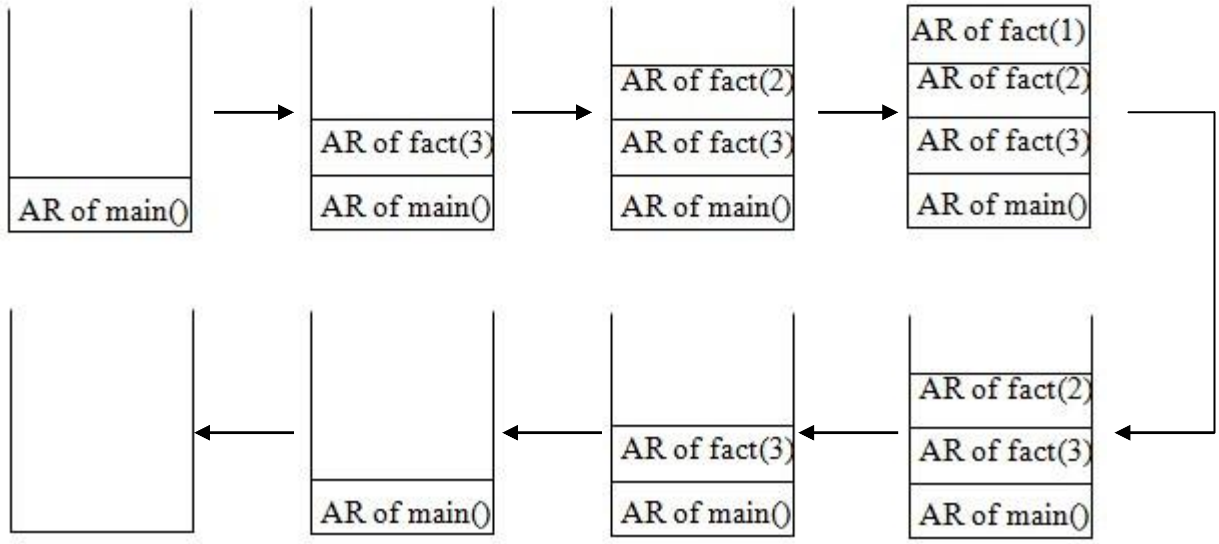
Implementation of Recursion

We came to know that recursive calls execute like normal function calls, so there is no extra technique to implement recursion. All function calls(Whether Recursive or Non-Recursive) are implemented through run time stack. Stack is a Last In First Out(LIFO) data structure. This means that the last item to be stored on the stack(PUSH Operation) is the first one which will be deleted(POP Operation) from the stack. Stack stores Activation Record(AR) of function during run time. Here we will take the example of function fact() in the previous recursive program to find factorial of a number.

Suppose fact() is called from main() with argument 3 i.e.

*fact(3);   /*From main()*/*

Now will see how the run time stack changes during the evaluation of factorial of 3.



The following steps will reveal how the above stack contents were expressed:

First main() is called, so PUSH AR of main() into the stack. Then main() calls fact(3) so PUSH AR of fact(3). Now fact(3) calls fact(2) so PUSH AR of fact(2) into the stack. Likewise PUSH AR of fact(1). After the above when fact(1) is completed, POP AR of fact(1), Similarly after completion of a specific function POP its corresponding AR. So when main() is completed POP AR of main(). Now stack is empty.

In the winding phase the stack content increases as new Activation Records(AR) are created and pushed into the stack for each invocation of the function. In the unwinding phase the Activation Records are popped from the stack in LIFO order till the original call returns.

Examples of Recursion

Q1. Write a program using recursion to find the summation of numbers from 1 to n.

Ans: We can say ‘sum of numbers from 1 to n can be represented as sum of numbers from 1 to n-1 plus n’ i.e.

Sum of numbers from 1 to n = n + Sum of numbers from 1 to n-1

= n + n-1 + Sum of numbers from 1 to n-2

= n+ n-1 + n-2 + ... +1

The program which implements the above logic is as follows:

```
#include<stdio.h>

void main()

{

int n,s;

 printf("Enter a number");

 scanf("%d",&n);

 s=sum(n);

 printf("Sum of numbers from 1 to %d is %d",n,s);

}
```

```
 int sum(int m) int r;

if(m==1)

        return (1);

else

        r=m+sum(m-1);/*Recursive Call*/

        return r;

}
```

Output:

```
Enter a number 5
15
```

Q2. Write a program using recursion to find power of a number i.e. n^{m}.

Ans: We can write,

$n_m = n*n_{m-1}$

$=n*n*n^{m-2}$

$=n*n*n*\ldots\ldots\ldots\ldots\text{m times} *n^{m-m}$

The program which implements the above logic is as follows:

```
#include<stdio.h>

int power(int,int);

 void main()

{

         int n,m,k;

         printf("Enter the value of n and m");

         scanf("%d%d",&n,&m);
```

```
            k=power(n,m);

            printf(“The value of n^m for n=%d and m=%d is %d”,n,m,k);

}

int power(int x, int y)

{

           if(y==0)

           {

                   return 1;

          }

          else

         {

                     return(x*power(x,y-1));

         }

}
```

Output:

```
Enter the value of n and m
3
5
The value of n^m for n=3 and m=5 is 243
```

Q3.Write a program to find GCD (Greatest Common Divisor) of two numbers.

Ans: The GCD or HCF (Highest Common Factor) of two integers is the greatest integer that divides both the integers with remainder equals to zero. This can be illustrated by Euclid’s remainder Algorithm which states that GCD of two numbers say x and y i.e.

GCD(x, y) = **x**   if y is 0

= **GCD(y, x%y)**   otherwise

The program which implements the previous logic is as follows:

```
#include<stdio.h>
int GCD(int,int);
void main()
{
 int a,b,gcd;
printf(“Enter two numbers”);
scanf(“%d%d”,&a,&b);
gcd=GCD(a,b);
printf(“GCD of %d and %d is %d”,a,b,gcd);
}
int GCD(int x, int y)
{
if(y==0)
return x;
 else
 return GCD(y,x%y);
}
```

Output:

```
Enter two numbers   21
35
GCD of 21 and 35 is 7
```

Q4:Write a program to print Fibonacci Series upto a given number of terms.

Ans: Fibonacci series is a sequence of integers in which the first two integers are 1 and from third integer onwards each integer is the sum of previous two integers of the sequence i.e.

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ……..........................

The program which implements the above logic is as follows:

```
#include<stdio.h>
int Fibonacci(int);
void main()
{
 int term,i;
printf(“Enter the number of terms of Fibonacci Series which is going to be printed”);
scanf(“%d”,&term);
 for(i=0;i<term;i++)
{
printf(“%d”,Fibonacci(i));
}
}
int Fibonacci(int x)
{
if(x==0 || x==1)
return 1;
else
return (Fibonacci(x-1) + Fibonacci(x-2));
}
```

Output:

```
Enter the number of terms of Fibonacci Series which is going to be printed 6
1 1 2 3 5 8 13
```

LECTURE NOTE 17

RECURSION VERSES ITERATION

Every repetitive problem can be implemented recursively or iteratively

Recursion should be used when the problem is recursive in nature. Iteration should be used when the problem is not inherently recursive

Recursive solutions incur more execution overhead than their iterative counterparts, but its advantage is that recursive code is very simple.

Recursive version of a problem is slower than iterative version since it requires PUSH and POP operations.

In both recursion and iteration, the same block of code is executed repeatedly, but in recursion repetition occurs when a function calls itself and in iteration repetition occurs when the block of code is finished or a continue statement is encountered.

For complex problems iterative algorithms are difficult to implement but it is easier to solve recursively. Recursion can be removed by using iterative version.

Tail Recursion

A recursive call is said to be tail recursive if the corresponding recursive call is the last statement to be executed inside the function.

Example: Look at the following recursive function

```
void show(int a)

{

if(a==1)

 return;

printf("%d",a);

show(a-1);

}
```

In the above example since the recursive call is the last statement in the function so the above recursive call is Tail recursive call.

In non void functions(return type of the function is other than void) , if the recursive call appears in return statement and the call is not a part of an expression then the call is said to be Tail recursive, otherwise Non Tail recursive. Now look at the following example

```
int hcf(int p, int q)
{
if(q==0)
return p;
else

return(hcf(q,p%q));    /*Tail recursive call*/
}
int factorial(int a)
{
if(a==0)
return 1;
else

 return(a*factorial(a-1));       /*Not a Tail recursive call*/
}
```

In the above example in hcf() the recursive call is not a part of expression (i.e. the call is the expression in the return statement) in the call so the recursive call is Tail recursive. But in factorial() the recursive call is part of expression in the return statement(a*recursive call) , so the recursive call in factorial() is not a Tail excursive call.

A function is said to be Tail recursive if all the recursive calls in the function are tail recursive.

Tail recursive functions are easy to write using loops,

In tail recursive functions, the last work that a function does is a recursive call, so no operation is left pending after the recursive call returns. Therefore in tail recursive functions , there is nothing to be done in unwinding phase, so we can jump directly from the last recursive call to the place where recursive function was first called.

Tail recursion can be efficiently implemented by compilers so we always will try to make our recursive functions tail recursive whenever possible.

Functions which are not tail recursive are called augmentive recursive functions and these types of functions have to finish the pending work after the recursive call finishes.

Indirect and Direct Recursion

If a function fun1() calls another function fun2() and the function fun2() in turn calls function fun1(), then this type of recursion is said to be **indirect** recursion, because the function fun1() is calling itself indirectly.

```
fun1( )
{
………………………     /* Some statements*/
fun2();
………………………     /* Some statements*/
}
fun2( )
{
………………………     /* Some statements*/
fun1();
………………………      /* Some statements*/
}
```

The chain of functions in indirect recursion may involve any number of functions.For example suppose n number of functions are present starting from f1() to fn() and they are involved as following: f1() calls f2(), f2() calls f3(), f3() calls f4() and so on with fn() calls f1().

If a function calls itself directly i.e. function fun1() is called inside its own function body, then that recursion is called as direct recursion. For example look at the following

```
fun1()
{
...      /* Some statements*/
fun2();
...     /* Some statements*/
}
```

Indirect recursion is complex, so it is rarely used.

Exercise:

Find the output of programs from 1 to 5.

1.
   ```
   void main()
   {
    printf(“%d\n”,count(17243));
    }
     int count(int x)
   {
    if(x==0)
           return 0;
    else
           return 1+count(x/10)
    }
   ```

2.
   ```
   void main()
   {
   printf(“%d\n”,fun(4,8));
   printf(“%d\n”,fun(3,8));
   }
   int fun(int x. int y)
   {
    if(x==y)
      return x;
    else
      return (x+y+fun(x+1,y-1));
    }
   ```

```
3. void main()

      {

       printf(“%d\n”,fun(4,9));

      printf(“%d\n”,fun(4,0));

       printf(“%d\n”,fun(0,4));

       }

       int fun(int x, int y)

       {

       if(y==0)

               return 0;

        if(y==1)

              return x;

         return x+fun(x,y-1);

       }
4.   void main()

       {

        printf(“%d\n”,fun1(14837));

       }

      int fun1(int m)

      {

      return ((m)? m%10+fun1(m/10):0);
}
```

5.
```
void main()
{
printf("%d\n",fun(3,8));
}
int fun(int x, int y)
{
if(x>y)
   return 1000;
return x+fun(x+1,y);
}
```

6. What is the use of recursion in a program, Explain?
7. Explain the use of stack in recursion.
8. What do you mean by winding and unwinding phase?
9. How to write a recursive function, Explain with example?
10. What is the difference between tail and non-tail recursion, explain with example.
11. What is indirect recursion?
12. What is the difference between iteration and recursion?
13. Write a recursive function to enter a line of text and display it in reverse order, without storing the text in an array.
14. Write a recursive function to count all the prime numbers between number p and q(both inclusive).
15. Write a recursive function to find quotient when a positive integer m is divided by a positive integer n.
16. Write a program using recursive function to calculate binary equivalent of a number.
17. Write a program using recursive function to reverse number.
18. Write a program using recursive function to find remainder when a positive integer m is divided by a positive integer n.
19. Write a recursive function that displays a positive integer in words, for example if the integer is 465 then it is displayed as -four six five.
20. Write a recursive function to print the pyramids

```
1               abcd
1 2             abc
1 2 3           ab
1 2 3 4         a
```

21. Write a recursive function to find the Binomial coefficient C(n,k), which is defined as: C(n,0)=1
C(n,n)=1
C(n,k)=C(n-1,k-1)+C(n-1,k)

LECTURE NOTE 18

STORAGE CLASSES

To completely define a variable one needs to mention its type along with its **storage class**. In other words we can say not only variables have a data type but also they have 'storage classes.

Till now the storage class of any variable is not mentioned because storage classes have defaults. If someone doesn't specify the storage class of a variable during its declaration, the compiler assumes a storage class depending upon the situation in which the variable is going to be used. Therefore we can now say every variable have certain default storage class.

Compiler identifies the physical location within the computer where the string of bits which represents the variable's values are stored from the variable name itself.

Generally there are two kinds of locations in a computer where such a value can be present, these are Memory and CPU registers. The storage class of a particular variable determines in which of the above two locations the variable's value is stored.

There are four properties by which storage class of a variable can be recognized.These are scope, default initial value, scope and life.

A variable's storage class reveals the following things about a variable

(i) Where the variable is stored.
(ii) What is the initial value of the variable if the value of the variable is not specified?
(iii)What is the scope of the variable (To which function or block the variable is available).
(iv) What is the life of particular variable (Up to what extent the variable exists in a program).

There are four storage classes in C, these are Automatic, Static, Register and External. The keywords auto, static, registers and exter are used for the above storage classes respectively.

We can specify a storage class while declaring a variable. The

general syntax is

storage_class datatype variable_name;

The following table shows different properties of a variable which according to its storage class.

Properties Storage Class	**Storage**	**Default Initial Value**	**Scope**	**Life**
Automatic	Memory	Garbage Value	Local to the block in which the variable is defined	Till the control remains within the block in which the variable is defined
Register	CPU Registers	Garbage Value	Local to the block in which the variable is defined	Till the control remains within the block in which the variable is defined
Static	Memory	Zero	Local to the block in which the variable is defined	Value of the variable continues to exist between different function calls
External	Memory	Zero	Global	Till the program's execution doesn't come to an end

Automatic Storage Class

Syntax to declare automatic variable is:

auto datatype variablename;

Example:

auto int i;

Features of Automatic Storage Class are as follows

Storage: Memory

Default Initial 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

All the variables which are declared inside a block or function without any storage class specifier are automatic variables. So by default every variable is automatic variable. We can use auto keyword to declare automatic variables but generally it is not done. Since automatic variables are known inside a function or block only, we may have variables of same name in different functions or blocks without any doubt. Look at the following two functions:

```
void fun1()
{
 int x,y;
/* Some statements*/
}
```

```
void fun1()
{
   auto int x,y;
  /*Some statements*/
  }
```

In the above two functions declaration statements are equivalent as both declare variables x and y as automatic variables.

