

DATA STRUCTURES AND C PROGRAMMING

[CONTENTS]

LESSON 1 PROGRAM DEVELOPMENT STYLES AND BASICS OF C	1
Program Development Methodologies - Programming Style - Stepwise Refinement and Modularity - Problem Solving Techniques - Algorithm – Flowcharts – Pseudocode – Sequence and Selection - Recursion vs. Iteration - Overview of Compilers and Interpreters - Structure of a C Program - Programming Rules - Executing the Program.	
LESSON 2 CONSTANTS & VARIABLES	14
Introduction - Character set - C Tokens - Keywords and Identifiers – Constants – Variables.	
LESSON 3 DATA TYPES	18
Introduction - Primary data types - Declaration of variables - Assigning values to variables - Reading data from keyword - Defining symbolic constants.	
LESSON 4 OPERATORS	21
Operators of C - Arithmetic operators - Relational operators - Logical operators – Assignment operators - Increment and decrement operators - Conditional operator - Bitwise operators - Special operators.	
LESSON 5 EXPRESSIONS	24
Expressions - Arithmetic expressions - Precedence of arithmetic operators - Type conversion in expressions - Mathematical functions - Managing input and output operations.	
LESSON 6 CONTROL STATEMENTS	32
Control Statements - Conditional Statements - The Switch Statement - Unconditional Statements- Decision Making and Looping.	
LESSON 7 ARRAYS & STRINGS	41
Introduction - One Dimensional Array - Two-Dimensional Arrays - Multidimensional Array- Handling of Character Strings - Declaring and Initializing String Variables - Arithmetic Operations on Characters - String Handling Functions.	
LESSON 8 USER-DEFINED FUNCTIONS	47
Introduction - Need for User-Defined Functions - The Form of C Functions - Category of Functions - Handling of Non-Integer Functions – Recursion - Functions with Arrays.	
LESSON 9 STORAGE CLASSES	50
Introduction - Automatic Variables (local/internal) - External Variables - Static Variables - Register Variables – ANSI C Functions.	
LESSON 10 POINTERS	52
Understanding Pointers - Accessing the Address of a Variable - Accessing a Variable through its Pointer - Pointer Expressions - Pointers and Arrays - Pointers and Character Strings.	

LESSON 11 POINTERS AND FUNCTIONS	58
Pointers as Function Arguments - Pointers and Structures - The Preprocessor.	
LESSON 12 STRUCTURES	61
Introduction - Structure Definition - Array Vs Structure - Giving Values to Members - Structure Initialization - Comparison of Structure Variables - Arrays of Structures.	
LESSON 13 STRUCTURES AND UNION	65
Introduction - Structures within Structures - Structures and Functions – Unions - Size of Structures - Bit Fields.	
LESSON 14 FILES	70
Introduction - Defining and Opening a File - Closing a File - Input/output operations on Files.	
LESSON 15 ERROR HANDLING DURING FILE I/O OPERATIONS	75
Introduction to Error Handling - Random Access to Files.	
LESSON 16 COMMAND LINE ARGUMENTS	77
Command Line Arguments - Program for Command Line Arguments.	
LESSON 17 LINEAR DATA STRUCTURES	78
Introduction - Implementation of a list - Traversal of a list - Searching and retrieving an element - Predecessor and Successor- Merging of lists.	
LESSON 18 STACK	84
Introduction – Stack - Representation and terms – Operations – Insertion - Deletion – Implementations.	
LESSON 19 LINKED LIST	86
Introduction - Linked list with header - Linked list without header.	
LESSON 20 DOUBLY LINKED LIST	91
Introduction - Doubly linked list - Insertion operation - Deletion operation - Difference between single and doubly linked list.	
LESSON 21 QUEUES, SEARCHING AND SORTING - BINARY SEARCH	95
Introduction - Queues - Operations of a queue - Searching - Linear Searching - Binary search Algorithm.	
LESSON 22 SORTING	98
Introduction – Sorting - Comparison with other method.	
LESSON 23 BUBBLE & QUICK SORTING	101
Introduction - Bubble sort: Time complexity - Quick sort: Time complexity.	

LESSON 24 TREE SORT & HEAP SORT 104

Introduction - Tree sort - Heap sort.

REFERENCES 107

APPENDICES 108

- **BHARATHIAR UNIVERSITY B.Sc. COM.SCI DEGREE COURSE SYLLABUS**
- **MODEL QUESTION PAPERS**
- **PRACTICAL LIST**

Online Student Resources available at
<http://thottarayaswamy.0fees.net>

LESSON 1 PROGRAM DEVELOPMENT STYLES AND BASICS OF C

Outline

Program Development Methodologies - Programming Style - Stepwise Refinement and Modularity - Problem Solving Techniques - Algorithm – Flowcharts – Pseudocode - Sequence and Selection - Recursion vs. Iteration - Overview of Compilers and Interpreters - Structure of a C Program - Programming Rules - Executing the Program

Programming Development Methodologies.

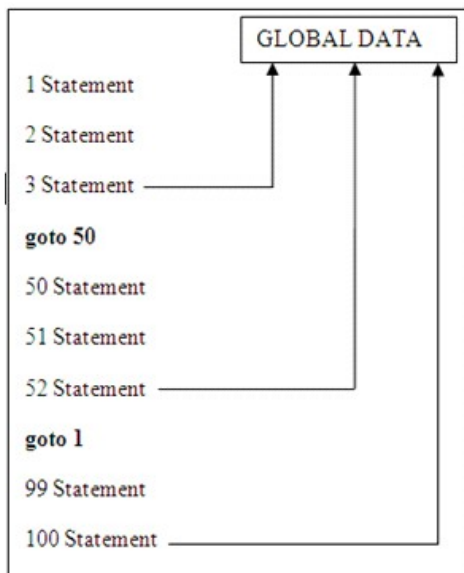
The programs in high level language can be developed by different technique. The different techniques are as follows:

Monolithic Programming

a) In monolithic programming languages, both in assembly and high level languages such as BASIC and assembly level languages of processor, the data variables declared are global and the statements are written in sequence.

b). The program contains jump statements such as **goto** that transfers control to any statement as specified in it (see Fig. 1). The global data can be accessed from any portion of the program. Due to this reason, the data is not fully protected.

c) The concept of sub-programs does not exist and hence this technique is useful for small programs.



Procedural Programming

(a) In the procedural programming languages such as FORTRAN and COBOL, programs are divided into a number of segments-called sub-programs. Thus, it focuses on functions apart from data. Fig. 2 describes a program of procedural type. It shows different subprograms accessing the same global data. Here also, the programmer can observe lack of secrecy.

(b) The control of program is transferred using unsafe **goto** statement.

(c) Data is global and all the sub-programs share the same data.

(d) These languages are used for developing medium-sized applications.

Structured Programming

BASIC is widely believed to be an unstructured language. Modularity is not supported by BASIC. It becomes difficult to debug a lengthy program written in BASIC. One should organize the programs in modules and should write programs that are easy for others to read. Modular programming concept is used in structured programming. A big program is to be split into modules or subroutines. Instead of running through a number of lines, divide it into small modules. These modules are easier to debug.

- a) Larger programs are developed in structured programming such as Pascal and C. Here programs are divided into multiple sub-modules and procedures.
- b) Each procedure performs different tasks.
- c) Each module has its own set of local variables and program code as shown in Fig. 3. Here, the different modules are shown accessing the global data.
- d) User-defined data types are introduced.

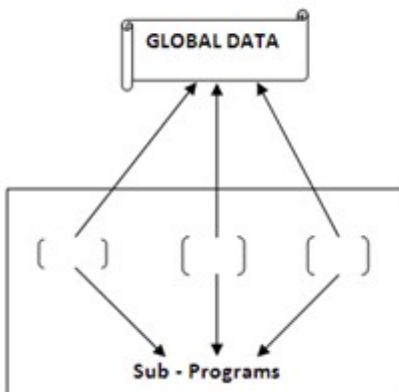


Figure. 2 Program in procedural programming

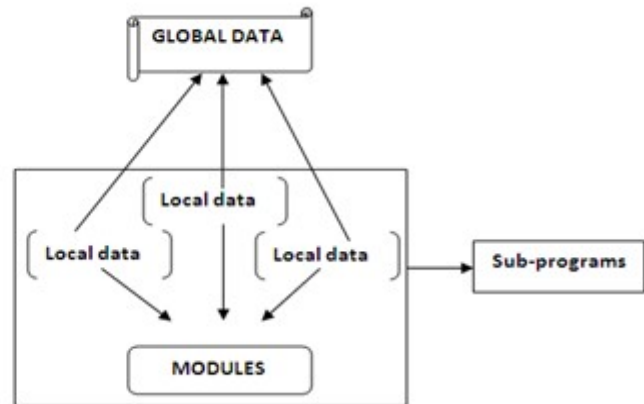


Figure 2 Program in structured programming

Programming Style.

The program development styles are of two types:

- 1) Top-down design method
- 2) Bottom-up design method

1) *Top-down* design method: In the top-down design method, designing the system is initiated from the top. The solution of the problem can be divided into smaller blocks. Initial view of solution is not clear just like when an aeroplane descends from a great height the view to the ground is not clear but when the plane approaches the ground things get clear and everything can be seen clearly with our naked eyes. In a similar way, design is carried out. The strategy of designing a system is to divide the task into many sub-tasks. We make a group that comprises task and sub-task. Each sub-task can be performed in a function. The function performs a well-defined job or task.

There are various drawbacks of the top-down approach. Here in the design strategy

importance is given to functions rather than data. For example, in turbo C, we cannot give data after `clrscr ()`. If data is declared after `clrscr ()`, error will be shown by the compiler.

2) *Bottom-up* design method: Here the design of a problem starts from the bottom and it goes up to the top. Some designers prefer this approach. They attempt to find the last blocks of design and implement them first. By building various blocks from the bottom they ascend towards the top, assemble all parts designed as planned, and make the final product. Most of the professional designers do not prefer this approach because it makes the design complex and difficult to implement.

Stepwise Refinement and Modularity

A lengthy program can be divided into small programs. This step enables one to follow the programs easily. It makes it easy to solve a program and bugs in it can be traced easily. The computer itself cannot do anything. It follows only the instructions given by the programmer. Even if the subprograms are difficult to understand after dividing the main, sub-programs may be further subdivided into smaller programs. This process can be continued until the programs are practical.

In any organization, the top management cannot look after every activity of the organization. They must concentrate on some specific aims or undertaking of the company and delegate responsibilities to subordinates. In addition, middle level managers also share their responsibilities with their subordinates. At every level, the work is divided and shared with the subordinates. The same concept is to be applied to computer programming. Even if a project seems to be small and one is capable of handling it alone, it is better to do the same among a group of programmers. This step helps in designing the project quickly and quality can be assured. However, it is a good approach to divide the problem into sub-programs and solve each sub-program in a systematic manner.

The above procedure is called top-down refinement. Large programs can be easily solved by this method. This approach involves deferment of detailed considerations at the first stage. Even at this stage attention must be paid to accuracy and inflexibility. The programmer decides the exact way of division of work between functions. Thus, before working it is essential to know what a particular function does. Normally it is somewhat difficult to divide the tasks into separate functions. Lot of thinking has to go at the initial stage. Moreover, some revision is always expected in the function and scope must be there for improvement. Therefore, a function can be altered in future. To overcome this confusing situation there are a few guidelines for dividing the program among functions.

These guidelines are as follows:

1) The programmer should explain the use of a function in a few words. If the programmer fails to do so and writes a lengthy unwanted description, it means that more than enough details are provided. The detailed descriptions are not to be disclosed and it should not come into view until the next stage of refinement comes. The programmer must rethink about the dividing of program. Thus, in short, we can say that each function should perform only one task with full accuracy.

2) We know that a big organization has hierarchical levels of operations. At the top, there are several ranks of managers controlling the operations. The middle level managers obtain information from the lower level. However, the entire information is not passed. As it is to upper level managers. Only the relevant information needed for that manager is sent. In the same fashion, whatever

instruction or information one gets from higher level managers is not passed as it is to junior level managers or subordinates but only applicable information is passed on to them. The function we are going to design should follow the above principle.

Thus, in short, we can say that each function must hide some details. The refinement decides what a particular function performs. It also decides a function's pre and post conditions, that is, what data the function takes and what output it will provide. The data used in these functions must be exactly declared.

The data that can be used in a function is of five types.

- 1) Input parameters: These parameters are utilized by the function. The function cannot change the value of the variables permanently if values are passed only by method 'pass by value'.
- 2) Output parameters: These parameters hold the values of results calculated. These values can be returned to the main function. 'Call by reference' can be used to avoid restriction of returning one value at a time.
- 3) In/out parameters: The parameters are used for both the purposes, that is, input as well as output. The function uses the initial value of the variable (parameter) and modifies it. As the same variable is used for both the purposes, the value is changed. In C, 'call by address' can be used to do this..In C++ call by address and reference can be applied to do this.
- 4) Local variables: These variables are declared inside the function body and their scope is limited to the same function body. When a function is invoked, the local variables come into existence and as soon as the execution is completed they are destroyed. Their life is subject to execution of the function.
- 5) Global variables: These variables are global in nature and can be accessed from anywhere in the program including sub-programs. These variables are insecure because their value can be changed by any function. Sometimes due to use of global variables a program gives unwanted secondary results. If the global and local variables are declared with the same name, a more complex situation may arise. If variables with the same name are declared with local and global declarations, the local variable gets first priority. To avoid such problems, avoid using global variables. Preferably, the variable can be declared as a constant.

Problem Solving Techniques

There are three ways to represent the logical steps for finding the solution to a given problem.

- 1) Algorithm
- 2) Flowchart
- 3) Pseudocode

In an algorithm, description of steps for a given problem is provided. Here stress is given on text.

A Flow chart represents solution to a given problem graphically. Pictorial representation of the logical steps is a flowchart. Another way to express the solution to a given problem is by means of pseudo code.

Algorithm

Algorithm is a very popular technique used to obtain solution for a given problem. An algorithm is defined as a finite set of steps which provide a chain of actions for solving a definite nature of problem. Each step in the algorithm should be well defined. This step will enable the reader to translate each step into a program. Gradual procedure for solving a problem is illustrated in this section.

An algorithm is a well-organized, pre-arranged, and defined textual computational module that receives some value or set of values as input and provides a single or a set of values as output. These well-defined computational steps are arranged in a sequence, which processes the given input into output. Writing precise description of the algorithm using an easy language is most essential for understanding the algorithm. An algorithm is said to be accurate and truthful only when it provides the exact required output. The lengthy procedure is sub-divided into smaller parts, which makes it easy to solve a given problem. Every step is known as an instruction. In our daily life we come across numerous algorithms for solving problems. We perform several routine tasks, for example, riding a bike, lifting a phone, making a telephone call, switching on the television, and so on.

Let us take an example. To establish a telephonic communication between two subscribers the following steps are to be followed.

- 1) Dial the phone number
- 2) Phone rings at the called party
- 3) Caller waits for the response
- 4) Called party picks up the phone
- 5) Conversation begins between them
- 6) Release of connection

Another real life example is accessing the Internet through an Internet Service Provider with dial up facility. To log on to the Internet, the following steps are to be followed.

- 1) Choose the Internet Service Provider for accessing the Internet
- 2) Obtain from service provider a dial up number
- 3) Acquire IP address of the service provider
- 4) Acquire login ID and password
- 5) Run the Internet browsing software

Analyzing algorithm: When one writes an algorithm, it is essential to know how to analyze the algorithm. Analyzing an algorithm refers to calculating or guessing resources needed for that algorithm. Resources means computer memory, processing time, logic gates and so on. In all the above factors, time is most important because the program developed should be faster in processing. The analysis can also be made by reading the algorithm for logical accuracy, tracing the algorithm, implementing it, checking with some data and with some mathematical technique to confirm its accuracy.

Algorithms can also be expressed in a simple method. It will help the user to put it into operation easily. However, this approach has a few drawbacks. It requires more space and time. It is very essential to consider factors such as time and space requirements of an algorithm. With minimum system resources such as CPU time and memory, an efficient algorithm must be developed.

Rate of growth: In practice, it is not possible to act upon a simple analysis of an algorithm to conclude the execution time of an algorithm. The execution time is dependent upon the machine and the way of implementation. Timing analysis depends upon the input required. To accurately analyze the time it is also very essential to know the exact directives executed by the hardware and the execution time passed for each statement.

Classification of Algorithms

Algorithm can be classified into three types. The classification of an algorithm is based on repetitive steps and based on control transfer from one statement to another. Based on repetitive steps, an algorithm can be classified into two types.

- 1) Direct algorithm
- 2) Indirect algorithm

1) **Direct algorithm:** In this type of algorithm the number of iterations is known in advance. For example, for displaying numbers from 1 to 10, the loop variable should be initialized from 1 to 10.

The statement would be as follows:

for (j=1;j<=10;j++)

In the above statement, it is predicted that the loop will iterate for ten times.

2) **Indirect algorithm:** In this type of algorithm repetitive steps are executed a number of times. Exactly how many repetitions are to be made is unknown.

For example, finding the first five Armstrong numbers from 1 to n , where n is the fifth Armstrong number and finding the first three palindrome numbers.

Based on control transfer the algorithms are classified into the following three types.

- a) Deterministic
- b) Non-deterministic
- c) Random

a) **Deterministic:** A deterministic algorithm is based on either following a 'yes' path or a 'no' path based on the condition. In this type of algorithm when control logic comes across a decision symbol, two paths 'yes' and 'no' are shown. Program control follows one of the routes depending upon the condition.

The examples include testing whether a number is even or odd or positive or negative.

b) **Non-deterministic:** In this type of algorithm, we have one of the multiple paths to reach to the solution.

Example Finding a day of a week

c) **Random:** After executing a few steps, the control of the program transfers to another step randomly in a random algorithm.

Example Random search.

Many a time, we have to follow one more algorithm in which the exact number of iterations is not known. Best estimation has to be done in order to get better results. More iteration is expected in this case.

3) **Infinite algorithm:** This algorithm is based on better estimates of results. The number of steps is unknown. The process will be continued until best results are achieved.

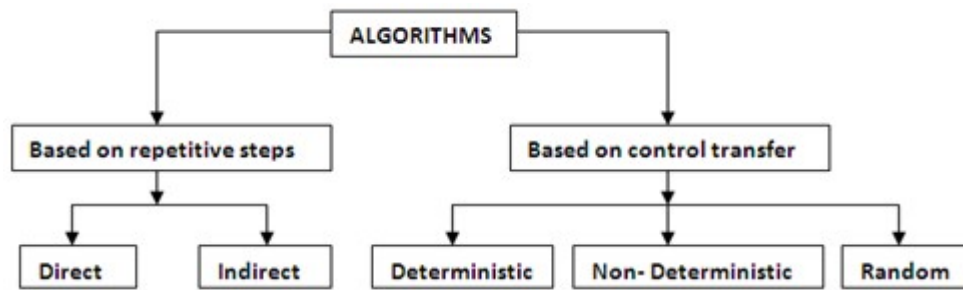


Fig.4 Classifications of algorithms

Example finding shortest paths from a given source to all destinations in the network.

Flowcharts

A flowchart is a visual representation of the sequence of steps for solving a problem. It enlightens what comes first, second, third, and so on. A completed flowchart enables you to organize your problem into a plan of actions. Even for designing a product a designer many times has to draw a flowchart. It is a working map of the final product. This is an easy way to solve the complex designing problems. The reader follows the process quickly from the flowchart instead of going through the text.

A flowchart is an alternative technique for solving a problem. Instead of descriptive steps, we use pictorial representation for every step. It shows a sequence of operations. A flowchart is a set of symbols, which indicates various operations in the program. For every process, there is a corresponding symbol in the flowchart. Once an algorithm is written, its pictorial representation. It can be done using flowchart symbols. In other words, a pictorial representation of a textual algorithm is done using a flowchart. We give below some commonly used symbols in flowcharts.

Start and end: The start and end symbols indicate both the beginning and the end of the flowchart. This symbol looks like a flat oval or is egg shaped. Fig.5 shows the symbol of Start/Stop; only one flow line is combined with this kind of symbol. We write START, STOP or END in the symbols of this kind. Usually this symbol is used twice in a flowchart, that is, at the beginning and at the end.

Decision or test symbol: The decision symbol is diamond shaped. This symbol is used to take one of the decisions. Depending on the condition the decision block selects one of the alternatives. While solving a problem, one can take a single, two or multiple alternatives depending upon the situation. All these alternatives are illustrated in this section. A decision symbol with a single alternative is shown in Fig.5. In case the condition is satisfied /TRUE a set of statement(s) will be executed otherwise for false the control transfers to exit.

Single alternative decision: Here more than one flow line can be used depending upon the condition. It is usually in the form of a 'yes' or 'no' question, with branching flow lines depending upon the answer. With a single alternative, the flow diagram will be as per Fig.6.

Two alternative decisions: In Fig.7 two alternative paths have been shown. On satisfying the condition statement(s) pertaining to 1 action will be executed, otherwise the other statement(s) for action 2 will be executed. **Multiple alternative decisions:** In Fig.8 multiple decision blocks are shown. Every decision block has two branches. In case the condition is satisfied, execution of statements of appropriate blocks take place, otherwise next condition will be verified. If condition 1 is satisfied then block 1 statements are executed. In the same way, other decision blocks are executed.

Connector symbol: A connector symbol has to be shown in the form of a circle. It is used to establish

the connection, whenever it is impossible to directly join two parts in a flowchart. Quite often, two parts of the flowcharts may be on two separate pages. In such a case, connector can be used for joining the two parts. Only one flow line is shown with this symbol. Only connector names are written inside the symbol, that is, alphabets or numbers. Fig.9 shows the connector symbol.

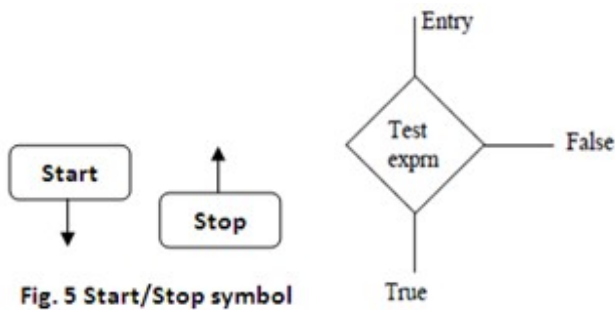


Fig.6 Single alternative decision

Process symbol: The symbol of process block should be shown by a rectangle. It is usually used for data handling, and values are assigned to the variables in this symbol. Fig.10 shows the process symbol. The operations mentioned within the rectangular block will be executed when this kind of block is entered in the flowchart. Sometimes an arrow can be used to assign the value of a variable to another. The value indicated at its head is replaced by the tail values. There are two flow lines connected with the process symbol. One line is incoming and the other line goes out.

Loop symbol: This symbol looks like a hexagon. This symbol is used for implementation of for loops only. Four flow lines are associated with this symbol. Two lines are used to indicate the sequence of the program and remaining two are used to show the looping area, that is, from the beginning to the end. For the sake of understanding, Fig.11 illustrates the working of for loop. The variable J is initialized to 0 and it is to be incremented by a step of 2 until it reaches the final value 10. For every increased value of J, body of the loop is executed. This process will be continued until the value of J reaches 10. Here the next block is shown for the repetitive operation.

Input/output symbol: Input/output symbol looks like a parallelogram, as shown in Fig.12. The input/ output symbol is used to input and output the data. When the data is provided to the program J for processing, then this symbol is used. There are two flow lines connected with the input/ output symbol. One line comes to this symbol and the other line goes from this symbol. As per Fig.12

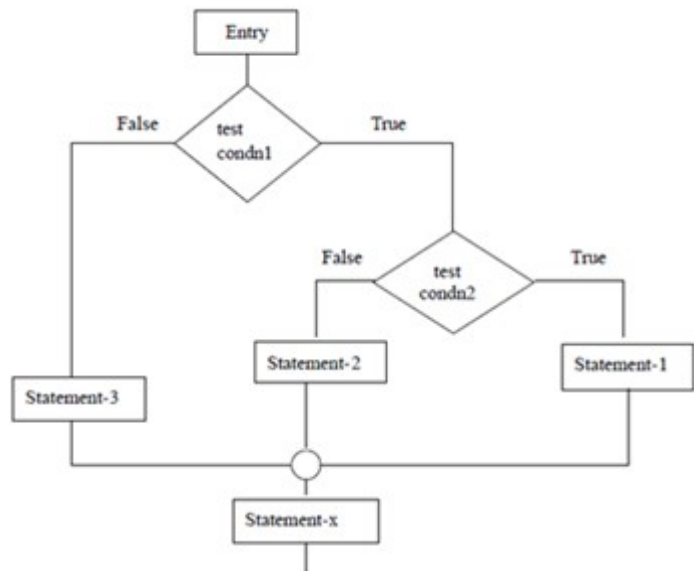
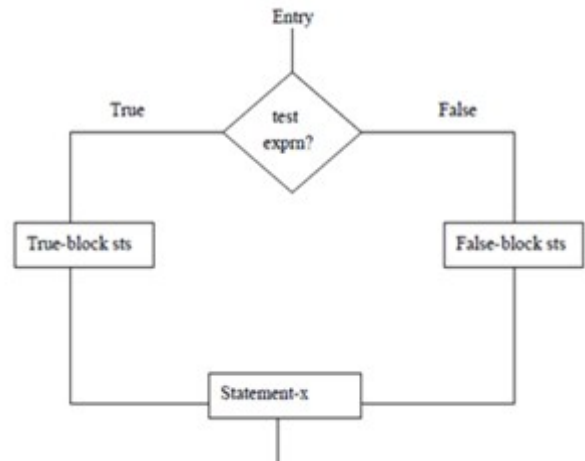


Fig. 8 Multiple alternative decisions

compiler reads the values of X, Y and in the second figure the result is displayed on the monitor or the printer.

Delay symbol: Symbol of delay is just like 'AND' gate. It is used for adding delay to the process. It is associated with two lines: One is incoming and the other is outgoing, as shown in Fig. 13.

Manual input symbol: This is used for assigning the variable values through the keyboard, where as in data symbol the values are assigned directly without manual intervention. Fig.14 represents the symbol of manual input.

In addition, the following symbols (Fig.15) can be used in the flowchart and they are parts of flowcharts.

