1.2.2 Computer Software

The software is what really gives the computer life. It is a set of instructions that tells the computer what to do when the computer operator does something. Software is installed onto your computer and stored in mass storage devices. Software can actually be placed into two separate categories system of applications.

1.2.2.1System software
System software, (*popularly called the operating system*) is the foundation software for a computer. An operating system (OS) controls all parts of the computer. The major functions of the OS ate to handle all input devices (*keyboard, mouse, disk*), handle all output devices (*screen, printer, disk*), coordinate and manage use of other resources (*memory, disk, CPU, etc.*), accept commands from users, provide an environment over which other programs (*software*) can run.
Examples of popular OS are DOS (*Disk Operating System*), Windows, Unix, Linux, OS/2, and Solaris etc. Some of these OSes are specific to a particular computer system – e.g. Solaris runs on Sun SPARC machines.

1.2.2.2Application software
Application software is the software that is developed for a particular use. These software run over the environment provided by the system software. Most of these process our data. There are thousands of application software. They are further divided into numerous categories depending on the operation they perform on our data, i.e. their application. Some of them are described below.
Word processing software allows you to process and manipulate test. Examples are Corel WordPerfect and Microsoft Word. Spreadsheets, very simply put, gives you 4 when you input 2+2. Examples are Quattro Pro and Excel. Databases are most useful for organizing data. For example, database software can keep track of your friend's name; address, telephone numbers and you can add or remove whenever you want. Some examples are Paradox and Access. Internet browsers give us access to the world. When this software is installed and you are connected to an ISP (*Internet Service Provider*), you have access to worldwide information. Common examples are Netscape and Explorer. There are other utility programs that check if the units inside your machine are functioning properly as they should. Examples are the disk scanner that checks if the surface of your disks are damaged, anti-virus that checks if your computer has been infected with some virus (a malicious program that may just annoy you or even delete all the content of your hard drive) and clean if found etc.

1.2.3 Programming Languages
There is another type of software that enables us to make software (*program*) on our own. This is called programming language. It operates in close conjunction with operating system and enables us (*the programmer*) to exploit certain capabilities of operating system while creating the program. This software is called language because it resembles different characteristics of human language (*for example syntax rules that are similar to our grammar*). Programming languages are categorized in different types according to the method by which a programmer creates the instructions. Programming languages are mainly categorized into two types – low level and high languages.

1.2.3.1Low-level language

A low-level language is a programming language much closer to the hardware. It requires a thorough knowledge (*low level*) of the hardware for which the program is being created. Program written for one type of machine may not run (*in fact does not run*) on other machines of different manufacture. These can be divided into two types – machine language and assembly language.

1.2.3.1.1Machine Language

Machine language is the lowest level programming language, which is the only language understood by a computer. While easily understood by computers, machine languages are almost impossible for humans to use because they consist entirely of numbers. Early computers were usually programmed using machine language.

1.2.3.1.2Assembly Language

As the machine language involves only numbers specific to a machine, developers thought that if they can represent those numbers by some symbols, it would be easier to make programs. So, they developed assembly languages in which those numbers are replaced by some symbols (*called mnemonic*). (*For example if 78 is the number that tells the computer to add two numbers, ADD could be used to represent it*.) Each number is replaced by a symbol so that by remembering the symbol for each number, programmers now can write programs. As machine language is the only language understood by computers, they also developed a program that could convert the programs written in symbols into machine language. This converting program is called assembler. Like the machine language, assembly language programs are also dependent on the machine because numbers may vary form machine to machine. But the flexibility it offered was it could be made to run on other machines by slightly changing the code or changing the assembler. Moreover, it's easier to learn assembly language than the machine language.

1.2.3.1.3 High Level Language

A high level language is a programming language that enables a programmer to write programs more or less independent of a particular type of computer. Such languages are considered high-level because they are closer to human languages and further from machine languages. Every high level language has its own set of rules to represent instructions. This rule is called syntax. The main advantage of high-level languages over low-level languages is that they are easier to read, write, and maintain.

The first high-level programming languages were designed in the 1950s. Now there are dozens of different languages, such as Ada, Algol, BASIC, COBOL, C, C^{++} , FORTRAN, LISP, Pascal, and Prolog etc.

Like assembly language programs, programs written in a high-level language also need to be translated into machine language. This can be done in two ways – by a compiler or interpreter.

Compiler

A compiler is a program that translates program (*called source code*) written in some high level language into object code. The compiler derives its name from the way it works, looking at the entire piece of source code and collecting and reorganizing the instructions. A compiler translates high-level instructions directly into machine language and this process is called compiling.

Interpreter

An interpreter translates high-level instructions into an intermediate form, which it then executes. Interpreter analyzes and executes each line of source code in succession, without looking at the entire program; the advantage of interpreters is that they can execute a program immediately. Compilers require some time before an executable program emerges. However, programs produced by compilers run much faster than the same programs executed by an interpreter.

Compiled programs generally run faster than interpreted programs. The advantage of an interpreter, however, is that it does not need to get through the compilation stage during which machine instructions are generated. This process can be time-consuming if the program is long. The interpreter, on the other hand, can immediately execute high-level programs. For this reason, interpreters are sometimes used during the development of a program, when a programmer wants to add small sections at a time and test them quickly.

Because compilers translate source code into object code, which is unique for each type of computer, many compilers are available for the same language. For example, there is a C compiler for PCs and another for Unix computers.

Compile

Compiling is a process of transforming a program written in a high-level programming language from source code into object code. Programmers write programs in a form called source code. Source code must go through several steps before it becomes an executable program. The first step is to pass the source code through a compiler, which translates the high-level language instructions in the source code follow its syntax rules. If not it will inform the programmer where rules have been violated.

The final step in producing an executable program –after the compiler has produced object code - is to pass the object code through a linker. The linker combines molecules (<u>different program segments</u>) and gives real values to all symbolic addresses (*memory locations*), thereby producing machine code.

Source code

Program written in a particular programming language in its original form is called source code (*program*). The word source differentiates code from various other forms that it can have (*for example, object code and executable code*). To execute the program, however, the programmer must translate it into machine language. The compiler translates the source code into a form called object code. Source code is the only format that s readable by humans. When we purchase programs, we usually receive them in their machine-language format. This means that we can execute them directly, but cannot read or modify them.

Object code

Object code is the code produced by a compiler. Object code is often the same as or similar to a computer's machine language. The final step in producing an executable program is to transform the object code into machine languages, if it is not already in this form. A program called linker does this job.

2. Problem solving using Computers

Standard software packages available in the market are intended for general-purpose applications. However, the users often require custom-tailored software, for performing specific data processing or computational tasks. Application software development is the process of creating such software, which satisfies the end user's requirements and needs. In simple language it can be said that problem solving using computers is the development of the application software.
Following steps needs to be followed for developing the application software.
   1. Problem Analysis
   2. Algorithm Development and Flowcharting
   3. Coding
   4. Compilation and Execution
   5. Debugging and Testing
   6. Program Documentation

2.1 Problem Analysis

Problem analysis is also known as defining problem, which involves the following six tasks.
   1. Specifying the objective of the program
   2. Specifying the outputs
   3. Specifying the input requirements
   4. Specifying the processing requirements
   5. Evaluating the feasibility of the program
   6. Documenting the program analysis
At this stage you not only want to determine the input and output requirements but also to get sense of how much everything will cost.

Specifying the objective of the program
In order to avoid having the right solution to the wrong problem, we need to be sure we know what the problem is actually is. Making a clear statement of the problem depends of course, on its size and complexity. If the problem is small and does not involve other systems, then we can probably state the problem easily and proceed immediately to the second step, "Program Design". However, many problems interact with an already existing information system or would require a series of programs and so require very complete analysis, meaning, careful coordination of people, procedures and programs.

Specifying the input requirements

Now that you have determined the outputs, you need to define the input and data. To do this, you list the inputs required and the source of the data. For example, in a payroll program, inputs could be employee timesheets and the source of the in put could be either the employees themselves or their supervisors. You need to be sure that the source of the data is consistent so that the date will be available in the future when you need it.

Specifying the processing requirements

Now you have to determine the processing requirements for converting the input data to output. If the proposed program is to replace or to supplement an existing one, you will want to make a particularly careful evaluation and identification of the present processing procedures, noting the logic used and any improvements that could be made. If the proposed system is not designed to replace and existing system, you would be well advised to look over another system in use that addresses a similar problem.

Evaluating the feasibility of the program

Now you need to see if what you have accomplished so far is enough to make a new program feasible. If the program is intended to replace an existing system, you need to determine id the potential improvements outweigh the costs and possible problems.

Documenting the program analysis

Before concluding the program analysis stage, it is best to write up a document, stating the results of this first phase. This document should contain statements on the program's objectives, output specifications, input requirements, processing requirements and feasibility. In other words, the documents should describe everything you have done so far.

2.2 Algorithm Development and Flowchart (Program design)

You know you have a problem and have identified it in the program analysis stage. Now you need to plan a solution to meet the objectives you have specified. This second phase is called the program design stage-it consists of designing a solution.

Algorithms

Algorithms are a verbal or say written form of the program. It can be defined as ordered description of instructions to be carried out in order to solve the given task. For instance to prepare a tea one can follow the following steps.

- ✓ Start.
- ✓ Fetch water and tealeaves along with sugar and milk.
- ✓ Boil the water.
- ✓ Put tealeaves and sugar in boiled water.
- ✓ Mix with milk.
- ✓ Serve the tea.
- ✓ Stop.

Basic Guidelines for writing algorithms.

- ▪ Use plain language.
- ▪ Do not use any language specific syntax. Same algorithm should hold true for any programming language.
- ▪ Do not make any assumptions. Describe everything clearly and explicitly.
- ▪ Ensure that the algorithm has single entry and exit point.

Important Features of Algorithm

1. Finiteness: Every algorithm should lead to a session of task that terminates after a finite number of steps.
2. Definiteness: Each step must be precisely defined. Actions should be unambiguously specified for each case.
3. Inputs: Any algorithm can have zero or more inputs. Inputs may be given initially or as the algorithm runs.
4. Outputs: Any algorithm may result in one or more outputs. Quantities that have specified relation to inputs.
5. Effectiveness: All operations must be sufficiently basic so as to be implemented with even paper and pencil.

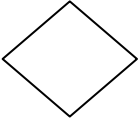For example: Read 2 numbers form user and display the resulting sum.
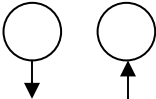
1. Start.
2. Read 2 numbers and solve in variables, say A and B.
3. Store the sum of A and B in C.
4. Display the value in C.
5. Stop.

Flowchart

One of the most widely used devices for designing programs is the flowchart, which graphically represents the logic needed to solve a programming problem.

A programming flowchart represents the detailed sequence of steps, needed to solve the problem. Program flowcharts are frequently used to visualize the logic and steps in processing. In other words it's a diagrammatic representation of algorithm.

Basic blocks used for drawing flowcharts:

Structure	Purpose
⬭	Start / Stop
▭	Processing
◇	Decision making
▱	Input / Outputs
◯ ◯	Connector

For examples: Read 2 numbers form user and display the resulting sum.

```
        ┌─────────────┐
        (    start    )
        └──────┬──────┘
               ▼
        ╱───────────────╲
       ╱   Read a, b    ╱
      ╱───────────────╱
               ▼
        ┌─────────────┐
        │   c=a+b     │
        └──────┬──────┘
               ▼
        ╱───────────────╲
       ╱   Display      ╱
      ╱───────────────╱
               ▼
        ┌─────────────┐
        (    Stop     )
        └─────────────┘
```
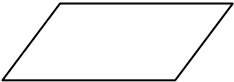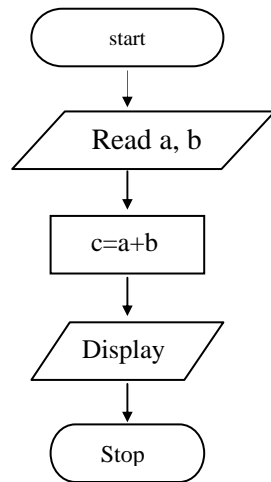
Flowchart to find the largest among three entered numbers:

```
                    ( Start )
                        │
                        ▼
                 ╱─────────────╲
                ╱     input     ╱
               ╱     A, B, C   ╱
              ╱─────────────╱
                        │
                        ▼
          Y      ◇─────────────◇      N
        ┌────────   Is A>B   ────────┐
        │        ◇─────────────◇     │
        ▼                            ▼
  ◇─────────────◇            ◇─────────────◇
  ╲   Is A>C   ╱  N       N  ╲   Is B>C   ╱
Y  ◇─────────◇              ◇─────────────◇  Y
│      │                        │           │
▼      │                        │           ▼
╱───────────╱    ╱───────────╱    ╱───────────╱
╱  Print A  ╱    ╱  Print C  ╱    ╱  Print B  ╱
╱─────────╱      ╱─────────╱      ╱─────────╱
                      │
                      ▼
                  ( Stop )
```
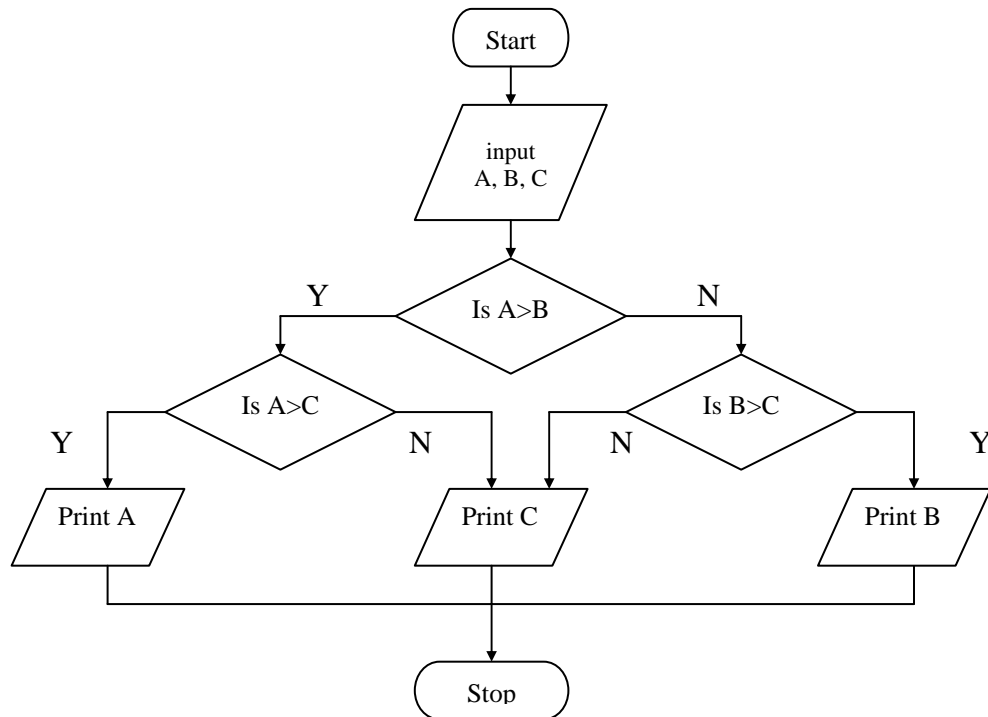
Coding

Writing the program is called coding. In this step, you use the logic you develop in the program design stage to actually write the program. The coding can be done in any kind of languages (*i.e. low level or high level languages*).