The following program illustrates the work of automatic variables.

```
void test();
void main()
{
test();
test();
test();
}
 void test()
{
auto int k=10;
printf(“%d\n”,k);
k++;
}
```

Output:

```
10
10
10
```

In the above program when the function test() is called for the first time ,variable k is created and initialized to 10. When the control returns to main(), k is destroyed. When function test() is called for the second time again k is created , initialized and destroyed after execution of the function. Hence automatic variables came into existence each time the function is executed and destroyed when execution of the function completes.

Register Storage Class

Syntax to declare register variable is:

register datatype variablename;

Features of Register Storage Class are as follows:

Storage: CPU Registers

Default Initial 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

Register storage class can be applied only to automatic variables. Its scope, life and default initial value are same as that of automatic variables. The only difference between register and automatic variables is the place where they are stored. Automatic variables are stored in memory but register variables are stored in CPU registers. Registers are small storage units present in the processor. The variables stored in registers can be accessed much faster than the variables stored in registers. Hence the variables which are frequently used in a program can be assigned register storage class for faster execution. For example loop counters are declared as register variables, which are defined as follows:

```
int main()
{
 register int a;
for(a=0;i<50000;i++)
printf(“%d\t”,a);
return 0;
 }
```

In the above program, variable **a** was used frequently as a loop counter so the variable a is defined as a register variable. Register is a not a command but just a request to the compiler to allocate the memory for the variable into the register. If free registers are available than memory will be allocated in to the registers. And if there are no free registers then memory will be allocated in the RAM only (Storage is in memory i.e. it will act as automatic variable).

Every type of variables can be stored in CPU registers. Suppose the microprocessor has 16 bit registers then they can't hold a float or a double value which requires 4 and 8 bytes respectively. But if you use register storage class for a float or double variable then you will not get any error message rather the compiler will treat the float and double variable as be of automatic storage class(i.e. Will treat them like automatic variables).

Static Storage Class

Syntax to declare static variable is:

```
static datatype variablename;
```

Example:

```
static int i;
```

Features of Static Storage Class are as follows

Storage: Memory

Default Initial Value: Zero

Scope: Local to the block in which the variable is defined

Life: Value of the variable continues to exist between different function calls

Now look at the previous program with k is declared as static instead of automatic.

```
void test();
void main()
{
 test();
 test();
 test();
 }
void test()
{
```

```
static int k=10;
 printf("%d\n",k);
 k++;
}
```

Output:

```
10
11
12
```

Here in the above program the output is 10, 11, 12, because if a variable is declared as static then it is initialized only once and that variable will never be initialized again. Here variable k is declared as static, so when test() is called for the first time k value is initialized to 10 and its value is incremented by 1 and becomes 11. Because k is static its value persists. When test() is called for the second time k is not reinitialized to 10 instead its old value 11 is available. So now 11 is get printed and it value is incremented by 1 to become 12. Similarly for the third time when test() is called 12(Old value) is printed and its value becomes 13 when executes the statement 'k++;'

So the main difference between automatic and static variables is that static variables are initialized to zero if not initialized but automatic variables contain an unpredictable value(Garbage Value) if not initialized and static variables does not disappear when the function is no longer active , their value persists, i.e. if the control comes back to the same function again the static variables have the same values they had last time.

General advice is avoid using static variables in a program unless you need them, because their values are kept in memory when the variables are not activc which means they occupies space in memory that could otherwise be used by other variables.

External Storage Class

Syntax to declare static variable is:

extern datatype variablename;

Example:

```
extern int i;
```

Features of External Storage Class are as follows

Storage: Memory

Default Initial Value: Zero

Scope: Global

Life: Till the program's execution doesn't come to an end

External variables differ from automatic, register and static variables in the context of scope, external variables are global on the contrary automatic, register and static variables are local. External variables are declared outside all functions, therefore are available to all functions that want to use them.

If the program size is very big then code may be distributed into several files and these files are compiled and object codes are generated. These object codes linked together with the help of linker and generate ".exe" file. In the compilation process if one file is using global variable but it is declared in some other file then it generate error called undefined symbol error. To overcome this we need to specify global variables and global functions with the keyword extern before using them into any file. If the global variable is declared in the middle of the program then we will get undefined symbol error, so in that case we have to specify its prototype using the keyword extern.

So if a variable is to be used by many functions and in different files can be declared as external variables. The value of an uninitialized external variable is zero. The declaration of an external variable declares the type and name of the variable, while the definition reserves storage for the variable as well as behaves as a declaration. The keyword **extern** is specified in declaration but not in definition. Now look at the following four statements

1. auto int a;
2. register int b;
3. static int c;
4. extern int d;

Out of the above statements first three are definitions where as the last one is a declaration.
Suppose there are two files and their names are File1.c and File2.c respectively. Their contents are as follows:

File1.c

```
int n=10;

void hello()

{

printf("Hello");

}
```

File2.c

```
extern int n;
extern void hello();
void main()
{
 printf(“%d”, n);
hello();
}
```

In the above program File1.obj+ File2.obj=File2.exe and in File2.c value of n will be 10.

Where to use which Storage Class for a particular variable:

We use different storage class in appropriate situations in a view to

1. Economise the memory space consumed by the variables
2. Improve the speed of execution of the program.

The rules that define which storage class is to be executed when are as follows:

(a) Use static storage class if you want the value of a variable to persist between different function calls.

(b) Use register storage class for those variables that are being used very often in a program, for faster execution. A typical application of register storage class is loop counters.

(c) Use extern storage class for only those variables that are being used by almost all the functions in a program, this avoids unnecessary passing of these variables as arguments when making a function call.

(d) If someone do not have any need mentioned above, then use auto storage class.

Exercise:

[A] Answer the following Questions.

(a) What do you mean by storage class of a variable?

(b) Why register storage class is required?

(c) How many storage classes are there in c and why they are used?

(d) Why someone needs static storage variable in a program?

(e) Which storage class is termed as default storage class, if the storage class of a variable is not mentioned in a program? Explain its working with the help of a suitable example.

(f) Can you store a floating point variable in CPU Registers, Explain?

(g) What do you mean by scope and life of a variable?

(h) Justify where to use which storage class.

(i) What do you mean by external variable and where it is defined?

(j) What is the difference between auto and static variables in a program?

(k) What do you mean by definition and declaration of a variable?

[B] What is the output of the following programs:

(a)
```
void main()
 {
 static int a=6;
printf(“\na=%d”,a--);
  if(a!=0)
 main();
 }
```

(b)
```
void main()
{
  int i,j;
 for(i=1;i<5;i++)
{
 j=fun1(i);
printf(“\n%d”,j);
 }
 }
  int fun1(int a)
```

```
{
 static int b=2;
   int c=3;
   b=a+b;
   return(a+b+c);
  }
```

(c)

```
void main()
{
fun1();
fun1();
}
void fun1()
{
 auto int x=0;
  register int y=0;
 static int z=0;
 x++;
 y++;
 z++;
 printf("\n%d%d%d",x,y,z);
}
```

```
(d) int a=10;
void main()
    {
    int a=20;
    {
    int a=30;
printf(“%d\n”,a);
     }
    printf(“%d”,a);
    }
```

[C] State whether the following statements are True or False:

(a) The register storage class variables cannot hold float values.

(b) The value of an automatic storage class variable persists between various function invocations.

(c) If the CPU registers are not available, the register storage class variables are treated as static storage class variables.

(d) The default value for automatic variable is zero.

(e) If a global variable is to be defined, then the extern keyword is necessary in its declaration.

(f) If we try to use register storage class for a float variable the compiler will flash an error message.

(g) Storage for a register storage class variable is allocated each time the control reaches the block in which the variable is present.

(h) An extern storage class variable is not available to the functions that precede its definition, unless the variable is explicitly declared in the above functions.

(i) The life of static variable is till the control remains within the block in which it is defined.

(j) The address of a register variable is not accessible.

[D] Following program calculates the sum of digits of the number 25634. Go through it and find out why is it necessary to declare the storage class of the variable sum as static.

```
#include<stdio.h>

 void main()

{ int m;

m=sumdigit(25634);

 printf("\n%d",m);

}

int sumdigit(int n)

{

static int sum;

 int p,q;

p=n%10;

q=(n-p)/10;

sum=sum+p;

if(q!=0) sumdigit(q);

else

return(sum);
 }
```

LECTURE NOTE 19

ARRAYS

Introduction

A *data structure* is the way data is stored in the machine and the functions used to access that data. An easy way to think of a data structure is a collection of related data items. An *array* is a data structure that is a collection of variables of one type that are accessed through a common name. Each element of an array is given a number by which we can access that element which is called an index. It solves the problem of storing a large number of values and manipulating them.

Arrays

Previously we use variables to store the values. To use the variables we have to declare the variable and initialize the variable i.e, assign the value to the variable. Suppose there are 1000 variables are present, so it is a tedious process to declare and initialize each and every variable and also to handle 1000 variables. To overcome this situation we use the concept of array .In an Array values of same type are stored. An array is a group of memory locations related by the fact that they all have the same name and same type. To refer to a particular location or element in the array we specify the name to the array and position number of particular element in the array.

One Dimensional Array

Declaration:

Before using the array in the program it must be declared

Syntax:

data_type array_name[size];

data_type represents the type of elements present in the array.

array_name represents the name of the array.

Size represents the number of elements that can be stored in the array.

Example:

int age[100];

float sal[15];

char grade[20];

Here age is an integer type array, which can store 100 elements of integer type. The array sal is floating type array of size 15, can hold float values. Grade is a character type array which holds 20 characters.

Initialization:

We can explicitly initialize arrays at the time of declaration.

Syntax:

data_type array_name[size]={value1, value2,……..valueN};

Value1, value2, valueN are the constant values known as initializers, which are assigned to the array elements one after another.

Example:

int marks[5]={10,2,0,23,4};

The values of the array elements after this initialization are:

marks[0]=10, marks[1]=2, marks[2]=0, marks[3]=23, marks[4]=4

NOTE:

1. In 1-D arrays it is optional to specify the size of the array. If size is omitted during initialization then the compiler assumes the size of array equal to the number of initializers.

Example:

int marks[]={10,2,0,23,4};

Here the size of array marks is initialized to 5.

2. We can't copy the elements of one array to another array by simply assigning it.

Example:

```
 int a[5]={9,8,7,6,5};
int b[5];
 b=a;      //not valid
```

we have to copy all the elements by using for loop.

```
for(a=i; i<5; i++)
      b[i]=a[i];
```

Processing:

For processing arrays we mostly use for loop. The total no. of passes is equal to the no. of elements present in the array and in each pass one element is processed.

Example:

```
#include<stdio.h>

main()

{

 int a[3],i;

 for(i=0;i<=2;i++)                        //Reading the array values

{

 printf("enter the elements");

scanf("%d",&a[i]);

}

 for(i=0;i<=2;i++)                     //display the array values

{

 printf("%d",a[i]);

printf("\n");

}

}
```

This program reads and displays 3 elements of integer type.

Example:1
C Program to Increment every Element of the Array by one & Print Incremented Array.

```
#include <stdio.h>

void main()

{

    int i;

   int array[4] = {10, 20, 30, 40};

for (i = 0; i < 4; i++)

    arr[i]++;

for (i = 0; i < 4; i++)

    printf("%d\t", array[i]);

}
```

Example: 2

C Program to Print the Alternate Elements in an Array

```
#include <stdio.h>
  void main()
{
    int array[10];
int i, j, temp;
    printf("enter the element of an array \n");
for (i = 0; i < 10; i++)
    scanf("%d", &array[i]);
    printf("Alternate elements of a given array \n");
for (i = 0; i < 10; i += 2)
     printf( "%d\n", array[i]) ;
}
```

Example-3

C program to accept N numbers and arrange them in an ascending order

```
#include <stdio.h>
void main()
{
 int i, j, a, n, number[30];
printf("Enter the value of N \n");
scanf("%d", &n);
printf("Enter the numbers \n");
for (i = 0; i < n; ++i)
        scanf("%d", &number[i]);
for (i = 0; i < n; ++i)
{
    for (j = i + 1; j < n; ++j)
    {
        if (number[i] > number[j])
        {
            a =number[i];
            number[i] = number[j];
            number[j] = a;
        }
    }
}
  printf("The numbers arranged in ascending order are given below \n");
for (i = 0; i < n; ++i)
      printf("%d\n", number[i]);
}
```

TWO DIMENSIONAL ARRAYS

Arrays that we have considered up to now are one dimensional array, a single line of elements. Often data come naturally in the form of a table, e.g. spreadsheet, which need a two-dimensional array.

Declaration:

The syntax is same as for 1-D array but here 2 subscripts are used.

Syntax:

data_type array_name[rowsize] [columnsize];

Rowsize specifies the no.of rows Columnsize

specifies the no.of columns.

Example:

int a[4] [5];

This is a 2-D array of 4 rows and 5 columns. Here the first element of the array is a[0][0] and last element of the array is a[3][4] and total no.of elements is 4*5=20.

	col 0	col 1	col 2	col 3	col 4
row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]
row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]
row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]	a[2][4]
row 3	a[3][0]	a[3][1]	a[3][2]	a[3][3]	a[3][4]

Initialization:

2-D arrays can be initialized in a way similar to 1-D arrays.

Example:

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

The values are assigned as follows:

m[0][0]:1	m[0][1]:2	m[0][2]:3
m[1][0]:4	m[1][1]:5	m[3][2]:6
m[2][0]:7	m[2][1]:8	m[3][2]:9
m[3][0]:10	m[3][1]:11	m[3][2]:12

The initialization of group of elements as follows:

```
int m[4][3]={{11},{12,13},{14,15,16},{17}};
```

The values are assigned as:

m[0][0]:1 1	m[0][1]:0	m[0][2]:0
m[1][0]:12	m[1][1]:13	m[3][2]:0
m[2][0]:14	m[2][1]:15	m[3][2]:16
m[3][0]:17	m[3][1]:0	m[3][2]:0

Note:

In 2-D arrays it is optional to specify the first dimension but the second dimension should always be present.

Example: *int*

```
m[][3]={

          {1,10},

          {2,20,200},

          {3},

          {4,40,400}   };
```

Here the first dimension is taken 4 since there are 4 roes in the initialization list. A 2-D array is known as matrix.

Processing:

For processing of 2-D arrays we need two nested for loops. The outer loop indicates the rows and the inner loop indicates the columns.