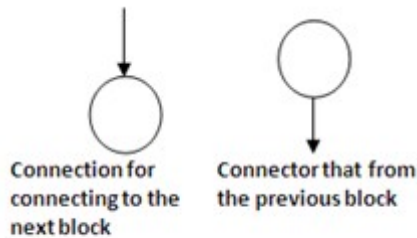


Fig. 9 Connector Symbol

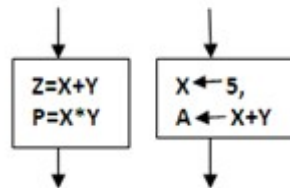


Fig. 10 Process Symbol

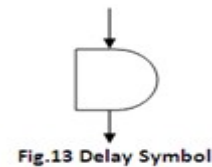


Fig.13 Delay Symbol

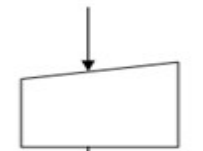


Fig. 14 Manual input

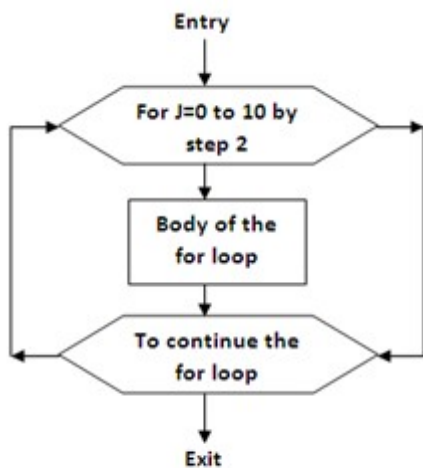


Fig. 11 For Loop

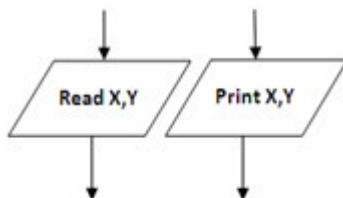


Fig.12 Input/output Symbol

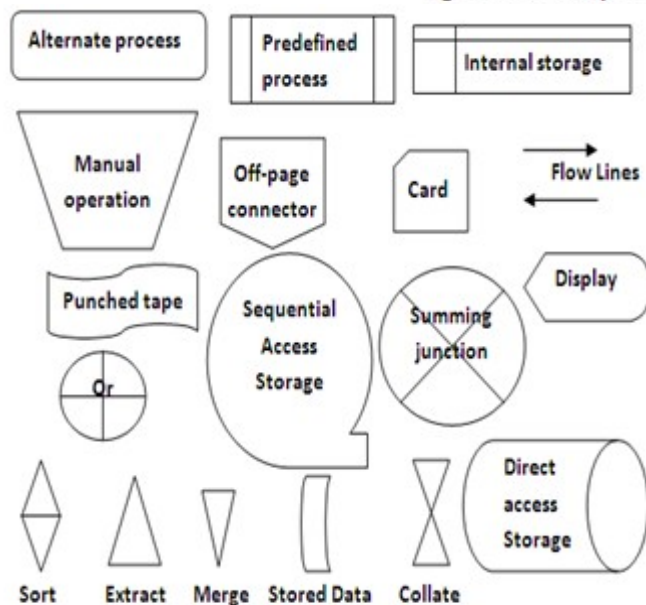


Fig. 15 some other symbols used in the flowchart

Pseudocode

In Pseudocode English-like words are used to represent the various logical steps. It is a prefix representation. Here solution of each step is described logically: The pseudocode is just the raw idea about the problem. By using this tip, one can try to solve the problem. The meaning of pseudocode is 'false code.' The syntax rule of any programming language cannot be applied to pseudocode.

Example (a): Assume a and b are two numbers and find the larger out of them. In such a case, comparison is made between them.

Few skilled programmers prefer to write pseudocode for drawing the flowchart. This is because using pseudocode is analogous to writing the final code in the programming language. Few programmers prefer to write the steps in algorithm.

Few programmers favor flowchart to represent the logical flow because with visualization things are clear for writing program statements.

For beginners a flowchart is a straightforward tool for writing a program.

Example (b): Example illustrates how the pseudocode is used to draw a flowchart.

- 1) Accept number
- 2) Calculate square of the number
- 3) Display number

All the steps of the program are written down in steps. Some programs follow pseudocode to draw flowcharts. Using pseudocode, final program can be written. Majority of programs have common tasks such as input, processing and output. These are fundamental tasks of any program. Using pseudocode a flowchart can be drawn as per the following steps. For the statement, that is, 'Accept number' the flowchart symbol is as per Fig.19.

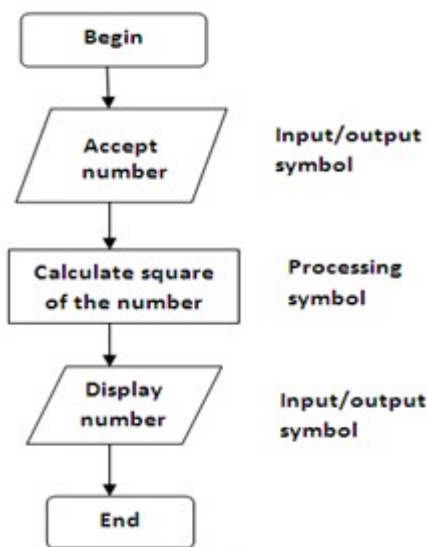


Fig. 19 pseudocode and flowchart of a program

Algorithm	Pseudocode
Input a and b.	Get numbers a & b
Is a>b.	Compare a & b
If yes a is larger than b.	if a is large max=a
If no b is larger than a.	if b is large max=b
Print the larger number.	larger number is max

1.8 Sequence and Selection

The steps in an algorithm can be divided into three categories.

1) *Sequence:* The steps described in the algorithm are performed successively one by one without skipping any step. The sequence of steps defined in the algorithm should be simple and easy to understand. Each instruction of such an algorithm is executed, because no selection procedure or conditional branching exists in a sequence algorithm. Fig.20 represents the sequence of steps for a telephonic conversation. Consider an example described in the topic 'Algorithm'.

- 1) Dial the phone number
- 2) Phone rings at the called party
- 3) Caller waits for the response

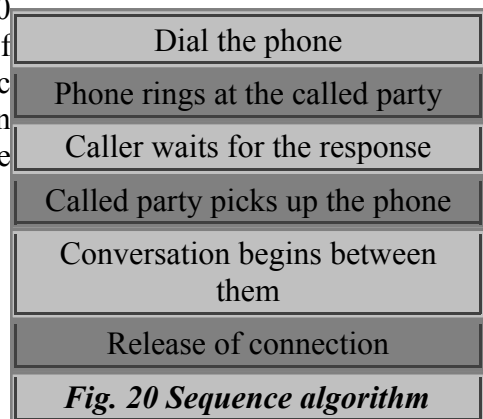
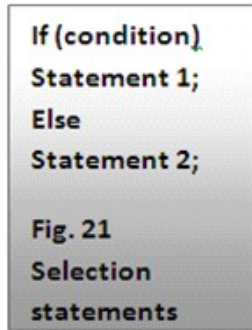


Fig. 20 Sequence algorithm

- 4) Called party picks up the phone
- 5) Conversation begins between them
- 6) Release of connection



2) *Selection*: We understood that the algorithms written in sequence fashion are not reliable. There must be a procedure to handle operation failure occurring during execution. The selection statements can be as shown in Fig.21.

Recursion versus iteration

Recursion	Iteration
Recursion is the term given to the mechanism of defining a set or procedure in terms of itself.	The block of statements executed repeatedly using loops.
A conditional statement is required in the body of the function for stopping the function execution.	The iteration control statements itself contain statement for stopping the iteration. At every execution, the condition is checked.
At some places, use of recursion generates extra overhead. Hence, better to skip when easy solution is available with iteration.	All problems can be solved with iteration.
Recursion is expensive in terms of speed and memory.	Iteration does not create any overhead. All the programming languages support iteration.

Overview of Compilers and Interpreters

A program is a set of instructions for performing a particular task. These instructions are just like English words. The computer processes the instructions as Is and Os. A program can be written in assembly language as well as high level language. This written program is called 'source program'. The source program is to be converted to the machine language, which is called 'object program'. Either an interpreter or compiler will do this activity.

a) **Interpreters**: An interpreter reads only one line of a source program at a time and converts it to the object codes. In case of errors, the same will be indicated instantly. The program written with an interpreter can easily be read and understood by other users. Therefore security is not provided. Anyone can modify the source code. Hence, it is easier than compilers. However, the disadvantage is that it consumes more time for converting a source program to an object program.

b) **Compilers**: Compiler reads the entire program and converts it to the object code. It provides errors not of one line but errors of the entire program. Only error-free programs are executed. It consumes little time for converting a source program to an object program. When the program length

for any application is large, compilers are preferred.

Structure of a C PROGRAM

Every C program contains a number of several building blocks known as functions. Each function performs a task independently. A function is a subroutine that may consist of one or more statements. A C program comprises different sections which are shown in Fig. 22.

a) **Include header file section:** A C program depends upon some header files for function definitions that are used in the program. Each header file, by default, has an extension **'.h'**. The file Should be included using **'# include'** directive as given below.

Example # include <stdio.h> or # include "stdio.h."

Include header file section

Global declaration section

/* comments */

main() function name

{

/* comments */

declaration part

executable part

}

user-defined functions

{

}

fig. 26. Structure of a C program

In this example <stdio.h> file is included, that is, all the definitions and prototypes of functions defined in this file are available in the current program. This file is also compiled with the original program.

b) **Global declaration:** This section declares some variables that are used in more than one function. These variables are known as global variables. This section must be declared outside of all the functions.

c) **Function main ():** Every program written in c language must contain main () function. Empty parenthesis after main is necessary. The function **main ()** is a starting point of every C program. The execution of the program always begins with the function main (). Except the main () function, other sections may not be necessary. The program execution starts from the opening brace ({) and ends with the closing brace(}). Between these two braces, the programmer should declare declaration and executable parts.

d) **Declaration part:** The declaration part declares the entire variables that are used in the executable part. Initializations of variables are also done in this section. Initialization means providing initial value to the variables.

e) **Executable part:** This part contains the statements following the declaration of the variables. This part contains a set of statements or a single statement. These statements are enclosed between braces.

f) **User-defined function:** The functions defined by the user are called user-defined functions. These functions are generally defined after the main () function. They can also be defined before **main()** function. This portion is not compulsory.

g) **Comments:** Comments are not necessary in the program. However, to understand the flow of a program, the programmer can include comments in the program. Comments are to be inserted by the programmer. These are useful for documentation. The clarity of the program can be followed if it is properly documented. Comments are nothing but some kind of statements which are to be placed between the delimiters **/* & */**. The compiler does not execute comments. Thus, we can say that comments are not the part of executable programs.

The user can frequently use any number of comments that can be placed anywhere in the

program. Please note that comment statements can be nested. The user should select the **OPTION MENU** of the editor and select the **COMPILER-SOURCE -NESTED COMMENTS ON /OFF**. The comments can be inserted with a single statement or in nested statements.

Example

```
/* This is single comment */  
/* This is an example of /* nested comments */*/  
/* This is an example of  
comments with multiple lines */ /* It can be nested */
```

Programming Rules

The programmer, while writing a program, should follow the following rules.

1) All statements should be written in lowercase letters. Uppercase letters are only used for symbolic constants.

2) Blank spaces may be inserted between the words. This improves the readability of the statements. However, this is not used while declaring a variable, keyword, constant and functions.

3) It is not necessary to fix the position of a statement in the program, that is, the programmer can write the statement anywhere between the two braces following the declaration part. The user can also write one or more statements in one line separating them with a semicolon (;). Hence, it is often called a free-form language.

The following statements are valid.

a=b+c;

d=b*c;

or

a=b+c;d=b*c;

4) The opening and closing braces should be balanced, that is, for example, if opening braces are four then closing braces should also be four.

Executing the Program:

The following steps are essential in executing a program in C.

a) *Creation of a program:* The program should be written in C editor. The file name does not necessarily include the extension '.c.' The default extension is '.c.' The user can also specify his/her own extension.

b) *Compilation and linking a program:* The source program statements should be translated into object program, which is suitable for execution by the computer. The translation is done after correcting each statement. If there is no error, compilation proceeds and the 'translated program is stored in another file with the same file name with extension' **.obj.** If at all errors are there the programmer should correct them. Linking is also an essential process. It puts all other program files and functions which are required by the program together. For example, if the programmer is using pow () function, then the object code of this function should be brought from **math.h** library of the system and linked to the main () program.

c) *Executing the program:* After the compilation,-the executable object code will be loaded in the computer's main memory and the program is executed. All the above steps can be performed using menu options of the editor.

LESSON 2 CONSTANTS & VARIABLES

Outline

Introduction - Character set - C Tokens - Keywords and Identifiers – Constants – Variables

INTRODUCTION

A programming language is designed to help certain kinds of *data process* consisting of numbers, characters and strings to provide useful output known as *information*. The task of processing of data is accomplished by executing a sequence of precise instructions called *program*.

CHARACTER SET

C characters are grouped into the following categories.

1. Letters
2. Digits
3. Special Characters
4. White Spaces

Note: The compiler ignores white spaces unless they are a part of a string constant.

Letters : Uppercase A....Z, Lowercase a.....z

Digits : All decimal digits 0.....9

White Spaces

- Blank Space
- Horizontal Tab
- Carriage Return
- New Line
- Form Feed

Delimiters

Language pattern of C uses special kinds of symbols, which are called as delimiters. They are given as Table

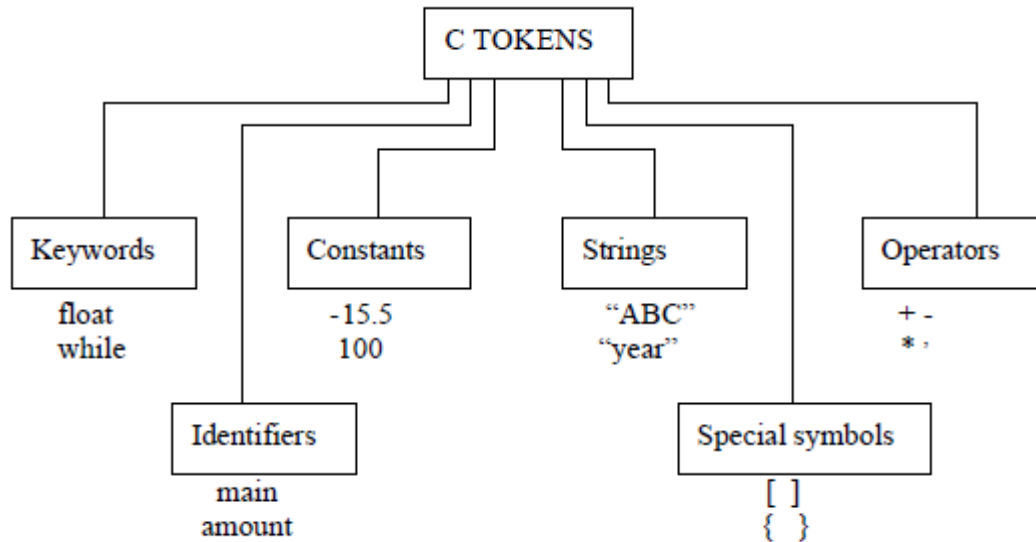
Special characters

, Comma	& Ampersand
. Period	^ Caret
; Semicolon	* Asterisk
: Colon	- Minus
? Question mark	+ Plus sign
' Apostrophe	< Less than
“ Quotation mark	> Greater than
! Exclamation	(Left parenthesis
Vertical Bar) Right parentheses
/ Slash	[Left bracket
\ Back slash] Right bracket
~ Tilde	{ Left brace
_ Underscore	} Right brace
\$ Dollar sign	# Number sign
% Percent sign	

: Colon	[] Square brackets
; Semi Colon	{ } Curly brace
() Parenthesis	# hash
, comma	

C TOKENS

In C programs, the smallest individual units are known as *tokens*.



KEYWORDS AND IDENTIFIERS

Every C word is classified as either a *keyword* or an *identifier*.

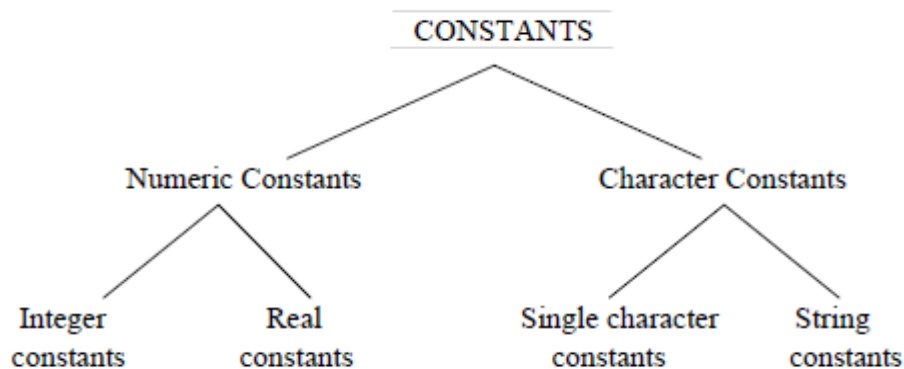
All keywords have fixed meanings and these meanings cannot be changed.

Eg: auto, break, char, void etc.,

Identifiers refer to the names of variables, functions and arrays. They are user-defined names and consist of a sequence of letters and digits, with a letter as a first character. Both uppercase and lowercase letters are permitted. The underscore character is also permitted in identifiers.

CONSTANTS

Constants in C refer to fixed values that do not change during the execution of a program.



Integer Constants

An integer constant refers to a sequence of digits, There are three types integers, namely, *decimal*, *octal*, and *hexa decimal*.

Decimal Constant

Eg: 123, -321 etc.,

Note: Embedded spaces, commas and non-digit characters are **not** permitted between digits.

Eg: 1) 15 750 2) \$1000

Octal Constant

An *octal* integer constant consists of any combination of digits from the set 0 through 7, with a leading 0.

Eg: 1) 037 2) 0435

Hexadecimal Constant

A sequence of digits preceded by 0x or 0X is considered as *hexadecimal* integer. They may also include alphabets A through F or a through f.

Eg: 1) 0X2 2) 0x9F 3) 0Xbcd

Program for representation of integer constants on a 16-bit computer.

```
/*Integer numbers on a 16-bit machine*/
main()
{
    printf("Integer values\n\n");
    printf("%d%d%d\n", 32767, 32767+1, 32767+10);
    printf("\n");
    printf("Long integer values\n\n");
    printf("%ld%ld%ld\n", 32767L, 32767L+1L, 32767L+10L);
}
```

OUTPUT

```
Integer values
32767 -32768 -32759
Long integer values
32767 32768 32777
```

Real Constants

Certain quantities that vary continuously, such as distances, heights etc., are represented by numbers containing functional parts like 17.548. Such numbers are called *real* (or *floating point*) constants.

Eg: 0.0083, -0.75 etc.,

A real number may also be expressed in *exponential or scientific notation*.

Eg: 215.65 may be written as 2.1565e2

Single Character Constants

A single character constants contains a single character enclosed within a pair of *single* quote marks.

Eg: '5'

'X'

','

'

String Constants

A string constant is a sequence of characters enclosed in *double* quotes. The characters may be letters, numbers, special characters and blank space.

Eg: "Hello!"

"1987" "?....!"

Backslash Character Constants

C supports special backslash character constants that are used in output functions. These character combinations are known as *escape sequences*.

Constant	Meaning
'\a'	audible alert
'\b'	backspace
'\f'	form feed
'\n'	new line
'\0'	null
'\v'	vertical tab
'\t'	horizontal tab
'\r'	carriage return

VARIABLES

Definition:

A *variable* is a data name that may be used to store a data value. A variable may take different values at different times of execution and may be chosen by the programmer in a meaningful way. It may consist of letters, digits and underscore character.

Eg: 1) Average

2) Height

Rules for defining variables

- ❖ They must begin with a letter. Some systems permit underscore as the first character.
- ❖ ANSI standard recognizes a length of 31 characters. However, the length should not be normally more than eight characters.
- ❖ Uppercase and lowercase are significant.
- ❖ The variable name should not be a keyword.
- ❖ White space is not allowed.

LESSON 3 DATA TYPES

Outline

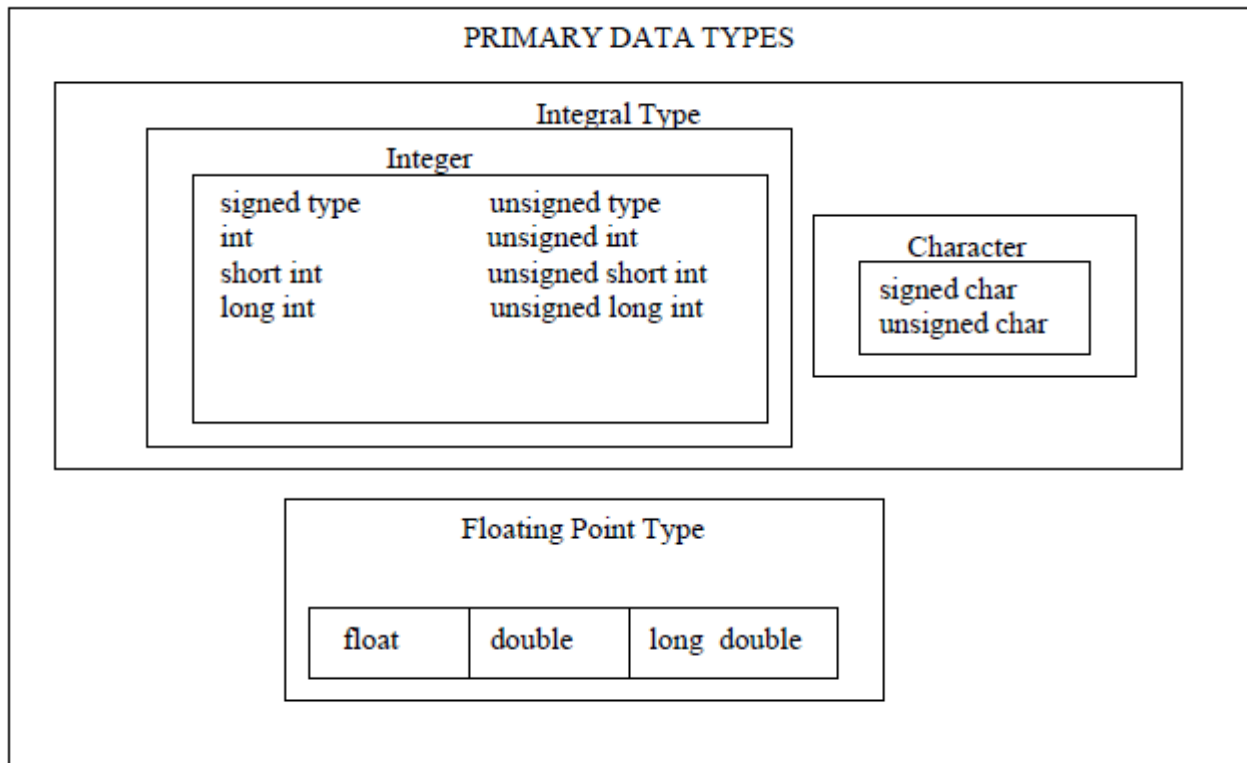
Introduction - Primary data types - Declaration of variables - Assigning values to variables -
Reading data from keyword - Defining symbolic constants

INTRODUCTION

ANSI C supports four classes of data types.

1. Primary or Fundamental data types.
2. User-defined data types.
3. Derived data types.
4. Empty data set.

PRIMARY DATA TYPES



Integer Types

Type	Size (bits)	Range
int or signed int	16	-32,768 to 32767
unsigned int	16	0 to 65535
short int	8	-128 to 127
unsigned short int	8	0 to 255
long int	32	-2,147,483,648 to 2,147,483,647
unsigned long int	32	0 to 4,294,967,295

Floating Point Types

Type		Size(bits)	Range
float		32	3.4E-38 to 3.4E+38
double	64		1.7E-308 to 1.7E+308
long double		80	3.4E-4932to 1.1E+4932

Character Types

Type	Size (bits)	Range
char	8	-128 to 127
unsigned char	8	0 to 255

DECLARATION OF VARIABLES

The syntax is

Data-type v1,v2.....vn;

Eg: 1.**int** count;
2.**double** ratio, total;

User-defined type declaration

C allows user to define an identifier that would represent an existing **int** data type.

The general form is **typedef** type identifier;

Eg: 1) **typedef int** units;
2) **typedef float** marks;

Another user defined data types is enumerated data type which can be used to declare variables that can have one of the values enclosed within the braces.

enum identifier {value1,value2,.....valuen};

Declaration of storage class

Variables in C can have not only data type but also storage class that provides information about their locality and visibility. Here the variable **m** is called the global variable. It can be used in all the functions in the program.

The variables **bal**, **sum** and **i** are called local variables. Local variables are visible and meaningful only inside the function which they are declared. There are four storage class specifiers, namely, auto, static, register and extern.

ASSIGNING VALUES TO VARIABLES

The syntax is

Variable_name=constant

Eg: 1) **int a=20**;
2) **bal=75.84**;
3) **yes='x'**;

/*Example of storage class*/

```
int m;
main()
{
int i;
float bal;
.....
.....
function1();
}
function1()
{
int i;
float sum;
.....
.....
}
```

in

C permits multiple assignments in one line.

Example:

```
initial_value=0;final_value=100;
```

Declaring a variable as constant

Eg: 1) **const int** class_size=40;

This tells the compiler that the value of the int variable class_size must not be modified by the program.

Declaring a variable as volatile

By declaring a variable as volatile, its value may be changed at any time by some external source.

Eg: 1) **volatile int** date;

READING DATA FROM KEYWORD

Another way of giving values to variables is to input data through keyboard using the **scanf** function. The general format of **scanf** is as follows.

```
scanf("control string",&variable1,&variable2,...);
```

The ampersand symbol **&** before each variable name is an operator that specifies the variable name's *address*.

Eg: 1) **scanf("%d",&number);**

DEFINING SYMBOLIC CONSTANTS

We often use certain unique constants in a program. These constants may appear repeatedly in a number of places in the program. One example of such a constant is 3.142, representing the value of the mathematical constant "**pi**". We face two problems in the subsequent use of such programs.

1. Problem in modification of the programs.
2. Problem in understanding the program.

A constant is defined as follows:

```
#define symbolic-name value of constant
```

Eg: 1) **#define pi** 3.1415

2) **#define** pass_mark 50

The following rules apply to a **#define** statement which define a symbolic constant

- ❖ Symbolic names have the same form as variable names.
- ❖ No blank space between the sign **#** and the word **define** is permitted
- ❖ **#** must be the first character in the line.
- ❖ A blank space is required between **#define** and *symbolic name* and between the *symbolic name* and the *constant*.
- ❖ **#define** statements must not end with the semicolon.
- ❖ After definition, the symbolic name should not be assigned any other value within the program by using an assignment statement.
- ❖ Symbolic names are NOT declared for data types. Their data types depend on the type of constant.
- ❖ **#define** statements may appear *anywhere* in the program but before it is referenced in the program.

LESSON 4 OPERATORS

Outline

Operators of C - Arithmetic operators - Relational operators - Logical operators -
Assignment operators - Increment and decrement operators - Conditional operator -
Bitwise operators - Special operators

OPERATORS OF C

C supports a rich set of operators. Operators are used in programs to manipulate data and variables. They usually form a part of the mathematical or logical expressions.

C operators are classified into a number of categories. They include:

1. *Arithmetic operators*
2. *Relational operators*
3. *Logical operators*
4. *Assignment operators*
5. *Increment and Decrement operators*
6. *Conditional operators*
7. *Bitwise operators*
8. *Special operators*

ARITHMETIC OPERATORS

The operators are

- + (Addition)
- (Subtraction)
- * (Multiplication)
- / (Division)
- % (Modulo division)

Eg: 1) $a-b$ 2) $a+b$ 3) $a*b$ 4) $p\%q$

The modulo division produces the remainder of an integer division. The modulo division operator cannot be used on floating point data.

Note: C does not have any operator for *exponentiation*.

Integer Arithmetic

When both the operands in a single arithmetic expression are integers, the expression is called an *integer expression*, and the operation is called *integer arithmetic*. During modulo division the sign of the result is always the sign of the first operand. That is

$$-14 \% 3 = -2$$

$$-14 \% -3 = -2$$

$$14 \% -3 = 2$$

Real Arithmetic

An arithmetic operation involving only real operands is called *real arithmetic*. If **x and y** are floats then we will have:

$$1) x = 6.0 / 7.0 = 0.857143$$

$$2) y = 1.0 / 3.0 = 0.333333$$

The operator % cannot be used with real operands.

Mixed-mode Arithmetic

When one of the operands is real and the other is integer, the expression is called a *mixed-mode arithmetic* expression and its result is always a real number.

Eg: 1) $15 / 10.0 = 1.5$

RELATIONAL OPERATORS

Comparisons can be done with the help of *relational operators*. The expression containing a relational operator is termed as a *relational expression*. The value of a relational expression is either *one* or *zero*.

- 1) < (is less than)
- 2) <= (is less than or equal to)
- 3) > (is greater than)
- 4) >= (is greater than or equal to)
- 5) == (is equal to)
- 6) != (is not equal to)

LOGICAL OPERATORS

C has the following three *logical operators*.