As in other areas of life there are many ways to get the same places, in programming also there are various ways to get things done. Here are some of the qualities of a good program:

- ✓ It should be easily readable and understandable by people other than the original programmer. This is accomplished by including comments within the program.
- ✓ It should be efficient, increasing the programmer's productivity.
- ✓ It should be reliable, able to work under all reasonable conditions and always get the correct output.
- ✓ It should be able to detect unreasonable or error conditions and indicate them to the programmer or user without stopping all operations – crashing the system.
- ✓ It should be easy to maintain and support after installation.

Three factors that make programs efficient are:

- ✓ The statements can be arranged into patterns that enhance readability.
- ✓ The variables – variables are symbolically named entities to which values are assigned – used are carefully and descriptively named.
- ✓ Comments can be inserted into the program to document the logic patterns and program flow.

2.3 Compilation and Execution

Generally coding is done in high-level language and sometimes in low-level language (*such as assembly language*). For these languages to work on the computer it must be translated into machine language. There are two kinds of translators – compilers and interpreters. The high level languages can be either called compiled languages or interpreted languages.

In a compiled language, a translation program is run to convert the programmer's entire high-level program, which is called the source code, into a machine language code. This translation process is called compilations.

The machine language code is called the object code and can be saved and either runs (*executed*) immediately or later. Some of the most widely used compiled languages are COBOL, C, C^{++}, FORTRAN, etc.

In an interpreted language, a translation program converts each program statement into machine code just before the program statement is to be executed. Translation and execution occur immediately, one after another, one statement at a time.

Unlike the compiled languages. No object code is stored and there is no compilation. This means that in a program where one statement is executed several times (*such as reading and employ's payroll record*), that statement is converted to machine language each time it is executed. The most frequently used interpreted language is BASIC. Compiler languages are better than interpreted languages as they can be executed faster and more efficiently once the object code has been obtained. On the other hand interpreted languages do not need to create object code and so are usually easier to develop- that is to code and test.

The compilation process

The object of the compiler is to translate a program written in a high level programming language from source code to object code. Programmers write programss in a form called source code. Source code must go through several steps before it becomes an executable program.
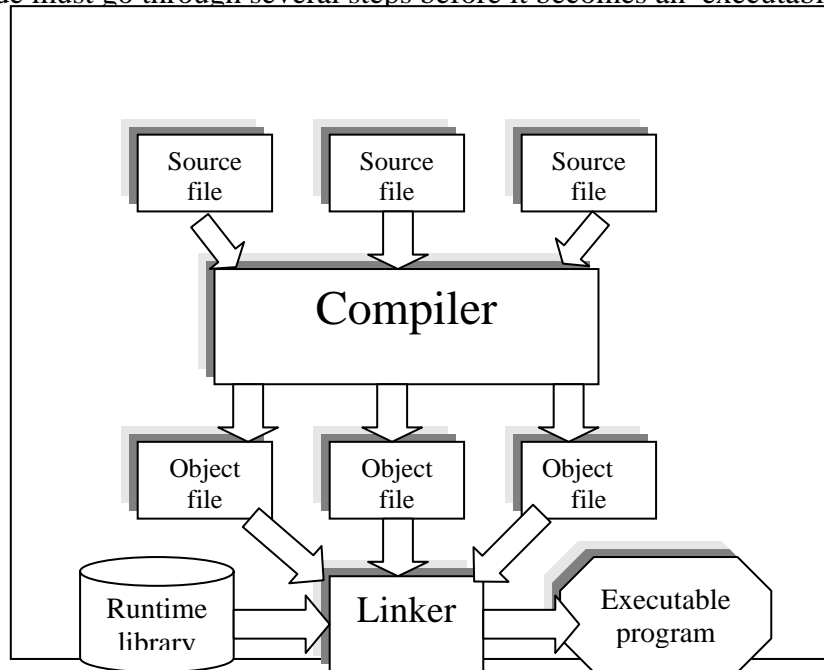
The first step is to pass the source code through a compiler ,which translate the high level language instructions into object code. The final step in producing and

Basic Structure of C programming:

The **documentation section** consists of a set of comment lines giving the name of the program, the author and other details, which the programmer would like to use later. The **link section** provides instructions to the compiler to link function from the system library. The **definition section** defines all symbolic constant.

There are some variables that are used in more than one functions, such variable are called *global* variables and are declared in the *global* declaration section that is outside of all the functions. This section also declares all the user-defined functions.
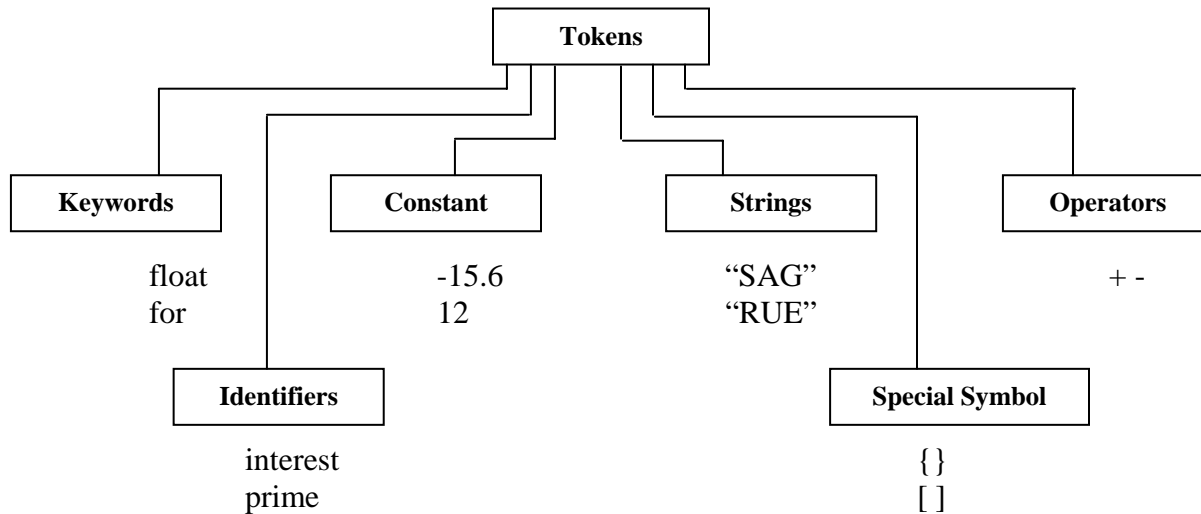
Every C program must have **one main**() function section. This section contains two parts, declaration part and executable part. The declaration part declares all the variables used in the executable part. There is at least one statement in the executable part. These two parts must appear between the opening and the closing braces. The program executing begins at the opening brace and ends at the closing brace. The closing brace of the main function section is the logical end of the program. All statements in the declaration and executable parts end with a semicolon (;).

The **subprogram section** contains all the user-defined functions that are called in the main function. User-defined functions are generally placed immediately after the main function, although they may appear in any order.

All section, except the **main** function section may be absent when they are not required.

Token:

Token is the smallest unit of a program i.e. *,int,+, etc. C has six type of tokens and they are:

```
                          ┌─────────────┐
                          │   Tokens    │
                          └─────────────┘

┌──────────┐    ┌──────────┐    ┌──────────┐    ┌──────────┐
│ Keywords │    │ Constant │    │ Strings  │    │Operators │
└──────────┘    └──────────┘    └──────────┘    └──────────┘
   float           -15.6           "SAG"            + -
   for              12             "RUE"

        ┌──────────────┐                 ┌──────────────────┐
        │ Identifiers  │                 │  Special Symbol   │
        └──────────────┘                 └──────────────────┘
           interest                              { }
           prime                                 [ ]
```
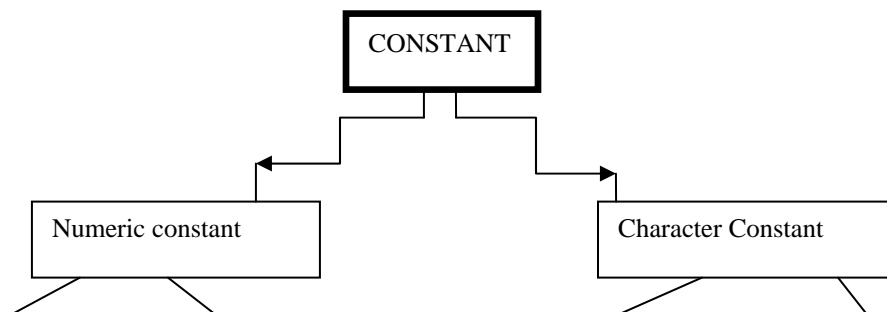
1. **Keyword:**

   All keyword have fixed meaning and these meaning cannot be changed. Keywords serve as basic building blocks for program statement. All keyword words must be written in lowercase.

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

   i.   First character must be an alphabet (or underscore).
   ii.   Must consist of only letters, digits or underscore.
   iii.   Only first 31 characters are significant.
   iv.   Cannot use a keyword.
   v.   Must not contain white space.

3. **Constant:**

   Constant is referring to fixed values that do not change during the execution of a program.

```
                    ┌─────────────┐
                    │  CONSTANT   │
                    └─────────────┘

  ┌──────────────────┐              ┌────────────────────┐
  │ Numeric constant │              │ Character Constant  │
  └──────────────────┘              └────────────────────┘
```

Variable: A variable is a data name that may be used to store a data value. A variable is a space in the computer's memory set aside for a certain kind of data and given a name for easy reference. Variable are used so that the same space in memory can hold different values at different time. Eg a,total..

Variable definitions:

The statement:
datatype variable_name;
eg. int a,total;
Defining a variable tells the compiler the *name* of the variable and the *type* of variable.

Defining Symbolic constant:
Syntax:
#define symbolic-name value of constant
Eg
#define pi 3.14
#define max 30
The following rules apply to#define statement which defines a symbolic constant:
1. Symbolic names have the same form as variable names.
2. No blank space between the pound sign **'#'** and the word **define** is permitted.
3. **'#'** must be the first character in the line.
4. A blank space is required between **#define** and symbolic name and between the **symbolic name** and the constant.
5. #define statements must not end with a semicolon.
6. After definition, the *symbolic name* should not be assigned any other value within the program by using an assignment statement.
7. Symbolic names are NOT declared for data types. Its data type depends on the type of constant.
8. **#define** statements may appear anywhere in the program but before it is referenced in the program.

Declaring a variable as constant:
We may like the value of certain variables to remain constant during the execution of a program. We can achieve this by declaring the variable with the qualifier const at the time of initialization.
Example: **const** int size=30;
const is a new data type qualifier defined by ANSI standard. This tells the compiler that the value of the int variable size must not be modified by the program. However, it can be used on the right-hand side of an assignment statement like any other variable.

Data type:
Data type are means to identify the type of data and associated operation for handling it.
C supports three classes of data type:
1. **Primary (or fundamental) data type**
2. **Derived data type**
3. **User-defined data type**

Fundamental data type:

Fundamental data types are those that are not composed of other data types. There are five fundamental data type and they are:

1. **int data type**: Integers are whole numbers, they have no fraction parts. An identifier declared as **int** becomes an integer variable and can hold integer value only. In order to provide some control over the range of number and storage space, c has three classes of integer storage namely **short int**, **int** and **long int** in both signed and unsigned form.
2. **Float data type (for floating-point)**: floating point numbers are stored in 32 bits with 6 digit of precision. An identifier declared as **float** becomes a floating-point variable and can hold floating-point number only.
3. **Double data type (for double precision floating point number):** the data type double is also used for handling floating-point numbers. But it is treated as a distinct data type because, it (double data type) occupies twice as much memory as type **float**, and stores floating-point numbers with much larger range and precision.
4. **Char data type (for character)**: A single character can be defined as a character (char) type data. Characters are usually stored in 8 bits (one byte) of internal storage. The qualifier **signed** or **unsigned** may be explicitly applied to char.
5. **Void data type (for empty set of value and non-returning functions):** the void type specifies an empty set of values. It is used as the return type for functions that do not return a value.

Type	Size(bits)	Range
char or signed char	8	-128 to 127
unsigned char	8	0 to 255
int or signed int	16	-32,768 to -32,767
unsigned int	16	0 to 65535
short int or		
signed short int	8	-128 to 127
unsigned short int	8	0 to 255
long int or		
signed long int	32	-2,147,483,648 to 2,147,483,647
unsigned long int	32	0 to 4,294,967,295
float	32	3.4E -38 to 3.4 E +38
double	64	1.7E-308 to 1.7E+308
long double	80	3.4E-4932 to 1.1E+4932

Operators:

Operators are words or symbols that cause a program to do something to variable. An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations. Operators are used in programs to manipulate data and variables.

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

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**

1. **Arithmetic   Operator:**

   This operator perform arithmetic operation with the operands(value). The arithmetic operators are:

Operator	Meaning
+	Addition or unary plus
-	Subtraction or unary minus
*	Multiplication
/	Division
%	Modulo division

2. **Relational Operator:**

   This operator is used to take certain decisions depending upon the relation between the operands (values or variable). The value of a relational expression is either one or zero. It is one if the specified relation is *true* and zero if the relation is *false*.

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

3. **Logical operators:**

   C has following three logical operators.

     && meaning logical AND
     || meaning logical OR
     ! meaning logical NOT

   The logical operators && and || are used when we want to test more than one condition and make decisions.

4. Assignment Operator:

Assignment operators are used to assign the result of an expression to a variable. It as the following form:

v=exp;

in above, exp value is assigned to v.

C has a set of 'shorthand' assignment operator of the form.

  v op= exp;

Where v is a variable, exp is an expression and op is the binary arithmetic operator.

The assignment statement

  v op=exp;

is equivalent to

  v = v op exp;

Example:

  X+=20;

This equivalent to

  X=X+20;

5. Increment and Decrement Operators:

These operators are:

           ++ and –

The operator ++ adds 1 to the operand, while – subtracts 1. Both are unary operators and takes   the   following   form:

           ++m; or m++;

           --m; or m--;

       ++m; is equivalent to m=m+1;

       --m; is equivalent to m=m-1;

Rules for ++ and -- Operators:

• Increment and decrement operators are unary operators and they require variable as their operands.