Example:

```
int a[4][5];
```

a) Reading values in a

```
for(i=0;i<4;i++)
    for(j=0;j<5;j++)
        scanf(“%d”,&a[i][j]);
```

b) Displaying values of a

```
for(i=0;i<4;i++)
    for(j=0;j<5;j++)
      printf(“%d”,a[i][j]);
```

Example 1:

Write a C program to find sum of two matrices

```
#include <stdio.h>
#include<conio.h>
void  main()
{
  float a[2][2], b[2][2], c[2][2];
   int i,j;
   clrscr();
  printf("Enter the elements of 1st matrix\n");
/* Reading two dimensional Array with the help of two for loop. If there is an array of 'n'
dimension, 'n' numbers of loops are needed for inserting data to array.*/
 for(i=0;i<2;I++)
  for(j=0;j<2;j++)
  {
     scanf("%f",&a[i][j]);
   }
  printf("Enter the elements of 2nd matrix\n");
   for(i=0;i<2;i++)
   for(j=0;j<2;j++)
   {
       scanf("%f",&b[i][j]);
       }
/* accessing corresponding elements of two arrays. */
for(i=0;i<2;i++)
        for(j=0;j<2;j++)
          {
                c[i][j]=a[i][j]+b[i][j];  /* Sum of corresponding elements of two arrays. */
          }
 /* To display matrix sum in order. */
printf("\nSum Of Matrix:");
for(i=0;i<2;++i)
      {
        for(j=0;j<2;++j)
          printf("%f", c[i][j]);
       printf("\n");
      }
getch();
}
```

Example 2: Program for multiplication of two matrices

```
#include<stdio.h>
#include<conio.h>
int main()
  { int i,j,k;
     int row1,col1,row2,col2,row3,col3;
     int mat1[5][5], mat2[5][5], mat3[5][5];
     clrscr();
    printf("\n enter the number of rows in the first matrix:");
   scanf("%d", &row1);
    printf("\n enter the number of columns in the first matrix:");
   scanf("%d", &col1);
    printf("\n enter the number of rows in the second matrix:");
   scanf("%d", &row2);
   printf("\n enter the number of columns in the second matrix:");
  scanf("%d", &col2);
 if(col1 != row2)
   {
       printf("\n The number of columns in the first matrix must be equal to the number of rows
       in the second matrix ");
```

```
        getch();
        exit();
      }
  row3= row1;
  col3= col3;
  printf(“\n Enter the elements of the first matrix”);
  for(i=0;i<row1;i++)
    {
        for(j=0;j<col1;j++)
         scanf(“%d”,&mat1[i][j]);
    }

   printf(“\n Enter the elements of the second matrix”);
   for(i=0;i<row2;i++)
   {
        for(j=0;j<col2;j++)
        scanf(“%d”,&mat2[i][j]);
    }
  for(i=0;i<row3;i++)
   {
      for(j=0;j<col3;j++)
         {
           mat3[i][j]=0;
           for(k=0;k<col3;k++)
               mat3[i][j] +=mat1[i][k]*mat2[k][j];
           }
        }
  printf(“\n The elements of the product matrix are”):
  for(i=0;i<row3;i++)
   {
        printf(“\n”);
       for(j=0;j<col3;j++)
        printf(“\t %d”, mat3[i][j]);
   }
  return 0;
  }
```

Output:

```
Enter the number of rows in the first matrix: 2
Enter the number of columns in the first matrix: 2
Enter the number of rows in the second matrix: 2
Enter the number of columns in the second matrix: 2
Enter the elements of the first matrix
 1 2 3 4
Enter the elements of the second matrix
 5 6 7 8
The elements of the product matrix are
 19 22
 43 50
```

Example 3:

Program to find transpose of a matrix.

```
#include <stdio.h>
 int main()
 {
     int a[10][10], trans[10][10], r, c, i, j;
     printf("Enter rows and column of matrix: ");
     scanf("%d %d", &r, &c);
     printf("\nEnter elements of matrix:\n");
     for(i=0; i<r; i++)
      for(j=0; j<c; j++)
     {
        printf("Enter elements a%d%d: ",i+1,j+1);
        scanf("%d", &a[i][j]);
     }
/* Displaying the matrix a[][] */
printf("\n Entered Matrix: \n");
for(i=0; i<r; i++)
    for(j=0; j<c; j++)
    {
        printf("%d  ",a[i][j]);
        if(j==c-1)
            printf("\n\n");
    }

/* Finding transpose of matrix a[][] and storing it in array trans[][]. */
for(i=0; i<r;i++)
    for(j=0; j<c; j++)
    {
      trans[j][i]=a[i][j];
   }

/* Displaying the array trans[][]. */
printf("\nTranspose of Matrix:\n");
for(i=0; i<c;i++)
   for(j=0; j<r;j++)
   {
      printf("%d  ",trans[i][j]);
       if(j==r-1)
          printf("\n\n");
   }
return 0;
}
```

Output

```
Enter the rows and columns of matrix: 2   3
Enter the elements of matrix:
Enter elements a11: 1
Enter elements a12: 2
Enter elements a13: 9
Enter elements a21: 0
Enter elements a22: 4
Enter elements a23: 7
Entered matrix:
1 2 9
0 4 7
Transpose of matrix:
1 0
2 4
9 7
```

Multidimensional Array

More than 2-dimensional arrays are treated as multidimensional arrays.

Example:

```
int a[2][3][4];
```

Here a represents two 2-dimensional arrays and each of these 2-d arrays contains 3 rows and 4 columns.

The individual elements are:

a[0][0][0], a[0][0][1],a[0][0][2],a[0][1][0]………..a[0][3][2]

a[1][0][0],a[1][0][1],a[1][0][2],a[1][1][0]…………..a[1][3][2]

the total no. of elements in the above array is 2*3*4=24.

Initialization:

```
int a[2][4][3]={
{
 {1,2,3},
{4,5},
{6,7,8},
{9}
},
{
{10,11},
{12,13,14},
{15,16},
{17,18,19}
}
}
```

The values of elements after this initialization are as:

a[0][0][0]:1   a[0][0][1]:2   a[0][0][2]:3

a[0][1][0]:4   a[0][1][1]:5   a[0][1][2]:0

a[0][2][0]:6   a[0][2][1]:7   a[0][2][2]:8

a[0][3][0]:9   a[0][3][1]:0   a[0][3][2]:0

```
a[1][0][0]:10     a[1][0][1]:11     a[1][0][2]:0

a[1][1][0]:12     a[1][1][1]:13     a[1][1][2]:14

a[1][2][0]:15     a[1][2][1]:16     a[1][2][2]:0

a[1][3][0]:17     a[1][3][1]:18     a[1][3][2]:19
```

Note:

The rule of initialization of multidimensional arrays is that last subscript varies most frequently and the first subscript varies least rapidly.

Example:

```
#include<stdio.h>
 main()
{
 int d[5];
int i;
for(i=0;i<5;i++)
{
 d[i]=i;
}
for(i=0;i<5;i++)
{
printf("value in array %d\n",a[i]);
}
}
```

pictorial representation of d will look like

d[0]	d[1]	d[2]	d[3]	d[4]
0	1	2	3	4

LECTURE NOTE 21

ARRAYS USING FUNCTIONS

1-d arrays using functions

Passing individual array elements to a function

We can pass individual array elements as arguments to a function like other simple variables.

Example:

```
#include<stdio.h>
void check(int);
void main()
{
 int a[10],i;
 clrscr();
 printf("\n enter the array elements:");
for(i=0;i<10;i++)
{
 scanf("%d",&a[i]);
 check(a[i]);
}
 void check(int num)
{
if(num%2==0)
  printf("%d is even\n",num);
else
    printf("%d is odd\n",num);
}
```

Output:

```
enter the array elements:

1 2 3 4 5 6 7 8 9 10

1   is odd

2   is even

3   is odd

4   is even

5   is odd

6   is even

7   is odd

8   is even

9   is odd

10 is even
```

Example:

C program to pass a single element of an array to function

```
#include <stdio.h>
 void display(int a)
   {
    printf("%d",a);
   }
int main()
{
    int c[]={2,3,4};
    display(c[2]);  //Passing array element c[2] only.
   return 0;
}
```

Output

```
2 3 4
```

Passing whole 1-D array to a function

We can pass whole array as an actual argument to a function the corresponding formal arguments should be declared as an array variable of the same type.

Example:

```
#include<stdio.h>
main()
{
int i, a[6]={1,2,3,4,5,6};
 func(a);
 printf(“contents of array:”);
for(i=0;i<6;i++)
            printf(“%d”,a[i]);
 printf(”\n”);
}
func(int val[])
{
int sum=0,i;
 for(i=0;i<6;i++)
{
           val[i]=val[i]*val[i];
           sum+=val[i];
}
printf(“the sum of squares:%d”, sum);
}
```

Output

```
contents of array: 1 2 3 4 5 6

the sum of squares: 91
```

Example.2:

Write a C program to pass an array containing age of person to a function. This function should find average age and display the average age in main function.

```
#include <stdio.h>
 float average(float a[]);
int main()
{
    float avg, c[]={23.4, 55, 22.6, 3, 40.5, 18};
     avg=average(c);   /* Only name of array is passed as argument. */
     printf("Average age=%.2f",avg);
    return 0;
 }
float average(float a[])
{
    int i;
    float avg, sum=0.0;
   for(i=0;i<6;++i)
            {
                   sum+=a[i];
         }
     avg =(sum/6);
    return avg;
}
```

Output

```
Average age= 27.08
```

Solved Example:

1. Write a program to find the largest of n numbers and its location in an array.

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

```
  #include<conio.h>
  void main()
 {
  int array[100], maximum, size, c, location = 1;
 clrscr();
 printf("Enter the number of elements in array\n");
 scanf("%d", &size);
 printf("Enter %d integers\n", size);

 for (c = 0; c < size; c++)
    scanf("%d", &array[c]);
 maximum = array[0];

  for (c = 1; c < size; c++)
  {
    if (array[c] > maximum)
    {
       maximum  = array[c];
       location = c+1;
    }
  }
  printf("Maximum element is present at location %d and it's value is %d.\n", location,
maximum);
  getch();
 }
```

Output:

```
Enter the number of elements in array
5
Enter 5 integers
2
4
7
9
1
```

Maximum element is present at location 4 and it's value is 9

2. Write a program to enter n number of digits. Form a number using these digits.

```
# include<stdio.h>
#include<conio.h>
#include<math.h>
void  main()
{
 int     number=0,digit[10],     numofdigits,i;
clrscr();
printf("\n Enter the number of digits:");
scanf("%d", &numofdigits);
for(i=0;i<numofdigits;i++)
     {
          printf("\n Enter the %d th digit:", i);
          scanf("%d",&digit[i]);
     }
i=0;
while(i<numofdigits)
{
   number= number + digit[i] * pow(10,i)
   i++;
}
printf("\n The number is : %d",number);
getch();
}
```

Output:

```
Enter the number of digits: 3
Enter the 0th digit: 5
Enter the 1th digit: 4
Enter the 2th digit: 3
The number is: 543
```

3. Matrix addition:

```
#include <stdio.h>
#include<conio.h>
 void main()
{
 int m, n, c, d, first[10][10], second[10][10], sum[10][10];
clrscr();
 printf("Enter the number of rows and columns of matrix\n");
 scanf("%d%d", &m, &n);
 printf("Enter the elements of first matrix\n");
 for ( c = 0 ; c < m ; c++ )
for ( d = 0 ; d < n ; d++ )
     scanf("%d", &first[c][d]);

 printf("Enter the elements of second matrix\n");

for ( c = 0 ; c < m ; c++ )
for ( d = 0 ; d < n ; d++ )
       scanf("%d", &second[c][d]);

 for ( c = 0 ; c < m ; c++ )
   for ( d = 0 ; d < n ; d++ )
     sum[c][d] = first[c][d] + second[c][d];

 printf("Sum of entered matrices:-\n");

 for ( c = 0 ; c < m ; c++ )
 {
   for ( d = 0 ; d < n ; d++ )
     printf("%d\t", sum[c][d]);

   printf("\n");
 }

 getch();
}
```

Output:

```
Enter the number of rows and columns of matrix
2
2
Enter the elements of first matrix
1 2
3 4
Enter the elements of second matrix
5 6
```

```
2 1
Sum of entered matrices:- 6
8
5 5
```

Exercise

1. Compute sum of elements of an array in a program?

2. Write a program for histogram printing using an array?

3. Write a program for dice-rolling using an array instead of switch?

4. Sorting an array with bubble sort?

5. Write a program for binary search using an array?

6. Write a program to interchange the largest and the smallest number in the array.

7. Write a program to fill a square matrix with value 0 on the diagonals, 1 on the upper right triangle, and -1 on the lower left triangle.

8. Write a program to read and display a 2x2x2 array.

9. Write a program to calculate the number of duplicate entries in the array.

10. Given an array of integers, calculate the sum, mean, variance and standard deviation of the numbers in the array.

11. Write a program that reads a matrix and displays the sum of the elements above the main diagonal.

12. Write a program to calculate XA + YB where A and B are matrices and X=2, and Y=3

LECTURE NOTE 22

FUNDAMENTALS OF STRINGS

A string is a series of characters treated as a single unit. A string may include letters, digits and various special characters such as +, -, *, / and $. String literals or string constants in C are written in double quotation marks as follows:

“1000 Main Street” (a street address)

“(080)329-7082” (a telephone number)

“Kalamazoo, New York” (a city)

In C language strings are stored in array of char type along with null terminating character ‘\0’ at the end.

When sizing the string array we need to add plus one to the actual size of the string to make space for the null terminating character, ‘\0’.

Syntax:

char fname[4];

The above statement declares a string called fname that can take up to 3 characters. It can be indexed just as a regular array as well.

fname[]={‘t’,’w’,’o’};

character	t	w	o	\0
ASCII code	116	119	41	0

Generalized syntax is:-

char str[size];

when we declare the string in this way, we can store size-1 characters in the array because the last character would be the null character. For example,

char mesg[10]; can store maximum of 9 characters.

If we want to print a string from a variable, such as four name string above we can do this.

e.g., *printf(“First name:%s”,fname);*
We can insert more than one variable. Conversion specification %s is used to insert a string and then go to each %s in our string, we are printing.

A string is an array of characters. Hence it can be indexed like an array.

char ourstr[6] = “EED”;

– ourstr[0] is ‘E’

– ourstr[1] is ‘E’

– ourstr[2] is ‘D’

– ourstr[3] is ‘\0’

– ourstr[4] is ‘\0’ – ourstr[5] is ‘\0’

‘E’	‘E’	‘D’	\0	‘\0’	‘\0’
ourstr[0]	ourstr[1]	ourstr[2]	ourstr[3]	ourstr[4]	ourstr[5]

Reading strings:

If we declare a string by writing

char str[100];

then str can be read from the user by using three ways;

1. Using scanf() function
2. Using gets() function
3. Using getchar(), getch(), or getche() function repeatedly

The string can be read using scanf() by writing
scanf(“%s”,str);
Although the syntax of scanf() function is well known and easy to use, the main pitfall with this function is that it terminates as soon as it finds a blank space. For example, if the user enters Hello World, then str will contain only Hello. This is because the moment a blank space is encountered, the string is terminated by the scanf() function.