&& (logical **AND**)

|| (logical **OR**)

! (logical **NOT**)

- Eg:
- 1) `if(age>55 && sal<1000)`
 - 2) `if(number<0 || number>100)`

ASSIGNMENT OPERATORS

The usual assignment operator is '='. In addition, C has a set of 'shorthand' assignment operators of the form, `v op = exp;`

Eg: `1.x += y+1;`

This is same as the statement

`x=x+(y+1);`

Shorthand Assignment Operators

Statement with shorthand operator	Statement with simple assignment operator
<code>a += 1</code>	<code>a = a + 1</code>
<code>a -= 1</code>	<code>a = a - 1</code>
<code>a *= n + 1</code>	<code>a = a * (n+1)</code>
<code>a /= n + 1</code>	<code>a = a / (n+1)</code>
<code>a %= b</code>	<code>a = a % b</code>

INCREMENT AND DECREMENT OPERATORS

C has two very useful operators that are not generally found in other languages. These are the *increment* and *decrement* operator:

++ and --

The operator ++ adds 1 to the operands while – subtracts 1. It takes the following form:

++m; or m++

--m; or m--

CONDITIONAL OPERATOR

A ternary operator pair “?:” is available in C to construct conditional expression of the form:

exp1 ? exp2 : exp3;

Here *exp1* is evaluated first. If it is true then the expression *exp2* is evaluated and becomes the value of the expression. If *exp1* is false then *exp3* is evaluated and its value becomes the value of the expression.

Eg: 1) if(a>b)

x = a;

else

x = b;

BITWISE OPERATORS

Operator	Meaning
&	Bitwise AND
	Bitwise OR
^	Bitwise XOR
<<	Shift left
>>	Shift right
~	One's complement

SPECIAL OPERATORS

C supports some special operators such as

- Comma operator ➤ Size of operator ➤ Pointer operators(& and *) and
- Member selection operators(. And →)

The Comma Operator

The comma operator can be used to link the related expressions together. A commalinked list of expressions are evaluated *left to right* and the value of *right-most* expression is the value of the combined expression.

Eg: value = (x = 10, y = 5, x + y);

This statement first assigns the value 10 to **x**, then assigns 5 to **y**, and finally assigns 15 (i.e, 10+5) to **value**.

The Size of Operator

The size of is a compiler time operator and, when used with an operand, it returns the number of bytes the operand occupies.

Eg: 1) m = **sizeof**(sum);
2) n = **sizeof**(long int)
3) k = **sizeof**(235L)

LESSON 5 EXPRESSIONS

Outline

Expressions - Arithmetic expressions - Precedence of arithmetic operators -
Type conversion in expressions - Mathematical functions - Managing input and output operations

EXPRESSIONS

The combination of operators and operands is said to be an expression.

ARITHMETIC EXPRESSIONS

An arithmetic expression is a combination of variables, constants, and operators arranged as per the syntax of the language.

Eg 1) $a = x + y$;

EVALUATION OF EXPRESSIONS

Expressions are evaluated using an assignment statement of the form $\text{variable} = \text{expression}$;

Eg: 1) $x = a * b - c$; 2) $y = b / c * a$;

PRECEDENCE OF ARITHMETIC OPERATORS

An arithmetic expression without parenthesis will be evaluated from left to right using the rule of precedence of operators. There are two distinct priority levels of arithmetic operators in C.

High priority: $*$ / $\%$ **Low priority:** $+$ -

Program

```
/*Evaluation of expressions*/
main()
{
float a, b, c, y, x, z;
a = 9;
b = 12;
c = 3;
x = a - b / 3 + c * 2 - 1;
y = a - b / (3 + c) * (2 - 1);
z = a - (b / (3 + c) * 2) - 1;
printf("x = %f\n",x);
printf("y = %f\n",y);
printf("z = %f\n",z);
}
```

OUTPUT

```
x = 10.000000
y = 7.000000
z = 4.000000
```

SOME COMPUTATIONAL PROBLEMS

When expressions include real values, then it is important to take necessary precautions to guard against certain computational errors. Some of these errors could be approximate values for real numbers, division by zero and overflow or underflow errors.

Program

```
/*Program showing round-off errors*/
/*Sum of n terms of 1/n*/
main()
{
float sum, n, term;
int count = 1;
sum = 0;
printf("Enter value for n \n");
scanf("%f",&n);
term = 1.0 / n;
while(count <= n)
{
sum = sum + term;
count ++;
}
printf("sum = %f\n",sum);
}
```

OUTPUT

```
Enter value of n
99
Sum = 1.000001
Enter value of n
143
Sum = 0.999999
```

We know that the sum of n terms of $1/n$ is 1. However due to errors in floating-point representation; the result is not always 1.

TYPE CONVERSION IN EXPRESSIONS

Automatic Type Conversion

C permits the mixing of constants and variables of different types in an expression. If the operands are of different types, the lower type is automatically converted to higher type before the operation proceeds. The result is of the higher type. Given below are the sequence of rules that are applied while evaluating expressions.

All **short** and **char** are automatically converted to **int**; then

- ❖ If one of the operands is long double, the other will be converted to long double and the result will be in long double.
- ❖ Else, If one of the operands is double, the other will be converted to double and the result will be in double.
- ❖ Else, If one of the operands is float, the other will be converted to float and the result will be in float.

- ❖ Else, If one of the operands is unsigned long int, the other will be converted to unsigned long int and the result will be in unsigned long int.
- ❖ Else, if one of the operands is long int and other is unsigned int, then:
 - (a) if unsigned int can be converted to long int, the unsigned int operand will be converted as such and the result will be long int.
 - (b) else, both the operands will be converted to unsigned long int and the result will be unsigned long int.
- ❖ Else, If one of the operands is long int, the other will be converted to long int and the result will be in long int.
- ❖ Else, If one of the operands is unsigned int, the other will be converted to unsigned int and the result will be in unsigned int.

Casting a Value

C performs type conversion automatically. However, there are instances when we want to force a type conversion in a way that is different from the automatic conversion.

Eg: 1) ratio = female_number / male_number

Since female_number and male_number are declared as integers the ratio would represent a wrong figure. Hence it should be converted to float.

ratio = (float) female_number / male_number

The general form of a cast is: **(type-name)**expression

MATHEMATICAL FUNCTIONS

Mathematical functions such as sqrt, cos, log etc., are the most frequently used ones. To use the mathematical functions in a C program, we should include the line **#include<math.h>** in the beginning of the program.

Function	Meaning
Trigonometric acos(x) asin(x) atan(x) atan2(x,y) cos(x) sin(x) tan(x)	Arc cosine of x Arc sine of x Arc tangent of x Arc tangent of x/y cosine of x sine of x tangent of x
Hyperbolic cosh(x) sinh(x) tanh(x)	Hyperbolic cosine of x Hyperbolic sine of x Hyperbolic tangent of x
Other functions ceil(x) exp(x) fabs(x) floor(x)	x rounded up to the nearest integer e to the power x absolute value of x x rounded down to the nearest integer

fmod(x,y)	remainder of x/y
log(x)	natural log of x, x>0
log10(x)	base 10 log of x.x>0
pow(x,y)	x to the power y
sqrt(x)	square root of x,x>=0

MANAGING INPUT AND OUTPUT OPERATIONS

All input/output operations are carried out through functions called as **printf** and **scanf**. There exist several functions that have become standard for input and output operations in C. These functions are collectively known as *standard i/o library*. Each program that uses standard I/O function must contain the statement

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

The file name `stdio.h` is an abbreviation of *standard input-output header file*.

READING A CHARACTER

Reading a single character can be done by using the function **getchar**. The **getchar** takes the following form:

```
variable_name = getchar();
```

Eg: char name;
 name=getchar();

Program

```
/*Reading a character from terminal*/
#include<stdio.h>
main()
{
    char ans;
    printf("Would you like to know my name? \n");
    printf("Type Y for yes and N for no");
    ans=getchar();
    if(ans == 'Y' || ans == 'y')
        printf("\n\n My name is India \n");
    else
        printf("\n\n You are good for nothing \n");
}
```

OUTPUT

```
Would you like to know my name?
Type Y for yes and N for no:Y
My name is India
Would you like to know my name?
Type Y for yes and N for no:n
You are good for nothing
```

WRITING A CHARACTER

Like **getchar**, there is an analogous function **putchar** for writing characters one at a time to the terminal. It takes the form as shown below:

putchar (variable_name);

Eg: 1) answer='y';

putchar(answer);

will display the character y on the screen.

The statement

putchar('\n');

would cause the cursor on the screen to move to the beginning of the next line.

Program

/*A program to read a character from keyboard and then prints it in reverse case*/

/*This program uses three new functions: **islower**, **toupper**, and **tolower**.

#include<stdio.h>

#include<ctype.h>

main()

{

char alphabet;

printf("Enter an alphabet");

putchar('\n');

alphabet = getchar();

if(islower(alphabet))

putchar(toupper(alphabet));

else

putchar(tolower(alphabet));

}

OUTPUT

Enter An alphabet

a

A

Enter An alphabet

Q

q

Enter An alphabet

z

Z

FORMATTED INPUT

Formatted input refers to an input data that has been arranged in a particular format. The formatted data can be read with the help of **scanf** function. The general form of **scanf** is

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

The control string specifies the field format in which the data is to be entered and the arguments arg1,arg2...argn specifies the address of locations where the data are stored. Control strings and arguments are separated by commas.

Control string contains field specification which direct the interpretation of input data.

It may include

- ❖ Field(or format)specifications, consisting of conversion character %, a data type character, and an optional number, specifying the field width.
- ❖ Blanks, tabs, or newlines.

Inputting integer numbers

The field specification for reading an integer number is

%wd

Eg: **scanf("%2d %5d",&num1, &num2);**

An input field may be skipped by specifying * in the place of field width. For eg ,
scanf("%d %*d %d",&a, &b);

Program

```
/*Reading integer numbers*/
main()
{
    int a, b, c, x, y, z;
    int p, q, r;
    printf("Enter three integer numbers \n");
    scanf("%d %*d %d",&a, &b, &c);
    printf("%d %d %d \n \n",a, b, c);
    printf("Enter two 4-digit numbers \n");
    scanf("%2d %4d",&x, &y);
    printf("%d %d \n \n",x, y);
    printf("Enter two integer numbers \n");
    scanf("%d %d",&a, &x);
    printf("%d %d \n \n",a, x);
    printf("Enter a nine digit numbers \n");
    scanf("%3d %4d %3d",&p, &q, &r);
    printf("%d %d %d \n \n",p, q, r);
    printf("Enter two three digit numbers \n");
```

```
scanf("%d %d",&x, &y);
printf("%d %d \n \n",x, y);
}
```

OUTPUT

```
Enter three integer numbers
1 2 3
1 3 -3577
Enter two 4-digit numbers
6789 4321
67 89
Enter two integer numbers
44 66
4321 44
Enter a nine digit numbers
123456789
66 1234 567
Enter two three digit numbers
123 456
89 123
```

Inputting Real Numbers

Unlike integer numbers, the field width of real numbers is not to be specified and therefore **scanf** reads real numbers using simple specification **%f** for both the notations, namely, decimal point notation and exponential notation.

Eg: **scanf("%f %f %f", &x, &y, &z);**

If the number to be read is of **double** type, then the specification should be **%lf** instead of simple **%f**.

Inputting Character Strings

Following are the specifications for reading character strings:

%ws or **%wc**

Some versions of **scanf** support the following conversion specifications for strings:

%[characters] and **%[^characters]**

The specification **%[characters]** means that only the characters specified within the brackets are permissible in the input string. If the input string contains any other character, the string will be terminated at the first encounter of such a character. The specification **%[^characters]** does exactly the reverse. That is, the characters specified after the circumflex (^) are not permitted in the input string.

Reading Mixed Data Types

It is possible to use one **scanf** statement to input a data line containing mixed mode data. In such cases, it should be ensured that the input data items match the control specifications in *order* and *type*.

Eg: **scanf("%d %c %f %s",&count, &code, &ratio, &name);**

Scanf Format Codes

Code	Meaning
%c	Read a single character
%d	Read a decimal integer
%e	Read a floating point value
%f	Read a floating point value
%g	Read a floating point value
%h	Read a short integer
%i	Read a decimal, hexadecimal, or octal integer
%o	Read an octal integer
%s	Read a string
%u	Read an unsigned decimal integer
%x	Read a hexa decimal integer
%[..]	Read a string of word(s)

Points To Remember while using scanf

- ❖ All function arguments, except the control string, must be pointers to variables.
- ❖ Format specifications contained in the control string should match the arguments in order.
- ❖ Input data items must be separated by spaces and must match the variables receiving the input in the same order.

- ❖ The reading will be terminated, when scanf encounters an ‘invalid mismatch’ of data or a character that is not valid for the value being read.
- ❖ When searching for a value, scanf ignores line boundaries and simply looks for the next appropriate character.
- ❖ Any unread data items in a line will be considered as a part of the data input line to the next scanf call.
- ❖ When the field width specifier *w* is used, it should be large enough to contain the input data size.

FORMATTED OUTPUT

The **printf** statement provides certain features that can be effectively exploited to control the alignment and spacing of print-outs on the terminals. The general form of **printf** statement is

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

Control string consists of three types of items:

1. Characters that will be printed on the screen as they appear.
2. Format specifications that define the output format for display of each item.
3. *Escape sequence* characters such as \n, \t and \b

Output of Integer Numbers

The format specification for printing an integer number is

%wd

Output of Real Numbers

The output of real numbers may be displayed in decimal notation using the following format specification:

%w.p f

The integer *w* indicates the minimum number of positions that are to be used for the display of the value and the integer *p* indicates the number of digits to be displayed after the decimal point.

We can also display real numbers in exponential notation by using the specification

%w.p e

Printing of Single Character

A single character can be displayed in a desired position using the format. The character will be displayed *right-justified* in the field of *w* columns. We can make the display *left-justified* by placing a minus sign before the integer *w*. The default value for *w* is 1.

Printing of Strings

The format specification for outputting strings is of the form

%wc

%w.ps

Mixed Data Output

It is permitted to mix data types in one printf statements.

Eg: **printf**("%d %f %s %c", a, b, c, d);

LESSON 6 CONTROL STATEMENTS

Outline

Control Statements - Conditional Statements - The Switch Statement - Unconditional Statements- Decision Making and Looping

CONTROL STATEMENTS

C language supports the following statements known as *control* or *decision making* statements.

1. **if** statement
2. **switch** statement
3. conditional operator statement
4. **goto** statement

CONDITIONAL STATEMENTS

IF STATEMENT

The **if** statement is used to control the flow of execution of statements and is of the form

If(test expression)

It allows the computer to evaluate the expression first and then, depending on whether the value of the expression is 'true' or 'false', it transfers the control to a particular statement.

Eg: **if**(bank balance is zero)
 Borrow money

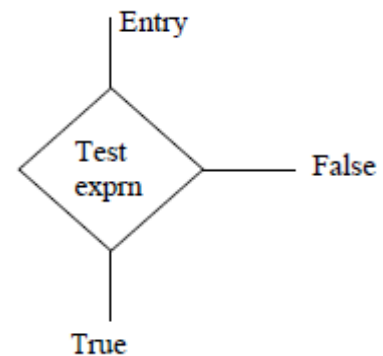
The **if** statement may be implemented in different forms depending on the complexity of conditions to be tested.

1. Simple **if** statement
2. **if.....else** statement
3. Nested **if.....else** statement
4. **elseif ladder**

SIMPLE IF STATEMENT

The general form of a simple **if** statement is The 'statement-block' may be a single statement or a group of statement. If the *test expression* is true, the *statement-block* will be executed; otherwise the statement-block will be skipped and the execution will jump to the *statement-x*.

```
If(test exprn)
{
    statement-block;
}
statement-x;
```



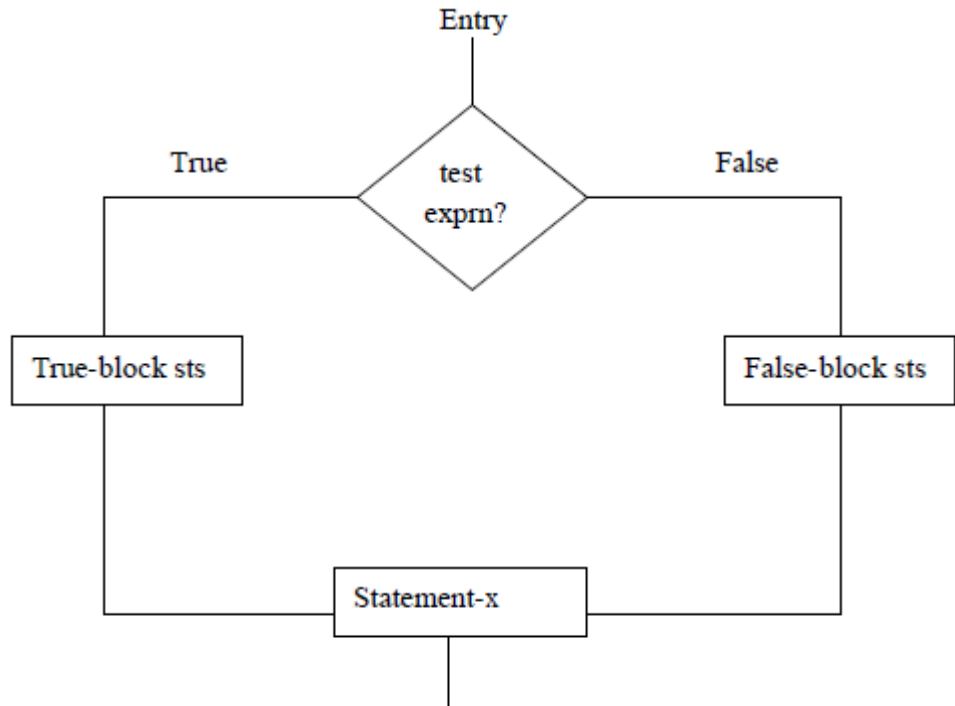
E.g.
if(category == SPORTS)
{
 marks = **marks** +
 bonus_marks;
}
printf("%"f ",marks);

THE IF...ELSE STATEMENT

The **if....else** statement is an extension of simple **if** statement. The general form is

```
If(test expression)
{
True-block statement(s)
}
else
{
False-block statement(s)
}
statement-x
```

If the *test expression* is true, then the true block statements are executed; otherwise the false block statement will be executed.

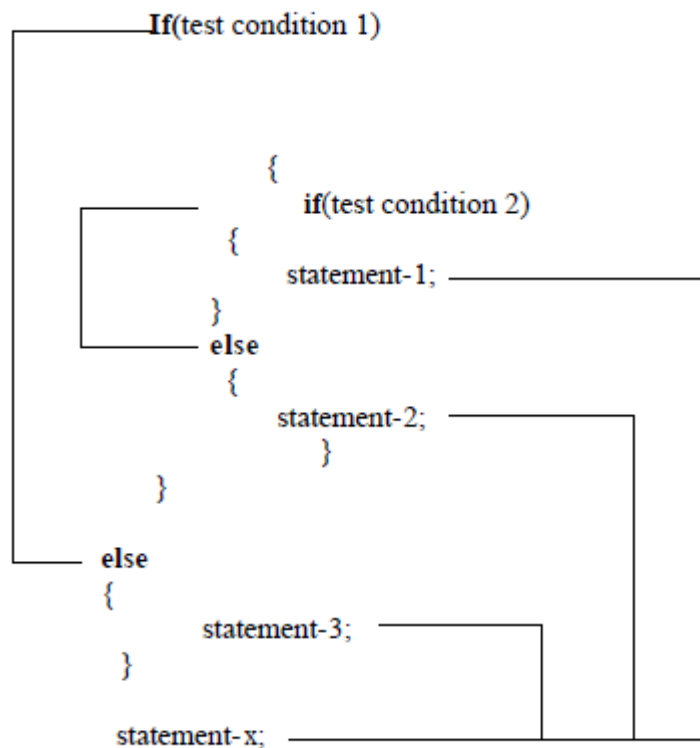


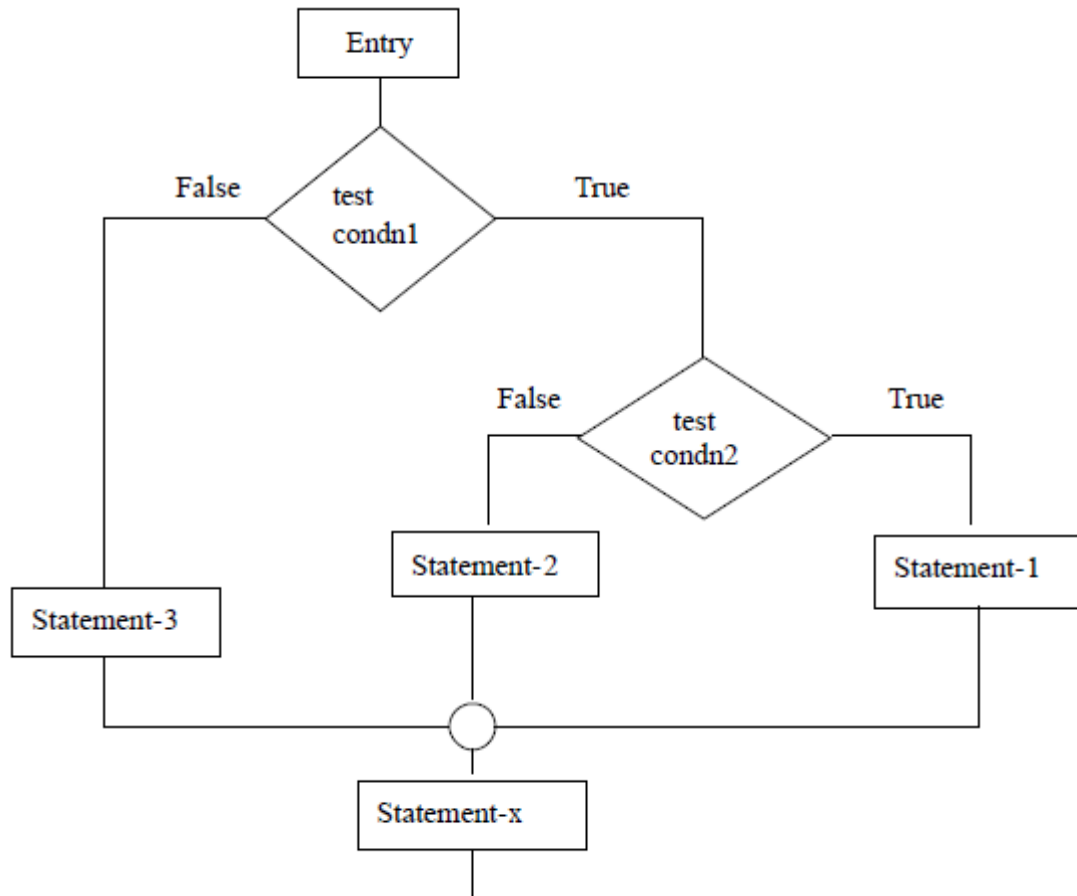
Eg:

```
.....
.....
if(code ==1)
boy = boy + 1;
if(code == 2)
girl =girl + 1;
.....
.....
```

NESTING OF IF.....ELSE STATEMENTS

When a series of decisions are involved, we may have to use more than one **if....else** statements, in *nested* form as follows.





Program

```

/*Selecting the largest of three values*/
main()
{
    float A, B, C;
    printf("Enter three values \n");
    scanf("%f %f %f",&A, &B, &C);
    printf("\nLargest value is:");
    if(A > B)
    { if(A > C)
      printf("%f\n",A);
    else
      printf("%f\n",C);
    }
    else

```

```

{
    if(C > B)
    printf("%f\n",C);
    else
    printf("%f\n",B);
}
}

```

OUTPUT

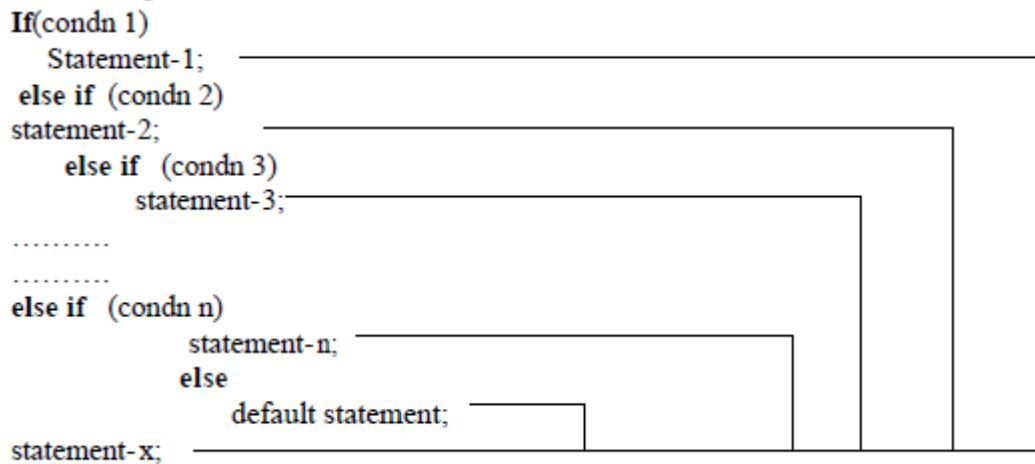
```

Enter three values:
5 8 24
Largest value is 24

```

THE ELSEIF LADDER

The general form is



Program

```
/*Use of else if ladder*/
main()
{
    int units, cno;
    float charges;
    printf("Enter customer no. and units consumed \n");
    scanf("%d %d",&cno, &units );
    if(units <= 200)
        charges = 0.5 * units;
    else if(units <= 400)
        charges = 100+ 0.65 * (units - 200)
    else if (units <= 600)
        charges = 230 + 0.8 * (units - 400)
    else
        charges = 390 + (units - 600);
    printf("\n \ncustomer no =%d,charges =%.2f \n",cno,charges);
}
```

OUTPUT

```
Enter customer no. and units consumed 101 150
Customer no=101 charges = 75.00
```

THE SWITCH STATEMENT

Switch statement is used for complex programs when the number of alternatives increases. The switch statement tests the value of the given variable against the list of **case** values and when a match is found, a block of statements associated with that case is executed. The general form of switch statement is

```
switch(expression)
{
case value-1:
block-1
break;
case value-2:
block-2
break;
.....
.....
default:
default-block
break;
}
statement-x;
```

THE ?: OPERATOR

The C language has an unusual operator, useful for making two-way decisions. This operator is a combination of **?** and **:** and takes three operands. It is of the form

exp1?exp2:exp 3

Here *exp1* is evaluated first. If it is true then the expression *exp2* is evaluated and becomes the value of the expression. If *exp1* is false then *exp3* is evaluated and its value becomes the value of the expression.