• When postfix ++(or--) is used with a variable in an expression, the expression is evaluated first using the original value of the variable and then the variable is incremented (or decremented) by one.

• When prefix ++ (or --) is used in an expression, the variable is incremented (or decremented) first and then the expression is evaluated using the new value of the variable.

•The precedence and associatively of ++ and – - operators are the same as those of unary + and unary -.

6. Conditional Operator (?:):

Syntax:
exp1? exp2: exp3
Where exp1, exp2 and exp3 are expression.
The operator?: works as follows: exp1 is evaluated first. If it in **nonzero (true)**, then the expression exp2 is evaluated and becomes the value of the expression. If exp1 is **false**, exp3 is evaluated and its value becomes of the expression. Note that only one of the expressions (either exp2 or exp3) is evaluated.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b,l;
printf("\nEnter two numbers:");
scanf("%d%d",&a,&b);
l=a>b?a:b;
printf("\nThe largest number is :%d ",l);
getch();
}
```

7. Bitwise operator:

This operator is used for manipulation of data at bit level. These operators are used for testing the bits, or shifting them right or left. Bitwise operators may not be applied to float or double.

Operator	Meaning
&	bitwise AND
|	bitwise OR
^	bitwise exclusive OR
<<	shift left
>>	shift right

8. Special operators:

C support some special operators of interest such as comma operator, sizeof operator, pointer operator (& and *)and member selection operator (. and ->).

8.1 **The comma operator:**
The comma operator can be used to link the related expression together. A comma linked list of expressions are evaluated left to right and the value of right-most expression is the value of the combined expression. For example
value=(x=5,y=10,x+y);
First assigns the value 5 to x. then assigns 10 to y and finally assigns 15 (i.e 5+10) to value.

8.2 The sizeof operator:

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

> sizeof(int)

It returns 2.

Type conversion in expression:

Implicit type conversion:

C automatically converts any intermediate value to the proper type so that the expression can be evaluated without losing any significance. This automatic conversion is known as implicit type conversion. During evaluation it adheres to very strict rules of type conversion. If the operands are of different type, the 'lower' type is automatically converted to the 'higher' type before the operation proceeds. The result is of higher type.

For example:

int a,b;

float b,c;

b=a+b+c;

the resultant of a+b+c expression is float

Explicit Conversion:

It is the process in which we want to force a type conversion in a way that is different from the automatic conversion. The general form of cast is:

> **(type-name) expression**

Where type-name is one of the standard C data type.

Example:

> X=(**int**)**7.5** i.e 7.5 is converted to integer by truncation

Operator precedence and associativity:

Precedence is used to determine how an expression involving more than one operator is evaluated. There are distinct levels of precedence and an operator may belong to one of these levels. The operator at the higher level of precedence is evaluated first. Precedence is defined as the rules that decides the order in which different operator are applied. The operator of the same precedence are evaluated either from left to right or from right to left depending on the level. This is known as the associativity property of an operator. Associativity is also defined as the rule that decides the order in which multiple occurrences of the same level operator are applied.

The following table provides a complete list of operators, their precedence levels, and their rules of association.

.

Managing input and output operation:

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

<p align="center">**variable_name=getchar ();**</p>

Variable_name is a valid C name that has been declared as char type. When this statement is encountered, the computer waits until a key is pressed and then assigns this character as a value to **getchar** function. Since **getchar** is used on the right –hand side of an assignment statement, the character value of getchar is in turn assigned to the variable name on the left. For example

<p align="center">char name;
name=getchar();</p>

The **getchar ()** function accept any character keyed in. This includes **RETURN** and **TAB**. This mean when we enter single character input, the newline character is waiting in the input queue after **getchar ()** return. This could create problem when we use **getchar ()** in a loop interactively. A dummy **getchar()** may be used to 'eat' the unwanted newline character. We can also use the **fflush** function to flush out the unwanted character.

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

<p align="center">putchar (variable_name);</p>

where variable_name is a type char variable containing a character. This statement displays the character contained in the variable_name at the terminal.
For example: answer='Y';

<p align="center">putchar(answer):   //this statement display the character Y</p>

```c
#include<stdio.h>
#include<conio.h>
void main()
{
char name[20];
char ch;
int i=0;
printf("\nEnter the name:");
while((ch=getchar())!='\n')
{
name[i]=ch;
i++;
}
name[i]='\0';
printf("\nThe name is :%s ",name);
getch();
}
```

Formatted input:

Formatted input refers to an input data that has been arranged in particular format. It can be performed using **scanf** function. The general format of scanf is

   scanf ("control string",arg1,arg2,……arnn);

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

Control string(also known as format string) contain field specification, which direct the interpretation of input data. It may include:

• field (or format) specifications, consisting of the conversion character %, a data type character (or type specifier), and an optional number, specifying the field width.

• Blanks, tabs, or newlines.

Blanks, tabs and newlines are ignored.

Inputting Integer Numbers:

The field specification for reading an integer number is:

   %w s d

The percentage sign(%) indicates that a conversion specification follows. w is an integer number that specifies the field width of the number to read and d, known as data type character, indicates that the number to be read is in integer mode.

When the scanf reads a particular value, reading of the value will be terminated as soon as the number of character specified by the field width is reached (if specified) or until a character that is not valid for the value being read in encountered. In the case of integers, valid characters are an optionally signed sequence of digits.

An input field may be skipped by specifying * in the place of field width. For example

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

will assign the data 123 456 789 as follows:

   123 to a

   456 skipped (because of *)

   789 to b

Inputting Real Numbers:

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

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

Inputting Character Strings:

Scanf can input strings containing more than one character. Following are the specifications for reading character strings:

    %ws or %wc

Formatted output:

The printf statement provides certain features that can be effectively exploited to control the alignment and spacing of print-out 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 character such as \n,\t and \b

The control string indicates how many arguments follow and what their types are. The arguments arg1,arg2…,argn are the variables whose values are formatted and printed according to the specifications of the control string. The arguments should match in number, order and type with the format specifications.

Commonly used printf format codes:

Code	Meaning
%c	printing a single character
%d	printing a decimal integer
%e	printing a floating point value in exponent form
%f	printing a floating point value without exponent
%g	printing a floating point value either e-type of f-type depending on
%i	printing a signed decimal integer
%o	printing an octal integer, without leading zero
%s	print an unsigned decimal integer
%x	print a hexadecimal integer, without leading Ox

The following letters may be used as prefix for certain conversion characters.

h   for short integer
l   for long integer or double
L for long double

Commonly used output format flags

Flag	Meaning
-	Output is left-justified within the field. Remaining field will be blank.
+	+ or – will precede the signed numeric item.
0	causes leading zeros to appear.
# (with o or x)	causing octal and hex item to be preceded by 0 and 0x. respectively

Decision making and Branching:

C language possesses such decision-making capabilities by supporting the following statements:
1. **if** statement
2. **switch** statement
3. **conditional operator** statement
4. **goto** statements

These statements are known as decision-making statements.

if statement:
It takes the following form:

if (test expression)

It allows the computer to evaluate the expression first and then, depending on whether the value of the expression (relation or condition) is **'true'** (non zero) or **'false'** (zero), it transfers the control to a particular statement. This point of program has two paths to follow, one for the true condition and the other for the false condition:

Entry

Test Expression
?
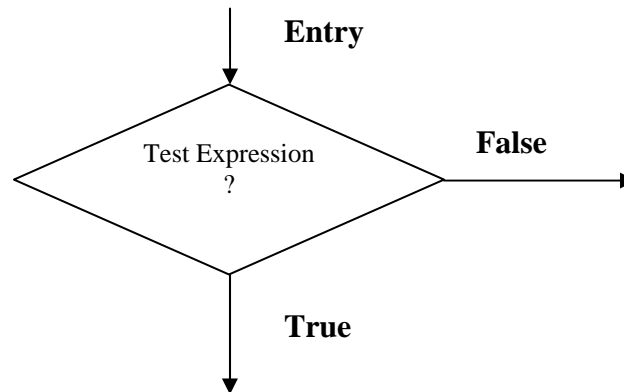
False

True

The if..else statement:
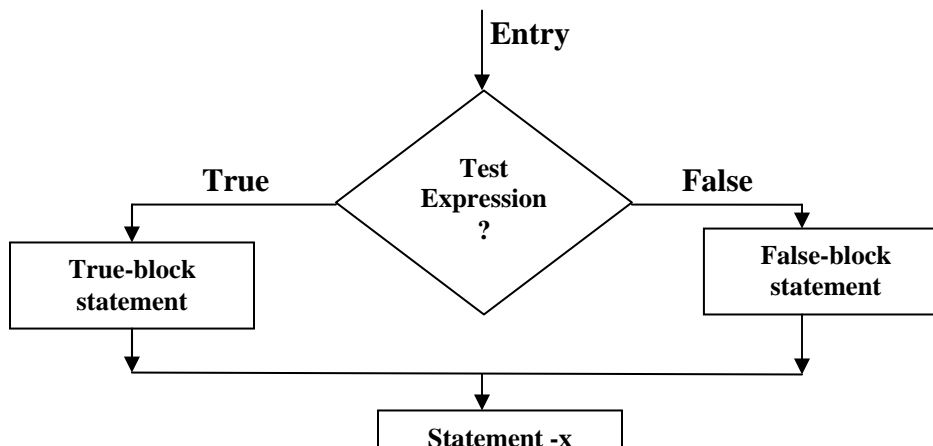The general form is:

```
if (test expression)
        {
                True-block statement(s)
        }

else
        {
                False-block statement(s)
        }
        statement-x
```
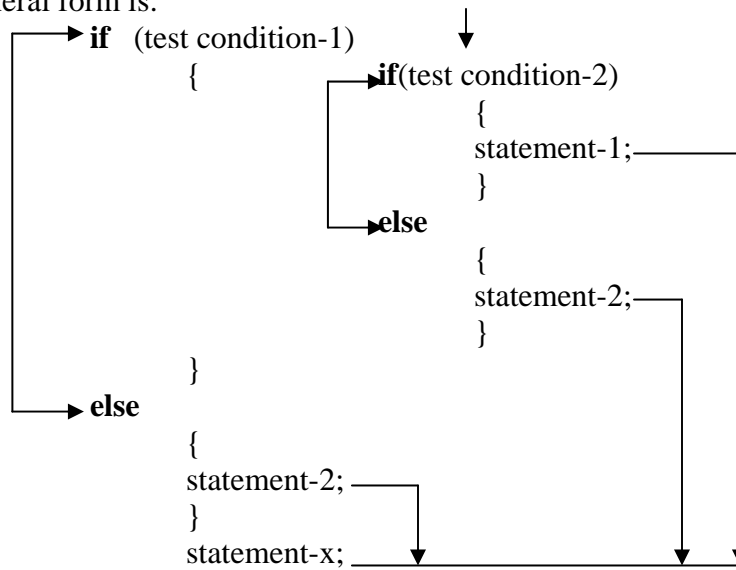
if the test expression is *true*, then the true-block statement(s), immediately following the if statements are executed: otherwise, the false-block statement(s) are executed. In either case, true-block or false-block will be executed, not both. In both the cases, the control is transferred subsequently to the statement-x.
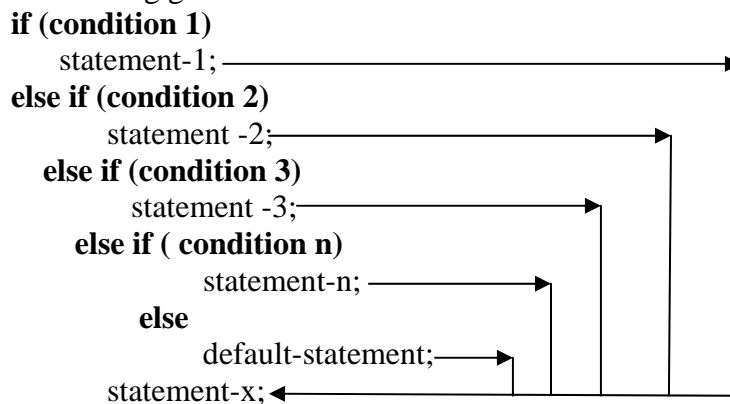
Entry

True

Test Expression ?

False

True-block statement

False-block statement

Statement -x

Nesting of if…else statement

The general form is:

```
        if   (test condition-1)
                {                   if(test condition-2)
                                        {
                                            statement-1;
                                        }
                                    else

                                        {
                                            statement-2;
                                        }

                }
        else

                {
                statement-2;
                }
                statement-x;
```

The else if ladder:

A multipath decision is a chain of **if**s in which the statement associated with each **else** is an **if**. It takes the following general form:

```
        if (condition 1)
                statement-1;
        else if (condition 2)
                statement -2;
            else if (condition 3)
                    statement -3;
              else if ( condition n)
                        statement-n;
                    else
                        default-statement;
                statement-x;
```

The switch statement:

The switch statement tests the value f a given variable (or expression) against a list of case values and when a match is found, a block of statements associated with that case is executed. The general form of the switch statement is as shown below:

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

The expression is an integer expression or characters. value-1,value2…… are constants or constant expressions(evaluable to an **integer constant**) and are known as **case labels**. Each these values should be unique within a **switch** statement. block 1, block-2….are statement lists and may contain zero or more statements. There is no need to put braces around these blocks. Note that **case** labels end with a colon(**:**);

The ? : operator:

Conditional operator is a combination of ? and :, an takes three operands. The general form of use of the conditional operator is as follows:

Conditional expression? expression1: expression2

The conditional expression is evaluated first. If the result is nonzero, expression1 is evaluated and is returned as the value of the conditional expression. Otherwise, expression2 is evaluated and its value is returned.
Example:
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b,c,l;
printf("\nEnter two numbers:");
scanf("%d%d%d",&a,&b,&c);
l=a>b?(a>c?a:c):(b>c?b:c);
printf("\nThe largest value is %d",l);
getch();
}

The goto statement

 It is the statement to branch unconditionally from one point to another in the program. 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 label is placed immediately before the statement where the control is to be transferred. The general forms of goto and label statements are shown below:

goto label:———┐	**label:** ◄———┐
-------------	statement;
-------------	----------
-------------	----------
label: ◄———┘	**goto** label;———┘
statement;	

Forward jump **Backward jump**

Note that a **goto** breaks the normal sequential execution of the program. If the label: is before the statement **goto label**; a loop will be formed and some statements will be executed repeatedly. Such jump is known as **a backward jump**. On the other hand, if the label: is placed after the **goto label**; some statements will be skipped and the jump is known as a **forward jump**.

Another use of the goto statement is to transfer the control out of the loop (or nested loop) when certain peculiar conditions are encountered.

Avoiding goto:

When a goto is used, many compilers generate a less efficient code. In addition, using many of them makes a program logic complicated and renders the program unreadable. If the backward jump is not properly terminated then it leads to infinite looping.
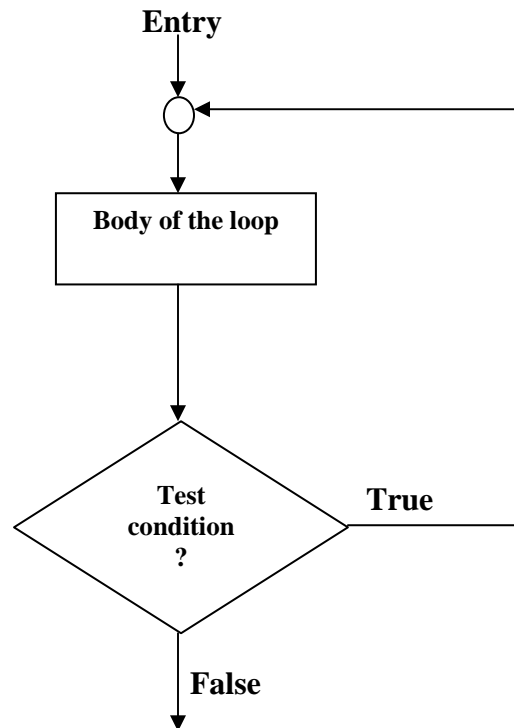
Decision making and looping:

Looping (or iteration) is the process of executing a sequence of statements until some condition for termination of loop is satisfied. These enable us to develop concise programs containing repetitive processes without the use of **goto** statements.
Looping is classified as:

1. **Entry-Controlled loop:** In entry-controlled loop, the control conditions are tested before the start of the loop execution. If the conditions are not satisfied, then the body of the loop will not be executed. It is also known as **pre-test loop**.
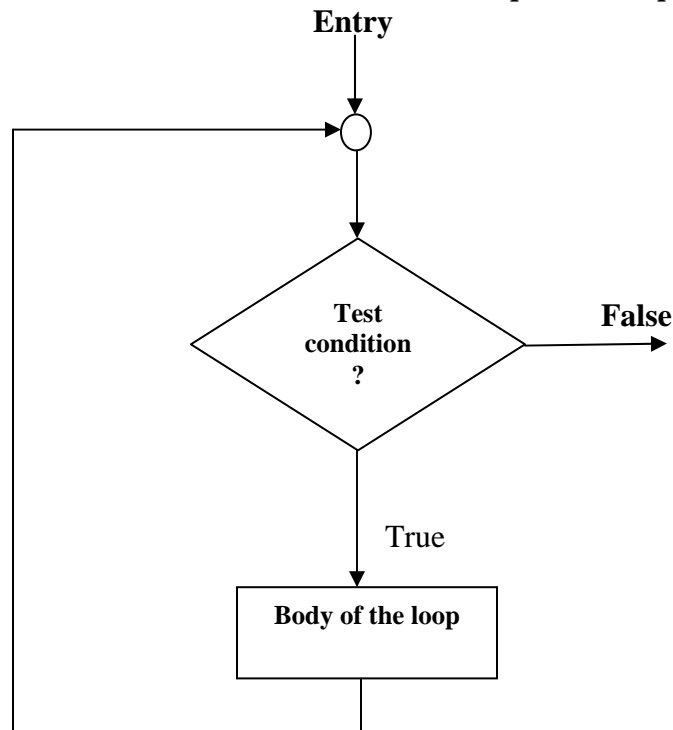
2. **Exit-Controlled loop**: In exit-controlled loop, the test is performed at the end of the body of the loop and therefore the body is executed unconditionally for the first time. It is also known as *post-test loop*.

```
                    Entry
                      |
                      v
                      O <----------------+
                      |                  |
                      v                  |
            +-------------------+        |
            |  Body of the loop |        |
            +-------------------+        |
                      |                  |
                      v                  |
                    /   \                |
                   /Test \   True        |
                  / condition >----------+
                  \    ?    /
                   \      /
                    \   /
                      |
                      | False
                      v
```

The C language provides three constructs for performing loop operation. They are:
1. **While statement.**
2. **The do statement.**
3. **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. ***On exit***, the program continues with the statement immediately after the body of the loop.

The do statement:
This takes the form:

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

On executing do statement, first the body of looping is executed. At the end of the loop, the test-condition in the **while** statement is evaluated. If the condition is **true**, the program continues to evaluate the body of the loop once again. This process continues as long as the condition is true. When the condition becomes false, the loop will be terminated and the control goes to the statement that appears immediately after the while statement.
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*.

The for statement:

For loop is the entry-controlled loop. The general form of for loop is:

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

The execution of for statement is as follows:

1. Initialization of the control variable is done first, using assignment statement.
2. The value of the control variable is tested using the test-condition. If the condition is true, the body of loop is executed; otherwise the loop is terminated and execution continues with the statement that immediately follows the loop.
3. When the body of the loop is executed, the control is transferred back to the for statement after the evaluating the last statement in the loop. Now the control variable is incremented using as assignment statement such as i=i+1 and the new value of the control variable is again tested to see whether it satisfies the loop condition. If the condition is satisfied, the body of the loop is again executed. This process continues till the value of the control variable fails to satisfy the test-condition.
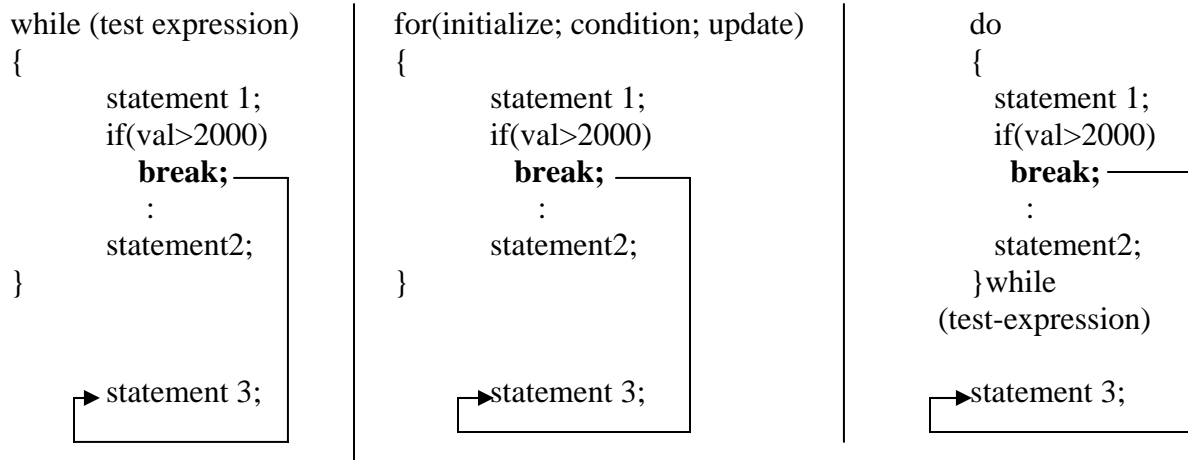

Jump Statement:
The jump statement unconditionally transfers program control one point to another point in a program. They are:
    1. The goto statement: Refer to page number **24**
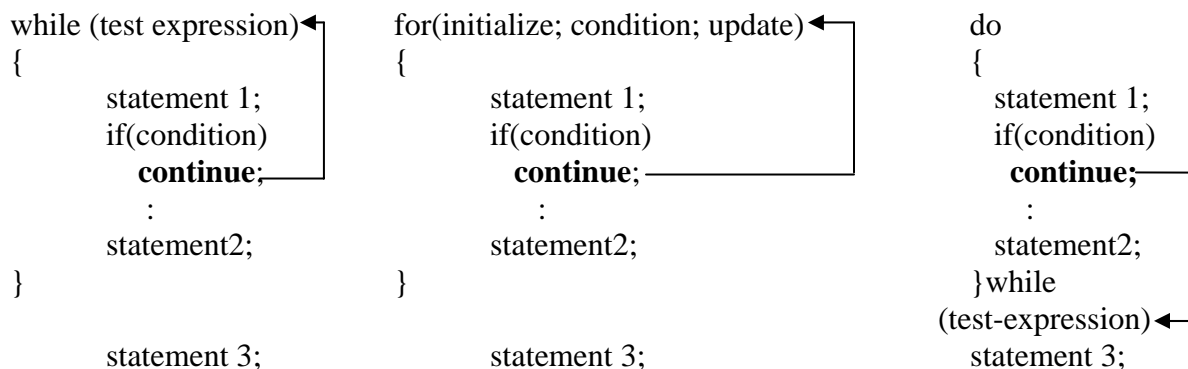
2. **The break Statement**:

The break statement enables a program to skip over part of the codes. A **break** statement terminates the smallest enclosing **while**, **do-while**, **for** or **switch** statement. Execution resumes at the statement immediately following the body of the terminated statement. The following figure explains the working of **break** statement.

while (test expression)	for(initialize; condition; update)	do
{	{	{
statement 1;	statement 1;	statement 1;
if(val>2000)	if(val>2000)	if(val>2000)
break;	**break;**	**break;**
:	:	:
statement2;	statement2;	statement2;
}	}	}while
		(test-expression)
statement 3;	statement 3;	statement 3;

The break statement skips the rest of the loop and jumps over to the statement following the loop.

3. **The continue Statement**:

The continue is another jump statement some what like the break statement as both the statement skip over a part of the code. But the continue statement is little different from break. Instead of forcing termination, it force the next iteration of the loop to take place, skipping any code in between. The following figure explains the working of continue statement.

while (test expression)	for(initialize; condition; update)	do
{	{	{
statement 1;	statement 1;	statement 1;
if(condition)	if(condition)	if(condition)
continue;	**continue;**	**continue;**
:	:	:
statement2;	statement2;	statement2;
}	}	}while
		(test-expression)
statement 3;	statement 3;	statement 3;

The continue statement skips the rest of the loop statements and causes the next iteration of the loop.

For the **for** loop, **continue** causes the next iteration by updating the variable and then causing the test-expression's evaluation. For the **while** and **do-while** loops, the program control passes to the conditional tests.

Practice Question:

1. Write an algorithm and a flowchart to read a five number and check whether the number is a palindrome or not.

   Algorithm:

   Step1: Start

   Step2: Declare variable n, r, sum, i

   Step3: **i=1**

   Step4: Repeat steps Step 4.1 to 4.8 until i is less than or equal to 5 else goto step 5

           Step 4.1: **input to n**

           Step 4.2: **sum=0**

           Step 4.3: **temp=n**

           Step 4.4: Repeat Steps 4.4.1 to 4.4.3 until n is greater than 0 else goto step 4.5

                   Step 4.4.1: **r=n mod of 10**

                   Step 4.4.2: **sum=sum*10+r**

                   Step 4.4.3: **n=n/10, then goto Step 4.4**

           Step 4.5: if sum is equal to temp than goto Step 4.6 else goto Step 4.7

           Step 4.6: Display n as palindrome, then goto step 4.8

           Step 4.7: Display n is not palindrome

           Step 4.8**: i=i+1, then goto step 4**

   Step5: Stop

   Flowchart:

Source code:

```c
#include<stdio.h>
#include<conio.h>
void main()
{
int n,i,r,sum,temp;
i=1;
clrscr();
while(i<=5)
{
printf("\nEnter the numbers:");
scanf("%d",&n);
sum=0;
temp=n;
while(n>0)
{
r=n%10;
sum=sum*10+r;
n=n/10;
}
if(sum==temp)
printf(" Palindrome %d",temp);
else
printf("Not Palindrome %d",temp);
i++;
}
getch();
}
```

2. Write an algorithm and flowchart to generate the following.
```
        *
      * * *
    * * ** *
  * * ** * * *
* * ** * * * * *
```

Algorithm:
Step1: start
Step 2: input to **n**   //Number of lines
Step 3: Initialize **c** as 0   //Deviation on each line
Step 4:**i=1**   //Number of lines
Step 5: Repeat step 5.1 to 5.3 until **i** less than or equal to **n** else goto step 6
      Step 5.1: **j=1**   //Number of column
      Step 5.2: Repeat step 5.2.1 to 5.2.2 until j less than or equal to (2*n-1) else goto step 5.3
           Step 5.2.1: Is **j** greater than or equal to (**n-c**) and less than or equal to (**n+c**) than
                goto Step 5.2.1.1 else goto Step 5.2.1.2:
                Step 5.2.1.1: Display **"*",** then goto step 5.2.2
                Step 5.2.1.2: Display " "
          Step 5.2.2:**j=j+1**, then goto step 5.2
      Step 5.3: Display **newline**
      Step 5.4: **c=c+1**
      Step 5.5:**i=i+1**, then goto **step 5**
Step 6:Stop

Source code:
```
#include<stdio.h>
#include<conio.h>
void main()
{
int n,c,i,j;
clrscr();

c=0;    //deviation

printf("\nEnter the number of lines:");