Example:

```
char str[10];
printf("Enter a string\n");
scanf("%s",str);
```

The next method of reading a string a string is by using gets() function. The string can be read by writing

```
gets(str);
```

gets() is a function that overcomes the drawbacks of scanf(). The gets() function takes the starting address of the string which will hold the input. The string inputted using gets() is automatically terminated with a null character.

Example:

```
char str[10];
 printf("Enter a string\n");
 gets(str);
```

The string can also be read by calling the getchar() repeatedly to read a sequence of single characters (unless a terminating character is encountered) and simultaneously storing it in a character array as follows:

```
int i=0;

char str[10],ch;

getchar(ch);

while(ch!='\0')

{

   str[i]=ch;      // store the read character in str

    i++;

   getch(ch);   // get another character

  }

str[i]='\0';      // terminate str with null character
```

Writing string

The string can be displayed on screen using three ways:

1. Using printf() function
2. Using puts() function
3. Using putchar() function repeatedly

The string can be displayed using pintf() by writing

```
printf("%s",str);
```

We can use width and precision specification along with %s. The width specifies the minimum output field width and the precision specifies the maximum number of characters to be displayed. Example:

```
    printf("%5.3s",str);
```

this statement would print only the first three characters in a total field of five charaters; also these three characters are right justified in the allocated width.

The next method of writing a string is by using the puts() function. The string can be displayed by writing:

```
    puts(str);
```

It terminates the line with a newline character ('\n'). It returns an EOF(-1) if an error occurs and returns a positive number on success.

Finally the string can be written by calling the putchar( ) function repeatedly to print a sequence of single characters.

```
int i=0;

char str[10];

while(str[i]!='\0')

{

   putchar(str[i]);     // print the character on the screen

   i++;

}
```

Example: Read and display a string

```
#include<stdio.h>
#include<conio.h>
void main()
{
 char str[20];
 clrscr();
 printf("\n Enter a string:\n");
gets(str);
 scanf("The string is:\n");
puts(str);
        getch();  }
```

Output:

```
 Enter a string:
vssut burla
The string is:
vssut burla
```

LECTURE NOTE 23

COMMON FUNCTIONS IN STRING

Type	*Method*	*Description*
char	*strcpy(s1, s2)*	Copy string
char	*strcat(s1, s2)*	Append string
int	*strcmp(s1, s2)*	Compare 2 strings
int	*strlen(s)*	Return string length
char	*strchr(s, int c)*	Find a character in string
char	*strstr(s1, s2)*	Find string s2 in string s1

strcpy():

It is used to copy one string to another string. The content of the second string is copied to the content of the first string.

Syntax:

strcpy (string 1, string 2);

Example:

char mystr[10];

mystr = “Hello”; *// Error! Illegal !!! Because we are assigning the value to mystr which is not possible in case of an string. We can only use "=" at declarations of C-String.*

strcpy(mystr, “Hello”);

It sets value of mystr equal to “Hello”.

strcmp():

It is used to compare the contents of the two strings. If any mismatch occurs then it results the difference of ASCII values between the first occurrence of 2 different characters.

Syntax:

int strcmp(string 1, string 2);

Example:

```
char mystr_a[10] = "Hello";
char mystr_b[10] = "Goodbye";

– mystr_a == mystr_b;    // NOT allowed!
```

The correct way is

```
if (strcmp(mystr_a, mystr_b ))
printf ("Strings are NOT the same.");
else
 printf( "Strings are the same.");
```

Here it will check the ASCII value of H and G i.e, 72 and 71 and return the diference 1.

strcat():

It is used to concatenate i.e, combine the content of two strings.

Syntax:

```
strcat(string 1, string 2);
```

Example:

```
char fname[30]={"bob"};

char lname[]={"by"};

printf("%s", strcat(fname,lname));
```

Output:

bobby.

strlen():

It is used to return the length of a string.

Syntax:

```
int strlen(string);
```

Example:

```
char fname[30]={“bob”};

int length=strlen(fname);
```

It will return 3

strchr():

It is used to find a character in the string and returns the index of occurrence of the character for the first time in the string.

Syntax:

```
            strchr(cstr);
```

Example:

```
char mystr[] = "This is a simple string";

char  pch = strchr(mystr,‘s’);
```

The output of pch is mystr[3]

strstr():

It is used to return the existence of one string inside another string and it results the starting index of the string.

Syntax:

```
            strstr(cstr1, cstr2);
```

Example:

```
Char mystr[]="This is a simple string";

char pch = strstr(mystr, “simple”);
```

here pch will point to mystr[10]

- **String input/output library functions**

Function prototype	Function description
int getchar(void);	Inputs the next character from the standard input and returns it as integer
int putchar(int c);	Prints the character stored in c and returns it as an integer
int puts(char s);	Prints the string s followed by new line character. Returns a non-zero integer if possible or EOF if an error occurs
int sprint(char s, char format,….)	Equivalent to printf,except the output is stored in the array s instead of printed in the screen. Returns the no.of characters written to s, or EOF if an error occurs
int sprint(char s, char format,….)	Equivalent to scanf, except the input is read from the array s rather than from the keyboard. Returns the no.of items successfully read by the function , or EOF if an error occurs

NOTE:

Character arrays are known as strings.

Self-review exercises:

1. Find the error in each of the following program segments and explain how to correct it:
   - *char s[10];*
   - *strcpy(s,"hello",5);*
   - *prinf("%s\n",s);*
   - *printf("%s",'a');*
   - *char s[12]; strcpy(s,"welcome home");*
   - *If (strcmp(string 1, sring 2))*

     { printf("the strings are equal\n");

     }
2. Show 2 different methods of initializing character array vowel with the string of vowels "AEIOU"?
3. Writ a program to convert string to an integer?

4. Write a program to accept a line of text and a word. Display the no. of occurrences of that word in the text?
5. Write a program to read a word and re-write its characters in alphabetical order.
6. Write a program to insert a word before a given word in the text.
7. Write a program to count the number of characters, words and lines in the given text.

MODULE 3

LECTURE NOTE 24

STRUCTURE AND UNION

Definition

A Structure is a user defined data type that can store related information together. The variable within a structure are of different data types and each has a name that is used to select it from the structure. C arrays allow you to define type of variables that can hold several data items of the same kind but **structure** is another user defined data type available in C programming, which allows you to combine data items of different kinds.

Structures are used to represent a record, Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book:

- Title
- Author
- Subject
- Book ID

Structure Declaration

It is declared using a keyword struct followed by the name of the structure. The variables of the structure are declared within the structure.

Example:

```
Struct struct-name

   {

        data_type var-name;


         data_type var-name;


   };
```

Structure Initialization

Assigning constants to the members of the structure is called initializing of structure.
Syntax:

```
        struct struct_name
        {
          data _type member_name1;

          data _type member_name2;

        } struct_var={constant1,constant2};
```

Accessing the Members of a structure

A structure member variable is generally accessed using a '.' operator.

Syntax: *strcut_var.*

member_name;

The dot operator is used to select a particular member of the structure. To assign value to the individual

Data members of the structure variable stud, we write,

stud.roll=01;

stud.name="Rahul";

To input values for data members of the structure variable stud, can be written as,

scanf("%d",&stud.roll);

scanf('"%s",&stud.name);

To print the values of structure variable stud, can be written as:

printf("%s",stud.roll);

printf("%f",stud.name);

QUESTIONS

1. Write a program using structures to read and display the information about an employee.
2. Write a program to read, display, add and subtract two complex numbers.
3. Write a program to enter two points and then calculate the distance between them.

LECTURE NOTE 25

NESTED STRUCTURES

The structure that contains another structure as its members is called a nested structure or a structure within a structure is called nested structure. The structure should be declared separately and then be grouped into high level structure.

1. Write a program to read and display the information of all the students in the class using nested structure.

Passing Structures through pointers

Pointer to a structure is a variable that holds the address of a structure. The syntax to declare pointer to a structure can be given as:

*strcut struct_name *ptr;*

To assign address of stud to the pointer using address operator(&) we would write

ptr_stud=&stud;

To access the members of the structure (->) operator is used.

for example

Ptr_stud->name=Raj;

SELF REFERENTIAL STRUCTURE

Self –referential structures are those structures that contain a reference to data of its same type as that of structure.

Example

```
struct node

{

int val;

struct node*next;

};
```

Pointers to Structures

You can define pointers to structures in very similar way as you define pointer to any other variable as follows:

*struct books *struct_pointer;*

Now, you can store the address of a structure variable in the above defined pointer variable. To find the address of a structure variable, place the & operator before the structure's name as follows:

struct_pointer = &book1;

To access the members of a structure using a pointer to that structure, you must use the -> operator as follows:

struct_pointer->title;

1 .Write a program to display, add and subtract two time defined using hour, minutes and values of seconds.

2. Write a program, using pointer to structure, to initialize the members in the structure. Use functions to print the students information.

3. Write a program using an array of pointers to a structure to read and display the data of a student.

LECTURE NOTE 26

UNION

Union is a collection of variables of different data types, in case of union information can only be stored In one field at any one time. A **union** is a special data type available in C that enables you to store different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multi-purpose.

Declaring Union

```
union union-name

{

data_type var-name;


data_type var-name;


};
```

The **union tag** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the union's definition, before the final semicolon, you can specify one or more union variables but it is optional. Here is the way you would define a union type named Data which has the three members i, f, and str. Now, a variable of **Data** type can store an integer, a floating-point number, or a string of characters. This means that a single variable ie. same memory location can be used to store multiple types of data. You can use any built-in or user defined data types inside a union based on your requirement.

The memory occupied by a union will be large enough to hold the largest member of the union. For example, in above example Data type will occupy 20 bytes of memory space because this is the maximum space which can be occupied by character string. Following is the example which will display total memory size occupied by the above union:

Accessing a Member of a Union

```
#include <stdio.h>
#include <string.h>
union Data
{
   int i;
  float f;
  char  str[20];
};
 int main( )
{
   union Data data;
   data.i = 10;
   data.f = 220.5;
   strcpy( data.str, "C Programming");
  printf( "data.i : %d\n", data.i);
 printf( "data.f : %f\n", data.f);
printf( "data.str : %s\n", data.str);
return 0;
}
```

Dot operator can be used to access a member of the union . he member access operator is coded as a period between the union variable name and the union member that we wish to access. You would use **union** keyword to define variables of union type. Following is the example to explain usage of union:

Exercises:

1. Write a program to define a union and a structure both having exactly the same members. Using the sizeof operator, print the size of structure variable as well as union variable and comment on the result.
2. Write a program to define a structure for a hotel that has the member's mane, address, grade, number of rooms, and room charges. Write a function to print the names of the hotels in a particular grade. Also write a function to print names of a hotel that have room charges less than the specified value.

LECTURE NOTE 27

POINTERS

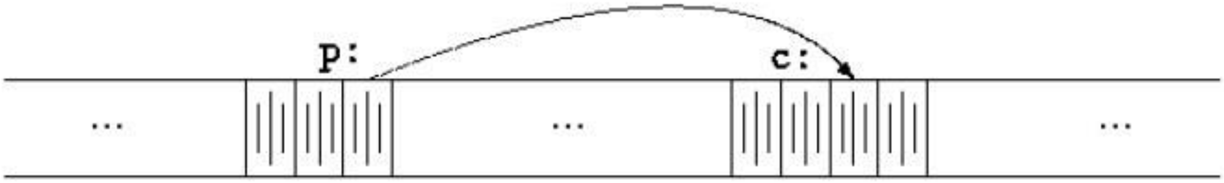
A pointer is a variable that contains the address of a variable. Pointers are much used in C, partly because they are sometimes the only way to express a computation, and partly because they usually lead to more compact and efficient code than can be obtained in other ways. Pointers and arrays are closely related; this chapter also explores this relationship and shows how to exploit it.

Pointers have been lumped with the `goto` statement as a marvelous way to create impossible to understand programs. This is certainly true when they are used carelessly, and it is easy to create pointers that point somewhere unexpected. With discipline, however, pointers can alsobe used to achieve clarity and simplicity. This is the aspect that we will try to illustrate.

The main change in ANSI C is to make explicit the rules about how pointers can be manipulated, in effect mandating what good programmers already practice and good compilers already enforce. In addition, the type `void *` (pointer to `void`) replaces `char *` as the proper type for a generic pointer.

Pointers and Addresses

Let us begin with a simplified picture of how memory is organized. A typical machine has an array of consecutively numbered or addressed memory cells that may be manipulated individually or in contiguous groups. One common situation is that any byte can be a `char`, a pair of one-byte cells can be treated as a `short` integer, and four adjacent bytes form a `long`. A pointer is a group of cells (often two or four) that can hold an address. So if `c` is a `char` and `p` is a pointer that points to it, we could represent the situation this way:



The unary operator `&`gives the address of an object, so the statement

```
p = &c;
```

assigns the address of `c` to the variable `p`, and `p` is said to ``point to" `c`. The `&`operator only applies to objects in memory: variables and array elements. It cannot be applied to expressions, constants, or `register` variables.

The unary operator `*` is the *indirection* or *dereferencing* operator; when applied to a pointer, it accesses the object the pointer points to. Suppose that `x` and `y` are integers and `ip`is a pointer to `int`. This artificial sequence shows how to declare a pointer and how to use `&`and `*`:

```
int x = 1, y = 2, z[10];
int *ip;
ip = &x;
y = *ip;
*ip = 0;
ip = &z[0];
```

The declaration of `x`, `y`, and `z` are what we've seen all along. The declaration of the pointer `ip`.

```
int *ip;
```

is intended as a mnemonic; it says that the expression `*ip`is an `int`. The syntax of the declaration for a variable mimics the syntax of expressions in which the variable might appear. This reasoning applies to function declarations as well. For example,

```
double *dp, atof(char *);
```

says that in an expression `*dp`and `atof(s)` have values of `double`, and that the argument of `atof`is a pointer to `char`.

You should also note the implication that a pointer is constrained to point to a particular kind of object: every pointer points to a specific data type. If `ip`points to the integer `x`, then `*ip`can occur in any context where `x` could, so

```
*ip = *ip + 10;
```

increments `*ip` by 10.

The unary operators `*` and `&`bind more tightly than arithmetic operators, so the assignment

```
y = *ip + 1
```

takes whatever `ip`points at, adds 1, and assigns the result to `y`, while

```
*ip += 1
```

increments what `ip`points to, as do

`++*ip` and `(*ip)++`

The parentheses are necessary in this last example; without them, the expression would increment `ip` instead of what it points to, because unary operators like `*` and `++` associate right to left. Finally, since pointers are variables, they can be used without dereferencing. For example, if `iq` is another pointer to `int`,

```
iq = ip
```

copies the contents of `ip`into `iq`, thus making `iq`point to whatever `ip`pointed to.