Eg:

```
if(x < 0)
flag = 0;
else
flag = 1;
can be written as
flag = (x < 0)? 0 : 1;
```

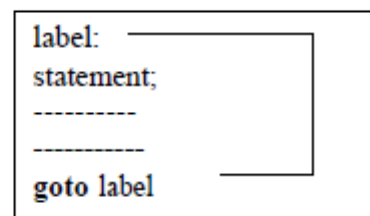
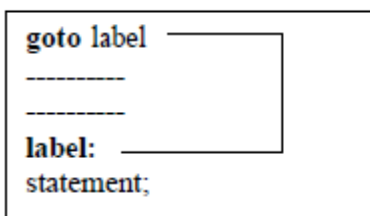
Eg: switch statement

```
.....
.....
index = marks / 10;
switch(index)
{
case 10:
case 9:
case 8:
grade = "Honours";
break;
case 7:
case 6:
grade = "first division";
break;
case 5:
grade = "second division";
break;
case 4:
grade = "third division";
break;
default:
grade = "first division";
break;
}
printf("%s \n",grade);
.....
```

UNCONDITIONAL STATEMENTS

THE GOTO STATEMENT

C supports the goto statement to branch unconditionally from one point of the program to another. The goto requires a *label* in order to identify the place where the branch is to be made. A **label** is any valid variable name and must be followed by a colon. The general form is

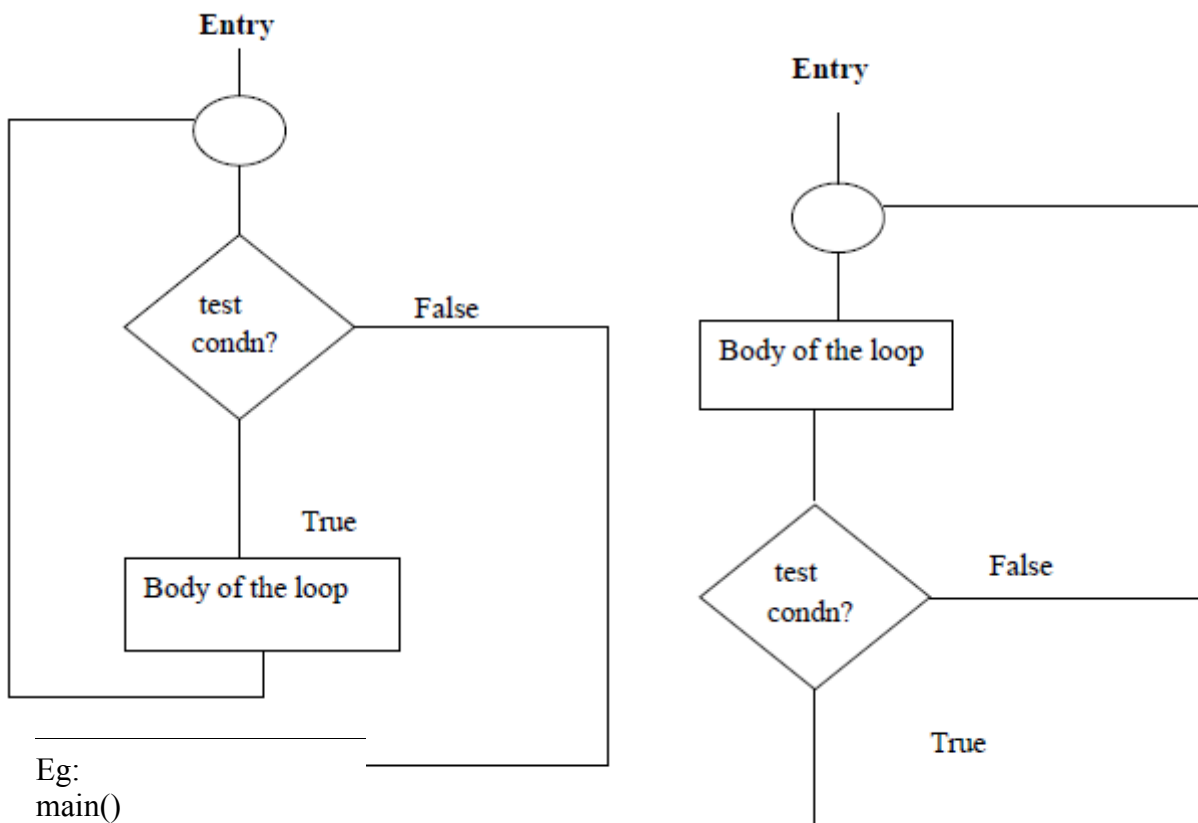


Note: A label can be anywhere in the program, either before or after the **goto** label; statement.

DECISION MAKING AND LOOPING

It is possible to execute a segment of a program repeatedly by introducing a counter and later testing it using the **if** statement. While this method is quite satisfactory for all practical purposes, we need to initialize and increment a counter and test its value at an appropriate place in the program for the completion of the loop.

In looping, a sequence of statements are executed until some conditions for termination of the loop are satisfied. A *program loop* therefore consists of two segments, one known as the *body of the loop* and the other known as the *control statements*. Depending on the position of the control statements in the loop, a control structure may be classified either as an *entry-controlled loop* or as the *exit-controlled loop*.



Eg:
 main()
 {
 double x, y;
read:
 scanf("%f",&x);
 if(x < 0) **goto read;**
 y = sqrt(x);
 printf("%f %f\n",x, y);
goto read;
 }
 }

The C language provides for three loop constructs for performing loop operations. They are

- The **while** statement
- The **do** statement
- The **for** statement

THE WHILE STATEMENT

The basic format of the **while** statement is

```
while(test condition)
{
    body of the loop
}
```

The while is an entry-controlled loop statement. The test-condition is evaluated and if the condition is true, then the body of the loop is executed. After execution of the body, the test-condition is once again evaluated and if it is true, the body is executed once again. This process of repeated execution of the body continues until the test-condition finally becomes *false* and the control is transferred out of the loop.

Eg:

```
sum = 0;
n = 1;
while(n <= 10)
{
    sum = sum + n * n;
    n = n + 1;
}
printf("sum = %d\n", sum);
```

THE DO STATEMENT

In while loop the body of the loop may not be executed at all if the condition is not satisfied at the very first attempt. Such situations can be handled with the help of the **do** statement

```
do
{
    body of the loop
}
while(test condition);
```

Since the *test-condition* is evaluated at the bottom of the loop, the **do.....while** construct provides an *exit-controlled* loop and therefore the body of the loop is *always executed at least once*.

Eg:

```
-----
do
{
    printf("Input a number\n");
    number = getnum();
}
while(number > 0);
```

THE FOR STATEMENT

Simple 'for' Loops

The **for** loop is another *entry-controlled* loop that provides a more concise loop control structure. The general form of the **for** loop is

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

The execution of the **for** statement is as follows:

- ❖ Initialization of the control variables is done first.
- ❖ The value of the control variable is tested using the *test-condition*. If the condition is *true*, the body of the loop is executed; otherwise the loop is terminated and the execution continues with the statement that immediately follows the loop.

- ❖ When the body of the loop is executed, the control is transferred back to the **for** statement after evaluating the last statement in the loop. Now, the control variable is either incremented or decremented as per the condition.

Eg 1)

```
for(x = 0; x <= 9; x = x + 1)
{
    printf("%d",x);
}
printf("\n");
```

The multiple arguments in the increment section are possible and separated by *commas*.

Eg 2)

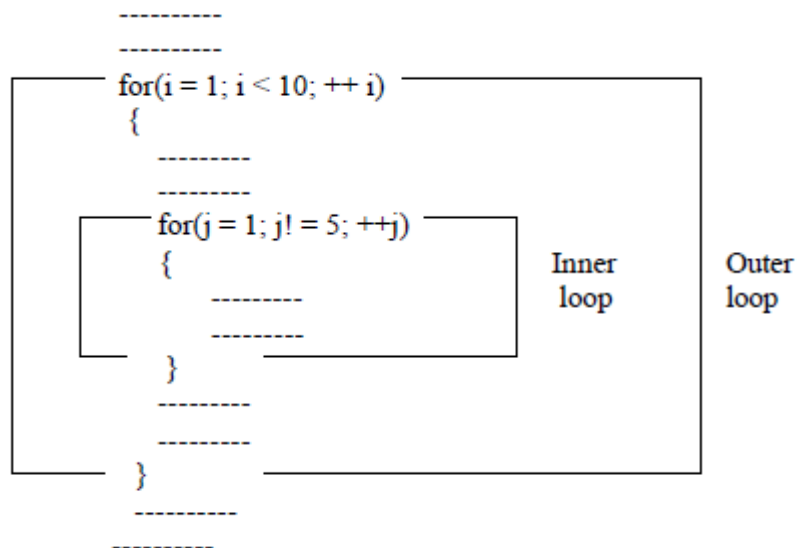
```
sum = 0;
for(i = 1; i < 20 && sum < 100; ++i)
{
    sum = sum + i;
    printf("%d %d \n",sum);
}
```

Eg: Nesting of For Loops

```
-----
-----
for(row = 1; row <= ROWMAX; ++row)
{
    for(column = 1; column <= COLMAX; ++column)
    {
        y = row * column;
        printf("%4d", y);
    }
    printf("\n");
}
-----
-----
```

Nesting of For Loops

C allows one **for** statement within another **for** statement.



JUMPS IN LOOPS

C permits a jump from one statement to another within a loop as well as the jump out of a loop.

Jumping out of a Loop

An early exit from a loop can be accomplished by using the **break** statement or the **goto** statement. When the **break** statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop. When the loops are nested, the **break** would only exit from the loop containing it. That is, the **break** will exit only a single loop.

Skipping a part of a Loop

Like the **break** statement, C supports another similar statement called the **continue** statement. However, unlike the **break** which causes the loop to be terminated, the **continue**, as the name implies, causes the loop to be *continued* with the *next iteration* after *skipping* any statements in between. The **continue** statement tells the compiler, “SKIP THE FOLLOWING STATEMENTS AND CONTINUE WITH THE NEXT ITERATION”. The format of the **continue** statement is simply

continue;

The use of **continue** statement in loops.

(a)

```
while(test-condition)
{
    -----
    if(-----)
        continue;
    -----
    -----
}
```

(b)

```
do
{
    -----
    if(-----)
        continue;
    -----
} while(test-condition);
```

(c)

```
for(initialization; test condition; increment)
```

```
{
    -----
    if(-----)
        continue;
    -----
}
```

LESSON 7 ARRAYS & STRINGS

Outline

Introduction - One Dimensional Array - Two-Dimensional Arrays - Multidimensional Array-
Handling of Character Strings - Declaring and Initializing String Variables -
Arithmetic Operations on Characters - String Handling Functions

INTRODUCTION

An array is a group of related data items that share a common name. For instance, we can define array name **salary** to represent a set of salary of a group of employees. A particular value is indicated by writing a number called index number or subscript in brackets after the array name.

Eg: **salary[10]**

ONE DIMENSIONAL ARRAY

An array with a single subscript is known as one dimensional array.

Eg: 1) **int number[5];**

The values to array elements can be assigned as follows.

Eg: 1) **number[0] = 35;**
number[1] = 40;
number[2] = 20;

Declaration of Arrays

The general form of array declaration is

type variable-name[size];

The type specifies the type of element that will be contained in the array, such as **int**, **float**, or **char** and the size indicates the maximum number of elements that can be stored inside the array.

Eg: 1) **float height[50];**
2) **int group[10];**
3) **char name[10];**

Initialization of Arrays

The general form of initialization of arrays is:

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

Eg: 1) **static int number[3] = {0,0};**

If the number of values in the list is less than the number of elements, then only that many elements will be initialized. The remaining elements will be set to zero automatically. Initialization of arrays in C suffers two drawbacks

- ❖ There is no convenient way to initialize only selected elements.
- ❖ There is no shortcut method for initializing a large number of array elements

like the one available in FORTRAN.

We can use the word 'static' before type declaration. This declares the variable as a static variable.

Eg :

1) **static int counter[] = {1,1,1};**

2)

```
.....
.....
for(i =0; i < 100; i = i+1)
{
if i < 50
sum[i] = 0.0;
else
sum[i] = 1.0;
}
.....
.....
```

TWO-DIMENSIONAL ARRAYS

Two-dimensional arrays are declared as follows

```
type array-name[row_size][column_size];
```

Eg: **product[i][j] = row * column;**

MULTIDIMENSIONAL ARRAY

C allows arrays of three or more dimensions. The exact limit is determined by the compiler. The general form of a multidimensional array is

```
type array_name[s1][s2][s3]...s[m];
```

Eg: 1. **int survey[3][5][12];**

2. **float table[5][4][5][3];**

3.

HANDLING OF CHARACTER STRINGS

INTRODUCTION

A string is a array of characters. Any group of characters(except the double quote sign) defined between double quotation marks is a constant string.

Eg: 1) "Man is obviously made to think"

If we want to include a double quote in a string, then we may use it with the back slash.

Eg: **printf("\well done!");**

will output

"well done!"

Program

/*Program showing one-dimensional array*/

```
main()
{
int i;
float x[10],value,total;
printf("Enter 10 real numbers:\n");
for(i =0; i < 10; i++)
{
scanf("%f",&value);
x[i] = value;
}
total = 0.0;
for(i = 0; i < 10; i++)
total = total + x[i] * x[i];
printf("\n");
for(i = 0; i < 10; i++)
printf("x[%2d] = %5.2f\n",i+1,x[i]);
printf("\nTotal = %.2f\n",total);
}
```

OUTPUT

Enter 10 real numbers:

1.1 2.2 3.3 4.4 5.5 6.6 7.7 8.8 9.9 10.10

x[1] = 1.10

x[2] = 2.20

x[3] = 3.30

x[4] = 4.40

x[5] = 5.50

x[6] = 6.60

x[7] = 7.70

x[8] = 8.80

x[9] = 9.90

x[10] = 10.10

Total = 446.86

Program (2 dim)

```

/*Program to print multiplication table*/
#define ROWS 5
#define COLUMNS 5
main()
{
int row, column, product[ROWS]
[COLUMNS];
int i, j;
printf("Multiplication table\n\n.");
printf(" ");
for(j = 1; j <= COLUMNS; j++)
printf("%4d", j);
printf("\n");
printf(" ");
for(i = 0; i < ROWS; i++)
{
row = i + 1;
printf("%2d|", row);
for(j = 1; j <= COLUMNS; j++)
{
column = j;
product[i][j] = row * column;
printf("%4d", product[i][j]);
}
printf("\n");
}
}

```

OUTPUT

Multiplication Table

1 2 3 4 5

1	1	2	3	4	5
2	2	4	6	8	10
3	3	6	9	12	15
4	4	8	12	16	20
5	5	10	15	20	25

The operations that are performed on character strings are

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

DECLARING AND INITIALIZING STRING VARIABLES

A string variable is any valid C variable name and is always declared as an array. The general form of declaration of a string variable is

char string_name[size];

Eg: **char city[10];**
char name[30];

When the compiler assigns a character string to a character array, it automatically supplies a null character ('\0') at the end of the string. Therefore, the size should be equal to the maximum number of characters in the string plus one. C permits a character array to be initialized in either of the following two forms

static char city[9] = "NEW YORK";
static char city[9] = {'N', 'E', 'W', ' ', 'Y', 'O', 'R', 'K', '\0'};

Reading Words

The familiar input function scanf can be used with %s format specification to read in a string of characters.

Eg: **char address[15];**
scanf("%s", address);

Program

```
/*Reading a series of words using scanf function*/
main()
{
    char word1[40],word2[40],word3[40],word4[40];
    printf("Enter text:\n");
    scanf("%s %s",word1, word2);
    scanf("%s", word3);
    scanf("%s",word4);
    printf("\n");
    printf("word1 = %s \n word2 = %s \n",word1,
    word2);
    printf("word3 = %s \n word4 = %s \n",word3,
    word4);
}
```

OUTPUT

Enter text:

Oxford Road, London M17ED

Word1 = Oxford

Word2 = Road

Word3 = London

Word4 = M17ED

Note: Scanf function terminates its input on the first white space it finds.

Reading a Line of Text

It is not possible to use scanf function to read a line containing more than one word. This is because the scanf terminates reading as soon as a space is encountered in the input. We can use the getchar function repeatedly to read single character from the terminal, using the function **getchar**. Thus an entire line of text can be read and stored in an array.

Program

```
/*Program to read a line of text from terminal*/
#include<stdio.h>
main()
{
    char line[81],character;
    int c;
    c = 0;
    printf("Enter text. Press<Return>at end \n");
    do
    {
        character = getchar();
        line[c] = character;
        c++;
    }
    while(character != '\n');
    c = c-1;
    line[c] = '\0';
    printf("\n %s \n",line);
}
```

OUTPUT

Enter text. Press<Return>at end

Programming in C is interesting

Programming in C is interesting

WRITING STRINGS TO SCREEN

We have used extensively the printf function with %s format to print strings to the screen. The format %s can be used to display an array of characters that is terminated by the null character. For eg, the statement

printf("%s", name);

can be used to display the entire contents of the array **name**.

ARITHMETIC OPERATIONS ON CHARACTERS

C allows us to manipulate characters the same way we do with numbers. Whenever a character constant or character variable is used in an expression, it is automatically converted into integer value by the system.

For eg, if the machine uses the ASCII representation, then,

```
x = 'a';
```

```
printf("%d \n",x);
```

will display the number 97 on the screen.

The C library supports a function that converts a string of digits into their integer values. The function takes the form `x = atoi(string)`

PUTTING STRINGS TOGETHER

Just as we cannot assign one string to another directly, we cannot join two strings together by the simple arithmetic addition. That is, the statements such as

```
string3 = string1 + string2;
```

```
string2 = string1 + "hello";
```

are **not** valid. The characters from string1 and string2 should be copied into string3 one after the other. The process of combining two strings together is called concatenation.

COMPARISON OF TWO STRINGS

C does not permit the comparison of two strings directly. That is, the statements such as

```
if(name1 == name2)
```

```
if(name == "ABC");
```

are **not** permitted. It is therefore necessary to compare the two strings to be tested, character by character. The comparison is done until there is a mismatch or one of the strings terminate into a null character, whichever occurs first.

STRING - HANDLING FUNCTIONS

C library supports a large number of string-handling functions that can be used to carry out many of the string manipulation activities. Following are the most commonly used string handling functions.

Function	Action
strcat()	Concatenates two strings
strcmp()	Compares two strings
strcpy()	Copies one string over another
strlen()	Finds the length of the string

strcat() Function

The strcat function joins two strings together. It takes the following form

```
strcat(string1,string2);
```

Eg: `strcat(part1, "GOOD");`

`strcat(strcat(string1,string2),string3);` Here three strings are concatenated and the result is stored in string1.

strcmp() Function

It is used to compare two strings identified by the arguments and has a value 0 if they

are equal. It takes the form:

```
strcmp(string1,string2);
```

Eg: 1) **strcmp(name1,name2);**
 2) **strcmp(name1,"john");**
 3) **strcmp("ram", "rom");**

strcpy() Function

This function works almost like a string assignment operator. It takes the form

```
strcpy(string1,string2);
```

This assigns the content of string2 to string1.

Eg: 1) **strcpy(city, "DELHI");**
 2) **strcpy(city1,city2);**

strlen() Function

This function counts and returns the number of characters in a string.

```
n = strlen(string);
```

Program

```
/*Illustration of string-handling functions*/
#include<string.h>
main()
{
char s1[20],s2[20],s3[20];
int x, l1, l2, l3;
printf("Enter two string constants \n");
printf("?");
scanf("%s %s", s1, s2);
x = strcmp(s1, s2);
if(x != 0)
printf("Strings are not equal \n");
strcat(s1, s2);
else
printf("Strings are equal \n");
strcpy(s3,s1);
l1 = strlen(s1);
l2 = strlen(s2);
l3 = strlen(s3);
printf("\ns1 = %s \t length = %d characters \n",s1, l1);
printf("\ns2= %s \t length = %d characters \n",s2, l2);
printf("\ns3 = %s \t length = %d characters \n",s3, l3);
}
```

OUTPUT

```
Enter two string constants
? New York
Strings are not equal
s1 = New York length = 7 characters
s2 = York length = 4 characters
s3 = New York length = 7 characters
Enter two string constants
? London London
Strings are equal
s1 = London length = 6 characters
s2 = London length = 6 characters
s3 = London length = 6 characters
```

LESSON 8 USER-DEFINED FUNCTIONS

Outline

Introduction - Need for User-Defined Functions - The Form of C Functions - Category of Functions - Handling of Non-Integer Functions – Recursion - Functions with Arrays

INTRODUCTION

C functions can be classified into two categories, namely, library functions and userdefined functions. **Main** is an example of user-defined functions, printf and scanf belong to the category of library functions. The main difference between these two categories is that library functions are not required to be written by us whereas a user-defined function has to be developed by the user at the time of writing the program.

NEED FOR USER-DEFINED FUNCTIONS

- It facilitates top-down modular programming.
- The length of the source program can be reduced by using functions at appropriate places.
- It is easy to locate and isolate a faulty function for further investigations.
- A function can be used by many other programs

THE FORM OF C FUNCTIONS

All functions have the form

```

Function-name(argument list)
argument declaration;
{
local variable declarations;
executable statement-1;
executable statement-2;
.....
.....
return(expression);
}

```

A function that does nothing may not include any executable statements.

For eg: **do_nothing() {}**

RETURN VALUES AND THEIR TYPES

The return statement can take the form:

return or return (expression);	Eg: if (x <= 0) return (0); else return (1);
---	---

CALLING A FUNCTION

A function can be called by simply using the function name in the statement.

When the compiler executes a function call, the control is transferred to the function mul(x,y).The function is then executed line by line as described and the value is returned, when a return statement is encountered. This value is assigned to p.

```

Eg:
main()
{
int p;
p = mul(10,5);
printf("%d \n", p);
}

```

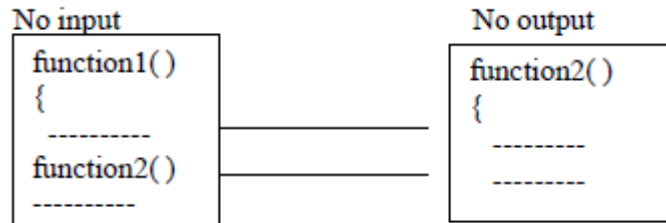
CATEGORY OF FUNCTIONS

A function may belong to one of the following categories.

- 1) Functions with no arguments and no return values.
- 2) Functions with arguments and no return values.
- 3) Functions with arguments and return values.

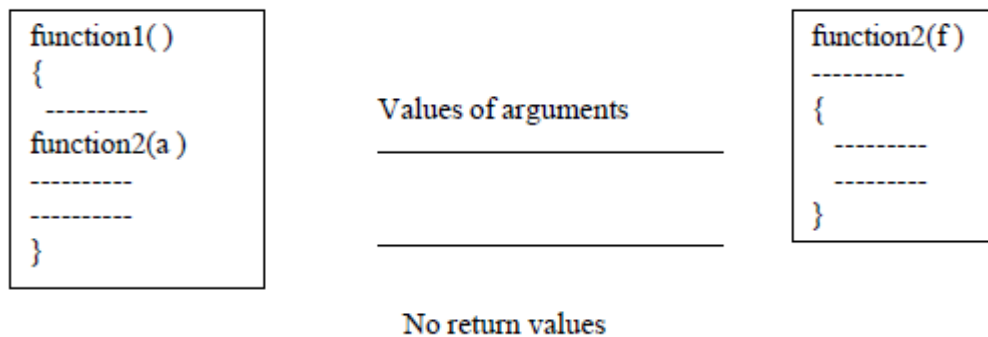
NO ARGUMENTS AND NO RETURN VALUES

A function does not receive any data from the calling function. Similarly, it does not return any value.



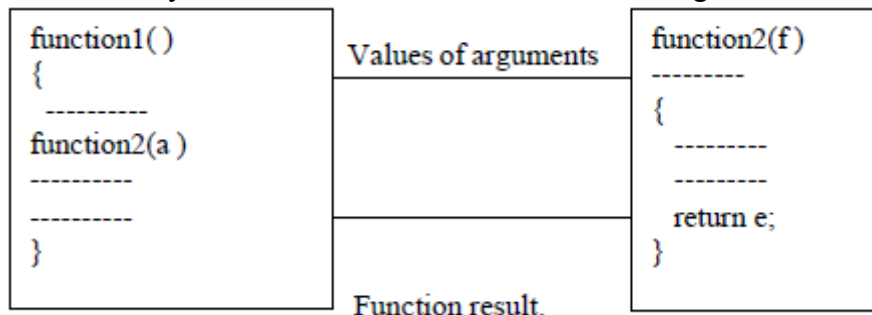
ARGUMENTS BUT NO RETURN VALUES

The nature of data communication between the calling function and the called function with arguments but no return values is shown in the diagram.



ARGUMENTS WITH RETURN VALUES

Here there is a two-way data communication between the calling and the called function.



HANDLING OF NON-INTEGER FUNCTIONS

We must do two things to enable a calling function to receive a non integer value from a called function:

1. The explicit type-specifier, corresponding to the data type required must be mentioned in the function header. The general form of the function definition is
type specifier function name(argument list)
argument declaration;
{
 function statements;
}
2. The called function must be declared at the start of the body in the calling function.

NESTING OF FUNCTIONS

C permits nesting of functions freely. main can call function1, which calls function2, which calls function3,and so on.

RECURSION

When a function in turn calls another function a process of ‘chaining’ occurs. **Recursion** is a special case of this process, **where a function calls itself**.

Eg:

```
1) main()
{
printf("Example for recursion");
main();
}
```

FUNCTIONS WITH ARRAYS

To pass an array to a called function, it is sufficient to list the name of the array, without any subscripts, and the size of the array as arguments.

Eg: 1) **largest(a,n);**

LESSON 9 STORAGE CLASSES

Outline

Introduction - Automatic Variables (local/internal) - External Variables - Static Variables - Register Variables - Ansi C Functions

INTRODUCTION

A variable in C can have any one of the four storage classes.

1. Automatic variables.
2. External variables.
3. Static variables.
4. Register variables.

AUTOMATIC VARIABLES (LOCAL/INTERNAL)

Automatic variables are declared inside a function in which they are to be utilized. They are created when a function is called and destroyed automatically when the function is exited.

We may also use the keyword **auto** to declare automatic variables explicitly.

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

EXTERNAL VARIABLES

```
Eg:
int number;
float length = 7.5;
main()
{
    -----
    -----
}
function1()
{
    -----
    -----
}
function2()
{
    -----
    -----
}
```

Variables that are both alive and active throughout the entire program are known as external variables. They are also known as global variables.

The keyword **extern** can be used for explicit declarations of external variables.

STATIC VARIABLES

As the name suggests, the value of a static variable persists until the end of the program. A variable can be declared static using the keyword **static**.

Eg: 1) **static int x;**
 2) **static int y;**
 3)

REGISTER VARIABLES

We can tell the compiler that a variable should be kept in one of the machine's registers, instead of keeping in the memory. Since a register access is much faster than a memory access, keeping the frequently accessed variables in the register will lead to faster execution of programs. This is done as follows:

register int count;

ANSI C FUNCTIONS

The general form of ANSI C function is

```
data-type function-name(type1 a1,type2 a2,.....typeN aN)
{
-----
----- (body of the function)
-----
}
```

Eg: 1) **double funct(int a, int b, double c)**

Function Declaration

The general form of function declaration is

```
data-type function-name(type1 a1,type2 a2,.....typeN aN)
```

Eg:main()

```
{
float a, b, x;
float mul(float length,float breadth); /*declaration*/
-----
-----
x = mul(a,b);
}
```

LESSON 10 POINTERS

Outline

Introduction - Understanding Pointers - Accessing the Address of a Variable -
Accessing a Variable through its Pointer - Pointer Expressions - Pointers and Arrays -
Pointers and Character Strings

INTRODUCTION

Pointers are another important feature of C language. Although they may appear a little confusing for a beginner, they are powerful tool and handy to use once they are mastered.

There are a number of reasons for using pointers.

1. A pointer enables us to access a variable that is defined outside the function.
2. Pointers are more efficient in handling the data tables.
3. Pointers reduce the length and complexity of a program.
4. They increase the execution speed.
5. The use of a pointer array to character strings result in saving of data storage space in memory.

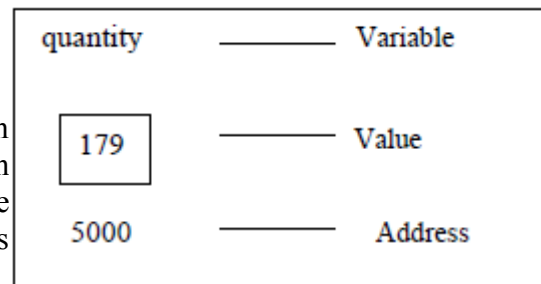
UNDERSTANDING POINTERS

Whenever we declare a variable, the system allocates, somewhere in the memory, an appropriate location to hold the value of the variable. Since, every byte has a unique address number, this location will have its own address number.

Consider the following statement:

```
int quantity = 179;
```

This statement instructs the system to find a location for the integer variable `quantity` and puts the value 179 in that location. Assume that the system has chosen the address location 5000 for `quantity`. We may represent this as shown below.