scanf("%d",&n); //number of lines

for(i=1;i<=n;i++)
{
for(j=1;j<=(2*n-1);j++)
{
if(j>=(n-c)&&j<=(n+c))
printf("*");
else
printf(" ");
}
printf("\n");
c++;
}
getch();
}
```
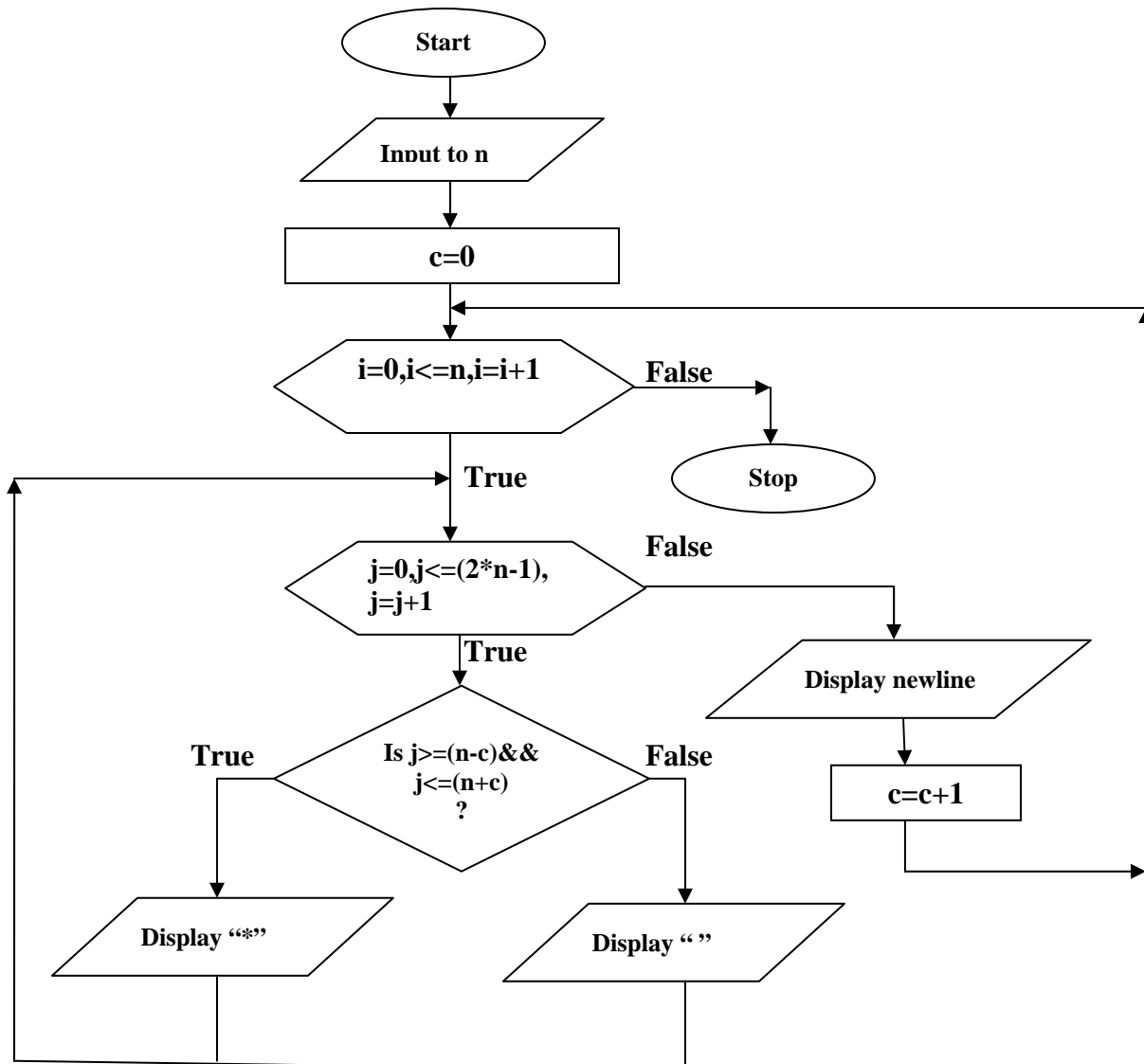
Flow Chart:

Array:

Array is a collection of variables of the same type that are referenced by a common base. It is also known as derived data type as it is derived from fundamental data type. The array is given a name and its elements are referred by their subscripts or indices. In c array's index numbering starts with 0. Arrays are of different types: (i) *one-dimensional* array, comprised of finite homogeneous element.(ii) *multi-dimensional array*, comprised of elements, each of which is itself an array. A two dimensional array is the simplest of multidimensional arrays.

Single dimensional Array:

The simplest form of an array is a single dimensional array. An array definition specifies a *variable type* and a *name* along with one more feature size to specify how many data items the array will contain. The general form of an array declaration is as shown below:

       type array-name[size];

Where type declares the **data type** of the array, which is the type of each element in the array. The *array-name* specifies the name with which the array will be referenced and *size* defined how many elements the array will hold. The size must be integer value or integer constant without any sign.

Initialization of one dimensional array:

After an array is declared, its elements must be initialized. Otherwise, they will contain "garbage". The general form of initialization of array is:

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

The values in the list are separated by commas. The element value in the list of value must have the same data type as that of type, of the array.

Example:

       int number [3]= { 1,2,3};

If the number of initializers may be less than the declared size. In such case, the remaining elements are initialized to zero, if the array type is numeric and NULL if the type is char.

Example: int num [10] = {1, 2};

Will initialize the first two elements of num as 1 and 2 respectively and the remaining elements to 0.

Practice:

Write to program to arrange list of 5 elements in ascending order:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
int n[5],temp;
int i,j;
printf("\nEnter the list of 5 elements:");
```

```c
for(i=0;i<5;i++)
{
scanf("%d",&n[i]);
}
for(i=0;i<(5-1);i++)
{
for(j=i+1;j<5;j++)
 {
  if(n[i]>n[j])
  {
    temp=n[i];
    n[i]=n[j];
    n[j]=temp;
   }
 }
}
  printf("\n The elements in assending order:");
for(i=0;i<5;i++)
{
printf("\n%d",n[i]);
}
getch();
}
```

Two –Dimensional Array:

A two-dimensional array is an array in which each element is itself an array. For instance, an array A [M][N] is a M by N table with M rows and N columns containing M X N element. The number of elements in a 2-D array can be determined by multiplying number of rows with number or columns. For example, the number of elements in an array A[7][9] is calculated as 7X9=63.

	0	1	...	...	...	N-1
0						
1						A[1][N-1]
Row 2						
:						
M-1						

A[2][1] ◄

The general form of a two-dimensional array is:

type array-name[row_size][column_size];

Where **type** is the data type of the array having name **array-name**: **row_size**, the first index, refers to the number of rows in the array and **column_size**, the second index, refers to the number of columns in the array. Following declares an int array sales of size 5,12.

int sale[5][12];

Initialization two- dimensional array:

Two dimensional arrays are also initialize in the same ways as single-dimension ones. For example,

        int table [2][3]= {0,0,0,1,1,1};

initializes the elements of the first row to zero and the second row to one. The initialization is done row by row. The above statement can be equivalently written as

        int table[2][3]= { {0,0,0},{1,1,1}};

Commas are require after each brace that closes off a row, except in the case of the last row. When the array is completely initialize with all values, explicitly, we need not specify the size of first dimension. That is,

        int table [ ][3]= {

                    {0, 0, 0}

                    {1, 1, 1}

          };

If the values are missing in an initialize, they are automatically set to zero.

        int table [ ][3]= {

                    {1, 1}

                    {2}

          };

Will initialize the first two elements of the first row to one, the first element of the second row to two, and all other elements to zero.

Practice:
Write a program that adds the individual rows of a two dimensional array of m by n and store the sums of rows into a single dimensional array using functions. Write a function that takes a two dimensional array and one-dimensional array and process the result and store in one-dimensional array.

```
#include<stdio.h>
#include<conio.h>
void add(int n[3][3],int r[3])
{
int i,j;
for(i=0;i<3;i++)
{
r[i]=0;
for(j=0;j<3;j++)
{
r[i]+=n[i][j];
}
}
}

void main()
{
int list[3][3],result[3],i,j;
printf("\nEnter the number :");
for(i=0;i<3;i++)
for(j=0;j<3;j++)
scanf("%d",&list[i][j]);
add(list,result);
printf("The result is:");
for(i=0;i<3;i++)
printf("\n%d",result[i]);
getch();
}
```

String:

C does not have a string data type rather it implements strings as single dimension character array. A string is defined as a character array that is terminated by null character **'\0'**. For this reason, the character arrays are declared one character longer than the largest string they can hold. The general form of declaration of a string variable is:

char string_name[size];

Example:

char name[10];

like numeric array, character arrays may be initialize when they are declared. C permits a character array to be initialized in either of the following two forms:

char name[11]="SAGARMATHA";

char name[11]={ 'S', 'A', 'G', 'A', 'R', 'M', 'A', 'T','H', 'A','\0'};

C also permits us to initialize a character array without specifying the number of elements. In such cases, the size of array will be determined automatically, based on the number of elements initialized. For example

char string[ ]= {'R', 'U','W','E','\0'};

defines the arrays string as a five elements array.

Reading line of text:

scanf with %s or %ws can read only string without whitespaces. That is, they cannot be used for reading a text containing more than one word. However, C supports a format specification known as the edit set conversion code %[. .] that can be used to read a line containing a variety of characters, including whitespace. For example;

char name[20];

scanf("%[^\n]",name);

printf("%s",name);

Note we can use getchar and gets functions to take input to character array variable. For example to getchar look at unformatted input example.

Program to count number of word in a line.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
char string[10];
int i,c=0;
clrscr();
printf("\n Enter the first string:");
gets(string);
i=strlen(string);
string[i++]=' ';
for(i=0;string[i]!='\0';i++)
{
if(string[i]==' ')
c++;
}
printf("\n The number of word in sentence:%d",c);
getch();
}
```

Array of string:

An array of string is a two-dimensional character array the size of first index (rows) determines the number of strings and the size of second index (column) determines maximum length of each string. The following code declares an array of 10 stings, each of which can hold maximum 50 valid character.

        char name[10][51];

Notice, that the second index has been given value 51 i.e. 1 extra to take care of the null character '\0'. A array of string appears in memory as shown below.

		0	1	2	3	4	5	6		
	0								stirng[0]
	1								stirng[1]
5	2								stirng[2]
	3								stirng[3]
	4								stirng[4]
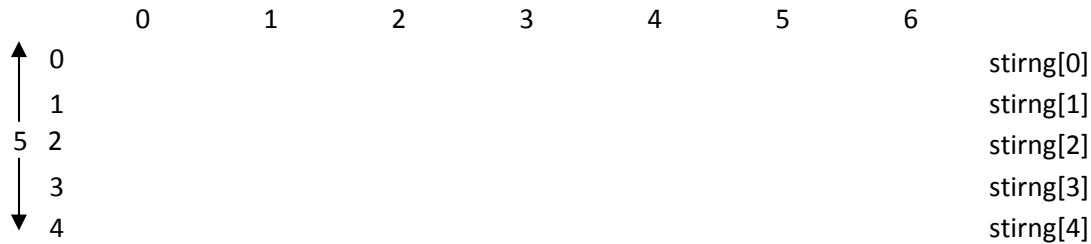
Write a program to arrange list of student name in ascending order:

```c
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
char name[5][10],temp[10];
int i,j;
for(i=0;i<5;i++)
{
printf("\nEnter the name of student:");
scanf("%s",name[i]);
}
for(i=0;i<(5-1);i++)
{
for(j=i+1;j<5;j++)
  {
  if(strcmp(name[i],name[j])>0)
  {
    strcpy(temp,name[i]);
    strcpy(name[i],name[j]);
    strcpy(name[j],temp);
    }
  }
}
  printf("\n the name of student is:");
for(i=0;i<5;i++)
{
printf("\n%s",name[i]);
}
getch();
}
```

User Defined Function

Function:
A function is a named unit of a group of program statements. The unit can be invoked from other part of the program. The elements of function:
1. Function definition
2. Function call
3. Function declaration or Function prototype.

Function Prototype:
A function prototype is a declaration of the function that tells the program about the type of the value returned by the function and the number and type of each argument. It consist of four parts.
- Function type (return type)
- Function name
- Parameter list
- Terminating Semicolon.

They are coded in the following format.

Function-type function-name (parameter list);

Example:

void sum(int a, int b);

when we place the declaration above all the function(in the global declaration section), the prototype is referred to as global prototype. Such declaration is available for all the function in the program.

When we place it in a function definition (in the local declaration section), the prototype is called a local prototype.

Function Definition:

A function must be defined before it is used anywhere in the program. The general form of a function definition is given below:

type function-name(parameter list)
{
      Body of the function
}

Where the type specifies the type of values that the return statement of the function returns. It may be any valid data type. If no type is specified, the compiler assumes the function returns as integer value. The parameter list is a comma-separated list of variables of a function referred to as its arguments. A function may be without any parameters, in which case, the parameter list is empty.

Function call:
A function can be called by simple using the function name followed by a list of actual parameter (or arguments),if any, enclosed in parentheses. The general form:

function-name(list of variable or value);

eg.

add(4,5);

When the compiler encounters the function call, the control is transferred to the function definition. And it executes each line of code in function and returns the control to the main program again.

Advantage of function:

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

The functions are categories as follows:

1. **Function with no arguments and no return values:**
   When a function has no argument, it does not receive any data from the calling function. Similarly, when it does not return value , the calling function does not receive any data from the called function. In effect, there is no data transfer between the calling function and the called function.
2. **Function with argument and no return type:**
   In this approach, we could make the calling function to read data from the terminal and pass it on the called function.
3. **Function with argument and return type:**
   In this, the calling function sends the data to the function and the called function returns the value to the calling function statement. It is a two-way data communication between the calling and called function.
4. **Function with no argument and return type:**
   In this function, the function does not take any input from the calling function but it returns value to the calling function.

Ways of passing arguments to functions:

There are two different mechanisms to pass arguments to function.

- Pass by value (also called as call by value)
- Pass by reference (also known as call by pointer)

Pass by value:

In pass by value, values of actual parameters are copied to the variable in the parameter list of the called function. The called function works on the copy and not on the original values of the actual parameters. This ensures that the original data in the calling function cannot be changed accidently.

Pass by pointer:

In pass by pointers (also known as pass by address), the memory addresses of the variables rather than the copies of value are sent to the called function. In this case, the called function directly work on the data in the calling function and the changed value are available in the calling function for its use.

Pass by pointers method is often used when manipulating arrays and strings. This method is also used when we require multiple values to returned by the called function.

Passing Array To Functions:

Three Rules to pass an array to a function
1. **Function must be called by passing only the name of the array.**
2. **In the function definition, the formal parameter must be an array type, the size of the array does not need to be specified.**
3. **The function prototype must show that the argument in a array.**

Note:
When an entire array is passed as an argument , the contents of the array are not copied into the formal parameter array. Instead, information about the addresses of array elements are passed on to the function. Therefore any changes introduced to the array elements are truly reflected in the original array in the calling function. However, this does not apply when an individual element is passed on argument.

```c
//One dimensional array
#include<stdio.h>
#include<conio.h>
#include<math.h>
float mean(float a[],int size)
{
int i;
float sum=0;
for(i=0;i<size;i++)
sum+=a[i];
return (sum/size);
}
float star(float a[],int size)
{
int i;
float xm,std=0.0;
xm=mean(a,size);
for(i=0;i<size;i++)
std+=(xm-a[i])*(xm-a[i]);
std=sqrt(std/size);
return std;
}
void main()
{
float a[5], i;
printf("\nEnter element:");
for(i=0;i<5;i++)
scanf("%f",&a[i]);
printf("\n The standard deviation is %f",star(a,5));
getch();
}
```

Passing Two dimensional Array to the function:
The rules are:

1. The function must be called by passing only the array name.
2. In the function definition, we must indicate that the array has two dimensions by including two set of brackets.
3. The size of the second dimension must be specified.
4. The prototype declaration should be similar to the function header.

Matrix Multiplication
```c
#include<stdio.h>
#include<conio.h>
void input(int a[][10],int r,int c)
{
int i,j;
for(i=0;i<r;i++)
for(j=0;j<c;j++)
scanf("%d",&a[i][j]);
}
void display(int a[][10],int r,int c)
{
int i,j;
for(i=0;i<r;i++)
{
for(j=0;j<c;j++)
printf("%d",a[i][j]);
printf("\n");
}
}
void multiply(int a[][10],int b[][10],int c[][10],int r1,int c1,int c2)
{
int i,j,k;
for(i=0;i<r1;i++)
for(j=0;j<c2;j++)
{
c[i][j]=0;
for(k=0;k<c1;k++)
c[i][j]+=a[i][k]*b[k][j];
}
}
void main()
{
int a[10][10],b[10][10],c[10][10],r1,r2,c1,c2;
clrscr();
printf("\nEnter the row and column for first matrix;");
scanf("%d%d",&r1,&c1);
printf("\nEnter the row and column for second matrix;");
scanf("%d%d",&r2,&c2);
if(c1!=r2)
```

```
{
printf("\nMultipication is not possible:");
}
else
{
printf("\nenter the value to first matrix:");
input(a,r1,c1);
printf("\nenter the value to second matrix:");
input(b,r2,c2);
multiply(a,b,c,r1,c1,c2);
printf("\n The resultant Matrix is :");
display(c,r1,c2);
}
getch();
}
```

Passing string to functions

The string are treated as character array in C and therefore the rules of passing string to function are very similar to those for passing array to functions.

Recursive:

A function is said to be recursive if a statement in the body of the function calls itself. In a recursive function, there must be a reachable condition for its termination; otherwise, the function will be invoked endlessly.

Write a program to find the HCF for any two numbe entered by user.