Pointers and Function Arguments

Since C passes arguments to functions by value, there is no direct way for the called function to alter a variable in the calling function. For instance, a sorting routine might exchange two outoforder arguments with a function called `swap`. It is not enough to write

```
swap(a, b);
```

where the `swap` function is defined as

```
void swap(int x, int y)
{
 int temp;
 temp = x;
 x = y;
 y = temp;
}
```

Because of call by value, `swap` can't affect the arguments `a` and `b` in the routine that called it. The function above swaps *copies* of `a` and `b`. The way to obtain the desired effect is for the calling program to pass pointers to the values to be changed:

```
swap(&a, &b);
```

Since the operator & produces the address of a variable, &a is a pointer to a. In swap itself, the parameters are declared as pointers, and the operands are accessed indirectly through them.

```
void swap(int *px, int *py) /* interchange *px and *py */

{

 int temp;

 temp = *px;

 *px = *py;

 *py = temp;

}
```

Pointer arguments enable a function to access and change objects in the function that called it. As an example, consider a function `getint` that performs free-format input conversion by breaking a stream of characters into integer values, one integer per call. `getint` has to return the value it found and also signal end of file when there is no more input. These values have to be passed back by separate paths, for no matter what value is used for `EOF`, that could also be the value of an input integer.

One solution is to have `getint` return the end of file status as its function value, while using a pointer argument to store the converted integer back in the calling function. This is the scheme used by `scanf`as well

The following loop fills an array with integers by calls to `getint`:

```
int n, array[SIZE], getint(int *);
for (n = 0; n < SIZE &&getint(&array[n]) != EOF; n++)
;
```

Each call sets `array[n]` to the next integer found in the input and increments `n`. Notice that it is essential to pass the address of `array[n]` to `getint`. Otherwise there is no way for `getint` to communicate the converted integer back to the caller.

Our version of `getint`returns `EOF` for end of file, zero if the next input is not a number, and a positive value if the input contains a valid number.

```
#include <ctype.h>
int getch(void);
void ungetch(int);
int getint(int *pn)
{
 int c, sign;
 while (isspace(c = getch()));
 if (!isdigit(c) && c != EOF && c != '+' && c != '-')
 {
  ungetch(c); return 0;
 }
 sign = (c == '-') ? -1 : 1;
 if (c == '+' || c == '-')
  c = getch();
for (*pn = 0; isdigit(c), c = getch())
*pn = 10 * *pn + (c - '0');
*pn *= sign;
 if (c != EOF)
 ungetch(c);
 return c;
}
```

Throughout `getint`, `*pn`is used as an ordinary `int`variable. We have also used `getch`and `ungetch`so the one extra character that must be read can be pushed back onto the input.

LECTURE NOTE 28

Pointers and Arrays

In C, there is a strong relationship between pointers and arrays, strong enough that pointers and arrays should be discussed simultaneously. Any operation that can be achieved by array subscripting can also be done with pointers. The pointer version will in general be faster but, at least to the uninitiated, somewhat harder to understand. The declaration

```
int a[10];
```

defines an array of size 10, that is, a block of 10 consecutive objects named `a[0]`, `a[1]`, .,`a[9]`. The notation `a[i]` refers to the `i`-th element of the array. If `pa` is a pointer to an integer, declared as

```
int *pa;
```

then the assignment

```
pa = &a[0];
```

sets `pa` to point to element zero of `a`; that is, `pa` contains the address of `a[0]`. Now the assignment

```
x =*pa;
```

will copy the contents of `a[0]` into `x`.

If `pa` points to a particular element of an array, then by definition `pa+1` points to the next element, `pa+i`points `i`elements after `pa`, and `pa-i`points `i`elements before. Thus, if `pa` points to `a[0]`, `*(pa+1)` refers to the contents of `a[1]`, `pa+i`is the address of `a[i]`, and `*(pa+i)` is the contents of `a[i]`. These remarks are true regardless of the type or size of the variables in the array `a`. The meaning of ``adding 1 to a pointer,'' and by extension, all pointer arithmetic, is that `pa+1` points to the next object, and `pa+i`points to the `i`-th object beyond `pa`. The correspondence between indexing and pointer arithmetic is very close. By definition, the value of a variable or expression of type array is the address of element zero of the array. Thus after the assignment

```
pa = &a[0];
```

`pa` and `a` have identical values. Since the name of an array is a synonym for the location of the initial element, the assignment `pa=&a[0]` can also be written as

```
pa = a;
```

Rather more surprising, at first sight, is the fact that a reference to `a[i]` can also be written as `*(a+i)`. In evaluating `a[i]`, C converts it to `*(a+i)` immediately; the two forms are equivalent. Applying the operator `&`to both parts of this equivalence, it follows that `&a[i]` and `a+i`are also identical: `a+i`is the address of the `i`-th element beyond `a`. As the other side of this coin, if `pa` is a pointer, expressions might use it with a subscript; `pa[i]` is identical to `*(pa+i)`. In short, an array-and-index expression is equivalent to one written as a pointer and offset.

There is one difference between an array name and a pointer that must be kept in mind. A pointer is a variable, so `pa=a` and `pa++` are legal. But an array name is not a variable; constructions like `a=pa` and `a++` are illegal.

When an array name is passed to a function, what is passed is the location of the initial element. Within the called function, this argument is a local variable, and so an array name parameter is a pointer, that is, a variable containing an address. We can use this fact to write another version of `strlen`, which computes the length of a string.

```
int strlen(char *s)
{
 int n;
 for (n = 0; *s != '\0', s++)
  n++;
 return n;
}
```

Since `s` is a pointer, incrementing it is perfectly legal; `s++` has no effect on the character string in the function that called `strlen`, but merely increments `strlen`'s private copy of the pointer. That means that calls like

```
strlen("hello, world");
strlen(array);
strlen(ptr);
```

all work.

As formal parameters in a function definition,

`char s[];` and `char *s;`

are equivalent; we prefer the latter because it says more explicitly that the variable is a pointer. When an array name is passed to a function, the function can at its convenience believe that it has been handed either an array or a pointer, and manipulate it accordingly. It can even use both notations if it seems appropriate and clear.

It is possible to pass part of an array to a function, by passing a pointer to the beginning of the subarray. For example, if `a` is an array,

`f(&a[2])`and `f(a+2)`

both pass to the function `f`the address of the subarray that starts at `a[2]`. Within `f`, the parameter declaration can read

`f(intarr[]) { ... }` or `f(int *arr) { ... }`

So as far as `f` is concerned, the fact that the parameter refers to part of a larger array is of no consequence.

If one is sure that the elements exist, it is also possible to index backwards in an array; `p[-1]`, `p[-2]`, and so on are syntactically legal, and refer to the elements that immediately precede `p[0]`. Of course, it is illegal to refer to objects that are not within the array bound

LECTURE NOTE 29

Address Arithmetic

If p is a pointer to some element of an array, then p++ increments p to point to the next element, and p+=i increments it to point i elements beyond where it currently does. These and similar constructions are the simples forms of pointer or address arithmetic. All types of arithmetic operations are not possible with pointers. The valid operations that can be performed using pointers are

(i) Addition of an integer to a pointer and increment operation.

(ii) Subtraction of an integer from a ponter and decrement operation.

(iii) Subtraction of a pointer from another pointer of same type.

The arithmetic operations that cannot be performed on pointers are as follows

(i) Addition, multiplication and division of two pointers.

(ii) Multiplication between pointer and any number.

(iii)Division of a pointer by any number.

(iv) Addition of float or double values to pointers.

The expression p+1 yields the correct machine address for the next variable of that type. Other valid pointer expressions:

p+i, ++p, p+=I, p-q

where p-q represents the No of array elements between p and q.
Since a pointer is just a mem address, we can add to it to traverse an array.
p+1 returns a ptr to the next array element.
Precedence level of * operator and increment/decrement operators is same and their associativity is from right to left. In reality, p+1 doesn't add 1 to the memory address, it adds the size of the array element.
Suppose p is an integer pointer and x is an integer variable. Now the main problem is to identify how the following pointer expressions given below are interpreted.

(i) x = *p++ is same as two expressions x = *p followed by p = p + 1.

(ii) x = (*p)++ is same as two expressions x = *p followed by *p = *p + 1.

(iii) x=*++p is same as two expressions p=p+1 followed by x=*p.

(iv) x=++*p is same as two expressions *p=*p+1 followed by x=*p

C is consistent and regular in its approach to address arithmetic; its integration of pointers, arrays, and address arithmetic is one of the strengths of the language. Let us illustrate by writing a rudimentary storage allocator. There are two routines. The first, alloc(n), returns a pointer to n consecutive character positions, which can be used by the caller of alloc for storing characters. The second, afree(p), releases the storage thus acquired so it can be reused later. The routines are ``rudimentary" because the calls to afree must be made in the opposite order to the calls made on `alloc`. That is, the storage managed by `alloc`and `afree` is a stack, or last-in, first-out. The standard library provides analogous functions called `malloc` and `free` that have no such restrictions.

The easiest implementation is to have `alloc`hand out pieces of a large character array that we will call `allocbuf`. This array is private to `alloc`and `afree`. Since they deal in pointers, not array indices, no other routine need know the name of the array, which can be declared `static` in the source file containing `alloc`and `afree`, and thus be invisible outside it. In practical implementations, the array may well not even have a name; it might instead be obtained by calling `malloc`or by asking the operating system for a pointer to some unnamed block of storage.

The other information needed is how much of `allocbuf`has been used. We use a pointer, called `allocp`, that points to the next free element. When `alloc`is asked for `n` characters, it checks to see if there is enough room left in `allocbuf`. If so, `alloc`returns the current value of `allocp`(i.e., the beginning of the free block), then increments it by `n` to point to the next free area. If there is no room, `alloc`returns zero. `afree(p)` merely sets `allocp`to `p`if `p` is inside `allocbuf`.

```
#define ALLOCSIZE 10000  static
char allocbuf[ALLOCSIZE];
static char *allocp = allocbuf;
char *alloc(int n)
{
if (allocbuf + ALLOCSIZE - allocp>= n) {
allocp += n;
return allocp - n;
} else
return 0;
}
void afree(char *p)
{
if (p >= allocbuf&& p <allocbuf + ALLOCSIZE) allocp = p;
}
```

In general a pointer can be initialized just as any other variable can, though normally the only meaningful values are zero or an expression involving the address of previously defined data of appropriate type. The declaration

```
static char *allocp = allocbuf;
```

defines `allocp`to be a character pointer and initializes it to point to the beginning of `allocbuf`, which is the next free position when the program starts. This could also have been written

```
static char *allocp = &allocbuf[0];
```

since the array name *is* the address of the zeroth element. The test

```
if (allocbuf + ALLOCSIZE - allocp>= n)
```

checks if there's enough room to satisfy a request for n characters. If there is, the new value of allocpwould be at most one beyond the end of allocbuf. If the request can be satisfied, allocreturns a pointer to the beginning of a block of characters (notice the declaration of the function itself). If not, allocmust return some signal that there is no space left. C guarantees that zero is never a valid address for data, so a return value of zero can be used to signal an abnormal event, in this case no space.

Pointers and integers are not interchangeable. Zero is the sole exception: the constant zero may be assigned to a pointer, and a pointer may be compared with the constant zero. The symbolic constant NULL is often used in place of zero, as a mnemonic to indicate more clearly that this is a special value for a pointer. NULL is defined in <stdio.h>. We will use NULL henceforth. Tests like

`if (allocbuf + ALLOCSIZE - allocp>= n)` and `if (p >= allocbuf&& p <allocbuf + ALLOCSIZE)`

show several important facets of pointer arithmetic. First, pointers may be compared under certain circumstances. If p and q point to members of the same array, then relations like ==, !=, <, >=, etc., work properly. For example,

```
p < q
```

is true if p points to an earlier element of the array than q does. Any pointer can be meaningfully compared for equality or inequality with zero. But the behavior is undefined for arithmetic or comparisons with pointers that do not point to members of the same array. (There is one exception: the address of the first element past the end of an array can be used in pointer arithmetic.) Second, we have already observed that a pointer and an integer may be added or subtracted. The construction

```
p + n
```

means the address of the n-th object beyond the one p currently points to. This is true regardless of the kind of object p points to; n is scaled according to the size of the objects p points to, which is determined by the declaration of p. If an intis four bytes, for example, the intwill be scaled by four.

Pointer subtraction is also valid: if p and q point to elements of the same array, and p<q, then q-p+1 is the number of elements from p to q inclusive. This fact can be used to write yet another version of strlen:

```
int strlen(char *s)
{
 char *p = s;
 while (*p != '\0')
 p++;
```

```
    return p - s;
   }
```

In its declaration, p is initialized to s, that is, to point to the first character of the string. In the while loop, each character in turn is examined until the '\0' at the end is seen. Because p points to characters, p++ advances p to the next character each time, and p-s gives the number of characters advanced over, that is, the string length. (The number of characters in the string could be too large to store in an int. The header <stddef.h>defines a type ptrdiff_tthat is large enough to hold the signed difference of two pointer values. If we were being cautious, however, we would use size_tfor the return value of strlen, to match the standard library version. size_tis the unsigned integer type returned by the sizeofoperator.

Pointer arithmetic is consistent: if we had been dealing with floats, which occupy more storage that chars, and if p were a pointer to float, p++ would advance to the next float. Thus we could write another version of allocthat maintains floats instead of chars, merely by changing char to float throughout allocand afree. All the pointer manipulations automatically take into account the size of the objects pointed to.The valid pointer operations are assignment of pointers of the same type, adding or subtracting a pointer and an integer, subtracting or comparing two pointers to members of the same array, and assigning or comparing to zero. All other pointer arithmetic is illegal. It is not legal to add two pointers, or to multiply or divide or shift or mask them, or to add float or double to them, or even, except for void *, to assign a pointer of one type to a pointer of another type without a cast.

LECTUER NOTE 30

CHARACTER POINTERS AND FUNCTIONS

A *string constant*, written as

```
"I am a string"
```

is an array of characters. In the internal representation, the array is terminated with the null character `'\0'` so that programs can find the end. The length in storage is thus one more than the number of characters between the double quotes.

Perhaps the most common occurrence of string constants is as arguments to functions, as in

```
printf("hello, world\n");
```

When a character string like this appears in a program, access to it is through a character pointer; `printf`receives a pointer to the beginning of the character array. That is, a string constant is accessed by a pointer to its first element. String constants need not be function arguments. If `pmessage`is declared as

```
char *pmessage;
```

then the statement

```
pmessage = "now is the time";
```

assigns to `pmessage`a pointer to the character array. This is *not* a string copy; only pointers are involved. C does not provide any operators for processing an entire string of characters as a unit. There is an important difference between these definitions:

```
                char amessage[] = "now is the time";
                char *pmessage = "now is the time";
```

`amessage` is an array, just big enough to hold the sequence of characters and `'\0'` that initializes it. Individual characters within the array may be changed but `amessage`will always refer to the same storage. On the other hand, `pmessage`is a pointer, initialized to point to a string constant; the pointer may subsequently be modified to point elsewhere, but the result is undefined if you try to modify the string contents.