Representation of a variable During execution of the

program, the system always associates the name `quantity` with the address 5000. To access the value 179 we use either the name `quantity` or the address 5000. Since memory addresses are simply numbers, they can be assigned to some variables which can be stored in memory, like any other variable. **Such variables that hold memory addresses are called pointers. A pointer is, therefore, nothing but a**

<i>Variable</i>	<i>Value</i>	<i>Address</i>
<code>quantity</code>	179	5000
<code>p</code>	5000	5048

variable that contains an address which is a location of another variable in memory. Since a pointer is a variable, its value is also stored in the memory in another location. Suppose, we assign the address of `quantity` to a variable `p`. The link between the variables `p` and `quantity` can be visualized as shown below. The address of `p` is 5048.

Pointer as a variable

Since the value of the variable p is the address of the variable quantity, we may access the value of quantity by using the value of p and therefore, we say that the variable p ‘points’ to the variable quantity. Thus, p gets the name ‘pointer’.

ACCESSING THE ADDRESS OF A VARIABLE

The actual location of a variable in the memory is system dependent and therefore, the address of a variable is not known to us immediately. However we determine the address of a variable by using the operand & available in C. The operator immediately preceding the variable returns the address of the variable associated with it. For example, the statement

```
p = &quantity;
```

would assign the address 5000(the location of quantity) to the variable p. The &operator can be remembered as ‘address of’.

The & operator can be only used with a simple variable or an array element. The following are **illegal** use of address operator:

1. &125 (pointing at constants).
2. int x[10];
 &x (pointing at array names).
3. &(x+y) (pointing at expressions).

If x is an array, then expressions such as &x[0] and &x[i + 3] are **valid** and represent the addresses of 0th and (i + 3)th elements of x. The program shown below declares and initializes four variables and then prints out these values with their respective storage locations.

DECLARING AND INITIALIZING POINTERS

Pointer variables contain addresses that belong to a separate data type, which must be declared as pointers before we use them. The declaration of the pointer variable takes the following form:

```
data type *pt_name;
```

This tells the compiler three things about the variable pt_name:

1. The asterisk(*) tells that the variable pt_name.
2. pt_name needs a memory location.
3. pt_name points to a variable of type data type.

Example:

1. int *p;
2. float *x;

Program

```

/*****
/* ACCESSING ADDRESSES OF VARIABLES
*/
*****/

main()
{
    char a;
    int x;
    float p, q;
    a = 'A';
    x = 125;
    p = 10.25 , q = 18.76;
    printf("%c is stored as addr %u . \n", a, &a);
    printf("%d is stored as addr %u . \n", x, &x);
    printf("%f is stored as addr %u . \n", p, &p);
    printf("%f is stored as addr %u . \n", q, &q);
}

A is stored at addr 44336
125 is stored at addr 4434
10.250000 is stored at addr 4442
18.760000 is stored at addr 4438.

```

Once a pointer variable has been declared, it can be made to point to a variable using an assignment operator such as

```
p = &quantity;
```

Before a pointer is initialized it should not be used.

Ensure that the pointer variables always point to the corresponding type of data.

Example:

```
float a, b;  
int x, *p;  
p = &a;  
b = *p;
```

will result in erroneous output because we are trying to assign the address of a float variable to an integer pointer. When we declare a pointer to be of int type, the system assumes that any address that a pointer will hold will point to an integer variable.

Assigning an absolute address to a pointer variable is prohibited. The following is wrong.

```
int *ptr;  
....  
ptr = 5368;  
....  
....
```

A pointer variable can be initialized in its declaration itself. For example, `int x, *p = &x;` is perfectly valid. It declares x as an integer variable and p as a pointer variable and then initializes p to the address of x. The statement

```
int *p = &x, x;  
is not valid.
```

ACCESSING A VARIABLE THROUGH ITS POINTER

To access the value of the variable using the pointer, another unary operator *(asterisk), usually known as the indirection operator is used. Consider the following statements:

```
int quantity, *p, n;  
quantity = 179;  
p = &quantity;  
n = *p;
```

The statement `n = *p` contains the indirection operator *. When the operator * is placed before a pointer variable in an expression (on the right-hand side of the equal sign), the pointer returns the value of the variable of which the pointer value is the address. In this case, *p returns the value of the variable quantity, because p is the address of the quantity. The * can be remembered as 'value at address'. Thus the value of n would be 179. The two statements

```
p = &quantity;  
n = *p;
```

are equivalent to

```
n = *&quantity;
```

which in turn is equivalent to

```
n = quantity;
```

The following program illustrates the distinction between pointer value and the value it points to and the use of indirection operator(*) to access the value pointed to by a pointer.

Program ACCESSING VARIABLES USING POINTERS

```
main( )
{
    int x, y ;
    int * ptr;
    x =10;
    ptr = &x;
    y = *ptr;
    printf ("Value of x is %d \n\n",x);
    printf ("%d is stored at addr %u \n" , x, &x);
    printf ("%d is stored at addr %u \n" , *&x, &x);
    printf ("%d is stored at addr %u \n" , *ptr, ptr);
    printf ("%d is stored at addr %u \n" , y, &*ptr);
    printf ("%d is stored at addr %u \n" , ptr, &ptr);
    printf ("%d is stored at addr %u \n" , y, &y);
    *ptr= 25;
    printf("\n Now x = %d \n",x);
}
```

pointed to by the pointer ptr to y.

Note the use of assignment statement ***ptr=25;**

This statement puts the value of 25 at a memory location whose address is the value of ptr. We know that the value of ptr is the address of x and therefore the old value of x is replaced by 25. This, in effect, is equivalent to assigning 25 to x. This shows how we can change the value of a variable indirectly using a pointer and the indirection operator.

POINTER EXPRESSIONS

Like other variables, pointer variables can be used in expressions. For example, if p1 and p2 are properly declared and initialized pointers, then the following statements are valid.

- 1) $y = *p1 * *p2$; same as $(* p1) * (* p2)$
- 2) $sum = sum + *p1$;
- 3) $z = 5 * - *p2 / *p1$; same as $(5 * (-(* p2)))/(* p1)$
- 4) $*p2 = *p2 + 10$;

Note that there is a blank space between / and * in the statement 3 above.

C allows us to add integers to or subtract integers from pointers, as well as to subtract one pointer from another. $p1 + 4$, $p2 - 2$ and $p1 - p2$ are all allowed. If p1 and p2 are both pointers to the same array, then $p2 - p1$ gives the number of elements between p1 and p2. We may also use short-hand operators with the pointers.

```
p1++;
--p2;
Sum += *p2;
```

Pointers can also be compared using the relational operators. Pointers cannot be used in division or multiplication. Similarly two pointers cannot be added. A program to illustrate the use of pointers in arithmetic operations.

Program POINTER EXPRESSIONS

```
main ( )
{
    int a, b, *p1,* p2, x, y, z;
    a = 12;
    b = 4;
    p1 = &a;
    p2 = &b;
    x = *p1 * *p2 - 6;
    y = 4* - *p2 / *p1 + 10;
    printf("Address of a = %u\n", p1);
    printf("Address of b = %u\n", p2);
    printf("\n");
    printf("a = %d, b = %d\n", a, b);
    printf("x = %d, y = %d\n", x, y);
    *p2 = *p2 + 3;
    *p1 = *p2 - 5;
    z = *p1 * *p2 - 6;
    printf("\n a = %d, b = %d," , a , b);
    printf("\n z = %d\n" , z);
}
```

POINTER INCREMENTS AND SCALE FACTOR

We have seen that the pointers can be incremented like

```
p1 = p2 + 2;
```

```
p1 = p1 + 1;
```

and so on .

Remember, however, an expression like

```
p1++;
```

will cause the pointer p1 to point to the next value of its type.

That is, when we increment a pointer, its value is increased by the length of the data type that it points to. This length is called the scale factor. The number of bytes used to store various data types depends on the system and can be found by making use of size of operator. For example, if x is a variable, then size of(x) returns the number of bytes needed for the variable.

POINTERS AND ARRAYS

When an array is declared, the compiler allocates a base address and sufficient amount of storage to contain all the elements of array in contiguous memory location. The base address is the location of the first element (index 0) of the array. The compiler also defines the array name as a constant pointer to the first element. Suppose we declare an array x as follows:

```
static int x[5] = {1,2,3,4,5};
```

Suppose the base address of x is 1000 and assuming that each integer requires two bytes, the five elements will be stored as follows:

Elements		x[0]	x[1]	x[2]	x[3]	x[4]
Value		1	2	3	4	5
Address		1000	1002	1004	1006	1008

The name x is defined as a constant pointer pointing to the first element x[0] and therefore value of x is 1000, the location where x[0] is stored . That is ,

```
x = &x[0] = 1000
```

Accessing array elements using the pointer

Pointers can be used to manipulate two-dimensional array as well. An element in a twodimensional array can be represented by the pointer expression as follows:

```
*(a+i+j) or (*(p+i)+j)
```

The base address of the array a is &a[0][0] and starting at this address, the compiler allocates contiguous space for all the elements, row-wise. That is, the first element of the second row is placed immediately after the last element of the first row, and so on.

A program using Pointers to compute the sum of all elements stored in an array is presented below:

POINTERS IN ONE-DIMENSIONAL ARRAY

```

main ( )
{
int *p, sum , i
static int x[5] = {5,9,6,3,7};
i = 0;
p = x;
sum = 0;
printf("Element Value Address \n\n");
while(i < 5)
{
printf(" x[%d] %d %u\n", i, *p, p);
sum = sum + *p;
i++, p++;
}
printf("\n Sum = %d \n", sum);
printf("\n &x[0] = %u \n", &x[0]);
printf("\n p = %u \n", p);
}

```

Output

Element Value Address

```

X[0] 5 166
X[1] 9 168
X[2] 6 170
X[3] 3 172
X[4] 7 174
Sum = 55
&x[0] = 166
p = 176

```

POINTERS AND CHARACTER STRINGS

We know that a string is an array of characters, terminated with a null character. Like in one-dimensional arrays, we can use a pointer to access the individual characters in a string. This is illustrated in the program given below.

/* Pointers and character Strings */

```

main()
{
char * name;
int length;
char * cptr = name;
name = "DELHI";
while ( *cptr != '\0')
{
printf( "%c is stored at address %u \n",
*cptr,cptr);
cptr++;
}
length = cptr-name;
printf("\n length = %d \n", length);
}

```

String handling by pointers

One important use of pointers in handling of a table of strings. Consider the following array of strings:

```
char name[3][25];
```

This says that name is a table containing three

names, each with a maximum length of 25 characters (including null character).

Total storage requirements for the name table are 75 bytes. Instead of making each row a fixed number of characters , we can make it a pointer to a string of varying length.

For example,

```

static char *name[3] = { "New zealand",
"Australia",
"India"
};

```

declares name to be an array of three pointers to characters, each pointer pointing to a particular name as shown below:

```

name[0] → New Zealand
name[1] → Australia
name[2] → India

```

LESSON 11 POINTERS AND FUNCTIONS

Outline

Pointers as Function Arguments - Pointers and Structures - The Preprocessor

Program POINTERS AS FUNCTION PARAMETERS

```
main ()
{
    int x , y;
    x = 100;
    y = 200;
    printf("Before exchange : x = %d y = %d \n\n", x , y);
    exchange(&x, &y);
    printf("After exchange : x = %d y = %d \n\n", x , y);
}

exchange(a, b)
int *a, *b;
{
    int t;
    t = *a; /*Assign the value at address a to t*/
    *a = *b; /*Put the value at b into a*/
    *b = t; /*Put t into b*/
}
```

the addresses of variable is known as call by reference. The function which is called by 'Reference' can change the value of the variable used in the call.

Passing of pointers as function parameters

1. The function parameters are declared as pointers.
 2. The dereference pointers are used in the function body.
 3. When the function is called, the addresses are passed as actual arguments.
- Pointers parameters are commonly employed in string functions.

Pointers to functions

A function, like a variable has an address location in the memory. It is therefore, possible to declare a pointer to a function,

POINTERS AS FUNCTION ARGUMENTS

In the example, we can pass the address of the variable a as an argument to a function in the normal fashion. The parameters receiving the addresses should be pointers. The process of calling a function using pointers to pass

Program

POINTERS TO FUNCTIONS

```
#include <math.h>
#define PI 3.141592
main ()
{
    double y( ), cos( ), table( );
    printf("Table of y(x) = 2*x*x-x+1\n\n");
    table(y, 0.0 , 2.0, 0.5);
    printf("\n Table of cos(x) \n\n");
    table(cos, 0.0, PI , 0.5);
}

double table(f, min, max, step)
double (*f) ( ), min, max , step;
{
    double a, value;
    for( a = min; a <= max; a += step)
    {
        value = (*f)(a);
        printf("%5.2f %10.4f\n", a, value);
    }
}

double y(x)
double x;
{
    return (2*x*x-x+1);
}
```

which can then be used as an argument in another function. A pointer to a function is declared as follows:

```
type (*fptr)( );
```

This tells the compiler that fptr is a pointer to a function which returns type value. A above program to illustrate a function pointer as a function argument.

POINTNTERS AND STRUCTERS

The name of an array stands for the address of its zeroth element. Consider the following declaration:

```
struct inventory
{
char name[30];
int number;
float price;
} product[2], *ptr;
```

This statement declares product as an array of two elements, each of type of struct inventory and ptr as a pointer to data objects of the type struct inventory.

The assignment

```
ptr = product;
```

would assign the address of the zeroth element of product to ptr. Its members can be accessed using the following notation .

```
ptr → name
ptr → number
ptr → price
```

Initially the pointer ptr will point to product[0], when the pointer ptr is incremented by one it will point to next record, that is product[1].

We can also use the notation

```
(*ptr).number
```

to access the member number.

A program to illustrate the use of structure pointers. While using structure pointers we should take care of the precedence of operators.

For example, given the definition

```
struct
{
int count;
float *p;
} *ptr;
```

Then the statement

```
++ptr → count;
```

increments count, not ptr.

Program

POINTERS TO STRUCTURE VARIABLES

```
struct invent
{
char *name[20];
int number;
float price;
};
main( )
{
struct invent product[3], *ptr;
printf("INPUT\n\n");
for(ptr = product; ptr < product + 3; ptr++)
scanf("%s %d %f", ptr → name, &ptr → number ,
& ptr → price);
printf("\Noutput\n\n");
ptr = product;
while(ptr < product +3)
{
printf("%-20s %5d %10.2f\n" ,
ptr → name,
ptr → number ,
ptr → price); ptr++;
}
}
```

However, `(++ptr) → count;`
increments `ptr` first and then links `count`.

THE PREPROCESSOR

The Preprocessor, as the name implies, is a program that processes the source code before it passes through the compiler. It operates under the control of preprocessor command lines or directives. Preprocessor directives are placed in the source program before the main line. Before the source code passes through the compiler, it is examined by the preprocessor for any preprocessor directives. If there are any, appropriate actions (as per the directives) are taken and then the source program is handed over to the compiler.

Preprocessor directives begin with the symbol `#` in column one and do not require a semicolon at the end.

Commonly used Preprocessor directives

Directive	Function
<code>#define</code>	Defines a macro substitution
<code>#undef</code>	Undefines a macro
<code>#include</code>	specifies the files to be include
<code>#ifdef</code>	Tests for a macro definition
<code>#endif</code>	specifies the end of <code>#if</code>
<code>#ifndef</code>	Tests whether a macro is not defined
<code>#if</code>	Tests a compile time condition
<code>#else</code>	specifies alternatives when <code>#if</code> fails

Preprocessor directives can be divided into three categories

- 1) Macro substitution directives
- 2) File Inclusion directives
- 3) Compiler control directives

Macro Substitution directive

The general form is `#define identifier string`

Examples

- 1) `#define COUNT 100` (simple macro substitution)
- 2) `#define CUBE(x) x*x*x` (macro with arguments)
- 3) `#define M 5`
`#define N M+1` (nesting of macros)

File Inclusion directive

This is achieved by

`#include "filename"` or `#include <filename>`

- Examples
- 1) `#include <stdio.h>`
 - 2) `#include "TEST.C"`

Compiler control directives

These are the directives meant for controlling the compiler actions. C preprocessor offers a feature known as conditional compilation, which can be used to switch off or on a particular line or group of lines in a program. Mostly `#ifdef` and `#ifndef` are used in these directives.

LESSON 12 STRUCTURES

Outline

Introduction - Structure Definition - Array Vs Structure - Giving Values to Members -
Structure Initialization - Comparison of Structure Variables - Arrays of Structures

INTRODUCTION

C supports a constructed data type known as structures, a mechanism for packing data of different types. A structure is a convenient tool for handling a group of logically related data items. For example, it can be used to represent a set of attributes, such as student _ name, roll _ number and marks. The concept of a structure is analogous to that of a 'record' in many other languages. More examples of such structures are:

time : seconds, minutes, hours
data : day, month, year
book : author, title, price, year
city : name, country, population

STRUCTURE DEFINITION

Unlike arrays, structure must be defined first for their format that may be used later to declare structure variables. Let us use an example to illustrate the process of structure definition and the creation of structure variables. Consider a book database consisting of book name, author, number of pages, and price. We can define a structure to hold this information as follows:

```
struct book _bank
{
    char title[20];
    char author[15];
    int pages;
    float price;
};
```

structure definition

```
struct tag _name
{
    data _type    member1;
    data _type    member2;
    -----      ----
    -----      ----
};
```

The keyword struct declares a structure to hold the details of four data fields, namely title, author, pages, and price. These fields are called structure elements or members. Each member may belong to different type of data. book _ bank is the name of the structure and is called the structure tag. The tag name may be used subsequently to declare variables that have the tag's structure. In defining a structure, we may note the following syntax:

1. The template is terminated with a semicolon.
2. While the entire definition is considered as a statement, each member is declared independently for its name and type in a separate statement inside the template.
3. The tag name such as book _ bank can be used to declare structure variables of its type, later in the program.

ARRAY VS STRUCTURE

1. An array is a collection of related data elements of same type. Structure can have elements of different types.
2. An array is derived data type whereas structure is a programmer-defined one.
3. Any array behaves like a built-in data type. All we have to do is to declare an array variable and use it. But in the case of a structure, first we have to design and declare a data structure before the variables of that type are declared and used.

GIVING VALUES TO MEMBERS

We can access and assign values to the members of a structure in a number of ways. The members themselves are not variables. They should be linked to the structure variables in order to make them meaningful members. For example, the word title has no meaning, whereas the phrase ‘title of book’ has a meaning. The link between a member and a variable is established using the member operator ‘.’, which is also known as ‘dot operator’ or ‘period operator’.

For example,

book1.price

is the variable representing the price of the book1 and can be treated like any other ordinary variable. Here is how we would assign values to the member of book1:

```
strcpy(book1.title, "COBOL");
strcpy(book1.author, "M.K.ROY");
book1.pages = 350;
book1.price = 140;
```

We can also use scanf to give the values through the keyboard.

```
scanf("%s\n", book1.title);
scanf("%d\n", &book1.pages);
```

are valid input statements.

Example :

Define a structure type, struct personal, that would contain person name, date of joining and salary. Using this structure, write a program to read this information for one person from the keyboard and print the same on the screen.

Structure definition along with the program is shown below. The scanf and printf functions illustrate how the member operator ‘.’ is used to link the structure members to the structure variables. The variable name with a period and the member name is used like an ordinary variable.

Program

```
/******
/* DEFINING AND ASSIGNING VALUES TO
/* STRUCTURE MEMBERS */
/******

struct personal
{
char name[20];
int day;
char month[10];
int year;
float salary;
};

main()
{
struct personal person;
printf("Input values\n");
scanf("%s %d %s %d %f",
person.name,
&person.day,
person.month,
&person.year,
&person.salary);
printf("%s %d %s %d %.2f\n",
person.name,
person.day,
person.month,
person.year,
person.salary);
}
```

STRUCTURE INITIALIZATION

Like any other data type, a structure variable can be initialized at compile time.

```
main()
{
    struct
    {
        int weight;
        float height;
    }
    student = {60, 180.75};
    .....
    .....
}
```

This assigns the value 60 to student. weight and 180.75 to student. height. There is a one-to-one correspondence between the members and their initializing values. A lot of variation is possible in initializing a structure. The following statements initialize two structure variables. Here, it is essential to use a tag name.

```
main()
{
    struct st _record
    {
        int weight;
        float height;
    };
    struct st_record student1 = {60, 180.75};
    struct st_record student2 = {53, 170.60};
    .....
    .....
}
```

C language does not permit the initialization of individual structure member within the template. The initialization must be done only in the declaration of the actual variables.

COMPARISON OF STRUCTURE VARIABLES

Two variables of the same structure type can be compared the same way as ordinary variables. If person1 and person2 belong to the same structure, then the following operations are valid:

Operation	Meaning
person1 = person2	Assign person2 to person1.
person1 == person2	Compare all members of person1 and person2 and return 1 if they are equal, 0 otherwise.
person1 != person2	Return 1 if all the members are not equal, 0 otherwise.

Note that not all compilers support these operations. For example, Microsoft C version does not permit any logical operations on structure variables. In such cases, individual member can be

compared using logical operators.

ARRAYS OF STRUCTURES

We use structure to describe the format of a number of related variables. For example, in analyzing the marks obtained by a class of students, we may use a template to describe student name and marks obtained in various subjects and then declare all the students as structure variables. In such cases, we may declare an array of structure, each elements of the array representing a structure variable. For example,

```
struct class student[100];
```

It defines an array called student, that consists of 100 elements. Each elements is defined to be of the type struct class. Consider the following declaration:

```
struct marks
{
int subject1;
int subject2;
int subject3;
};
main()
{
static struct marks student[3] = { {45, 68, 81}, {75, 53, 69}, {57,36,71} };
```

This declares the student as an array of three elements students[0], student[1], and student[2] and initializes their members as follows:

```
student[0].subject1=45;
student[0].subject2=68;
.....
.....
student[2].subject3=71;
```

An array of structures is stored inside the memory in the same way as a multi- dimensional array.

LESSON 13 STRUCTURES AND UNION

Outline

Introduction - Structures within Structures - Structures and Functions – Unions -
Size of Structures - Bit Fields

INTRODUCTION

C permits the use of arrays as structure members. We have already used arrays of characters inside a structure. Similarly, we can use single or multi-dimensional arrays of type int or float. For example, the following structure declaration is valid:

```
struct marks
{
    int number;
    float subject[3];
    student[2];
}
```

Here, the member subject contains three elements, subject [0], subject [1] and subject [2]. These elements can be accessed using appropriate subscripts. For example, the name student[1].subject[2]; would refer to the marks obtained in the third subject by the second student.

STRUCTURES WITHIN STRUCTURES

Structures within structures means nesting of structures. Nesting of structures is permitted in C. Let us consider the following structure defined to store information about the salary of employees.

```
struct salary
{
    char name[20];
    char department[10];
    int basic _ pay;
    int dearness _ allowance;
    int house _ rent _ allowance;
    int city _ allowance;
}
employee;
```

This structure defines name, department , basic pay and three kinds of allowances. We can group all items related to allowance together and declare them under a substructure as shown below:

```
struct salary
{
    char name[20];
    char department[10];
    struct
    {
```

```
int dearness;  
int house_rent;  
int city;  
}  
allowance;  
}  
employee;
```

is
legal:

An inner structure can have more than one variable. The following form of declaration

```
struct salary  
{  
.....  
struct  
{  
nt dearness;  
.....  
}  
allowance,  
arrears;  
}  
employee[100];
```

It is also possible to nest more than one type of structures.

```
struct personal_record  
{  
struct name_part name;  
struct addr_part address;  
struct date _ of _ birth  
.....  
.....  
};  
struct personal_record person 1;
```

The first member of this structure is name which is of the type struct name_part.
Similarly, other members have their structure types.

STRUCTURES AND FUNCTIONS

C supports the passing of structure values as arguments to functions. There are three methods by which the values of a structure can be transferred from one function to another. The first method is to pass each member of the structure as an actual argument of the function call. The second method involves passing of a copy of the entire structure to the called function. The third approach employs a concept called pointers to pass the structure as an argument. The general format of sending a copy of a structure to the called function is:

```
function name(structure variable name)
```

The called function takes the following form:

```
data_type function name(st_name)
struct_type st_name;
{
    .....
    .....
    return (expression);
}
```

The following points are important to note:

1. The called function must be declared for its type, appropriate to the data type it is expected to return. For example, if it is returning a copy of the entire structure, then it must be declared as struct with an appropriate tag name.
2. The structure variable used as the actual argument and the corresponding formal argument in the called function must be of the same struct type.
3. The return statement is necessary only when the function is returning some data. The expression may be any simple variable or structure variable or an expression using simple variables.
4. When a function returns a structure, it must be assigned to a structure of identical type in the calling function.
5. The called function must be declared in the calling function for its type, if it is placed after the calling function.

UNIONS

Like structures, a union can be declared using the keyword union as follows:

```
union item
{
    int m;
    float x;
    char c;
} code;
```

This declares a variable code of type union item. The compiler allocates a piece of storage that is large enough to hold the largest variable type in the union. To access a union member, we can use the same syntax that we use for structure members. That is,

code.m

code.x

code.c

are all valid member variables. During accessing, we should make sure that we are accessing the member whose value is currently stored. For example, the statement such as

```
code.m = 379;
code.x=7859.36;
printf("%d", code.m);
```

would produce erroneous output.

In effect, a union creates a storage location that can be used by any one of its members at a time. When a different member is assigned a new value, the new value supercedes the previous

member's value.

SIZE OF STRUCTURES

We normally use structures, unions and arrays to create variables of large sizes. The actual size of these variables in terms of bytes may change from machine to machine. We may use the **unary** operator **sizeof** to tell us the size of a structure. The expression **sizeof(struct x)** will evaluate the number of bytes required to hold all the members of the structure x. If y is a simple structure variable of type **struct x**, then the expression **sizeof(y)** would also give the same answer. However, if y is an array variable of type **struct x**, then **sizeof(y)** would give the total number of bytes the array requires. This kind of information would be useful to determine the number of records in a database. For example, the expression **sizeof(y) / sizeof(x)** would give the number of elements in the array y.

BIT FIELDS

C permits us to use small bit fields to hold data items and thereby to pack several data items in a word of memory. Bit fields allow direct manipulation of string of a string of preselected bits, as if it is represented an integral quantity. A bit field is a set of adjacent bits whose size can vary from 1 to 16 bits in length. A word can be divided into a number of bit fields. The name and size of bit fields are defined using a structure.

The general form of bit field definition is

```
struct tag-name
{
    data-type name1 : bit-length;
    data-type name2 : bit-length;
    data-type name3 : bit-length;
    -----
    -----
    -----
    data-type nameN : bit-length;
}
```

The data type is either int or unsigned int or signed int and the bit-length is the number of bits used for the specific name. The bit-length is decided by the range of value to be stored. The largest value that can be stored is 2^{n-1} , where n is bit-length. The internal representation of bit-field is machine dependent. It depends on the size of int and the ordering of bits.

Example :

Suppose we want to store and use the personal information of employees in compressed form.

This can be done as follows:

```
struct personal
{
    unsigned sex: 1
    unsigned age : 7
    unsigned m_status: 1
    unsigned children: 3
    unsigned : 4
} emp;
```


This defines a variable name emp with 4 bit fields. The range of values each field could have is as follows:

Bit Filed	Bit length	Range of values
sex	1	0 or 1
age	7	0 to 127 ($2^7 - 1$)
m_status	1	0 or 1
children	3	0 to 7 ($2^3 - 1$)

The following statements are valid :

```
emp.sex =1 ;  
emp.age = 50;
```

It is important to note that we can not use scanf to read the values in to the bit field.

LESSON 14 FILES

Outline

Introduction - Defining and Opening a File - Closing a File - Input/Output operations on Files

INTRODUCTION

Many real-life problems involve large volumes of data and in such situations, the console oriented I/O operations pose two major problems.

1. It becomes cumbersome and time consuming to handle large volumes of data through terminals.
2. The entire data is lost when either the program is terminated or the computer is turned-off.