```
#include<stdio.h>
#include<conio.h>
int hcf(int f,int s)
{
if(f%s==0)
return s;
return (s,f%s);
}
void main()
{
int a,b;
clrscr();
printf("\nEnter the number:");
scanf("%d%d",&a,&b);
printf("The Hcf is %d ",hcf(a,b)) ;
getch();
}
```

Storage classes:

The available storage classes are:
1. Automatic variables
2. External variables
3. Static variables
4. Register variables

We explain all of above class of variable in terms of visibility, longevity and scope. The scope of variable determines over what region of the program a variable is actually available for use. The longevity refers to the period during which a variable retain a given value during execution of program (alive). The visibility refers to the accessibility of a variable from the memory.

Automatic variable:

Automatic variable are declared inside a functions in which they are to be utilized. They are created when the function is called and destroyed automatically when the function is exited, hence the name automatic. Automatic variables are therefore private (or local) to the function in which they are declared. We may also use the keyword auto to declare automatic variables explicitly.

External variables:

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

Write a program to evaluate the following series using recursive function.

$1^2-2^2+3^2-4^2\ldots n^2$

```
#include<stdio.h>
#include<conio.h>
float sum=0.0;
int sign=-1;
void add(int n,int i)
{
int term;
if(i<=n)
{
sign=sign*-1;
term=i*i*sign;
sum=sum+term;
add(n,i+1);
}
else
printf("\n The sum of series is %f:",sum);
}
```

```
void main()
{
int n;
printf("\nEnter the number of term:");
scanf("%d",&n);
add(n,1);
getch();
}
```

Static variable:

A variable can be declared static using the keyword static like

        static int a;

Internal static variable are those which are declared inside a function. The scope of internal static variables extends up to the end of the function in which they are defined. Therefore, internal static variables are similar to auto variables, except that they remain in existence (alive) throughout the remainder of the program.

Reverse a number using recursive function:

```
#include<stdio.h>
#include<conio.h>
void fib(int);
void main()
{
int n=124;
rev(n);
getch();   }
void rev(int n)
{ static int r=0;
int d;
if(n>0)
{
d=n%10;
r=r*10+d;
n=n/10;
rev(n);
}
else
printf("%d",r);
}
```

Register variable:

We can tell the compiler that a variable should be kept in one of the machine's registers, instead of keeping in the memory (where normal variables are stored). 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 program. This is done as follows:

        register auto int a;

Most compilers allow only int or char type variable to be placed in register.

Pointer:

//refer to book for proper knowledge as this content is from class note slide

Declaring Pointer Variables

Syntax:

data_type *pt_name;

This tells the compiler three things about the variable pt_name.

1. This asterisk(*) tells that the variable pt_name is a pointer variable.
2. pt_name needs a memory location.
3. pt_name points to a variable type of data_type.

Example:

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

Pointer Expression:

C allows us to add integers to or subtract integers from pointers, as well as to subtract one pointer from another.

Eg. **P1+4,p2-2,p1-p2;**

In addition to arithmetic operations discussed, pointer can also be compared using the relational operators. The expressions such **as p1>p2,p1==p2 and p1!=p2 are allowed**. However, any comparisons of pointer that refers to separate and unrelated variables make no sense. Comparisons can be used meaningfully in handling array and string.

We may no use pointer in division or multiplication. Eg**. p1/p2 or p1*p2 or p1/3 are not allowed**. Similarly, two pointer cannot be added. **i.e p1+p2 Is illegal**.

Pointer increment and scalar factor:

When we increment a pointer, its value is increased by the 'length' of the data type that it points to. This length called the scale factor.

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int x,*b=&x;
int y,*a=&y;
int z;
clrscr();
printf("\nEnter the value:");
scanf("%d",&x);
printf("\nEnter the value:");
scanf("%d",&y);
z=x+y-*a**b;
printf("\nThe value of z is:%d",z);
printf("\nOriginal value of b is %u",b);
b++;
printf("\nChanged value of b is %u",b);
printf("\nOriginal value of a is %u",a);
a=a+2;
printf("\nChanged value of a is %u",a);
```

```
getch();
}
```

Pointer and Arrays:

When an array is declared, the compiler allocate a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory allocation. 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:

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

Suppose the base address of x is 1000 then:

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

Example:

Here's the array version:

```c
#include<stdio.h>
#include<conio.h>
void main()
{
int a[]={1,2,3,4,5}; int i;
clrscr();
for(i=0;i<5;i++)
printf("\n%d  %u",a[i],&a[i]);
getch();
}
```

Now let's see how this program would look using pointer notation.

```c
#include<stdio.h>
#include<conio.h>
void main()
{
int a[]={1,2,3,4,5};
int i;
clrscr();
for(i=0;i<5;i++)
printf("\n%d  %u",*(a+i),a+i);
getch();
}
```

Note:
1. *(array + index) is the same as array[index]
2. (array +index) is the same as &array[index]

Pointer to Array in Function:

As an example to explain that a program where each element of array will be added by a constant.

```
#include<stdio.h>
#include<conio.h>
void add(int *,int,int);
void main()
{
int a[]={5,4,3,2,1};
int n,i;
printf("\n Enter the constant to add:");
scanf("%d",&n);
add(a,5,n);
for(i=0;i<5;i++)
printf("\n%d",*(a+i));
getch();
}
void add(int *array,int size,int n)
{
int i;
for(i=0;i<size;i++)
*(array+i)=*(array+i)+n;
}
```

Exception of Array variable over Pointer:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int array[]={1,2,3,4,5};
int i,*p;
p=array;
for(i=1;i<=5;i++)
{
printf("\n%d",*p);
p++;
}
/*
for(i=1;i<=5;i++)
{
printf("\n%d",*array);
array++;// This no possible as array is constant variable its value cannot be changed
}
*p=3; //*/
getch();
}
```

Pointer and character strings:

C support an alternative method to create strings using pointer variables of type char.

Eg. char *string="Sagar"; this create a string for the literal and then stores its address in the pointer variable string. The pointer string now points to the first character of the string "Sagar" as:
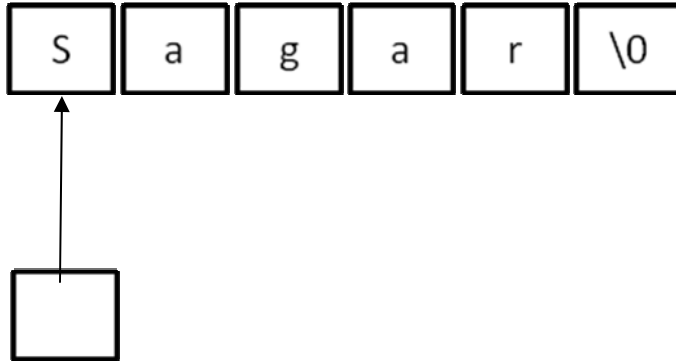
 String

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
char *string,*string1;
clrscr();
printf("\nEnter the string:");
gets(string);
string1=string;
while(*string1!='\0')
{
printf("\n%c is stored at address %u",*string1,string1);
string1++;
}
printf("\nThe length of string is:%d",string1-string);
getch();
}
```

Write a program that reads two different strings. Pass these to a function which reverse the second string and then appends it at the end of the first string. Print the new string from the calling function.

```
#include<stdio.h>
#include<conio.h>
char* rev(char name[20])
{
int l,j;
char r[20];
l=strlen(name);
```

```c
for(l=l-1,j=0;l>=0;l--,j++)
{
r[j]=name[l];
}
r[j]='\0';
return r;
}
void display(char *name1,char *name2)
{
char *n;
n=name1;      //This copy the name1 address to n
printf("\n the copy string is %s",n);
name2=rev(name2);
strcat(n,name2);
printf("\n The new string is:%s",n);
}
void main()
{
char *string1,*string2;
clrscr();
printf("\n Enter the first string:");
scanf("%s",string1);
printf("\n Enter the second string:");
scanf("%s",string2);
display(string1,string2);
getch();
}
```

Structure:

A structure is a collection of variable (of different data types) referenced under one name, providing of convenient means of keeping related information together. A structure definition forms a template that may be used to create structure variable. The general format of a structure definition is as follows:

```
stuct   tag_name
{
        data_type       member1;
        data_type       member2;
        ----------      ------
        -----------     -----
};
```

Array VS Structure:

Both the array and structures are classified as structured data types as they provides mechanism that enables us to access and manipulate data in relatively easy manner. But they are different in number of ways:
1. An array is a collection of related data element of same type. Structure can have elements of different types.
2. An array is derived data type whereas a structure is 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 variable of that type are declared and used.

Rules of initializing structure:

There are few rules to keep in mind while initializing structure variable at compile-time.
1. We cannot initialize individual members inside the structure template.
2. The order of values enclosed in braces must match the order of member in the structure definition.
3. It permitted to have a partial initialization. We can initialize only the first few members and leave the remaining blank. The uninitialized members should be only at the end of the list.
4. The uninitialized members will be assigned default values as follows:
   - **Zero** for integer and floating point numbers.
   - **'\0'** for character and string.

Note:
        Two variables of the same structure type can be copied the same ways as ordinary variables. And C programming doesn't permit any logical operations on structure variables. In case, we need to compare them, we may do so by comparing members individually.

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

Practice:
Create a structure containing real and imaginary as its member. Write a program that uses this structure to input two complex number, multiply then and display the resultant.

```
#include<stdio.h>
#include<conio.h>
struct complex
{
int r;
int m;
};
void main()
{
struct complex c1,c2,c3;
printf("\nEnter the real n imaginary for first complex:");
scanf("%d%d",&c1.r,&c1.m);
printf("\nEnter the real n imaginary for second complex:");
scanf("%d%d",&c2.r,&c2.m);
c3.r=c1.r*c2.r-c1.m*c2.m;
c3.m=c1.r*c2.m+c1.m*c2.r;
printf("\nThe resultant is:");
printf("\nReal is %d",c3.r);
printf("\nImaginary is %d",c3.m);
getch();
}
```

Structure and function:
There are three methods by which the values of a structure can be transferred from one function to another.

1. The first method is to pass each member of the structure as an actual argument of the function call.

The general format of sending a copy of a structure to the called function as:

function_name(structure_variable_name);

The called function takes the following form:

data_type function_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 back to the calling function. 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 functions must be declared in the calling function appropriately.

Pointer and Structure:

Pointer used with simple variable (like type int and char) provides increased power; the ability to do things in your program that are difficult or impossible any other way. The same is true of pointers used with structures.

Defining pointer to Structure:

 Struct sturucture_name * pointer_variable;

Example:
Struct complex s1,*s2;
s2=&s1   //assign the address of the structure variable s1 to s2

Accessing member of structure using pointer variable:
Pointer_variable -> member_name=value;

Example:
s2->r=0;

Note:
The . operator connects a structure with a member of the structure; the -> operator connects a pointer with a member of the structure.

Practice:

Create a structure TIME containing hour, minutes and seconds as its member. Write a program the uses this structure to input start time and stop time to a function. Which returns the sum and difference of the start time and stop time in the main program?

```c
#include<stdio.h>
#include<conio.h>
struct time
{
int hr;
int min;
int sec;
};
void input(struct time *t)
{
printf("\nEnter the hr min and sec:");
scanf("%d%d%d",&t->hr,&t->min,&t->sec);
}
void display(struct time t)
{
printf("\nhr:%d\tmin:%d\tsec:%d\t",t.hr,t.min,t.sec);}
}
struct time add(struct time t1,struct time t2)
{
struct time t3;
t3.sec=t1.sec+t2.sec;
t3.min=t1.min+t2.min;
t3.hr=t1.hr+t2.hr;
t3.min+=t3.sec/60;

t3.sec=t3.sec%60;
t3.hr+=t3.min/60;
t3.min=t3.min%60;
return t3;
}
struct time sub(struct time t1,struct time t2)
{
struct time t3;
t3.sec=t2.sec-t1.sec;
t3.min=t2.min-t1.min;
t3.hr=t2.hr-t1.hr;
if(t3.sec<0)
{
t3.min-=1;
t3.sec+=60;
}
if(t3.min<0)
{
t3.hr-=1;
t3.min=t3.min+60;
}
return t3;
}

void main()
{
struct time t3,t4,t5;
printf("\nEnter the start time in hr min sec:");
input(&t3);
printf("\nEnter the stop time in hr min sec:");
input(&t4);
t5=add(t3,t4);
printf("\naddition of two time is:");
display(t5);
t5=sub(t3,t4);
printf("\nThe substraction of time is:");
display(t5);
getch();
}
```

Array 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 templates 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 structures, each element of the array representing a structure variable. An array of structures is stored inside the memory in the same way as a multi-dimensional array.