We will illustrate more aspects of pointers and arrays by studying versions of two useful functions adapted from the standard library. The first function is `strcpy(s,t)`, which copies the string `t` to the string `s`. It would be nice just to say `s=t` but this copies the pointer, not the characters. To copy the characters, we need a loop. The array version first:

```
void strcpy(char *s, char *t)
{
inti;
 i = 0;
while ((s[i] = t[i]) != '\0')
i++;
```

```
 }
```

For contrast, here is a version of `strcpy`with pointers:

```
void strcpy(char *s, char *t)
{
 inti;
i = 0;
while ((*s = *t) != '\0')
 { s++; t++; }
}
```

Because arguments are passed by value, `strcpy` can use the parameters `s` and `t` in any way it pleases. Here they are conveniently initialized pointers, which are marched along the arrays a character at a time, until the `'\0'` that terminates `t` has been copied into `s`. In practice, `strcpy`would not be written as we showed it above. Experienced C programmers would prefer

```
void strcpy(char *s, char *t)
{
while ((*s++ = *t++) != '\0')
;
}
```

This moves the increment of `s` and `t` into the test part of the loop. The value of `*t++` is the character that `t` pointed to before `t`was incremented; the postfix `++` doesn't change `t` until after this character has been fetched. In the same way, the character is stored into the old `s` position before `s` is incremented. This character is also the value that is compared against `'\0'` to control the loop. The net effect is that characters are copied from `t` to `s`, up and including the terminating `'\0'`.

As the final abbreviation, observe that a comparison against `'\0'` is redundant, since the question is merely whether the expression is zero. So the function would likely be written as

```
void strcpy(char *s, char *t)
{
while (*s++ = *t++)
;
}
```

Although this may seem cryptic at first sight, the notational convenience is considerable, and the idiom should be mastered, because you will see it frequently in C programs. The `strcpy`in the standard library (`<string.h>`) returns the target string as its function value. The second routine that we will examine is `strcmp(s,t)`, which compares the character strings `s` and `t`, and returns negative, zero or positive if `s` is lexicographically less than, equal to, or greater than `t`. The value is obtained by subtracting the characters at the first position where `s` and `t` disagree.

```
int strcmp(char *s, char *t)
```

```
{
 inti;
 for (i = 0; s[i] == t[i]; i++)
if (s[i] == '\0')
 return 0;
return s[i] - t[i];
}
```

The pointer version of strcmp:

```
int strcmp(char *s, char *t)
{
 for ( ; *s == *t; s++, t++)
if (*s == '\0')
 return 0;
 return *s - *t;
}
```

Since ++ and -- are either prefix or postfix operators, other combinations of * and ++ and -- occur, although less frequently. For example,

```
*--p
```

decrements p before fetching the character that p points to. In fact, the pair of expressions

```
*p++ = val;

val = *--p;
```

are the standard idiom for pushing and popping a stack;The header <string.h> contains declarations for the functions mentioned in this section, plus a variety of other string-handling functions from the standard library.

Pointer Arrays; Pointers to Pointers

Syntax to declare pointer to an array is

*datatype (*pointer_variable)[size];*

For example

int (*ptr)[10]; ,Here ptr is a pointer that can point to an array of 10 integers, where we can initialize ptr with the base address of the array then by incre menting the value of ptr we can access different elements of array a[].
Since pointers are variables themselves, they can be stored in arrays just as other variables can. Let us illustrate by writing a program that will sort a set of text lines into alphabetic order, a stripped-down version of the UNIX program sort.

We need a data representation that will cope efficiently and conveniently with variable-length text lines. This is where the array of pointers enters. If the lines to be sorted are stored end-to-end in one long character array, then each line can be accessed by a pointer to its first character. `The`pointers themselves can bee stored in an array. Two lines can be compared by passing their pointers to `strcmp`. When two out-of-order lines have to be exchanged, the pointers in the pointer array are exchanged, not the text lines themselves.

This eliminates the twin problems of complicated storage management and high overhead that would go with moving the lines themselves.

The sorting process has three steps:
read all the lines of input
sort them
print them in order

As usual, it's best to divide the program into functions that match this natural division, with the main routine controlling the other functions. Let us defer the sorting step for a moment, and concentrate on the data structure and the input and output. The input routine has to collect and save the characters of each line, and build an array of pointers to the lines. It will also have to count the number of input lines, since that information is needed for sorting and printing. Since the input function can only cope with a finite number of input lines, it can return some illegal count like `-1` if too much input is presented. The output routine only has to print the lines in the order in which they appear in the array of pointers.

```
#include <stdio.h>

#include <string.h>

#define MAXLINES 5000

char *lineptr[MAXLINES];

intreadlines(char *lineptr[], intnlines);
```

```
void writelines(char *lineptr[], intnlines);
void qsort(char *lineptr[], int left, int right);
main()
{
 intnlines;
 if ((nlines = readlines(lineptr, MAXLINES)) >= 0) {
qsort(lineptr, 0, nlines-1);
 writelines(lineptr, nlines); return 0;
 } else {
printf("error: input too big to sort\n");
 return 1;
}
}
#define MAXLEN 1000
 intgetline(char *, int);
 char *alloc(int);
intreadlines(char *lineptr[], intmaxlines)
{
intlen, nlines;
 char *p, line[MAXLEN];
 nlines = 0;
 while ((len = getline(line, MAXLEN)) > 0)
if (nlines>= maxlines || p = alloc(len) == NULL)
return -1;
 else {
line[len-1] = '\0'; /* delete newline */
strcpy(p, line);
lineptr[nlines++] = p;
}
return nlines;
}
void writelines(char *lineptr[], intnlines)
{
inti;
 for (i = 0; i<nlines; i++)
printf("%s\n", lineptr[i]);
}
```

The main new thing is the declaration for `lineptr`:

```
char *lineptr[MAXLINES]
```

says that `lineptr`is an array of `MAXLINES` elements, each element of which is a pointer to a `char`. That is, `lineptr[i]` is a character pointer, and `*lineptr[i]` is the character it points to, the first character of the `i`-th saved text line.

Since `lineptr`is itself the name of an array, it can be treated as a pointer in the same manner as in our earlier examples, and `writelines`can be written instead as

```
void writelines(char *lineptr[], intnlines)
{
while (nlines-- > 0)
printf("%s\n", *lineptr++);
}
```

Initially, `*lineptr` points to the first line; each element advances it to the next line pointer while `nlines`is counted down.

With input and output under control, we can proceed to sorting.

```
void qsort(char *v[], int left, int right)
{
inti, last;
void swap(char *v[], inti, int j);
if (left >= right) /* do nothing if array contains */
return; /* fewer than two elements */
swap(v, left, (left + right)/2);
last = left;
for (i = left+1; i<= right; i++)
if (strcmp(v[i], v[left]) < 0)
swap(v, ++last, i);
swap(v, left, last);
qsort(v, left, last-1);
qsort(v, last+1, right);
}
```

Similarly, the swap routine needs only trivial changes:

```
void swap(char *v[], inti, int j)
{
char *temp;
temp = v[i];
v[i] = v[j];
v[j] = temp;
}
```

Since any individual element of `v` (alias `lineptr`) is a character pointer, `temp` must be also, so one can be copied to the other.

Multi-dimensional Arrays

C provides rectangular multi-dimensional arrays, although in practice they are much less used than arrays of pointers. In this section, we will show some of their properties.

Consider the problem of date conversion, from day of the month to day of the year and vice versa. For example, March 1 is the 60th day of a non-leap year, and the 61st day of a leap year.

Let us define two functions to do the conversions: `day_of_year`converts the month and day into the day of the year, and `month_day`converts the day of the year into the month and day. Since this latter function computes two values, the month and day arguments will be pointers:

```
month_day(1988, 60, &m, &d)
```

sets `m` to 2 and `d` to 29 (February 29th).

These functions both need the same information, a table of the number of days in each month (``thirty days hath September ..."). Since the number of days per month differs for leap years and non-leap years, it's easier to separate them into two rows of a two-dimensional array than to keep track of what happens to February during computation. The array and the functions for performing the transformations are as follows:

```
static char daytab[2][13] = {
{0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
{0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};
intday_of_year(int year, int month, int day)
{
inti, leap;
leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
for (i = 1; i< month; i++)
day += daytab[leap][i];
return day;
}
void month_day(int year, intyearday, int *pmonth, int *pday)
{

inti, leap;

leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
for (i = 1; yearday>daytab[leap][i]; i++)
yearday -= daytab[leap][i];
*pmonth = i;
*pday = yearday;
}
```

Recall that the arithmetic value of a logical expression, such as the one for leap, is either zero (false) or one (true), so it can be used as a subscript of the array daytab.

The array daytabhas to be external to both day_of_yearand month_day, so they can both use it. We made it char to illustrate a legitimate use of char for storing small non-character integers. daytabis the first two-dimensional array we have dealt with. In C, a two-dimensional array is really a one-dimensional array, each of whose elements is an array. Hence subscripts are written as

daytab[i][j] rather than daytab[i,j]

Other than this notational distinction, a two-dimensional array can be treated in much the same way as in other languages. Elements are stored by rows, so the rightmost subscript, or column, varies fastest as elements are accessed in storage order.

An array is initialized by a list of initializers in braces; each row of a two-dimensional array is initialized by a corresponding sub-list. We started the array daytabwith a column of zero so that month numbers can run from the natural 1 to 12 instead of 0 to 11. Since space is not at a premium here, this is clearer than adjusting the indices.

If a two-dimensional array is to be passed to a function, the parameter declaration in the function must include the number of columns; the number of rows is irrelevant, since what is passed is, as before, a pointer to an array of rows, where each row is an array of 13 ints. In this particular case, it is a pointer to objects that are arrays of 13 ints. Thus if the array daytabis to be passed to a function f, the declaration of f would be: f(intdaytab[2][13]) { ... }

It could also be

```
f(intdaytab[][13]) { ... }
```

since the number of rows is irrelevant, or it could be

```
f(int (*daytab)[13]) { ... }
```

which says that the parameter is a pointer to an array of 13 integers. The parentheses are necessary since brackets [] have higher precedence than *. Without parentheses, the declaration

```
int *daytab[13]
```

is an array of 13 pointers to integers. More generally, only the first dimension (subscript) of an array is free; all the others have to be specified.

Initialization of Pointer Arrays

Consider the problem of writing a function month_name(n), which returns a pointer to a character string containing the name of the n-th month. This is an ideal application for an internal static array. month_namecontains a private array of character strings, and returns a pointer to the proper one when called. This section shows how that array of names is initialized. The syntax is similar to previous initializations:

```
char *month_name(int n)
{
static char *name[] = {
```

```
"Illegal month",
"January", "February", "March",
"April", "May", "June",
"July", "August", "September",
"October", "November", "December"
};
return (n < 1 || n > 12) ? name[0] : name[n];
}
```

The declaration of `name`, which is an array of character pointers, is the same as `lineptr`in the sorting example. The initializer is a list of character strings; each is assigned to the corresponding position in the array. The characters of the `i`-th string are placed somewhere, and a pointer to them is stored in `name[i]`. Since the size of the array `name` is not specified, the compiler counts the initializers and fills in the correct number.

Pointers vs. Multi-dimensional Arrays

Newcomers to C are sometimes confused about the difference between a two-dimensional array and an array of pointers, such as `name` in the example above. Given the definitions

```
int a[10][20];
int *b[10];
```

then `a[3][4]` and `b[3][4]` are both syntactically legal references to a single `int`. But `a` is a true two-dimensional array: 200 `int`-sized locations have been set aside, and the conventional rectangular subscript calculation 20 * *row +col* is used to find the element `a[row,col]`. For `b`, however, the definition only allocates 10 pointers and does not initialize them; initialization must be done explicitly, either statically or with code. Assuming that each element of `b` does point to a twenty-element array, then there will be 200 `int`s set aside, plus ten cells for the pointers. The important advantage of the pointer array is that the rows of the array may be of different lengths. That is, each element of `b` need not point to a twenty-element vector; some may point to two elements, some to fifty, and some to none at all. Although we have phrased this discussion in terms of integers, by far the most frequent use of arrays of pointers is to store character strings of diverse lengths, as in the function `month_name`. Compare the declaration and picture for an array of pointers:

```
char *name[] = { "Illegal month", "Jan", "Feb", "Mar" };

with those for a two-dimensional array:

char aname[][15] = { "Illegal month", "Jan", "Feb", "Mar" };
```

Array of Pointers

We can declare an array that contains pointers as its elements. Syntax to declare array of pointer:

```
datatype *arrayname[size];
```

For example to declare an array of size 20 that contains integer pointers we can write

int *a[10];

where we can initialize each element of a[] with addresses of variables.

Functions returning Pointer

We can have a function that returns a pointer. Syntax to declare such type of function is

```
type *function_name(type1,type2,……);
```

For Example:

```
void main()
{
 int *p;
p=fun();
---------------------------
}
int *fun()
{
int a=5;
int *q=&a;
--------------------------- return
q;
}
```

Pointers to Functions

How to declare a pointer to a function?

Syntax: returntype_of_function (*pointer variable)(List of arguments); For example:

int (*p)(int,int); can be interpreted as p is a pointer to function which takes two integers as argument and returntype is integer.

How to make a pointer to a function?

Syntax:
pointer_variable=function_name_without_parantheses;
For Example:
p=test;
can be read as p is a pointer to function test.

How to call a function using pointer?

Syntax:
pointer_variable(ist of arguments);
OR
*(*pointer_variable)(List of arguments);*

The following program illustrates pointer to function

```
int getc()
{
return 10;
}
void put(int a)
{
printf("%d",a);
}
void main()
{
 int k;
int (*p)();                    /*You can write void *p();*/
void (*q)(int);                /*You can write void *q(int);*/
p=get; q=put; k=p(); q(k);
}
```

NOTE:

(i) In C every function is stored into physical memory location and entry point of that function address is stored into function name. If you assign a pointer to to the function then the base address of the function is copied into the pointer. Using this control is shifting from calling function to called function.
(ii) By default all functions are global, we can access from wherever we want. So there is no such significance to make a pointer to function.

In C, a function itself is not a variable, but it is possible to define pointers to functions, which can be assigned, placed in arrays, passed to functions, returned by functions, and so on. We will illustrate this by modifying the sorting procedure written earlier in this chapter so that if the optional argument -n is given, it will sort the input lines numerically instead of lexicographically.