It is therefore necessary to have a more flexible approach where data can be stored on the disk and read whenever necessary, without destroying the data. This method employs the concept of files to store data.

There are two distinct ways to perform file operations in C. The first one is known as the low-level I/O and uses UNIX system calls. The second method is referred to as the highlevel I/O operation and uses functions in C's standard I/O library.

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

DEFINING AND OPENING A FILE

Data structure of a file is defined as `FILE` in the library of standard I/O function definitions. Therefore all files should be declared as type `FILE` before they are used. `FILE` is a defined data type.

The following is the general format for declaring and opening a file:

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

Mode can be one of the following:

- r** open the file for reading only.
- w** open the file for writing only.
- a** open the file for appending(or adding)data to it.

The additional modes of operation are:

- r+** the existing file is opened to the beginning for both reading and writing.
- w+** same as w except both for reading and writing.
- a+** same as a except both for reading and writing.

CLOSING A FILE

A file must be closed as soon as all operations on it have been completed. This ensures that all outstanding information associated with the file is flushed out from the buffers and all links to the file are broken. It also prevents any accidental misuse of the file. The syntax for closing a file is

```
fclose( file_pointer);
```

This would close the file associated with the FILE pointer file_pointer .

Example:

```
.....  
.....  
FILE *p1, *p2;  
p1 = fopen("INPUT", "w");  
p2 = fopen("OUTPUT", "r");  
.....  
.....  
fclose(p1);  
fclose(p2);  
.....
```

This program opens two files and closes them after all operations on them are completed. Once a file is closed, its file pointer can be reused for another file. All files are closed automatically whenever a program terminates.

INPUT/OUTPUT OPERATIONS ON FILES

The **getc** and **putc** functions

The file i/o functions **getc** and **putc** are similar to **getchar** and **putchar** functions and handle one character at a time. The statement

```
putc(c, fp1);
```

writes the character contained in the character variable **c** to the file associated with FILE pointer **fp1**. Similarly, **getc** is used to read a character from a file that has been opened in the read mode. For example, the statement

```
c = getc(fp2);
```

would read a character from the file whose file pointer is **fp2**.

A program to illustrate to read data from the keyboard, write it to a file called INPUT, again read the same data from the INPUT file, and then display it on the screen is given below:

Program

```

/*****
/* WRITING TO AND READING FROM A FILE */
*****/
#include <stdio.h>
main ( )
{
    FILE *fp1;
    char c;
    printf("Data input\n\n");
    fl = fopen("INPUT", "w")
    while((c = getchar())!=EOF)
    putc(c,fl)
    fclose(fl);
    printf("\n Data OUTPUT\n\n");
    fl = fopen("INPUT", "r")
    while ((c =getc( fl))!=EOF)
    printf("%c", c);
    fclose(fl);
}

```

Output*Data input*

This a program to test the file handling
features in the system ^z

Data output

This is a program to test the file handling
features in the system

Character oriented read/write operations on a file.

The getw and putw functions

The getw and putw are integer oriented functions. They are similar to getc and putc functions and are used to read and write integer values only. The general forms of getw and putw are:

```

putw(integer,fp);
getw(fp);

```

A program to read a series of integer numbers from the file DATA and then write all odd numbers into the file ODD and even numbers into file EVEN is given below:

Program HANDLING OF INTEGER DATA FILES

```

#include <stdio.h>
main ( )
{
FILE *f1, *f2, *f3;
int number, i;
printf("Contents of DATA file\n\n");
f1 = fopen("DATA", "w"); /*Create DATA file */
for(i =1; i<=30; i++)
{
scanf("%d", &number);
if(number == -1)break;
putw(number,f1);
}
fclose(f1);
f1 = fopen("DATA", "r");
f2 = fopen("ODD", "w");
f3 = fopen("EVEN", "w");
while((number = getw(f1))!= EOF) /*Read from DATA file */
{
if(number %2 == 0)
putw(number,f3); /*Write to EVEN file */
else
putw(number,f2); /*Write to ODD file */
}
fclose(f1);
fclose(f2);
fclose(f3);
f2 = fopen("ODD", "r");
f3 = fopen("EVEN", "r");
printf("\n\nContents of ODD file \n\n");
while((number = getw(f2)) != EOF)
printf("%4d", number);
printf("\n\n Contents of EVEN file\n\n");
while((number = getw(f3)) != EOF)
printf("%4d", number);
fclose(f2);
fclose(f3);
}

```

The fprintf and fscanf functions

The functions fprintf and fscanf perform I/O operations that are identical to the printf and scanf functions except of course that they work on files. The general form of fprintf is

```
fprintf(fp, "control string", list);
```

The general format of fscanf is

```
fscanf(fp, "control string", list);
```

Write a program to open a file named INVENTORY and store in it the following data:
(see next page)

Item name	Number	Price	Quantity
AAA-1	111	17.50	115
BBB-2	125	36.00	75
C-3	247	31.75	104

Extend the program to read this data from the file INVENTORY and display the inventory table with the value of each item

/*Program HANDLING OF FILES WITH MIXED DATA TYPES */

```

/* (scanf and fprintf ) */
#include <stdio.h>
main( )
{
FILE *fp;
int number,quantity,i;
float price,value;
char item[10],filename[10];
printf("Input file name\n");
scanf("%s",filename);
fp = fopen(filename, "w");
printf("Input inventory data\n\n");
printf("Item name Number Price Quantity\n");
for(i=1;i <=3;i++)
{
fscanf(stdin, "%s %d %f %d",
item, &number, &price, &quantity);
fprintf(fp, "%s %d %2f %d",
item, number, price, quantity);
}
fclose(fp);
fprintf(stdout, "\n\n");
fp = fopen(filename, "r");
printf("Item name Number Price Quantity\n");
for(i=1; i <=3; i++)
{
fscanf(fp, "%s %d %f %d",
item, &number, &price, &quantity);
value = price * quantity;
fprintf(stdout, "%-8s %7d %8.2f %8d %11.2f\n",
item, number, price, quantity, value); }
fclose(fp);
}

```

LESSON 15

ERROR HANDLING DURING FILE I/O OPERATIONS

Outline

Introduction to Error Handling - Random Access To Files

INTRODUCTION TO ERROR HANDLING

The `feof` function can be used to test for an end of file condition. It takes a `FILE` pointer as its only argument and returns non zero integer value if all of the data from the specified file has been read, and returns zero otherwise. If `fp` is a pointer to file that has just been opened for reading , then the statement

```
if(feof(fp))  
    printf("End of data.\n");
```

would display the message "End of data." on reaching the end of file condition. The `ferror` function reports the status of the file indicated. It also takes a `FILE` pointer as its argument and returns a nonzero integer if an error has been detected up to that point, during processing. It returns zero otherwise. The statement

```
if(ferror(fp) !=0)  
    printf("An error has occurred.\n");
```

would print the error message, if the reading is not successful. If the file cannot be opened for some reason then the function returns a null pointer. This facility can be used to test whether a file has been opened or not.

Example:

```
if(fp == NULL)  
    printf("File could not be opened.\n");
```

RANDOM ACCESS TO FILES

There are occasions, where we need accessing only a particular part of a file and not in reading the other parts. This can be achieved by using the functions `fseek`, `ftell`, and `rewind` available in the I/O library.

ftell takes a file pointer and returns a number of type `long`, that corresponds to the current position and useful in saving the current position of a file, which can be used later in the program. It takes the following form:

```
n = ftell(fp);
```

rewind takes a file pointer takes a file pointer and resets the position to the start of the file.

Example :

```
rewind(fp);  
n = ftell(fp);
```

would assign 0 to `n` because the file position has been set to the start of the file by `rewind`.

fseek function is used to move the file position to a desired location within the file.

The syntax is

```
fseek(file ptr, offset, position);
```

The position can take any one of the following three values:

Value	Meaning
0	Beginning of file.
1	Current position.
2	End of file.

The offset may be positive, meaning move forwards, or negative , move backwards. The following examples illustrates the operation of the fseek function:

Statement	Meaning
fseek(fp,0L,0);	Go to the beginning .
fseek(fp,0L,1);	Stay at the current position.
fseek(fp, 0L,2);	Go to the end of the file, past the last character of the file.
fseek(fp,m,0);	Move to (m+1)th bytes in the file.
fseek(m,1);	Go forward by m bytes.
fseek(fp,-m,1);	Go backward by m bytes from the current position.
fseek(fp,-m,2);	Go backward by m bytes from the end.

When the operation is successful, fseek returns 0 or if the operation fails it returns -1.

LESSON 16

COMMAND LINE ARGUMENTS

Outline

Command Line Arguments - Program for Command Line Arguments

COMMAND LINE ARGUMENTS

It is a parameter supplied to a program when a program is invoked. The main can take two arguments called argc and argv. The variable argc is an argument counter that counts the number of arguments on the command line. The argv is an argument vector and represents an array of character pointers that point to the command line arguments. The size of this array will be equal to the value of the argc. In order to access the command line arguments, we must declare the main function and its parameters as follows.

```
main(argc,argv)
int argc;
char *argv[];
{
.....
.....
}
```

A program that will receive a file name and a line of text as command line arguments and write the text to the file.

PROGRAM FOR COMMAND LINE ARGUMENTS

COMMAND LINE ARGUMENTS

```
#include<stdio.h>
main(argc,argv) /* main with arguments */
int argc; /* argument count */
char argv[]; /* list of arguments */
{
FILE *fp;
int i;
char word[15];
fp = fopen(argv[1], "w"); /* Open file with name argv[1] */
printf("\nNo of arguments in command line = %d\n\n", argc);
for(i=2;i<argc;i++)
fprintf(fp,"%s",argv[i]); /* Write to file argv[1] */
fclose(fp);
/* Writing content of the file to screen */
printf("Contents of %s file\n\n",argv[1]);
fp = fopen(argv[1], "r");
for(i=2;i<argc;i++)
{
fscanf(fp,"%s",word);
printf("%s",word);
}
fclose(fp);
printf("\n\n");
}
```

LESSON 17

LINEAR DATA STRUCTURES

Outline

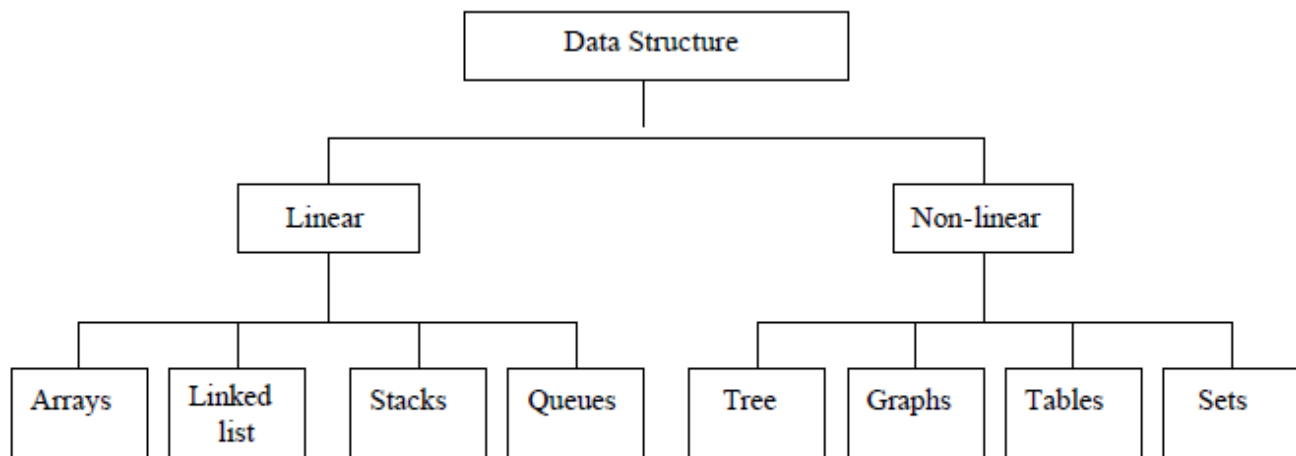
Introduction - Implementation of a list - Traversal of a list - Searching and retrieving an element - Predecessor and Successor- Merging of lists

INTRODUCTION

Data structure is one of the method of representation of logical relationships between individual data elements related to the solution of a given problem. Data structure is the most convenient way to handle data of different data types including abstract data type for a known problem. For example, the characteristics of a house can be represented by house name, house number, location, number of floors, number of rooms on each floor etc. Data structure is the base of the programming tools and the choice of data structure provides the following things:

1. The data structure should be satisfactory to represent the relationship between data elements.
2. The data structure should be easy so that the programmer can easily process the data, as per requirement.

Data structures have been classified in several ways as outlined below:



Types of data structure

Linear: In the linear method, values are arranged in a linear fashion. An array, linked list, stacks and queues are examples of linear data structure in which values are stored in a sequence. There is a relation between the adjacent elements of a linear list.

Non-linear: This is just opposite to linear. The data values in this structure are not arranged in order. Tree, graph, table and sets are examples of nonlinear data structure.

Homogenous: In this type of data structure, values of same data types are stored. An example of it can be an array that stores similar data type elements having different locations for each element of them.

Non-homogenous: In this type, data values of different types are grouped like in structure and classes.

Dynamic: In dynamic data structure like references and pointers, size and memory locations can be changed during program execution.

Static: A static keyword in C/C++ is used to initialize the variable to 0(NULL). Static variable value remains in the memory throughout the program. Value of the static variable persists. In C++, a member function can be declared as static and such a function can be invoked directly.

IMPLEMENTATION OF A LIST

There are two methods to implement the list: They are classified as Static and Dynamic.

Static implementation: Static implementation can be achieved using arrays. Though it is a very simple method, it has a few limitations. Once the size of an array is declared, its size cannot be altered during program execution. In array declaration, memory is allocated equal to array size. Thus, the program itself decides the amount of memory needed and it informs accordingly to the compiler. Hence memory requirement is determined in advance before compilation and the compiler provides the required memory. Static allocation of memory has a few disadvantages. It is an inefficient memory allocation technique. It is suitable only when we exactly know the number of elements to be stored.

Dynamic implementation: Pointers can also be used for implementation of a stack. The linked list is an example of this implementation. The limitations noticed in static implementation can be removed using dynamic implementation. The dynamic implementation is achieved using pointers. Using pointer implementation at run time there is no restriction on number of elements. The stack may be expanded. It is efficient in memory allocation because the program informs the compiler its memory requirement at run time. Memory is allocated only after an element is pushed.

TRAVERSAL OF A LIST

A simple list can be created using an array in which elements are stored in successive memory locations. The following program is an example. Given below is a program to create a simple list of elements. The list is displayed in original and reverse order.

<pre>#include <stdio.h> #include <conio.h> main() { int sim[5],j; clrscr(); printf("\n Enter five elements :"); for(j=0; j<5; j++) scanf("%d", &sim[j]); printf("\n List :");</pre>	<pre> for(j=0; j<5; j++) printf("%d", sim[j]); printf("\n Reverse List :"); for(j=4; j>=0; j--) printf("%d",sim[j]); }</pre>
--	---

OUTPUT

Enter five elements: 1 5 9 7 3

List : 1 5 9 7 3

Reverse list: 3 7 9 5 1

Explanation: In the above program, using the declaration of an array a list is implemented. Using for loop and scanf() statements five integers are entered. The list can be displayed using the printf() statement. Once a list is created, various operations such as sorting and searching can be applied.

SEARCHING AND RETRIEVING AN ELEMENT

Once a list is created, we can access and perform operations with the elements. One can also specify some conditions such as to search numbers which are greater than 5, or equal to 10 or any valid condition. If the list contains more elements, then it may be difficult to find a particular element and its position. Consider the following program which creates a list of integer elements and also search for the entered number in the list.

<pre>#include <stdio.h> #include <conio.h> main() { int sim[7], j, n, f=0; clrscr(); printf("\n Enter seven Integers :"); for (j=0; j<7; j++) scanf("%d", &sim[j]); printf("\n Enter Integer to search :"); scanf("%d", &n); for(j=0; j<7; j++) { if(sim[j] == n)</pre>	<pre>{ f=1; printf("\n Found ! position of integer %d is %d", n, j+1); break; } if(f==0) printf("\n Element not found !"); } }</pre>
---	--

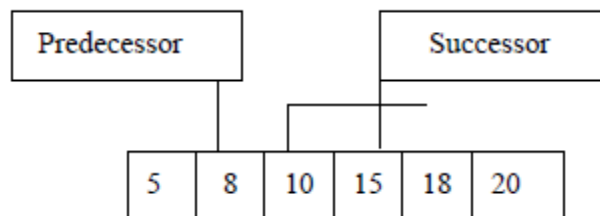
OUTPUT

```
Enter seven integers: 24 23 45 67 56 89
Enter integer to search: 67
Found ! position of integer 67 is 5.
```

Note: In the above program an array `sim[7]` is declared and it stores seven integers, which are entered by the programmer. Followed by this, an element is entered to search in the list. This is done using the for loop and the embedded if statement. The `if()` statement checks the entered element with the list elements. If the match is found the flag is set to '1', else it remains '0'. When a match is found, the number's position is displayed.

PREDECESSOR AND SUCCESSOR

The elements of a list are located with their positions. For example, we can place the elements at different locations and call them as element n , $(n-1)$ as predecessor and $(n+1)$ as successor. Once the position of an element is fixed, we can easily find its predecessor and successor. In other words, the elements have relative positions in the list. The left element is the predecessor and the right element is the successor. The first element of a list does not have a predecessor and the last one does not have a successor.



Predecessor and successor

The above diagram shows the predecessor and successor elements of number 10.

The following program displays the predecessor and successor elements of the entered element from

```
#include<stdio.h>
#include<conio.h>
main()
{
    int num[8], j, n, k=0, f=0;
    clrscr();
    printf("\n Enter eight elements :

    for(j=0; j<8; j++)
        scanf("%d", &num[j]);
    printf("\n Enter an element :");
    scanf("%d", &n);
    for(j=0; j<8; j++)
    {
        if (n== num[j])
        {
            f=1;
            (j>0) ? printf("\n The predecessor of %d
                    is %d", num[j],num[j-1]);
            printf(" No predecessor");
            (j==7) ? printf("\n No successor");
            printf("\n The successor of %d is %d",
                    num[j], num[j+1]);
            break;
        }
        k++;
    }
    if(f == 0)
        printf("\n The element %d is not found in list",
                n);
}
```

OUTPUT

Enter eight elements :
1 2 5 8 7 4 46
Enter an element : 5
The predecessor of 5 is 2
The successor of 5 is 8
Enter eight elements :
1 2 3 4 5 6 7 8
Enter an element:1
No predecessor
The successor of 1 is 2
Enter eight elements:
12 34 54 76 69 78 85 97
Enter an element : 3
The element 3 is not found in the list.

the list. Program to find the predecessor and successor of the entered number in a list.

INSERTION

When a new element is added in the list it called as **appending**. However, when an element is added in between two elements in the list, then the process is called as **insertion**. The insertion can be done at the beginning, inside or anywhere in the list. For successful insertion of an element, the array implementing the list should have at least one empty location. If the array is full, insertion cannot be possible. The target location where element is to be inserted is made empty by shifting elements downwards by one position and the newly inserted element is placed at that location.

5	7	9	10	12		
0	1	2	3	4	5	6

(a)

5	7		9	10	12	
0	1	2	3	4	5	6

(b)

5	7	3	9	10	12	
0	1	2	3	4	5	6

(c)

Insertion

There are two empty spaces available. Suppose we want to insert 3 in between 7 and 9. All the elements after 7 must be shifted towards end of the array. The entered number 3 can be assigned to that memory location. The above example describes the insertion of an element.

The following program illustrates the insertion operation

<pre>#include<stdio.h> #include<conio.h> #include<process.h> main() { int num[8]={0}, j, k=0, p, n; clrscr(); printf("\n Enter elements (0 to exit) : "); for(j=0; j<8; j++) { scanf("%d", &num[j]); if(num[j] == 0) break; } if(j<8) { printf("\n Enter position number and element :");</pre>	<pre>sacnf("%d %d", &p,&n); --p; } else while(num[k]!=0) k++; k--; for(j=k;j>=p;j--) num[j+1]=num[j]; num[p]=n; for(j=0;j<8;j++) printf("%d", num[j]); }</pre>
--	--

OUTPUT

```
Enter elements(0 to Exit): 1 2 3 4 5 6 0
Enter position number and element: 5 10
1 2 3 4 10 5 6 0
```

DELETION

Like insertion, deletion of an element can be done from the list. In deletion, the elements are moved upwards by one position.

<pre>#include<stdio.h> #include<conio.h> #include<process.h> main() int num[8]={0}, j, k=0, p, n; clrscr(); printf("\n Enter elements(0 to Exit) :"); for(j=0;j<8;j++) { scanf("%d", &num[j]); if(num[j] == 0) break; } printf("\n Enter an element to remove:");</pre>	<pre>scanf("%d", &n); while(num[k]!=n) k++; for(j=kl;j<7;j++) num[j]=num[j+1]; num[j]=0; for(j=0;j<8;j++) printf("%d", num[j]);</pre>
--	---

OUTPUT

```
Enter elements(0 to exit): 5 8 9 4 2 3 4 7
Enter element to remove: 9
5 8 4 2 3 4 7 0
```

Explanation:

In the above program, elements are entered by the user. The user can enter maximum eight elements.

The element which is to be removed is also entered. The while loop calculates the position of the element in the list. The second for loop shifts all the elements next to the specified element one position up towards the beginning of the array. The element which is to be deleted is replaced with successive elements.

SORTING

Sorting is a process in which records are arranged in ascending or descending order. The records of the list of these telephone holders are to be sorted by the name of the holder. By using this directory, we can find the telephone number of any subscriber easily. Sort method has great importance

<pre>#include<stdio.h> #include<conio.h> main() { int num{8}={0}; int k=0,h,a,n,tmp; clrscr(); printf("\nEnter numbers :"); for(a=0;a<8;++a)</pre>	<pre>scanf("%d",&num[a]); while(k<7) { for (h=k+1;h<8;++h) if (num[k]>num[h]) { tmp=num[k]; num[k]=num[h]; num[h]=tmp; } printf ("\n Sorted array:");</pre>	<pre>for (k=0;k<8;++k) printf("%d",num[k]); }</pre> <p>OUTPUT: Enter numbers:4 5 8 7 9 3 2 1 Sorted array: 1 2 3 4 5 7 8 9</p>
---	--	--

in data structures. Different sort methods are used in sorting elements/records.

MERGING LISTS

Merging is a process in which two lists are merged to form a new list. The new list is the sorted list. Before merging individual lists are sorted and then merging is done. Write a program to create two array lists with integers. Sort and store elements of both of them in the third list.

<pre># include<stdio.h> # include<conio.h> # include<math.h> main() { int m,n,p,sum=0; int listA[5],listB[5],listC[10]={0}; clrscr(); printf("\nEnter the elements for first list:"); for (m=0;m<5;m++) { scanf("%d",&listA[m]); if (listA[m]==0) m--; sum=sum+abs(listA[m]); } }</pre>	<pre>printf("\n Enter element for the second list:"); for (n=0;n<5;n++) { scanf("%d",&listB[n]); if(listA[n]==0) n--; sum=sum+abs(listB[n]); } p=n=m=0; m=m-sum; while (m<sum) { for (n=0;n<5;n++) { if (m==listA[n] m==listB[n]) listC[p++]=m; if (m==listA[n] &&</pre>	<pre>m==listB[n]) listC[p++]=m; } m++; } puts("Merged sorted list:"); for (n=0;n<10;++n) printf ("%d", listC[n]); }</pre> <p>OUTPUT: Enter elements for first list: 1 5 4 -3 2 Enter elements for second list: 9 5 1 2 10 Merged sorted list: -3 1 1 2 2 4 5 5 9 10</p>
--	---	---

LESSON 18 STACK

Outline

Introduction – Stack - Representation and terms – Operations – Insertion -
Deletion – Implementations

INTRODUCTION

There are two common data objects found in computer algorithm called stacks and queues. Data objects in an ordered list is $A(a_1, a_2, \dots, a_n)$ where $n \geq 0$ and a_i are the atoms taken from the set. If the list is empty or Null then $n=0$.

STACK

Stack is an array of size N , where N is an unsigned integer. It is an ordered list in which all insertions and deletions are made at one end called **TOP**.

REPRESENTATION AND TERMS

Storage: A function contains local variables and constants. These are stored in a stack. Only global variables in a stack frame.

Stack frames: This data structure holds all formal arguments, return address and local variables on the stack at the time when function is invoked.

TOP: The top of the stack indicates its door. The stack top is used to verify the stack's current position, that is, underflow and overflow. Some programmers refer to -1 assigned to top as initial value. This is because when an element is added the top is incremented and it would become zero. It is easy to count elements because the array element counting also begins from zero. Hence it is convenient to assign -1 to top as initial value.

Stack underflow: When there is no element in the stack or the stack holds elements less than its capacity, then this stack is known as stack underflow. In this situation, the TOP is present at the bottom of the stack. When an element is pushed, it will be the first element of the stack and top will be moved to one step upper.

Stack overflow: When the stack contains equal number of elements as per its capacity and no more elements can be added such status of stack is known as stack overflow. In such a position, the top rests at the highest position.

STACK OPERATIONS

CREATE (S)	←	Creates an empty stack
ADD (i,S)	←	Add i to rear of stack
DELETE (S)	←	Removes the top element from stack
FRONT (S)	←	Returns the top element of stack.
ISEMT(S)	←	returns true if stack is empty else false.

INSERTION OPERATION

```
CREATE ( ) := declare Stack(1:n) ,top<- 0
ISEMPTS (S) =If TOP=0 then true else false.
TOP(S) = if top= 0 then error else stack (top)
ADD & DELETE operations are implemented as,
```

Procedure ADD (item, stack,n,top)

```
If top >=n then call stack-Full
Top<- top + 1
Stack (top) <- item
End ADD.
```

DELETION OPERATION

```
Procedure DELETE (item, stack,top)
If top <=0 then call stack-empty
Item <- stack(top)
Top<- top-1
End delete.
```

IMPLEMENTATION OF A STACK

The stack implementation can be done in the following ways:

Static implementation: Static implementation can be achieved using arrays. Though, it is a very simple method, but it has few limitations. Once the size of an array is declared, its size cannot be altered during program execution. While in array declaration, memory is allocated equal to array size. The vacant space of stack (array) also occupies memory space. Thus, it is inefficient in terms of memory utilization. In both the cases if we store less argument than declared, memory is wasted and if we want to store more elements than declared, array could not expand. It is suitable only when we exactly know the number of elements to be stored.

Elements are entered stored in the stack .The element number that is to be deleted from the stack will be entered from keyboard. Recall that in a stack, before deleting any element, it is compulsory to delete all elements before that element. In this program the elements before a target element are replaced with value NULL(0). The elements after the target element are retained. Recall that when an element is inserted in the stack the operation is called 'push', When an element is deleted from the stack then the operation is pop.

The push() operation inserts an element into the stack. The next element pushed() is displayed after the first element and so on. The pop() operation removes the element from the stack. The last element inserted is deleted first.

LESSON 19 LINKED LIST

Outline

Introduction - Linked list with header - Linked list without header

INTRODUCTION

Data representation have the property that successive nodes of data object were stored at a fixed distance. Thus,

- (i) if element a_{ij} table stored at the location L_{ij}
- (ii) if the i th node in a queue at location L_i .
- (iii) if the top most node of the stack will be at the location L_t .

When a sequential mapping is used for ordered list, operations like insertion and deletion becomes expensive.

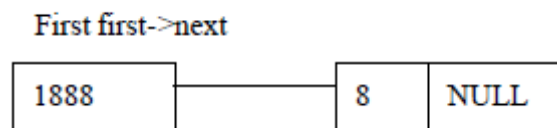
For eg., consider some English words. Three letter word ending with “AT” like BAT, CAT, EAT, FAT, HAT,.....,WAT. If we want to add GAT in the list we have to move BAT,CAT.....,WAT. If we want to remove the word LAT then delete many from the list. The problem is list having varying sizes, so sequential mapping is inadequate. By storing each list with maximum size storage space may be wasted. By maintaining the list in a single array large amount of data movement is needed. The data object polynomial are ordered by exponent while matrices ordered rows and columns. The alternative representation to minimize the time needed for insertion and deletion is achieved by linked representation.