Defining an Array of structure:

struct structure array_name[size];

Example
struct student class[50];
This statement provides space in memory for 50 structures of type student

Accessing Members of Array of Structure:

Individual members of a structure in our array of structures are accessed by referring to the structure variable followed by a subscript, followed by the dot operator and ending with the structure element desired.

array_name[i].member_name;

Example:
class[0].roll=61012;

Practice:

Define a structure to hold the roll no. of a student and marks obtained by him in 5 subjects.Declare array to hold the data of 20 students. Pass this to a function that displays the marks of student who has a highest total marks.

```
#include<stdio.h>
#include<conio.h>
struct student
{
int roll;
int marks[5];
int total;
};
void input(struct student s[20])
{
int i,j;
for(i=0;i<20;i++)
{
printf("\nEnter the roll number of student:");
scanf("%d",&s[i].roll);
printf("\nEnter marks for 5 subjects:");
s[i].total=0;
for(j=0;j<5;j++)
{
scanf("%d",&s[i].marks[j]);
s[i].total=s[i].total+s[i].marks[j];
}
}
}
```

```c
void display(struct student s[20])
{
int i,location,max;
max=s[0].total;
location=0;
for(i=1;i<20;i++)
{
if(max<s[i].total)          //serching highest marks location
{
max=s[i].total;
location=i;
}
}
printf("\nRecord of student who score highest marks:");
printf("\nRoll number of student:%d",s[location].roll);
printf("\nEnter marks for 5 subjects:");
for(i=0;i<5;i++)
{
printf("\nMarks in  %d subject%d",i+1,s[location].marks[i]);
}

}
void main()
{
struct student civil[5];
input(civil);
display(civil);
getch();
}
```

Create a structure STUDENT containing name, symbol number, name of 6 subjects, mark of each subject and total mar as its members. Write a program that uses this structure and reads data for a student and gives the total marks as the output.
//Refer to above program and display only total marks of all the student

Write a program to compute any two instant of distances in a format 1feet=12inches using structure. Build functions to add and subtract given distances and display the results in the main function.

```c
#include<stdio.h>
#include<conio.h>
struct distance
{
int feet;
int inch;
};
```

```c
void input(struct distance *d)
{
printf("\nEnter the feet and inches:");
scanf("%d%d",&d->feet,&d->inch);
}
void display(struct distance d)
{
printf("\nFeet:%d \t Inche:%d\t",d.feet,d.inch);
}

struct distance add(struct distance d1,struct distance d2)
{
struct distance d3;
d3.feet=d1.feet+d2.feet;
d3.inch=d1.inch+d2.inch;
d3.feet+=d3.inch/12;
d3.inch=d3.inch%60;
return d3;
}
struct distance sub(struct distance d1,struct distance d2)
{
struct distance d3;
d3.inch=d2.inch-d1.inch;
d3.feet=d2.feet-d1.feet;
if(d3.inch<0)
{
d3.feet=d3.feet-1;
d3.inch=d3.inch+12;
}
return d3;
}
void main()
{
struct distance t3,t4,t5;
printf("\nEnter the starting distance in feet and inches:");
input(&t3);
printf("\nEnter the stoping distance  in feet and inches:");
input(&t4);
t5=add(t3,t4);
printf("\naddition of two time is:");
display(t5);
t5=sub(t3,t4);
printf("\nThe substraction of time is:");
display(t5);
getch();
}
```

Nested Structure:

Structure within a structure means *nesting of structure*. Let us consider following example to explain it.

```c
#include<stdio.h>
#include<conio.h>
struct student
{
char name[10];
struct date
{
int dd;
int mm;
int yy;
}d;
char loc[10];
char city[10];
};
void input(struct student *t)
{
printf("\nEnter Name:");
scanf("%s",&t->name);
printf("\nEnter the birth date (dd,mm,yy):");
scanf("%d",&t->d.dd);
scanf("%d",&t->d.mm);
scanf("%d",&t->d.yy);
printf("\nEnter the location:");
scanf("%s",&t->loc);
printf("\nEnter the city:");
scanf("%s",&t->city);
}

void display(struct student t)
{
printf("\nName:%s",t.name);
printf("\nBirth date:%d/%d/%d",t.d.dd,t.d.mm,t.d.yy);
printf("\nlocation:%s",t.loc);
printf("\nCity:%s",t.city);
}
void main()
{
struct student t1;
clrscr();
input(&t1);
display(t1);
getch();
}
```

In above example, the student structure contains a member named d, which itself is a structure with three members. The members contained in the inner structure namely dd, mm and yy can be referred to as:

        t1.d.dd;
        t1.d.mm;
        t1.d.yy;

Note:

An inner-most member in a nested structure can be accessed by chaining all the concerned structure variables (from outer-most to inner-most) with the member using dot operator.

We can also use tag name to define inner structures. Example

```
struct date
{
int dd;
int mm;
int yy;
};
struct student
{
char name[10];
struct date d;
char loc[10];
char city[10];
};
```

Access all of its members same as above in a given program
//for further example refer to class note

File:

Until now we have been using the functions such as scanf ,printf,getchar(),putchar() etc to read and write data. These are console oriented I/O functions, which always use the terminal (keyboard and screen) as the target place. This works fine as long as the data is small. However, 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 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. A file is a place on the disk where a group of related data is stored. The basic file operations performed are:

- Naming a file,
- Opening a file,
- Reading data from a file,
- Writing data from a file,
- Closing a file.

Defining and opening a file:

If we want to perform operation in a file in the secondary memory, we must specify certain things about the file, to the operating system. They include:

1. Filename
2. Data structure
3. Purpose

Filename is a string of characters that make up a valid filename for the operating system. Example:

Student.txt

Employ.dat

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.

When we open a file, we must specify what we want to do with the file. For example, we may write data to the file or read the already existing data.

Following is the general format for declaring and opening a file:

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

the first statement declares the variable fp as a "pointer to the data type FILE". The second statement opens the file named filename and assigns an identifier to the FILE type pointer fp. This pointer, which contains all the information about the file is subsequently used as a communication link between the system and the program.

File opening Modes:

The different types of file opening modes are:

"r": This mode open file for reading only. While opening the file in this mode if file exists, then the file is opened with the current content safe otherwise and error occurs i.e fopen() returns NULL. In this mode, FILE pointer points to the starting of bite of file.

"w": This mode open file for writing only. In this mode, a file with specified name is created if the file does not exist. The contents are deleted, if the file already exists.

"a": This mode open file for appending (or adding) data to it. When the file is opened in this mode the file is opened with the current content safe. A file with the specified name is created if the file does not exist. When the file is opened in this mode, FILE pointer points to end of file.

"r+": This mode opens the file for reading existing content, writing new contents and modifying existing content of the file.

"w+" This mode opens the file for writing new content, reading them back and modifying the existing content of the file.

"a+": This mode opens the file for appending new content to the end of file, reading existing content from the file, But cannot modify existing contents.

Error handling during I/O operation:

It is possible that an error may occur during I/O operations on a file. Typical error situation include:

1. Trying to read beyond the end of-file mark.
2. Device Overflow.
3. Trying to use a file that has not been opened.
4. Trying to perform an operation on a file, when the file is opened for another type of operation.
5. Opening a file with an invalid filename.
6. Attempting to write a write-protected file.

If we fail to check such read and write errors, a program may behave abnormally when an error occurs. An unchecked error may result in premature termination of the program or incorrect output. We have two status-inquiry library functions: **feof** and **ferror** that can help up detect I/O errors in the file.

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

if(feof(fp))
printf("End of data");

would display the message "End of data" on reaching the end of file condition.

The ferror function report 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 return zero otherwise. The statement:

if(ferror(fp)!=0)
printf("An error has occurred:");

Would print the error message, if the reading is no successful.

Random access to files:

This can be achieved with the help of the function fseek, ftell and rewind available in I/O library. ftell takes a file pointer and return a number of type long, that corresponds to the current position. This function is useful in saving the current position of a file, which can be used later in the program. It takes the following form:

N= fte11(fp);

n would give the relative offset (in bytes) if the current position. This means that n bytes have already been read(or written).

Rewind takes a file pointer and resets the position to the start of the file. For example, the statement.

rewind nd(fp);
n=fte11(fp);

Would assign 0 to n because the file position has been set to the start of the file be rewind. Remember , the first byte in the file is numbered as 0, second as 1, and so on. This function helps us in reading a file more than once, without having to close and open the file. Remember that whenever a file is opened for reading or writing , a rewind is done implicitly.

fseek function is used to move the file position to a desired location within the file. It takes the following form:

fseek(file_ptr,offset,position);

file_ptr is a pointer to the file concerned, offset is a number or variable of type long and position is an integer number . the offset specifies the number of position (byte) to be moved from the location specified by position. The position can take one of the following three values:

value   meaning
0   Beginning of file
1   Current position
2   End of file

Example:
fseek(fp,0L,0) meaning go to begining

//refer to class note for example

1.1 FORTAN character set

The following is the set of character used in FORTRAN.
Capital alphabets:A-Z
Digit:0-9
Symbols:+ - / * . , ' $ ( )

1.2 Fortran Constant:

A number on a string of FORTRAN characters is called a constant. Number are called numeric constant. 243,-21,3.14 etc are some numeric constants. A string of character is called a character constant.
There are two types of Numeric Constant:
1. Integer constant
2. Real Constant

INTEGER Constant:

Integer written without decimal point is called fixed point constant or integer constants. The following rule defines a integer constant.
1. An integer constant is formed with digit 1,2….9,0 and the symbols +or-. No other character should occur in fixed point constant.
2. The symbol + or – can occur only at the left most end of the number.
3. For negative number – symbol is used at the most position and for the positive number + symbol is used. If no symbol occurs the number is assumed to be positive.
4. Any intermediate blank space in a constant has no effect.

Real Constant:

Any number written with one decimal point is called a floating point constant of real constant. A real constant can be expressed in any one of the following two forms
   i)   Fractional form
  ii)   Exponential form

The following rules apply for the real constant in fractional form.
1. A real constant is written in the decimal form with the digitals 0,1,….9 and the decimal point.
2. There is one and only one decimal point. This means that there must be one decimal point and there should not be more than one decimal points.
3. A negative number must be written with the –symbol.
4. For a positive number the + symbol is optional. If there is no sign, the number is assumed to be positive.
5. No special symbols such as *,etc are allowed in a real constant.

Eg. -0.17 1.4 +1.5 0.0

Exponent Form:
The following rules apply to the exponential form of real constant
1. The exponential form has two parts
   (i) mantissa
   (ii) exponent
2. The alphabet E is written in between the mantissa and the exponent. The general form of the exponential floating point constant is
   Mantissa E exponent
3. The mantissa must be valid real constant in fractional form. All the rules of the fractional form applied to the mantissa.
4. The exponent is always an integer with at most two digit.
5. The exponent can have sign(+ or -). If there is no sign, the exponent is assumed to be positive.

Character constant:
Character constant are any string of character enclosed within quotes Notes the only single quote('). Note only single quote(') must be used and not the double quotes("). The maximum length of a character is 127.

Rules of naming variable:
1. Variable name can be form one to six character in length.
2. The first character of the variable name must be an alphabet and the succeeding characters can be alphabets or numeric digits.
3. No special is allowed in a variable name.
4. FORTRAN verbs which have special meaning in FORTRAN cannot be used as variable names.

Type of variable:
Depending upon the data contained in the memory location, the variable are classified as integer, real and character variables.
There are three type of variables
1. Integer variables
2. Real variables
3. Character variables
A variable which acquires only integer values is called an integer value.
A variable which acquires only real values is called a real variable.
A variable which acquires only character string is called a character variable.

Variable Type Declaration:
it is not compulsory to declare the variables. If the variables are not declared, the variable names starting with letters I,J,K,L,M or N are considered to be integer variable and other as real variables. For character variables the declaration is compulsory.
The general format for declaring integer variables is shown below:
   INTEGER list of variables
Eg. INTEGER I,SUM,AREA
The general format is shown below:
REAL list of variables

Declaration statement of the character variable also contains the length of the string that the variable can hold. If the length is not specified, the length considered as 1.

If the name is a character variable which will hold the name of length at most 25 character then must be declared as follows:

CHARACTER name *25

CHARACTER *10 A,B ,then A and B are both character variable to hold data of length at the most 10 each.

Implicit type declaration:

The general format is given below:

IMPLICIT type(….),type(…)..

Example

IMPLICIT INTEGER(A)

The above statement declares that all the variable name starting with the alphabet A are integer variables. This does not affect the general rule that the variables starting with the letters I,J,K,L,M or N are also integer variable. So the above statement declare that all the variable starting with the letters A,I,J,K,L,M or N are integer variable in the program.

The following is the rule of Hierarchy arithmetic operators in FORTRAN

1. Any expression within parenthesis is first evaluated.
2. Exponentiation is given the top priority and evaluated first in an expression.
3. Multiplication and division are given the next priority.
4. Addition and subtraction are performed finally.

FORMAT specification:

When data are to be input or the result to be output, we fully mention the type of the data (integer , real or character) and also its size. The specification of the types of the data and its size is called FORMAT specification. This is a non-executable statement.

FORMAT statement:

The general form of a FORMAT statement is

n FORMAT($s_1,s_2….s_r$)

where n is the statement number

$s_1,s_2,……s_r$ are the format specifications

Rules:

1. The format specifications $s_1….s_r$ must be enclosed within parenthesis.
2. The specification $s_1….sr$ must be separated by commas.
3. Every FORMAT statement must be given a statement number.

Carriage control:

In any output the first character of the output is lost. That is considered as the carriage control. So, the first character of the output must be made a blank space so that the loss does not affect the output.

I format:

The symbol I is used to denote the integer quantities. The general I format specification is

I w

Where w is the width of the integer data. In the width one space is allotted for the sign.
Example:
-215 458
This data can be describe by the format statement

FORMAT(I4,I4)

This can be also be written as

FORMAT(2I4)

F Format:
The symbol F is used to denote the real data expressed in decimal form. The general form of the F format is

F w.d

Where

w is the total width of the number
d is the decimal width

c. Temperature conversion
```
      REAL F,C
      WRITE(*,1)
1     FORMAT(1X, 'Enter the temperatrue',1X/,'THE VALUE')
      READ(*,2)C
2     FORMAT(F10.3)
      F=C*1.8+32.0
      WRITE(*,3)F
3     FORMAT(1X,'Temperature',f10.3)
      STOP
      END
```

Unformatted Input and output
c. Temperature conversion
```
      REAL F,C
      PRINT *, 'Enter the temperatrue'
      READ*,C
      F=C*1.8+32.0
      PRINT *,'Temperature',F
      STOP
      END
```

Control Statement:
1: Unconditional GOTO statement:
This statement is used to transfer the control to any other statement unconditionally. The general form is
 GO TO n
where n is the statement number to which the control must be transferred.
The blank between GO and TO is optional.
for example.

GO TO 35

In the GO TO statement, the number referenced cannot be a variable
example:

GO TO I is not valid.

Computed GO TO statement:

The computed GO TO statement cause the transfer of control depending upon value of an integer variable. the destination
(where to go) is decided by the value in the integer variable.
the general form is
GO TO $(n_1,n_2,......n_k)$, i
where i is the integer variable
$n_1,n_2....n_k$ are statement numbers.

Arithmetic IF statement:

This statement is used to transfer the control depending upon the value of an expression whether negative, zero or positive.
The general form is

If (expression) n_1,n_2,n_3

Where
(expression) is a valid FORTRAN arithmetic expression enclosed within parenthesis. n_1, n_2,n_3 are statement numbers. The value of the expression is evaluated first. If the value is *negative* the control goes to statement number n_1, if it is *zero,* it goes to n_2 and if it is *positive*, it goes to n_3.