A sort often consists of three parts - a comparison that determines the ordering of any pair of objects, an exchange that reverses their order, and a sorting algorithm that makes comparisons and exchanges until the objects are in order. The sorting algorithm is independent of the comparison and exchange operations, so by passing different comparison and exchange functions to it, we can arrange to sort by different criteria. This is the approach taken in our new sort. Lexicographic comparison of two lines is done by strcmp, as before; we will also need a routine numcmpthat compares two lines on the basis of numeric value and returns the same kind of condition indication as strcmpdoes. These functions are declared ahead of main and a pointer to the appropriate one is passed to qsort. We have skimped on error processing for arguments, so as to concentrate on the main issues.

```
#include <stdio.h>
#include <string.h>
#define MAXLINES 5000 /* max #lines to be sorted */

char *lineptr[MAXLINES]; /* pointers to text lines */

int readlines(char *lineptr[], intnlines);
void writelines(char *lineptr[], intnlines);
void qsort(void *lineptr[], int left, int right, int (*comp)(void *, void *));
int numcmp(char *, char *);
/* sort input lines */
main(intargc, char *argv[])
{
intnlines; /* number of input lines read */
int numeric = 0; /* 1 if numeric sort */
if (argc> 1 &&strcmp(argv[1], "-n") == 0)
numeric = 1;
if ((nlines = readlines(lineptr, MAXLINES)) >= 0)
{
qsort((void**) lineptr, 0, nlines-1,
(int (*)(void*,void*))(numeric ? numcmp : strcmp));
writelines(lineptr, nlines);
return 0;
}
else
{
printf("input too big to sort\n");
return 1;
}
}
```

In the call to qsort, strcmpand numcmpare addresses of functions. Since they are known to be functions, the & is not necessary, in the same way that it is not needed before an array name. We have written qsortso it can process any data type, not just character strings. As indicated by the function prototype, qsortexpects an array of pointers, two integers, and a function with two pointer arguments. The generic pointer type `void *` is used for the pointer arguments. Any pointer can be cast to `void *` and back again without loss of information, so we can call `qsort`by casting arguments to `void *`. The elaborate cast of the function argument casts the arguments of the comparison function. These will generally have no effect on actual representation, but assure the compiler that all is well.

```
void qsort(void *v[], int left, int right,
int (*comp)(void *, void *))
{
inti, last;
void swap(void *v[], int, int);
if (left >= right) /* do nothing if array contains */
return; /* fewer than two elements */
swap(v, left, (left + right)/2);
last = left;
for (i = left+1; i<= right; i++)
if ((*comp)(v[i], v[left]) < 0)
swap(v, ++last, i);
swap(v, left, last);
qsort(v, left, last-1, comp);
qsort(v, last+1, right, comp);
}
```

The declarations should be studied with some care. The fourth parameter of `qsort`is

```
int (*comp)(void *, void *)
```

which says that `comp` is a pointer to a function that has two `void *` arguments and returns an `int`.

The use of `comp` in the line `if ((*comp)(v[i], v[left]) < 0)`

is consistent with the declaration: `comp` is a pointer to a function, `*comp` is the function, and

```
(*comp)(v[i], v[left])
```

is the call to it. The parentheses are needed so the components are correctly associated; without them,

```
int *comp(void *, void *)
```

says that `comp` is a function returning a pointer to an `int`, which is very different. We have already shown `strcmp`, which compares two strings. Here is `numcmp`, which compares two strings on a leading numeric value, computed by calling `atof`:

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

```
/* numcmp: compare s1 and s2 numerically */
intnumcmp(char *s1, char *s2)
{ double v1, v2;
v1 = atof(s1);
v2 = atof(s2);
if (v1 < v2)
return -1;
else if (v1 > v2)
return 1;
else
return 0;
}
```

The `swap` function, which exchanges two pointers, is as follows

```
void swap(void *v[], inti, int j;)
{ void *temp;
temp = v[i];
v[i] = v[j];
v[j] = temp;
}
```

DYNAMIC MEMORY ALLOCATION

The memory allocation that we have done till now was static memory allocation.. So the memory that could be used by the program was fixed. So we couldnot allocate or deallocate memory during the execution of the program. It is not possible to predict how much memory will be needed by the program at run time. For example assume we have declared an array with size 20 elements, which is fixed. So if at run time values to be stored in array is less than 20 then wastage of memory occurs or our program may fail if more than 20 values are to be stored in to that array. To solve the above problems and allocate memory during runtime we are using dynamic memory allocation.

The following functions are used in dynamic memory allocation and are defined in <stdlib.h>

1. **malloc()**

   Declaration: *void *malloc(size_t size);*

   This function is used to allocate memory dynamically. The argument size specifies the number of bytes to be allocated. On success, malloc() returns a pointer to the first byte vof allocated memory. The returned pointer is of type void, which can be type cast to appropriate type of pointer. The memory allocated by malloc() contains garbage value
2. **calloc()**

   Declaration: *void *calloc(size_t n,size_t size);*

   This function is used to allocate multiple blocks of memory. The first argument specifies the number of blocks and the second one specifies the size of each block. The memory allocated by calloc() is initialized to zero.
3. **realloc()**

   Declaration: *void *realloc(void *ptr,size_t newsize);*

   The function realloc() is used to change the size of the memory block. It alters the size of the memory block without losing the old data. This function takes two arguments, first is a pointer to the block of memory that was previously allocated by mallloc() or calloc() and second one is the new size for that block.
4. **free();**

   Declaration: *void free(void *p);*

   This function is used to release the memory space allocated dynamically. Rhe memory released by free() is made available to the heap again and can be used for some other purpose. We should not try to free any memory location that was not allocated by malloc(), calloc() or realloc().

The following program illustrates Dynamic memory allocation.

```
#include<stdio.h>

#include<stdlib.h> void

main()

{ int *p,n,i;

 printf(“Enter the number of integers to be entered”);

scanf(“%d”,&n);

p=(int *)malloc(n*sizeof(int));    /* This is same as “(int *)calloc(n,sizeof(int))”*/

 /* If we write “(int *)malloc(sizeof(int))” then only 2 byte of memory will be allocated

dynamically*/

if(p==NULL)

{

printf(“Memory is not available”);

exit(1);

 }

 for(i=0;i<n;i++)

{

 printf(“Enter an integer”);

scanf(“%d”,p+i);

}

 for(i=0;i<n;i++)

printf(”%d\t”,*(p+i));

}
```

LECTURE NOTE 32

POINTER TO STRUCTURES

You may recall that the name of an array stands for the address of its *zero-th element*. Also true for the names of arrays of structure variables.

Consider the declaration:

```
struct stud {
int roll;
char dept_code[25];
float cgpa;

} class[100], *ptr ;
```

The name class represents the address of the zero-th element of the structure array. ptr is a pointer to data objects of the type struct stud. The assignment ptr = class; will assign the address of class[0] to ptr.
When the pointer ptr is incremented by one (ptr++) :
The value of ptr is actually increased by sizeof(stud).

It is made to point to the next record.

Once ptr points to a structure variable, the members can be accessed as:

```
ptr –> roll;
ptr –> dept_code;
ptr –> cgpa;
```

The symbol "–>" is called the *arrow* operator.

LECTURE NOTE 33

FILE

A File is a collection of data stored on a secondary storage device like hard disk. File operation is to combine all the input data into a file and then to operate through the C program. Various operations like insertion, deletion, opening closing etc can be done upon a file. When the program is terminated, the entire data is lost in C programming. If you want to keep large volume of data, it is time consuming to enter the entire data. But, if file is created these information can be accessed using few commands.

There are large numbers of functions to handle file I/O in C language. In this tutorial, you will learn to handle standard I/O(High level file I/O functions) in C. High level file I/O functions can be categorized as:

1. Text file
2. Binary file

A file can be open in several modes for these operations. The various modes are:

r	open a text file for reading
w	truncate to zero length or create a text file for writing
a	append; open or create text file for writing at end-of-file
rb	open binary file for reading
wb	truncate to zero length or create a binary file for writing
ab	append; open or create binary file for writing at end-of-file
r+	open text file for update (reading and writing)
w+	truncate to zero length or create a text file for update
a+	append; open or create text file for update
r+b or rb+	open binary file for update (reading and writing)
w+b or wb+	truncate to zero length or create a binary file for update
a+b or ab+	append; open or create binary file for update

fopen and freopen opens the file whose name is in the string pointed to by filename and associates a stream with it. Both return a pointer to the object controlling the stream, or if the open operation fails a null pointer. The error and end-of-file(EOF) indicators are cleared, and if the open operation fails error is set. freopen differs from fopen in that the file pointed to by stream is closed first when already open and any close errors are ignored.

Q1. Write a program to open a file using fopen().

Ans:

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

{

fopen()
file *fp;

fp=fopen(“student.DAT”, “r”);

 if(fp==NULL)

{

printf(“The file could not be open”);
exit(0);

}
```

Q2. Write a C program to read name and marks of n number of students from user and store them in a file. If the file previously exits, add the information of n students.

Ans:

```
#include <stdio.h>

int main()

{

char name[50];
int marks, i,n;
printf(“Enter number of students”);

scanf(“%d”, &n);

FILE *fptr;
fptr=(fopen(“C:\\student.txt”,”a”));

if (fptr==NULL){ printf("Error!");
exit(1);

 }

 for(i=0;i<n;++i)

{ printf("For student%d\nEnter name: ",i+1);
scanf("%s",name);

printf(“Enter marks”);

scanf(“%d”, &marks);

fprintf(fptr, “\nName: %s\nMarks=%d\n”, name, marks);

} fclose(fptr);

Return 0;
}
```

The fclose function causes the stream pointed to be flushed and the associated file to be closed. Any unwritten buffered data for the stream are delivered to the host environment to be written to the file; any unread buffered data are discarded. The stream is disassociated from the file. If the associated buffer was automatically allocated, it is deallocated. The function returns zero if the stream was successfully closed or **EOF** if any errors were detected.

Q.3. Write a program to read data from file and close using fclose function.

Ans:

```
#include <stdio.h>

int main()


int n
```

```
FILE *fptr;

if ((fptr=fopen("C:\\program.txt","r"))==NULL){

printf("Error! opening file");


exit(1);     // Program exits if file pointer returns NULL.


}

fscanf(fptr,"%d",&n);

printf("Value of n=%d",n);

fclose(fptr);


return 0;


}
```

Q4. Write a C program to write all the members of an array of strcures to a file using fwrite(). Read the array from the file and display on the screen.

Ans:

```
#include<stdio.h>

Struct s

{

Char name[50];

Int height;

};

Int main()

{

Struct s a[5], b[5];

FILE *fptr;

Int I;
Fptr=fopen(“file.txt”, “wb”);

For(i=0; i<5; ++i)

{
```

```
fflush(stdin);
printf("Enter name: ") ;
gets(a[i].name);
printf("Enter height: ");
scanf("%d",&a[i].height);
 }
fwrite(a,sizeof(a),1,fptr);
fclose(fptr);
fptr=fopen("file.txt","rb");
fread(b,sizeof(b),1,fptr);
for(i=0;i<5;++i)
{
printf("Name: %s\nHeight: %d",b[i].name,b[i].height);
   }    fclose(fptr);
}
```

MODULE 4

LECTURE NOTE – 34

ALGORITHM AND DATA STRUCTURE

Algorithm

- An algorithm is a step-by-step procedure of solving a problem.
- It is defined as any well-defined computational procedure that takes some value as input, produces some value known as output within a finite amount of time.
- Different algorithms can be devised to solve a single problem.

Real world applications of algorithms

- The Human Genome project
- The Internet
- E-commerce

Data Structure

- A data structure is a way to store and organize data in order to facilitate access and modifications.
- For example, an array stores similar types of data sequentially while a structure stores data of different types. These two are basic data structures and are used to formulate other data structures.
- For example, we can create a structure to store the information of a student. An array of such a structure will store the details of several students. So this array of structure will represent another data structure.
- Other well-known data structures include stack, queue, linked list, trees etc.
- No single data structure works well for all purposes, so it is important to understand multiple data structures.

Difference between algorithm and data structure

- An algorithm is a way to solve a problem where as a data structure is a way to store similar or heterogeneous data.
- Every algorithm must work on some specific data structure (be it an array or a tree or a list).
- In simpler terms, the way we store data is data structure and the way we access and modify that data is the algorithm.

Properties of Algorithm

An algorithm must have the following properties.

- Input: An algorithm must take some (possibly one) input value.
- Output: An algorithm must produce some definite outputs.
- Finiteness: An algorithm must terminate after a finite number of steps.
- Definiteness: An algorithm should be definite, i.e. it shouldn’t contain any ambiguity at any step.
- Effectiveness: One must be able to perform the steps in algorithm without applying any intelligence.

Types of algorithm

Algorithms generally fall under two categories:

- Iterative: These algorithms sequentially execute the input using loops and conditional statements. E.g. Linear search
- Recursive: These algorithms recursively break a larger problem into some smaller problems and solve those problems. Then they combine the result of smaller problems to solve the main given problem.

LECTURE NOTE – 35

ANALYSIS OF ALGORITHMS

As mentioned before, different algorithms can be devised to solve a single problem. The most efficient algorithm is always desired. This efficiency can be in terms of space required or time required to solve the problem.

Analysis of algorithm is a technique to compare the relative efficiency of different algorithms. Since speed of an algorithm can be different on different computers (depending on computer speed), time required to solve the problem cannot be expressed without ambiguity in terms of number of seconds required to solve a problem. Instead, it is measured by number of calculations required to solve the problem.

In real life situations, the actual number of operations is not used. Rather, the time required is expressed as a mathematical function of the input size. Two algorithms are compared on the basis of rate of growth of that function. Higher rate of growth means the algorithm will take more time as the input size increases.

Representation of analysis

The mathematical function is generally represented using big-O notation. Big-O notation gives an asymptotic upper bound of the function.

Let the function be a polynomial function

$$F(n)=an^k+bn^{(k-1)}+\ldots+z$$

where n is the variable, a,b,…,z are co-efficient.
The growth of the function is high. It is higher than any polynomial equation of power less than k. But it is lower than any polynomial of power greater than k.
So

$$F(n) < n^{(k+1)} \text{ and so on.}$$

Moreover, we can always find a constant α such that

$$F(n) <= \alpha\, n^k$$

So, n^k, $n^{(k+1)}$ and so on represent an upper bound to the function. It is in symbolic natation represented as

$$F(n)= O\,(n^k)$$
$$F(n)= O\,(n^{(k+1)}) \qquad \text{and so on}$$

Definition of Big-O

For two functions F(n) and G(n), if there exists a positive constant c, such that

$$0 <= F(n) <= c\, G(n), \text{ for some } n>=n0$$

Then F(n)=O(G(n))

Other representations

- Big-Ω :
  - This gives an asymptotic lower bound of a function.

    - Informally, it is the minimum time an algorithm will take to solve a problem.
    - For two functions F(n) and G(n), if there exists a positive constant c, such that
      0 <= c G(n) <= F(n), for some n>=n0
      Then F(n)= Ω (G(n))
- Big-Θ:
    - For two functions F(n) and G(n), if there exists two positive constant c1 and c2, such that
      0 <= c1 G(n) <= F(n) <=c2 G(n) , for some n>=n0
      Then F(n)= Θ (G(n))

Comparison of speed of two algorithms (an example)

Let there be two computers, A and B. Let A can execute 10^10 instructions per second while B can execute 10^7 instructions per second. Let there be two algorithms to solve a given problem, one of time complexity 2n^2 and another 50nlogn. Let the first algorithm be run on faster computer A and the other one in computer B.

When input size n=10^7,
Time required by computer A = 2 x (10^7)^2 / 10^10
= 20000 seconds (about 5.5 hours)
Time required by computer B = 50 x (10^7) x log(10^7) / 10^10
= 1163 seconds (less than 20 minutes)

Even though first computer was about 1000 times faster than second computer, it took a lot more time to execute an algorithm of higher complexity.

Cases considered in analysis

- Best case :
    - This is the analysis when the input is in favour of output.
    - For example, in sorting problem, the analysis done when the list is already sorted is called best case analysis.
    - This is not of much interest.
- Worst case :
    - Worst case analysis is the longest running time of an algorithm for a particular size of input.
    - For sorting problem, the worst case is when the number are already sorted but in reverse order.
    - This gives an upper bound of an algorithm.
- Average case :
    - It is the general case (not best not worst).
    - But in practice, it is as bad as the worst case.
    - It occurs fairly often (as often as worst case).
    - In sorting example, sorting any random list falls under this category.

An example of finding time complexity

Consider the following code snippet for sorting an array.