LINKED LIST WITH HEADER

The following steps are used to create a linked list with header.

1. **Three pointers –header, first and rear** –are declared. The header pointer is initially initialized with NULL. For example, $\text{header} = \text{NULL}$, where header is pointer to structure. If it remains NULL it implies that the list has no element. Such a list is known as a NULL list or an empty list.
2. In the second step, memory is allocated for the first node of the linked list. For example, let the address of the first node be 1888. An integer, say 8, is stored in the variable num and the value of header is assigned to pointer next.

Header



Both header and rear are initialized the address of the first node.

The statement would be

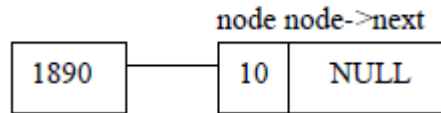
$\text{Header} = \text{first};$

$\text{Rear} = \text{first};$

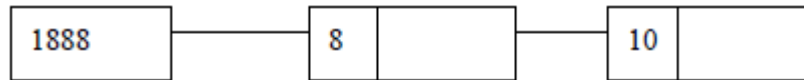
3. The address of pointer first is assigned to pointers header and rear. The rear is used to identify the end of the list and to detect the NULL pointer.

4. Again, create a memory location, suppose 1890, for the successive node.

node node->next



5. Join the element of 1890 by assigning the value of node rear->next. Move the rear pointer to the last node.



Consider the following statements

1. node->next=NULL;

The above statement assigns NULL to the pointer next to current node. If the user does not want to create more nodes, the linked list can be closed here.

2. rear->next=node;

The rear pointer keeps track of the last node, the address of current node(node) is assigned to link field of the previous node.

3. rear=node;

Before creating the next node, the address of the last created node is assigned to pointer rear, which is used for statement(2). In function main(), using while loop the elements of the linked list are displayed.

Header: The pointer header is very useful in the formation of a linked list. The address of first node(1888) is stored in the pointer header. The value of the header remains unchanged until it turns out to be NULL. The starting location of the list can only be determined by the pointer header.

```

While ( header!=NULL)
{
printf("%d", header->num);
header=header->next;
}
  
```

LINKED LIST WITHOUT HEADER

In the last topic we discussed how a linked list can be created using header. The creation of a linked list without header is same as that of a linked list with header. The difference in manipulation is that, in the linked list with header, the pointer header contains the address of the first node. In the without header list, pointer first itself is the starting of the linked list.

OPERATIONS OF A SINGLY LINKED LIST INSERTION

Insertion of an element in the linked list leads to several operations. The following steps are involved in inserting an element.

Creation of node: Before insertion, the node is created. Using malloc () function memory space is booked for the node.

Assignment of data: Once the node is created, data values are assigned to members.

Adjusting pointers: The insertion operation changes the sequence. Hence, according to the sequence, the address of the next element is assigned to the inserted node. The address of the current node (inserted) is assigned to the previous node.

The node can be inserted in the following positions in the list.

Insertion of the node at the starting: The created node is inserted before the first element. After insertion, the newly inserted element will be the first element of the linked list. In this insertion only the contents of the successive node's pointer are changed.

Insertion at the end of the list: A new element is appended at the end of the list. This is easy as compared to other two (a) and (c) operations. In this insertion only the contents of the previous node's pointer are changed.

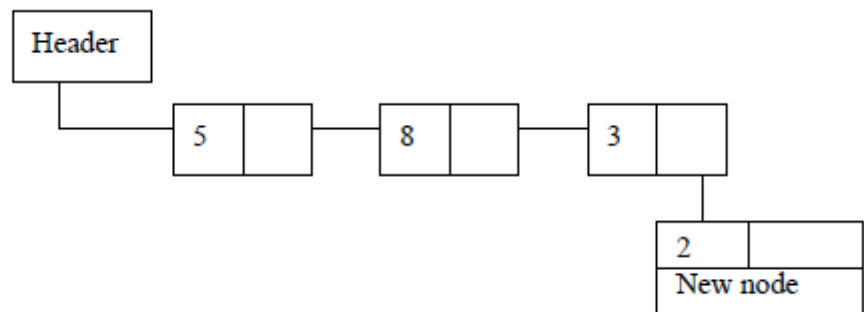
Insertion at the given position in the list: In this operation, the user has to enter the position number. The given pointer is counted and the element is inserted. In this insertion contents of both the previous and next pointers are altered.

Insertion of the Node at the starting

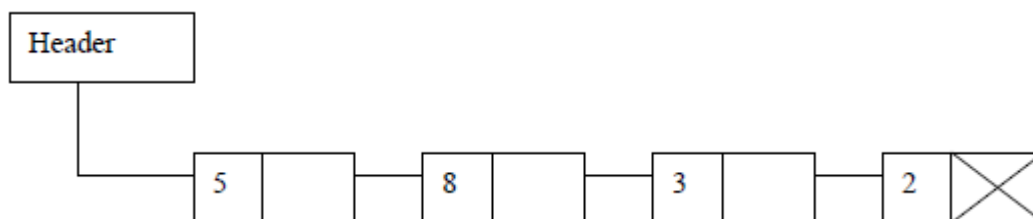
Inserting an element at the beginning involves updating links between link fields of two pointers. After insertion of new node, the previously existing nodes are shifted ahead. The new node, which is to be inserted, is formed and the arrow indicates the position where it will be inserted. After insertion, the new node will be the first node and its link field points to the second element, which was previously the first element.

Insertion of the Node at the end

A new element is inserted or appended at the end of the existing linked list. The address of the newly created node is linked to the previous node that is NULL. The new node link field is assigned a NULL value.



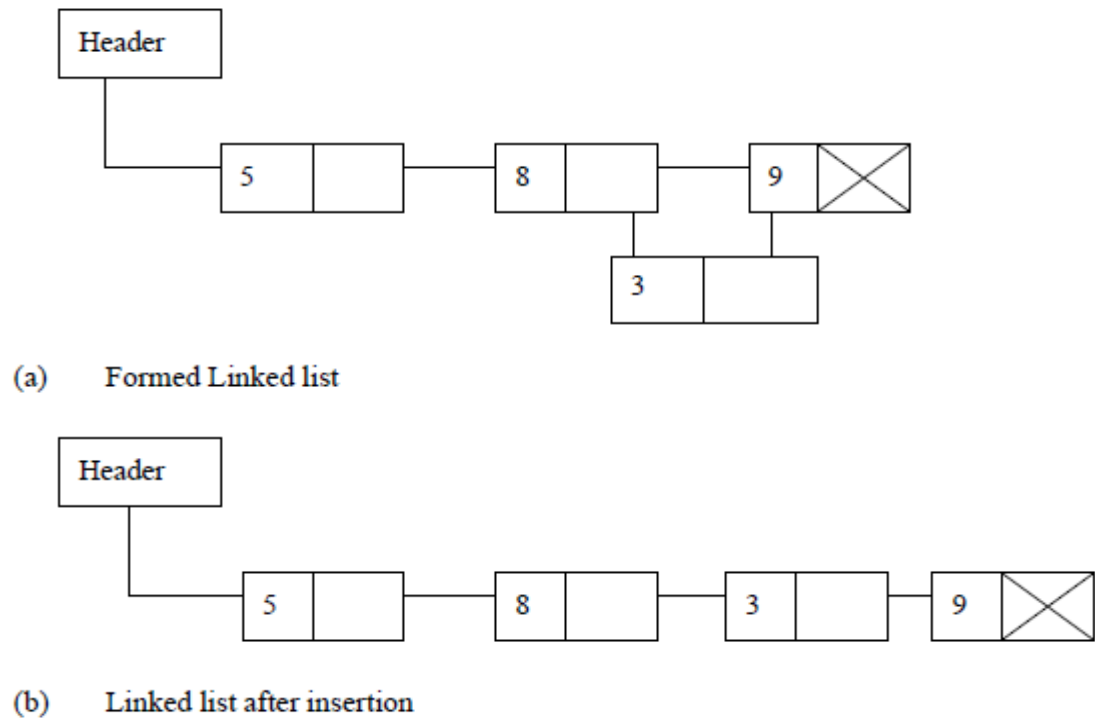
(a) Before insertion



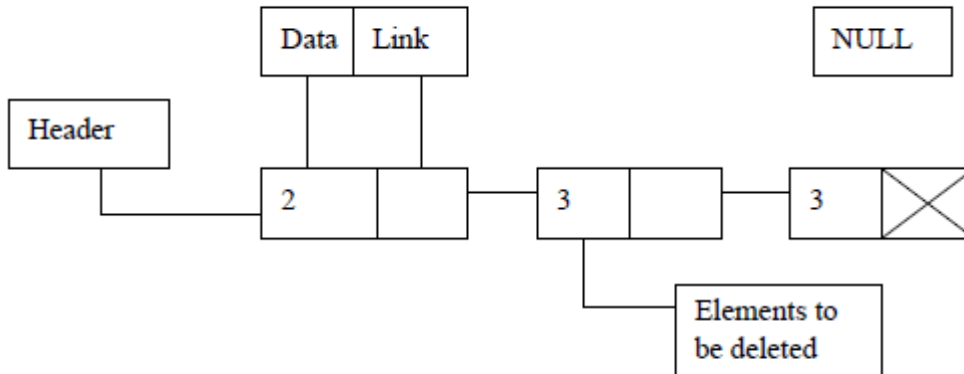
(b) After insertion

Insertion of a Node at a given position

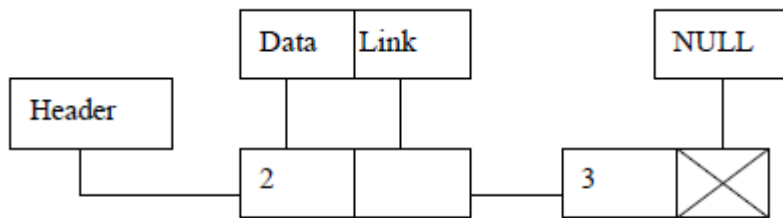
Insertion of a node can be done at a specific position in the linked list. The following figure explain the insertion of a node at the specific position in the linked list. Suppose we want to insert a node at the third position, then



DELETION



(a) Deletion from the linked list



(b) Linked list after deletion of element 3

In figure (a) linked list elements are shown. The element 3 is to be deleted. After deletion the linked list would be shown as in figure(b).

Deleting a node from the list has the following three cases.

1. Deleting the first node
2. Deleting the last node
3. Deleting the specific node

While deleting the node, the node which is to be deleted is searched first. If the first node is to be deleted, then the second node is assigned to the header. If the last node is to be deleted, the last but one node is accessed and its link field is assigned a NULL value. If the node which is to be deleted is in between the linked list, then the predecessor and successor nodes of the specified node are accessed and linked.

LESSON 20 DOUBLY LINKED LIST

Outline

Introduction - Doubly linked list - Insertion operation - Deletion operation -
Difference between single and doubly linked list

INTRODUCTION

The doubly linked list has two link fields one linking in forward direction and another one in backward direction.

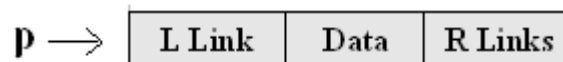
DOUBLY LINKED LIST

If we point to a specific node say P, we can move only in the direction of links. The only way to find which precedes P is to start back at the beginning. The same is followed to delete the node. The problem is to which direction the link is to be moved. In such cases doubly linked list is used. It has two link fields one linking in forward direction and another one in backward direction.

It has at least three fields as follows:

L Link	Data	R Links
---------------	-------------	----------------

It has one special node called head node. It may or may not be **Circular**. An empty list is not really empty since it will always have its head node.



Insert algorithm:

```

Procedure DINSERT (P,X)
    (insert node p to right of node x)
    LLINK (P) ← X
    RLINK (P) ← RLINK (X) (set LLINK & RLINK of nodes)
    LLINK (RLINK (X)) ← P
End Dinsert
    
```

Delete algorithm:

```

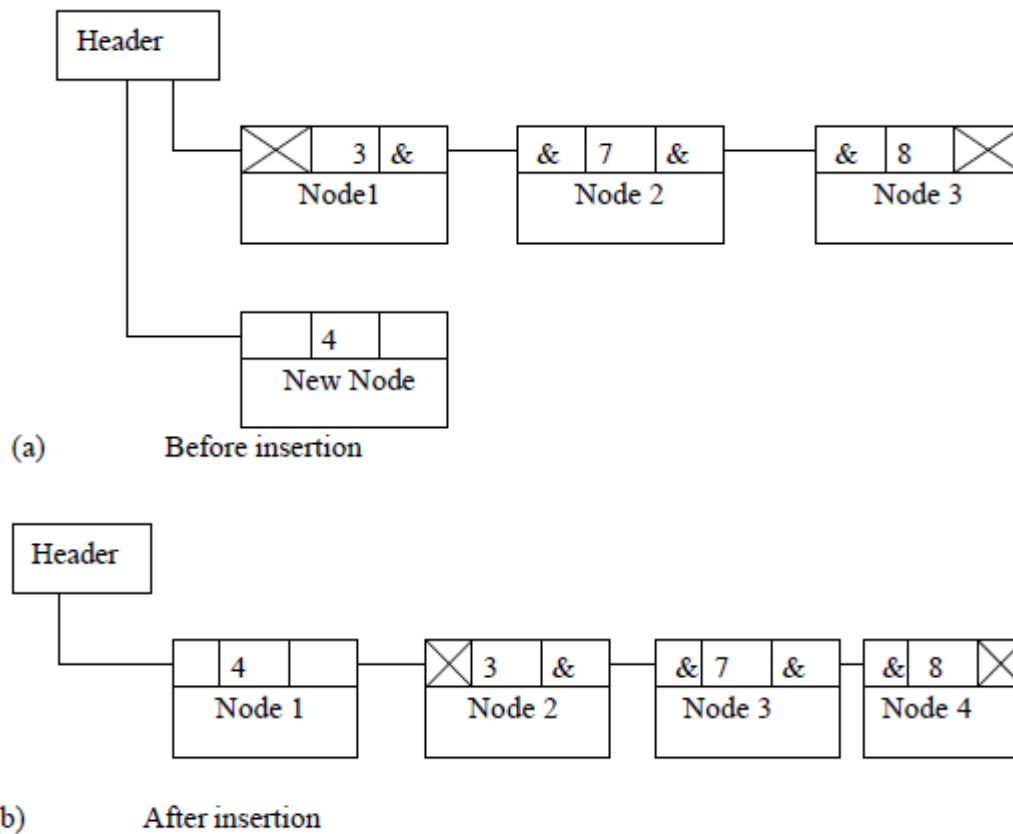
Procedure DDELETE (X,L)
    If X=L then no more nodes (L ← list)
    RLINK ( LLINK (X) ) ← RLINK (X)
    LLINK ( RLINK(X) ) ← LLINK(X)
    Call RET (x)
End DDelete
    
```

INSERTION

Insertion of an element in the doubly linked list can be done in three ways.

1. Insertion at the beginning
2. Insertion at the specified location
3. Insertion at the end

Insertion at the beginning: The process involves creation of a new node, inputting data and adjusting the pointers.



Insertion at the end or a specified position: An element can be inserted at a specific position in the linked list. The following program can be used to insert an element at a specific position.

```
int addafter(int n, int p)
{
    struct doubly *temp, *t;
    int j;
    t= first;
    for(j=0;j<p-1;j++)
    {
        t=t-next;
        if(t==NULL)
        {
            printf("\n There are less than %d element:",p);
            return 0;
        }
    }
}
```



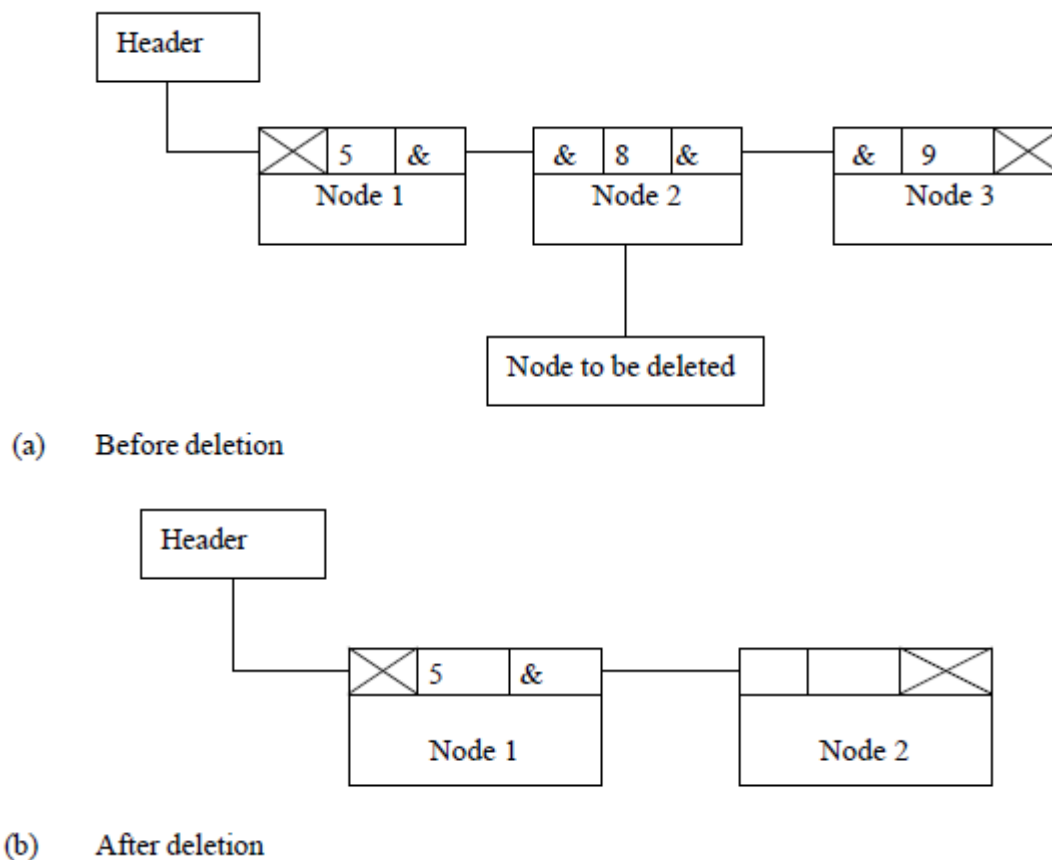
```

}
temp=(struct doubly*) malloc(sizeof(struct doubly));
temp->next=t->next;
temp->number=n;
t->next=temp;
return 0;
}

```

DELETION

An element can be removed from the linked list by an operation called deletion. Deletion can be done at the beginning or at a specified position.



In the deletion process the memory of the node to be deleted is released using `free()` function. The previous and next node are linked.

DIFFERENCE BETWEEN SINGLE AND DOUBLY LINKED LIST

Single linked list

- Only one link is available to show the next node
- There is no head-node
- Circular links are not available
- Insertion and deletion are expensive
- More space is wasted due to unusage of memory allocation

Doubly linked list

- Two links are available to show the node as LLINK and RLINK
 - It has a special node called as Head-node
 - Circular links are possible with the help of the two links
 - Insertion and deletion are easily possible
 - Empty space is not present because in empty space the head node is present
-

LESSON 21

QUEUES, SEARCHING AND SORTING -

BINARY SEARCH

Outline

Introduction - Queues - Operations of a queue - Searching - Linear Searching -
Binary search Algorithm

INTRODUCTION

Queue is used for the application of job scheduling. It is used in batch processing .when the jobs are queued –up and executed one after the other they were received.

QUEUES

Queues ignore the existence of priority. There are two different ends called FRONT and REAR end. Deletions are made from the FRONT end. If job is submitted for execution it joins at the REAR end. job at the front of queue is next to be executed.

Queue is an common data objects found in computer algorithm. In queue the insertions are made at one end called rear and the deletion at one end called front. Queue follows the FIFO basis (**First in First Out**).

Deletions are made from the Front end and joins at the REAR end. Job at the FRONT of queue is next to be executed. Some operations performed on queues are:

CREATE Q (Q)	←	Creates an empty queue
ADD (i,Q)	←	Add i to rear of queue
DELETE (Q)	←	Removes the front element
FRONT (Q)	←	Returns the front element of queue.
ISEMTQ	←	returns true if queue is empty else false.

OPERATIONS OF A QUEUE

Add procedure on a queue:

```
Procedure ADDQ (item , Q, n, rear)
    If rear = n then Queue- full
    Rear <- rear +1
    Q(rear) <- item
end ADDQ.
```

Delete Procedure of Queue:

```
Procedure DELETE Q(item, Q, Front)
    If front = rear then queue –empty
    Front <- front +1
    Item <- Q(front)
End DeleteQ
```

SEARCHING

The searching of an item starts from the first item and goes up to the last item until the expected item is found. An element that is to be searched is checked in the entire data structure in a sequential way from starting to end. Sorting is a term used to arrange the data items in an order.

The searching methods are classified into two types as Linear search and Binary search methods.

LINEAR SEARCH

The linear search is a usual method of searching data. The linear search is a sequential search. It is the simple and the easiest searching method. An element that is to be searched is checked in the entire data structure in a sequential way from starting to end. Hence, it is called linear search. Though, it is straight forward, it has serious limitations. It consumes more time and reduces the retrieval rate of the system. The linear or sequential name implies that the items are stored in a systematic manner. The linear search can be applied on sorted or unsorted linear data structure.

The number of iterations for searching depends on the location of an item. If it were located at first position then the number of iteration required would be 1. Here, least iterations would be needed hence, this comes under the best case. In case the item to be searched is observed somewhere at the middle then the number of iterations would be approximately $N/2$, where N is the total number of iterations. This comes under average number of iterations. In the worst case to search an item in a list, N iterations would be required provided the expected item is at the end of the list.

Next to linear and sequential search the better known methods for searching is binary search. This search begins by examining the record in the middle of file rather at any one end. It takes $O(\log n)$ time for the search. It begins by examining the record in the middle of file rather at any one end. File being searched is ordered by non-decrementing values of the key.

Based on the result of comparison with the middle key k_m one of the following conclusion is obtained.

- a. If $k < k_m$ record being searched in lower half of the file.
- b. If $k = k_m$ is the record which is being searched for.
- c. If $k > k_m$ records begin searched in upper half of the file.

After each comparison either it terminates successfully or the size of the file searching is reduced to one half of the original size. $O(\log_2 n)$. The file be examined at most $[(n/2)^k]$ where n is the number of records, k is key. In worst case this method requires $O(\log n)$ comparisons.

Always in binary search method the key in the middle of subfile is currently examined. Suppose there are 31 records. The first tested is k_{16} because $[(1+31)/2]=16$. If k is less than k_{16} then k_8 is tested next because $[(1+15)/2]=8$ or if $k > k_{16}$ then k_{24} is tested & it is repeated until the desired record is reached.

BINARY SEARCH ALGORITHM

Procedure BINSRCH

//search an ordered sequential file.F with records $R_1 \dots R_n$ and the keys

$K_1 \leq k_2 \leq \dots \leq k_n$ for a record R_i such that $k_i = k$;

$i=0$ if there is no such record else $k_i = k$

Throughout the algorithm, l is the smallest index such that k_l may be k and u the largest index such that k_u may be k //

$l \leftarrow -1; u \leftarrow n$

while $l \leq u$ do

$m \leftarrow \lfloor (l+u)/2 \rfloor$ //compute index of middle record//

case:

: $k > k_m: l \leftarrow m+1$ //look in upper half//

: $k = k_m: l \leftarrow m$; return

: $k < k_m: u \leftarrow m-1$ //look in lower half//

end

end

$I \leftarrow 0$ //no record with key k //

end BINSRCH

LESSON 22 SORTING

Outline

Introduction – Sorting - Comparison with other method

INTRODUCTION

Sorting is an important phenomenon in the programming domain. If large volumes of records are available, then sorting of them becomes very crucial. If the records are sorted either in ascending or descending order based on keys, then searching becomes easy.

SORTING

Sorting is a process in which records are arranged in ascending or descending order. In real life we come across several examples of such sorted information. For example, in a telephone directory the names of the subscribers and their phone numbers are written in ascending alphabets. The records of the list of these telephone holders are to be sorted by their names. By using this directory, we can find the telephone number and address of the subscriber very easily. The sort method has great impact on data structures in our daily life.

For example, consider the five numbers 5,9,7,4,1.

The above numbers can be sorted in ascending or descending order.

The representations of these numbers in

Ascending order (0 to n): 1 4 5 7 9

Descending order (n to 0): 9 7 5 4 1

Similarly, alphabets can be sorted as given below.

Consider the alphabets B, A, D, C, E. These are sorted in

Ascending order (A to Z): A B C D E

Descending order (Z to A): E D C B A.

INSERTION SORT

In insertion sort an element is inserted at the appropriate place. Here the records must be sorted from R_1, R_2, \dots, R_n in non-decreasing value of the key k . Assume n is always greater than 1.

Algorithm:

Procedure INSORT(R,N)	ordered sequence is also ordered on key K .
$K_0 \leftarrow -\alpha$	Assume that R_0 is a dummy record such
For $I \leftarrow 2$ TO N DO	that $K \geq K_0$)
$T \leftarrow R_I$	$j \leftarrow I$
Call INSERT (T, $I-1$)	while $K < k_j$ do
END {for}	$R_{j+1} \leftarrow R_j$
END {INSORT}	$J \leftarrow j-1$
Procedure INSERT (R,I)	END {while}
(Insert record R with key K into the ordered	$R_{j+1} \leftarrow R$
sequence such that resulting	End {insert}

case 1:

```

K0 ← -α
for j ← 2 to 6 do
  T ← R2
  Call Insert (R2,1)
  Procedure insert (R,I)
  j←I i←1 (i.e.) j=1
  while K<kj do
  3<5 (True)
  R2←R1
  j←1-1 = 0 (ie.-j=0)
  end {while}
  R0+1←R
  R1←R
  R1 ← 3 (ie.R1=3)

```

SELECTION SORT

The selection sort is nearly same as exchange sort. Assume we have a list containing n elements. By applying selection sort, the first element is compared with remaining $(n-1)$ elements. The smallest element is placed at the first location. Again, the second element compared with the remaining $(n-2)$ elements. If the item found is lesser than the compared elements in the remaining $n-2$ list then the swap operation is done. In this type, the entire array is checked for the smallest element and then swapped.

In each pass, one element is sorted and kept at the left. Initially the elements are temporarily sorted and after next pass, they are permanently sorted and kept at the left. Permanently sorted elements are covered with squares and temporarily sorted with encircles. Element inside the circle 'O' is chosen for comparing with the other elements marked in a circle and sorted temporarily. Sorted elements inside the square 'y' are shown.

Time Complexity

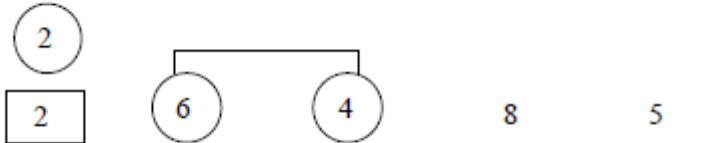
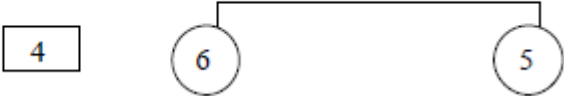
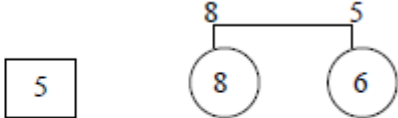


The performance of sorting algorithm depends upon the number of iterations and time to compare them. The first element is compared with the remaining $n-1$ elements in pass 1. Then $n-2$ elements are taken in pass 2. This process is repeated until the last element is encountered. The mathematical expression for these iterations will be equal to $(n-1) + (n-2) + \dots + (n-(n-1))$. Thus the expression becomes $n*(n-1) / 2$. Thus the number of comparisons is proportional to (n^2) . The time complexity of selection sort is $O(n^2)$.