PRACTICE:
$1+X^1+X^2+X^3+...$
```
C     SERIES USING ARITHMETIC IF
      INTEGER P
      PRINT *,'ENTER THE VALUE OF N AND X'
      READ *,N,X
      I=1
      P=0
      SUM=0.0
30    IF(I-N)10,10,20
10    SUM=SUM+X**P
      P=P+1
      I=I+1
      GOTO 30
20    PRINT *,'SUM IS ',SUM
      STOP
      END
```

Logical IF statement:

The logical if condition checks any given logical condition and transfer the control accordingly.
The general form of the statement is

   IF(condn) Statement

Where condn is a logical condition, statement is an executable statement. If the condition is *true* the **statement** is executed and then goes to the **next statement**. If the condition is **false** the control goes to the **next statement**.

Logical Condition
The following are the relational operators and their symbols used in FORTRAN

PRACTICE:

Write a program to display FIBNOACCI series until term value is less than 500 in FORTRAN.

```
C     FIBONACCI SERIES
      INTEGER F,S,T
      PRINT *,'ENTER THE NUMBER OF TERMS'
      F=0
      S=1
      WRITE (*,*) F,S
20    T=F+S
      IF(T.GT.500) GOTO 10
      WRITE (*,*) T
      F=S
      S=T
      GOTO 20
10    STOP
      END
```

IF-THEN-ELSE statement:
The IF-THEN-ELSE statement is more useful and easy to handle than the logical if statement. The general form of the statement is

      IF(condn) THEN
            S1,
            S2,

```
                    :
                    :
        ELSE
                S1'
                S2'
                 :
                 :
        ENDIF
```
Where condn is a logical condition,

  S1,S2….. are the statement to be executed when cond is true

  S1',S2'….. are the statement to be executed when cond is false.

Here the ELSE clause is optional.

That is,

```
        IF (condn) THEN
        …………
        ……….
        ELSEIF
```

PRACTICE:

Write a program to display Fibonacci series up to N term in FORTRAN.

```
C     FIBONACCI SERIES
      INTEGER F,S,T
      PRINT *,'ENTER THE NUMBER OF TERMS'
      READ (*,*) N
      F=0
      S=1
      IF(N.EQ.1) THEN
      WRITE (*,*) F
      ELSE
      WRITE (*,*) F,S
      DO 10 I=1,N-2,1
      T=F+S
      WRITE (*,*) T
      F=S
      S=T
10    CONTINUE
      ENDIF
      STOP
      END
```

Write a program to find the HCF for any two number entered by user in FORTRAN

```
C     WRITE A PROGRAM TO FIND HCF (GREATEST COMMON FACTOR)

      READ *,I1,I2
3     IR=I1-I1/I2*I2
```

```
        IF(IR.EQ.0) THEN
        PRINT *,'THE HCF IS',I2
        STOP
        ENDIF
        I1=I2
        I2=IR
        GOTO 3
        STOP
        END
```

Nested IF-THEN-ELSE:
In certain cases we may have to use one IF-THEN-ELSE structure within another IF-THEN.
This is called Nested usage of IF-THEN-ELSE.

```
IF (condn) THEN
        ………………..
        IF( condn) THEN
                …………..
                IF(condn) THEN
                ………………
                ELSE
                ……………..
                ENDIF
                ……………..
        ELSE
                ………………
        ENDIF
ELSE
        …………..
        IF (condn) THEN
        ……………..
        ELSE
        …………….
        ENDIF
        …………..
ENDIF
```

ELSE-IF-THEN structure:
The general format is
```
        IF (condn) THEN
                …………
        ELSE IF (condn) THEN
                …………..
```

ELSE IF(condn) THEN

………..

ELSE

………..

ENDIF

Write a program to check whether a given number is Armstrong or not in FORTRAN.

```
C     PROGRAM TO FIND ARMSTRONG NUMBER OR NOT
      PRINT *, 'ENTER THE NUMBER'
      READ *,N
      TEMP=N
      SUM=0
60    IF(N.GT.0) THEN
      IR=N-N/10*10;
      SUM=SUM+IR**3
      N=N/10
      GOTO  60
      ENDIF
      IF(SUM.EQ.TEMP)  THEN
      PRINT *,TEMP,'IS ARMSTORNG'
      ELSE
      PRINT *,TEMP,'IS NOT ARMSTRONG'
      ENDIF
      STOP
      END
```

DO LOOPS:

The **DO LOOP** is used whenever a particular job is to be repeated number of times. The general form of the DO loop is

DO I v=v1, v2, v3

…………………

…………………

………………..

Where **I** is the last statement in the DO loop.

v is an integer variable. It is called the ruining variable for the DO loop.

v1 is an integer variable or constant. This is the initial values for the running variable.

v2 is an integer variable or constant quantity. The running variable will not take a value beyond **v2**.

v3 is an integer variable or constant. This is the increment value. If the increment is **1** then this can be omitted. This is also permitted to be an expression (real or integer). The following is the work done.

1. assign **v=v1**
2. execute all the statements between **DO** and statement 1.
3. Increment v by **v3** i.e. **v=v+v3**

4. If **v>v2**, then **stop** otherwise go to step-2

PRACTICE

```
C      PRIME NUMBER
       INTEGER COUNT
       READ *,NUM
       COUNT=0
       DO 10 I=1,NUM,1
       IF(MOD(NUM,I).EQ.0) THEN
       COUNT=COUNT+1
       ENDIF
10     CONTINUE
       IF(COUNT.EQ.2) THEN
       PRINT *,NUM,'IS PRIME'
       ELSE
       PRINT *,NUM,'IS NOT PRIME'
       ENDIF
       STOP
       END
```

Rules for subscripted variables:
The following rules may be strictly followed while defining the subscripted variables.
1. The subscript is always an integer.
2. The subscript value cannot be negative.
3. The subscript must be given within parenthesis after the variable name.
4. If there is more than one subscript, they are separated by commas.

Array declarative statement:
The DIMENSION statement must be occur before the first occurrence of the subscripted variable. The general format is

DIMENSION variable (max value of subscript)

Consider the dimension statement

DIMENSION X(10),M(4,5)

This statement declares that X is a one dimensional array with the subscript varying from 1 to 10 and M is a two dimensional array with the first subscript varying from 1 to 4 and the second subscript form 1 to 5
Instead of DIMENSION statement, the following statements can be also be used to declare the arrays.
1. INTEGER statement
2. REAL statement
3. CHARACTER statement

Practice:

Write a program to find the range from the list of 5 element using array in FORTRAN.

```
C      RANGE OF MATRIX
       INTEGER A(5),L,S
       PRINT *,'ENTER THE ELEMENT TO ARRAY'
```

```fortran
      DO 10 I=1,5,1
      READ *,A(I)
10    CONTINUE
      L=A(1)
      S=A(1)
      DO 20 I=2,5,1
      IF(L.LT.A(I)) THEN
      L=A(I)
      ENDIF
      IF(S.GT.A(I)) THEN
      S=A(I)
      ENDIF
20    CONTINUE
      PRINT *,'LAGREST ELEMENT:',L
      PRINT *,'SMALEST ELEMENT:',S
      PRINT *,'RANGE:',L-S
      STOP
      END
```

IMPLIED DO loop:

Suppose we want to read all the entries of a one dimensional array A with array length 5. The following DO loop will do the reading operation.

```fortran
      INTEGER A(5)
      DO 10 I=1,5,1
      READ *,A(I)
10    CONTINUE
```

FORTRAN also has the facility of reading or writing the entire array with one statement. This statement is called **the implied DO loop**. For example the following is the implied DO loop which read the Array A of length 5.

```fortran
      READ (*,*) (A(I),I=1,5)
```

Similarly the following is the implied DO loop to print the elements of the array.

```fortran
      WRITE (*,*) (A(I),I=1,5)
```

PRACTICE:

Write a program to take input to one dimensional array and display.

```fortran
      INTEGER IN(5);
      PRINT *,'ENTER THE ELEMENTS TO ARRAY'
      READ (*,*) (IN(I),I=1,5)
      PRINT *,'THE ELEMENTS OF ARRAY IS:'
      WRITE (*,*) (IN(I),I=1,5)
      STOP
      END
```

Write a program to arrange one dimensional array in ascending order in FORTRAN.

```fortran
C     ASCENDING ORDER OF MATRIX
```

```
      INTEGER A(5),TEMP
      PRINT *,'ENTER THE ELEMENT TO ARRAY'
      READ (*,*) (A(I),I=1,5)

      DO 20 I=1,4,1
      DO 20 J=I+1,5,1
      IF(A(I).GT.A(J)) THEN
        TEMP=A(I)
        A(I)=A(J)
        A(J)=TEMP
      ENDIF
20    CONTINUE
      PRINT *,'THE ELEMENT TO ARRAY'
      WRITE (*,*) (A(I),I=1,5)
      STOP
      END
```

Implied DO loop for multidimensional array:

The implied DO loop can also be used for multidimensional arrays. For example, consider a two dimensional array A(I,J) where I varies from 1 to 3 and J varies from 1 to 3. If we want to read all the elements of the array, we write the nested DO loop as follows.

```
      INTEGER A(3,3)
      DO 10 I=1,3,1
      DO 10 J=1,3
      READ *,A(I,J)
10    CONTINUE
```

But we can use the implied DO loop and write this as a single statements as follows.
```
      READ (*,*) ((A(I,J),J=1,3),I=1,3)
```
Notice that the outer loop has I as running variable and the inner loop has J.

PRACTICE:

Write a program to take input and display two dimensional arrays.
```
C     input and display elements OF MATRIX
      INTEGER IN(3,3);
      PRINT *,'ENTER THE ELEMENTS TO ARRAY'
      READ (*,*) ((IN(I,J),J=1,3),I=1,3)
      PRINT *,'THE ELEMENTS OF ARRAY IS:'
      WRITE (*,*) ((IN(I,J),J=1,3),I=1,3)
      STOP
      END
```

Write a program to add two matrixes and display the resultant matrix in FORTRAN.

```
C      ADDING CORRESPONDING ELEMENT OF TWO MATRIX TO THIRD
MATRIX
      INTEGER A(5,5),B(5,5),C(5,5),R1,R2,C1,C2
      PRINT *,'ENTER THE ROW AND COLUMN OF FIRST MATRIX'
      READ *,R1,C1
      PRINT *,'ENTER THE ROW AND COLUMN OF SECOND MATRIX'
      READ *,R2,C2
      IF((R1.EQ.R2).AND.(C1.EQ.C2)) THEN
      PRINT *,'ENTER ELEMENT TO FIRST ARRAY'
      DO 10 I=1,R1,1
      DO 10 J=1,C1
      READ *,A(I,J)
10       CONTINUE
       PRINT *,'ENTER ELEMENT TO SECOND ARRAY'
      DO 20 I=1,R1,1
      DO 20 J=1,C1
      READ *,B(I,J)
20       CONTINUE
      DO 30 I=1,R1,1
      DO 30 J=1,C1,1
       C(I,J)=A(I,J)+B(I,J)
30       CONTINUE
      PRINT *,'RESULTANT ARRAY IS'
      DO 40 I=1,R1,1
      DO 40 J=1,C1
      PRINT *,C(I,J)
40       CONTINUE
      ELSE
      PRINT *,'ADDITION IS NOT POSSIBLE'
      ENDIF
      STOP
      END
```

Write a program perform matrix multiplication using nested DO loop

```
C      MATRIX MULTIPICATION
      INTEGER A(5,5),B(5,5),C(5,5),R1,R2,C1,C2
      PRINT *,'ENTER THE ROW AND COLUMN OF FIRST MATRIX'
      READ *,R1,C1
      PRINT *,'ENTER THE ROW AND COLUMN OF SECOND MATRIX'
      READ *,R2,C2
      IF(R2.EQ.C1) THEN
      PRINT *,'ENTER ELEMENT TO FIRST ARRAY'
      DO 10 I=1,R1,1
      DO 10 J=1,C1
      READ *,A(I,J)
```

```fortran
10      CONTINUE
        PRINT *,'ENTER ELEMENT TO SECOND ARRAY'
        DO 20 I=1,R2,1
        DO 20 J=1,C2
        READ *,B(I,J)
20      CONTINUE
        DO 30 I=1,R1,1
        DO 30 J=1,C2,1
        C(I,J)=0
        DO 30 K=1,R2,1
        C(I,J)=C(I,J)+A(I,K)*B(K,J)
30      CONTINUE
        PRINT *,'RESULTANT ARRAY IS'
        DO 40 I=1,R1,1
        DO 40 J=1,C2
        PRINT *,C(I,J)
40      CONTINUE
        ELSE
        PRINT *,'MULTIPICATION IS NOT POSSIBLE'
        ENDIF
        STOP
        END
```

Write a program to perform matrix multiplication using implied DO loop

```fortran
C       MATRIX MULTIPICATION
        INTEGER A(5,5),B(5,5),C(5,5),R1,R2,C1,C2
        PRINT *,'ENTER THE ROW AND COLUMN OF FIRST MATRIX'
        READ *,R1,C1
        PRINT *,'ENTER THE ROW AND COLUMN OF SECOND MATRIX'
        READ *,R2,C2
        IF(R2.EQ.C1) THEN
        PRINT *,'ENTER ELEMENT TO FIRST ARRAY'
        READ (*,*)((A(I,J),J= 1,C1),I=1,R1)
        PRINT *,'ENTER ELEMENT TO SECOND ARRAY'
        READ (*,*)((B(I,J),J=1,C2),I=1,R2)
        DO 30 I=1,R1,1
        DO 30 J=1,C2,1
        C(I,J)=0
        DO 30 K=1,C1,1
        C(I,J)=C(I,J)+A(I,K)*B(K,J)
30      CONTINUE
        PRINT *,'RESULTANT ARRAY IS'
        WRITE(*,*)((C(I,J),J=1,C2),I=1,R1)
        ELSE
        PRINT *,'MULTIPICATION IS NOT POSSIBLE'
```

```
        ENDIF
        STOP
        END
```

Write a program to sum the following series up to n term:

$1-x^2/2!+x^4/4!-x^6/6!......$

```
 PRINT *,'ENTET THE VALUE OF N AND X'
        READ *,N,X
        P=0
        SIGN=-1
        FACT=0
        SUM=0.0

        DO 10,I=1,N,1
        SIGN=SIGN*-1

        DEN=1
        DO 20,J=1,FACT,1
        DEN=DEN*J
20      CONTINUE

        NUM=SIGN*X**P
        SUM=SUM+NUM/DEN
        FACT=FACT+2
        P=P+2
10      CONTINUE

        PRINT *,'THE SUM IS:',SUM
        STOP
        END
```

Additional Topic:
Look for
Preprocessor and Dynamic Memory Allocation