```
for(i=0;i<n;i++)
        for(j=i+1;j<n;j++)
                if(arr[i]>arr[j])
                        swap(arr[i],arr[j]);
```

Here the first element is compared with remaining n-1 elements, second element with remaining n-2 elements and so on.

So total number of comparisons is

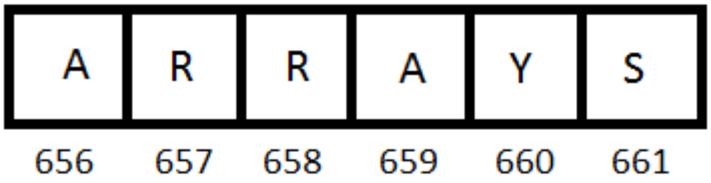
$$(n-1) + (n-2) + \ldots + 1 = n(n-1)/2$$

This is a polynomial equation of power 2. So the algorithm is said to be O(n^2).

LECTURE NOTE – 36

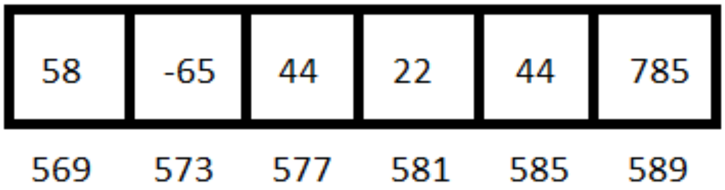
STORAGE STRUCTURE OF ARRAYS

- Arrays are linear data structure that store homogenous elements in contiguous memory locations.
- Every memory location has an address. The address is basically an integer.
- If more than one memory locations are required to store a value of a given data type (say x number of locations), then each element of the array will be stored at x locations apart from its predecessor and its successor.
- Consider an array of five characters where each character takes one byte storage (Which is a single memory location size in most architectures).



Here, ‘A’,’R’,’R’,’A’,’Y’,’S’ are the elements and the number under each is the memory location. Since each character takes a single byte, each elements are stored one element apart.

- Consider another array of five integers where each element takes four bytes storage.



Here, each element is four bytes (hence four memory locations) apart.

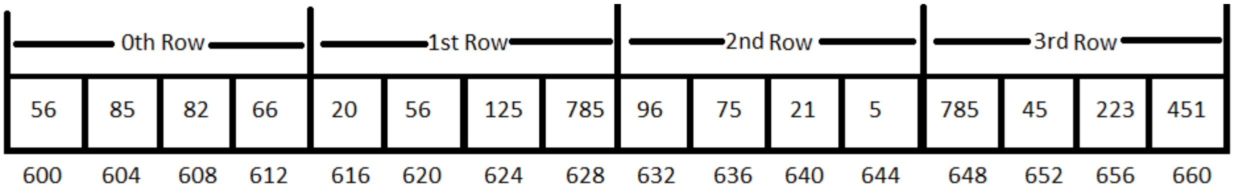
- In order to find the address of each elements, we can use the following code snippet –

```
int arr[n],i;       //n is the size of array
for(i=0;i<n;i++)
        printf(“%d\t”,&arr[i]);
```

2D array

- A 2-dimensional array is a collection of homogenous elements arranged in m rows and n columns.
- Since the actual memory is sequential, the array elements are stored sequentially in memory as shown in the figure below.

```
int arr[4][4]={
                        {56, 85, 82, 66},
                        {20, 56, 125, 785},
                        {96, 75, 21, 5},
                        {785, 45, 223, 451},
                };
```



0th Row				1st Row				2nd Row				3rd Row			
56	85	82	66	20	56	125	785	96	75	21	5	785	45	223	451
600	604	608	612	616	620	624	628	632	636	640	644	648	652	656	660

- Since it is an integer array, each element took four bytes of storage. So the memory location of each is separated by four units as well.
- The above method of storing is called row-major order. Here, the values are stored row wise i.e. all elements of zeroth row are stored first, followed by that of first row and so on.
- The address of (i,j)th cell can be found by

  Base address + (i*n) + j //m X n matrix
- Similarly in column major order, elements are stored column wise.
- C programming language uses row-major order.

Multi-dimensional array

- A multi-dimensional array has more than two dimensions.
- Consider a three by m by n array. It is basically a collection of three two dimensional array.
- Hence, to store it, the first two dimensional array is stored as mentioned before, followed by second 2D array followed by 3rd 2D array.

LECTURE NOTE – 37

SPARSE MATRICES

- A sparse matrix is a matrix in which most of the elements of the array are 0.
- There is no exact definition of how many zeroes should be there for a matrix to be called a sparse matrix.
- The following matrix can be considered sparse.

0	22	0	1	0	0	0	0	55
0	0	0	0	0	1	55	0	9
0	8	2	0	0	4	0	0	0
4	0	0	0	3	0	0	0	0
78	0	0	0	0	0	0	4	9
0	4	0	2	0	71	3	0	0

- A sparse matrix can be represented as a general matrix. But considering the fact that most of the elements are zero, we can use a different representations that uses less amount of space.
- One such representation is 3-tuple form.
- In three tuple form, each element is represented by three fields. The first field is row number, second is column number and the third is the value.
- But we need not store the three tuple form of zeroes.
- The three tuple form of the above matrix (first three rows) is as follows:

```
int sparse[][]={        0,1,22,
                        0,3,1,
                        0,8,55,
                        1,5,1,
                        1,6,55,
                        1,8,9,
                        2,1,8,              // And so on
                };
```

- Consider an n X m matrix. Let the number of non-zero elements be x. In 3-tuple form, we need 3*x number of storage. So if

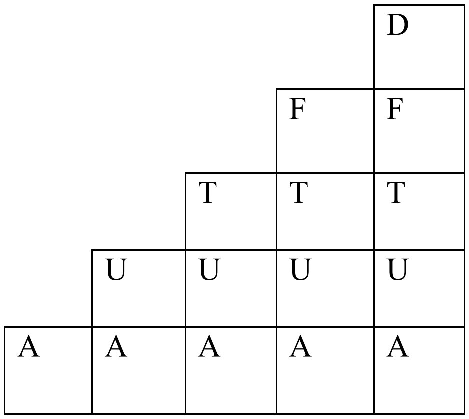
  3*x < n*m

  then 3-tuple form saves space. Otherwise, this method is not very efficient.

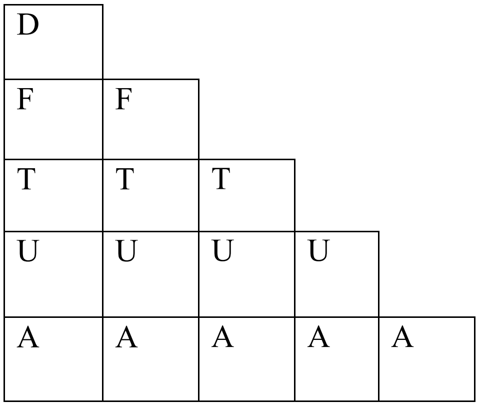
LECTURE NOTE – 38

STACKS

- A stack is a linear data structure like array with some restrictions.
- In a stack, insertion and deletion of elements is permitted only at one end. This end is called stack TOP.
- Because of this property, stack is also called Last-In-First-Out (LIFO) list.
- Insertion of element is called PUSH operation and deletion is called POP operation.
- A pictorial representation of stack insertion and deletion operation is as follows.



PUSH OPERATION



POP OPERATION

- A stack can be represented using an array or a linked list.
- Since insertion and deletion is done at one end, we don't need to traverse the entire list for these operations. So stack supports insertion and deletion in O(1) time i.e. in constant amount of time.
- When stack is empty, if we try to remove an element, it is called "Stack underflow".
- Similarly, when we add an element to a stack which is full, it is called "Stack overflow".
- A stack can theoretically grow to an infinite size. But in practice, there is limit. For array representation, the limit is array size whereas for linked list representation it is the amount of available memory.

Application of Stack

- A stack is useful for converting an arithmetic expression into Polish and reverse-Polish notation.

- In any programming language, each arithmetic operator has a priority. An expression can be evaluated based on that priority. But still, it is difficult to evaluate an expression in a computer.

- Polish mathematician Jan Lukasiewicz gave two methods of representing an arithmetic expression called pre-fix notation (aka Polish notation) and post-fix notation (aka reverse-Polish notation). The general arithmetic expression are said to be in in-fix form.

- In pre-fix and post-fix notations, the expressions are represented in such a way that the operator appears before or after the operands. As a result, no parenthesis are required to find which part is to be evaluated first.

- Consider the expression

  a+b*c

  This is to be evaluated as

  (a+(b*c))

  The pre-fix notation of it is

  +a*bc

  And post-fix notation is

  abc*+

- Now, the expressions can be evaluated using a stack.

- For evaluation of pre-fix operation, we scan the expression and push it in a stack one element at a time. Whenever we encounter two operands, we pop the two operands and the operator just beneath it and evaluate that part with the operator. The result is then pushed to the stack. The process continues till stack is not empty.

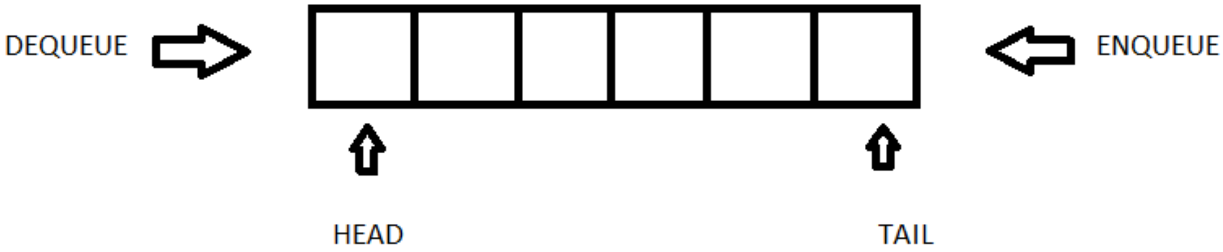
- Similarly for post-fix operation, we scan the expression and push it in a stack one element at a time. Whenever we encounter an operator, we pop the operator and the two operands just beneath it and evaluate that part with the operator. The result is then pushed to the stack. The process continues till stack is not empty.

- For +a*bc, + is pushed followed by a and then * and then b and then c. When c is pushed, we have two operands b and c together. It is evaluated with operator *. Let the result be D. When D is pushed, the structure of the stack is +aD. Now a and D are two operands together and are evaluated with +. When plus is popped, the process stops as stack is now empty. The result a+D is the final result.

LECTURE NOTE – 39

QUEUE

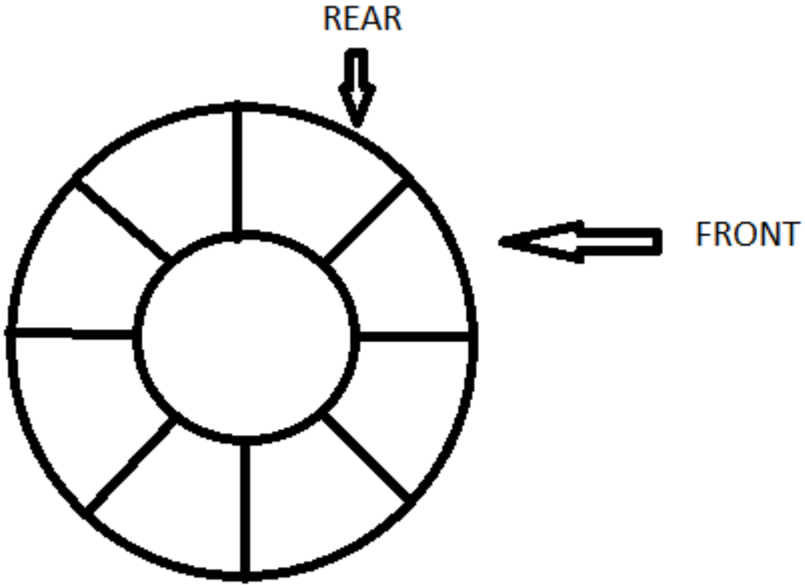
- A queue is also a linear data structure like stack. But unlike stack, here insertion is done at one end and deletion at the other end.
- Because of this property, it is also called First-In-First-Out (FIFO) list.
- The insertion process is called ENQUEUE and deletion process is called DEQUEUE.
- The end where element is inserted is called the TAIL of the queue, whereas the other end is called HEAD. So initially head=tail.



- Queue can also be represented using an array or a linked list.
- If we DEQUEUE an empty queue, it is called queue underflow. Similarly queue overflow is when we ENQUEUE a full queue.
- In array representation, when head=tail+1, then queue is full.

Circular queue

- A circular queue is one in which after reaching the last position, next element is added in the first position (provided it is free).
- In general queue, on reaching the end of the queue, no more elements can be added even though elements in the beginning are empty (as a result of many dequeue process). Circular queue overcomes this limitation of the queue.
- This also can be represented using an array or a linked list.



- In array representation, the list is empty when front=rear.
- Also in array representation, the list is full when front=0 and rear=n-1 where n is the size of the array; or when rear=front-1.

De-queue

- De-queue is a double ended queue. It supports insertion and deletion at both end of the list but not anywhere in between.
- It is basically a generalization of stack and queue.

REFERENCES

1. Fundamentals of computer by P K Sinha
2. Programming in C by Reema Thereja
3. Programming in ANSI C by E Balagurusamy
4. O'REILLY, "Practical C Programming", 3rd Edition
5. Yashavant P.kanetkar, "Let Us C", 5th Edition
6. Brian W. kernighan and Dennis M. Ritchie, "The C Programming Language"
7. Greg Perry, "C by Example"
8. Stephen Prata, "C Primer Plus", 5th Edition