COMPARISON WITH OTHER METHODS

1. This method requires more number of comparisons than quick sort and tree sort.
2. It is easier to implement than other methods such as quick sort and tree sort. The performance of the selection sort is quicker than bubble sort.

Consider the elements 2, 6, 4, 8 and 5 for sorting under selection sort method.

1. In pass 1, select element 2 and check it with the remaining elements 6, 4, 8 and 5. There is no smaller value than 2, hence the element 2 is placed at the first position.
2. Now, select element 6 for comparing with remaining (n-2) elements i.e., 4, 8, and 5. The smaller element is 4 than 6. Hence we swap 4 and 6 and 4 is placed after 2.
3. Again we select the next element 6 and compare it with the other two elements 8, 5 and swapping is done between 6 and 5.
4. In the next pass element 8 is compared with the remaining one element 6. In this case, 8 is swapped with 6.
5. In the last pass the 8 is placed at last because it is highest number in the list.

Pass	No. 1	2	3	4	5
1					
2					
3					
4	2				
5	2	4	5		
Sorted No.	2	4	5	6	8

LESSON 23 BUBBLE & QUICK SORTING

Outline

Introduction - Bubble sort: Time complexity - Quick sort: Time complexity

INTRODUCTION

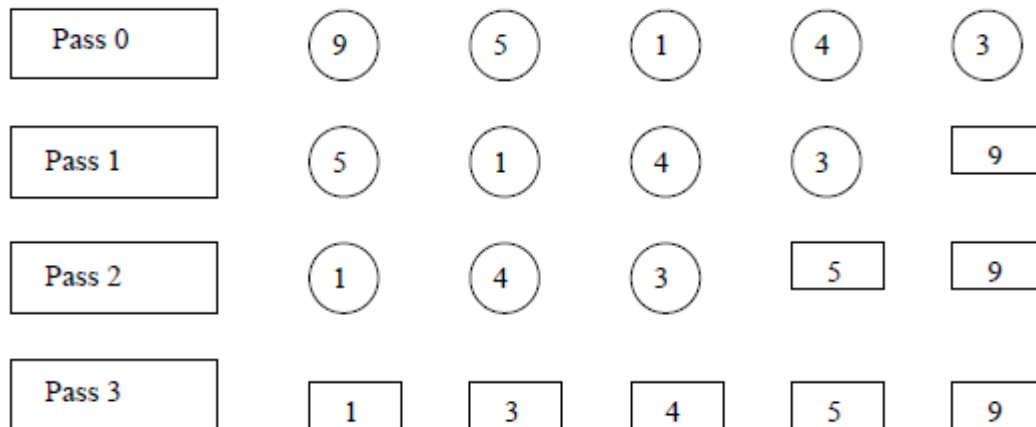
Bubble sort is a commonly used sorting algorithm and it is easy to understand. In this type, two successive elements are compared and swapping is done if the first element is greater than the second one. The elements are sorted in ascending order. Though it is easy to understand, it is time consuming. The quick sort method works by dividing into two partitions.

BUBBLE SORT

The bubble sort is an example of exchange sort. In this method comparison is performed repetitively between the two successive elements and if essential swapping of elements is done. Thus, step-by-step the entire array elements are checked. It is different from the selection sort. Instead of searching the minimum element and then applying swapping, two records are swapped instantly upon noticing that they are not in order.

Let us consider the elements 9, 5, 1, 4 and 3 for sorting under bubble sort.

1. In pass 1, first element 9 is compared with its next element 5. The next element is smaller than the 9. Hence, it is swapped. Now the list is 5, 9, 1, 4, 3 again the 9 is compared with its next element 1 the next element is smaller than the 9 hence swapping is done. The same procedure is done with the 4 and 3 and at last we get the list as 5, 1, 4, 3, 9.
2. In pass 2, the elements of pass 1 are compared. The first element 5 is compared with its next element 1. 5 and 1 are swapped because 1 is smaller than 5. Now the list becomes 1, 5, 4, 3, 9. Next, 5 is compared with element 4. Again, the 5 and 4 are swapped. This process is repeated until all successive elements are compared and if the succeeding number is smaller than the present number then the numbers are swapped. The final list at the end of this pass is 1, 4, 3, 5, 9.
3. In pass 3, the first element 1 is compared with the next element 4. The element 4 is having the higher value than the first element 1, hence they remain at their original positions. Next 4 is compared with the subsequent element 3 and swapped due to smaller value of 3 than 4.
4. At last, the sorted list obtained is as 1, 3, 4, 5, 9.



TIME COMPLEXITY

The performance of bubble sort in worst case is $n(n-1)/2$. This is because in the first pass $n-1$ comparisons or exchanges are made; in the second pass $n-2$ comparisons are made. This is carried out until the last exchange is made. The mathematical representation for these exchanges will be equal to $(n-1) + (n-2) + \dots + (n(n-1))$. Thus the expression becomes $n*(n-1)/2$.

Thus, the number of comparisons is proportional to (n^2) . The time complexity of bubble sort is $O(n^2)$.

QUICK SORT

It is also known as partition exchange sort. It was invented by C A R Hoare. It is based on partition. Hence, the method falls under divide and conquer technique. The main list of element is divided into two sub-lists. For example, suppose lists of X elements are to be sorted. The quick sort marks an element in a list called as pivot or key. Consider the first element J as a pivot. Shift all the elements whose value is less than J towards the left and elements whose value is greater than J to right of J . Now, the key element divides the main list in to two parts. It is not necessary that selected key element must be at middle. Any element from the list can act as key element. However, for best performance is given to middle elements. Time consumption of the quick sort depends on the location of the key in the list.

Consider the following example in which five elements 8, 9, 7, 6, 4 are to be sorted using quick sort.

Consider pass 1 under which the following iterations are made. Similar operations are done in pass 2, pass 3, etc.

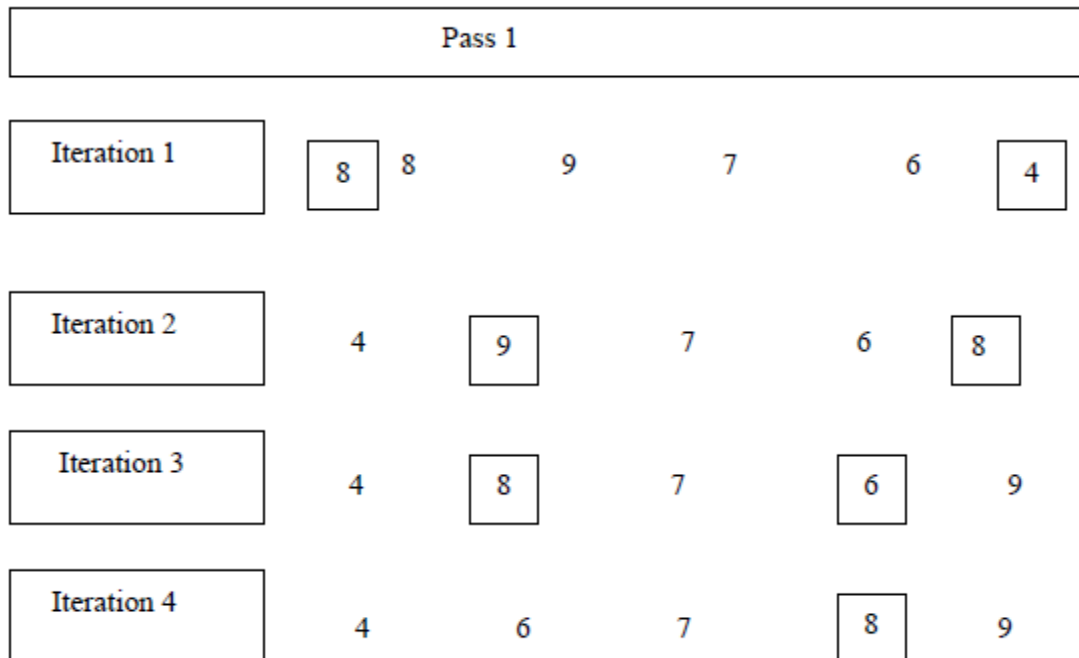
In iteration 1 the first element 8 is marked as pivot one. It is compared with the last element whose value is 4. Here 4 is smaller than 8 hence the number are swapped. Iteration 2 shows the swapping operation.

In the iteration 3 and 4, the position of 8 is fixed. In iteration 2, 8 and 9 are compared and swapping is done after Iteration 2.

In iteration 3, 8 and 6 are compared and necessary swapping is done. After this, it is impossible to swap. Hence, the position of 8 is fixed. Because of fixing the position of 8 the main list is divided into two parts. Towards left of 8 elements having smaller than 8 are placed and towards the right greater than 8 are placed.

Towards the right of 8 only one element is present hence there is no need of further swapping. This may be considered under Pass2.

However towards the left of 8 there are three elements and these elements are to be swapped as per the procedure described above. This may be considered under Pass3.



TIME COMPLEXITY

The efficiency of quick sort depends upon the selection of pivot. The pivot can bifurcate the list into compartments. Sometimes, the compartments may have the same sizes or dissimilar sizes. Assume a case in which pivot is at middle. Thus, the total elements towards left of it and right are equal. We can express the size of list with the power of 2. The mathematical representation for the size is $n=2^s$. The value of s can be calculated as $\log_2 n$.

After the first pass is completed there will be two compartments having equal number of elements that is, $n/2$ elements are on the left side as well as right side. In the subsequent pass, four portions are made and each portion contains $n/4$ elements. In this way, we will be proceeding further until the list is sorted. The number of comparisons in different passes will be as follows.

Pass 1 will have n comparisons. Pass 2 will have $2*(n/2)$ comparisons. In the subsequent passes will have $4*(n/4)$, $8*(n/8)$ comparisons and so on. The total comparisons involved in this case would be $O(n)+O(n)+O(n)+\dots+s$.

The value of expression will be $O(n \log n)$. Thus time complexity of quick sort is $O(n \log n)$.

LESSON 24 TREE SORT & HEAP SORT

Outline

Introduction - Tree sort - Heap sort

INTRODUCTION

If the nodes in the binary tree are in specific prearranged order, then heap sorting method can be used. A **Heap** is defined to be a complete binary tree with a property that the value of each node is at least as large as the value of its children nodes.

TREE SORT

In binary tree, we know that the elements are inserted according to the value greater or less in between node and the root in traversing. If the value is less than traversing node then, insertion is done at left side. If the value is greater than traversing node, it is inserted at the right side. The elements of such a tree can be sorted. This sorting involves creation of binary tree and then in order traversing is done.

HEAP SORT

In heap sort, we use the binary tree, in which the nodes are arranged in specific prearranged order. The rule prescribes that each node should have bigger value than its child node. The following steps are to be followed in heap sort.

1. Arrange the nodes in the binary tree form.
2. Node should be arranged as per specific rules.
3. If the inserted node is bigger than its parent node then replace the node.
4. If the inserted node is lesser than its parent node then do not change the position.
5. Do not allow entering nodes on right side until the child nodes of the left are fulfilled.
6. Do not allow entering nodes on left side until the child nodes of the right are fulfilled.
7. The procedure from step 3 to step 6 is repeated for each entry of the node.

Consider the numbers 56, 23, 10, 99, 6, 19, 45, 45, 23 for sorting using heap. Sorting process is shown in the below figure.

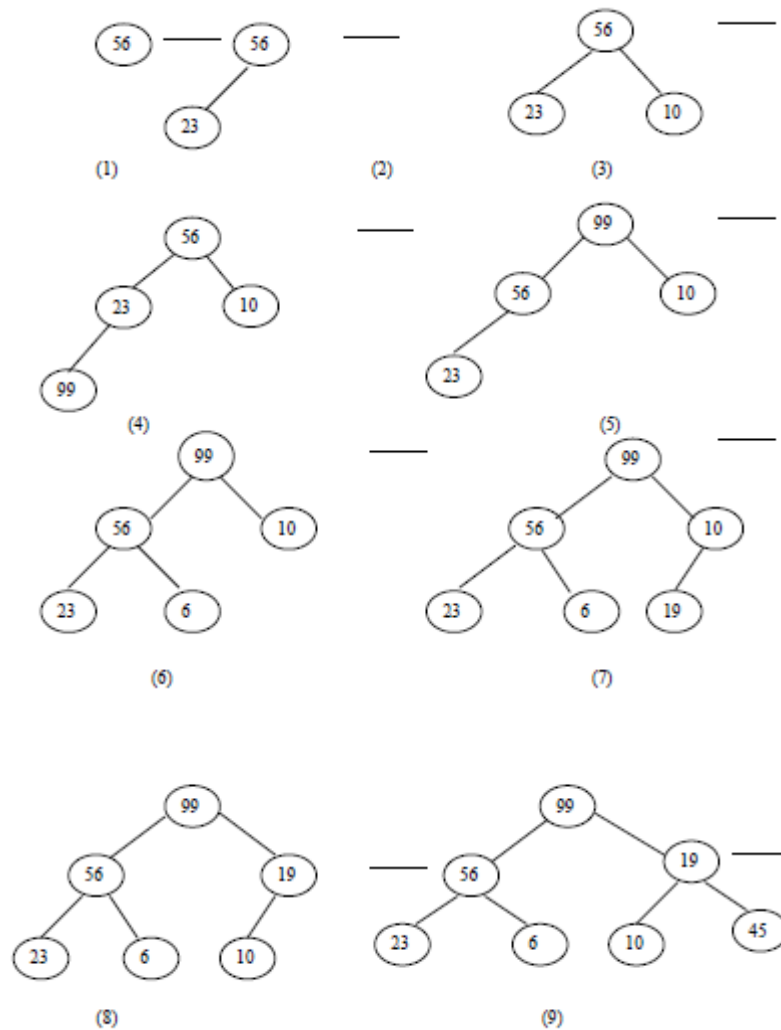
1. At first, we pick up the first element 56 from the list. Make it as the root node.
2. Next, take the second element 23 from the list. Insert this to the left of the root node 56. Refer to Fig. 16.7(2).
3. Then take the third element 10 from the list for insertion. Insert it to the right of the root node.
4. Take the fourth element 99. Insert it to the left side of node 23. The inserted element is greater than the parent element hence swap 99 with 23. But the parent node 56 is smaller than the child node 99 hence 99 and 56 are swapped. After swapping 99 becomes the root node.
5. Consider the next element 6 to insert in the tree. Insert it at the left side. There exists a left node hence insert it to the right of 56.
6. Element 19 is inserted to the right side of 99 because the left side gets full. Insert the element

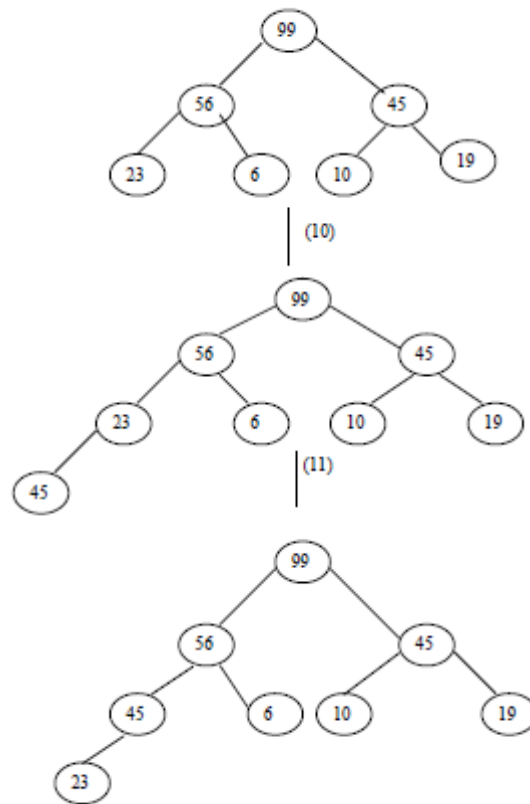
19 to the right side of node 10. However, the parent element is lesser than the child hence swap 19 with 10.

7. Now element 45 is to be inserted at the right side of 19. However, the parent element is having value lesser than the child element hence swap 45 with 19.

8. Now the right side is fully filled hence add the next element 45 to the left. The element 45 is inserted at the last node of left i.e., 23. However, the parent element is having value lesser than the child element hence swap 45 with 23.

9. Insert the last element 23 to the left side node i.e. 45. The left of 45 is already filled hence insert element 23 at the right of 45. 10. At last, we get a sorted heap tree.





:: REFERENCES

- Programming and Data Structures by Ashok N Kamthane – Pearson Education, First Indian Print 2004, ISBDN 81-297-0327-0.
- Programming in ANSI C by E. Balagurusamy, Tata McGraw-Hill, 1998.
- Data Structure using C by Aaron M Tanenbaum, Yedidyeh langsam, Moshe J Augenstein - PHI PUB.
- Fundamentals of Data Structure by Ellis Horowitz and Sartaj Sahni - Galgotia Book Source.
- The C programming language By Brian W. Kernighan, Dennis M. Ritchie
- C programming FAQs: frequently asked questions By Steve Summit
- Expert C programming: deep C secrets By Peter Van der Linden
- C programming: the essentials for engineers and scientists By David R. Brooks
- C programming By Larry Edward Ullman, Marc Liyanage
- C programming for the absolute beginner: the fun way to learn programming By Michael A. Vine
- C programming guide By Jack Jay Purdum
- Advanced Data Structures by Peter Brass
- Data Structures and Algorithms by Alfred V. Aho, Jeffrey D. Ullman, and John E. Hopcroft
- Data Structures and Their Algorithms by Harry R. Lewis and Larry Denenberg
- Data Structures Using C by Aaron M. Tenenbaum
- Schaum's outline of theory and problems of programming with C By Byron S. Gottfried
- Advanced C Struct Programming: Data Structure Design and Implementation in C by John W. L. Ogilvie
- Data Structures: A Pseudocode Approach With C by Richard F. Gilberg and Behrouz A. Forouzan
- Data Structures Using C by Aaron M. Tenenbaum
- C Programming with Data Structures by T. Sudha and B. Poornima
- Practical Data Structures Using C/C++ by James L. Antonakos and Kenneth C. Mansfield
- An Introduction to Data Structures and Algorithms (Progress in Theoretical Computer Science) by J.A. Storer and John C. Cherniavsky
- Data Structures and Program Design in C++ by Robert L. Kruse and Alex Ryba
- Data Structures and Abstraction Using C by Geoff Whale
- DATA-STRUCTURES AND PROGRAMMING by Malcolm C. Harrison
- Algorithms and Data Structures: An Approach in C by Charles F. Bowman
- Data Structures and C. Programmes by Christopher J. Van Wyk
- Data Structure for C Programming by Kumar Ajay

:: APPENDICES

BHARATHIAR UNIVERSITY B.Sc. COM.SCI DEGREE COURSE SYLLABUS CORE 3 : DATA STRUCTURES AND C PROGRAMMING

Subject Description:

This subject deals with the methods of data structures using C programming language.

Goal: To learn about C programming language using data structural concepts.

Objective:

On successful completion of this subject the students should have writing programming ability on data structures dealing with Stacks, Queues, List, Searching and Sorting algorithms etc.,

UNIT I:

Programming development methodologies Programming style Problem solving techniques: Algorithm, Flowchart, Pseudocode - Structure of a C program C character set Delimiters Keywords Identifiers Constants Variables Rules for defining variables Data types Declaring and initializing variables Type conversion. Operators and Expressions Formatted and Unformatted I/O functions Decision statements Loop control statements.

UNIT II:

Arrays String and its standard functions. Pointers Functions Preprocessor directives: #define, #include, #ifndef, Predefined macros.

UNIT III:

Structure and Union: Features of structure, Declaration and initialization of structure, Structure within structure, Array of structure, Pointer to structure, Bit fields, Enumerated data types, Union. Files: Streams and file types, Steps for file operation, File I/O, Structures read and write, other file functions, Command line arguments, I/O redirection.

UNIT IV:

Linear data structures: Introduction to data structures List: Implementations, Traversal, Searching and retrieving an element, Predecessor and Successor, Insertion, Deletion, Sorting, Merging lists Stack: Representation, Terms, Operations on stack, Implementation. Single linked list, Linked list with and without header, Insertion, Deletion, Double linked list Queues: Various positions of queue, Representation

UNIT V:

Searching and Sorting Searching: Linear, Binary. Sorting Insertion, Selection, Bubble, Quick, Tree, Heap.

BHARATHIAR UNIVERSITY MODEL QUESTION PAPERS

APR - 2009 Semester -[2008 - Data Structures and C Programming (Semester 2)]

Answer all the questions.

Max.Marks: 75

section A (10x1=Marks)

1. C has _____ facility.
a. OOPS b. Pointer c. Set d. None
2. Number of bytes for 'int' is _____.
a. 4 b. 2 c. 1 d. none
3. Strcmp("raja", "rani") returns _____ value.
a. zero b. negative c. positive d. none
4. _____ compares atmost n characters of string x to string y.
a. strncmp(x,y) b. strcmp(x,y,n) c. strcat(x,y) d. none
5. _____ opens file for reading and writing.
a. r+ b. r c. w d. a
6. 'arg c' and 'arg r' are _____.
a. reserved words b. user defined words c. built-in-functions d. none
7. Stack follows _____.
a. FIFO b. LILO c. FILO d. FCFS
8. 'PUSH' and 'POP' are special terms used in _____.
a. QUEUE b. TREE c. STACK d. None
9. Adjacent elements are compared in _____ sort.
a. selection b. insertion c. bubble d. none
10. Best sort is _____ sort.
a. selection b. heap c. quick d. none

Section B (5X5=25 Marks)

11. a. Write a C program to raise a base 'a' to power 'n'. Or
b. Explain different if statements available in 'C'.
12. a. Write a program to read a name and to reverse it. Or
b. Write a function to exchange the values of two integer variables.
13. a. Explain enumerated data types with an example. Or
b. Write a program to create a structure of two fields 'Roll Number' and 'Mark' and to display it. 14. a.
Write an algorithm to display a linked list P. Or
b. Write deletion algorithm in queue.
14. a. Write an algorithm to display a linked list P. Or
b. Write deletion algorithm in queue.

15. a. Write an algorithm for sequential search in an array. Or
b. Apply insertion sort for the following data. Give order of data pass by pass. 4, 2, 1, 5, 3.

section C (5X8=40Marks)

16. a. Explain different loop statements available in C with examples. Or
b. Write a program to read an integer and to find the sum of digits.
17. a. Write a program to read to sort n names. Or
b. Write a program with a function to find the length of the string using pointers.
18. a. Discuss various file I/O functions. Or
b. Write a program to create a pay roll file and to calculate the net pay of employee using sequential file handling.
19. a.
i. Write an algorithm to delete a last node of linked list P.
ii. Write an algorithm to add a new node as a last node of linked list P. Or
b. Write insertion and deletion algorithms in a STACK.
20. a. Write an algorithm for binary search. Explain it with an example. Or
b. Write an algorithm for quick sort. Explain it with an example.

APR - 2009 Semester -[2007 - Data Structures and C Programming (Semester 2)]

Answer all the questions.

Max.Marks: 100

section A (10x1=10Marks)

1. Define the term flow chart.
2. List the data types in C.
3. What is meant by an array?
4. Name any two preprocessor directives.
5. Give the difference between structures and unions.
6. Mention the use of command line arguments.
7. Name the linear data structures.
8. What is a stack?
9. Give the advantage of binary searching.
10. List any one representation of a queue.

section B (5x6=30Marks)

11. a. Explain the structure of a C Program. Or
b. Write a C program to find the sum of the digits of an integer number.
12. a. Discuss about functions in C. Or
b. Write a C program to find the sum of the n given numbers using pointer.
13. a. Write about enumerated data types. Or
b. Discuss on I/O redirection.
14. a. Write about inserting an element into the stack. Or
b. Discuss on deleting an element in a single linked list.
15. a. Explain binary search algorithm. Or
b. Sort the following numbers using selection sort.
81, 17, 12, 32, 45

section C (5X12=60Marks)

16. a. Discuss the following:
 - i. Data types (3 marks)
 - ii. Declaring and initializing variables (3 marks)
 - iii. Type conversion (4 marks)
 - iv. Formatted I/O (2 marks) Or
b. Write a C program to find the sum of even numbers from 2 to 100.
17. a. Explain about arrays in C. Or
b. Write a C program to find whether a given number is available in a given set of numbers (or) not.
18. a. Explain the following:

- i. Pointer to structures (6 marks)
 - ii. Array of structures (6 marks) Or
 - b. Write a C program to read and display the contents of a file.
19. a. Explain singly linked list insertion and deletion algorithm. Or
- b. Discuss on various operations on queue.
20. a. Explain tree sort algorithm with an example. Or
- b. Sort the following numbers using insertion and bubble sort.
85, 35, 12, 17, 45

APR - 2008 Semester -[2007 - Data Structures and C Programming (Semester 2)]

Answer all the questions.

Max.Marks: 100

section A (10x1=10Marks)

1. Mention any two logical operators in C.
2. Specify any two control statements.
3. Write the declaration of a string.
4. Mention the difference between int a and int *a.
5. Specify the difference between structure and union.
6. Write any two file functions.
7. Distinguish stack and queue.
8. Mention any one applications of stack.
9. Specify the meaning of space complexity.
10. Mention the time complexity of binary search.

Section B(5X6=30Marks)

11. a. Explain the structure of a C program. Or
b. Write a C program to reverse a given integer number.
12. a. What is an array? Explain its use. Or
b. Write a C program to find the largest of n given numbers using pointer.
13. a. Explain enumerated data types with example. Or
b. Write a C program to add two numbers using command line arguments.
14. a. Write an algorithm to push an element onto the stack. Or
b. Explain the method of inserting an element in the list.
15. a. Explain linear search algorithm. Or
b. Search the element 43 using binary search from the sorted list: 15, 22, 31, 40, 43, 52, 60.

Section B(5X12=60Marks)

16. a. Explain different data types available in C with examples. Or
b. Write a C program to sum all the prime numbers from 2 to 100.
17. a. Explain parameter passing mechanisms used in C with examples. Or
b. Write a C program to sort n given names.
18. a. Explain structure within structure and array of structure with examples. Or
b. Write a C program to read and display the content of a text file on the screen.
19. a. Explain singly linked list insertion and deletion algorithm with examples. Or
b. Explain various operations on queue.
20. a. Explain quicksort algorithm with example. Or
b. Sort the following numbers using selection sort and bubble sort: 20, 10, 60, 30, 15, 45.

BHARATHIAR UNIVERSITY
B.Sc. COMPUTER SCIENCE DEGREE COURSE
CORE LAB-2: PROGRAMMING LAB C (DATA STRUCTURES)
PRACTICAL LIST

- 1. Write a C program to create two array list of integers. Sort and store the elements of both of them in the third list.
- 2. Write a C program to experiment the operation of STACK using array implementation.
- 3. Write a C program to create menu driven program to implement QUEUE to perform the following:
 - (i) Insertion
 - (ii) Deletion
 - (iii) Modification
 - (iv) Listing of elements using points
- 4. Write a C program to create LINKED LIST representation of employee records and do the following operations using pointers:
 - a. To add a new record
 - b. To delete an existing record
 - c. To print the information about an employee
 - d. To find the number of employees in the structure.
- 5. Write a C program to count the total nodes of the linked list.
- 6. Write a C program to insert an element at the end of the linked list.
- 7. Write a C program to insert an element at the beginning of the Double linked list.
- 8. Write a C program to display the hash table, which is to be prepared by using the Mid-square method.
- 9. Write a C program to demonstrate Binary search.
- 10. Write a C program to insert nodes into a Binary tree and to traverse in pre-order.
- 11. Write a C program to arrange a set of numbers in ascending order using QUICKSORT.
- 12. Write a C program to arrange a set of numbers in descending order using EXCHANGE-SORT.