PROGRAMMING FOR PROBLEM SOLVING

DIGITAL NOTES

B.TECH
(I YEAR II SEM)

DEPARTMENT OF CSE & IT

PROGRAMMING FOR PROBLEM SOLVING SYLLABUS

Course Objectives

- To understand the various steps in Program development.
- To understand the basic concepts in C Programming Language.
- To learn how to write modular and readable C Programs
- To learn to write programs (using structured programming approach) in C to solve problems.

Course Outcomes:

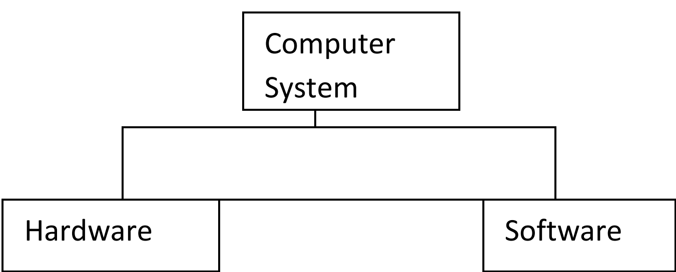
- Demonstrate the basic knowledge of computer hardware and software.
- To formulate simple algorithms for arithmetic and logical problems.
- To translate the algorithms to programs (in C language).
- To test and execute the programs and correct syntax and logical errors.
- Ability to apply solving and logical skills to programming in C language and also in other languages.

UNIT - I

Introduction to Computing:

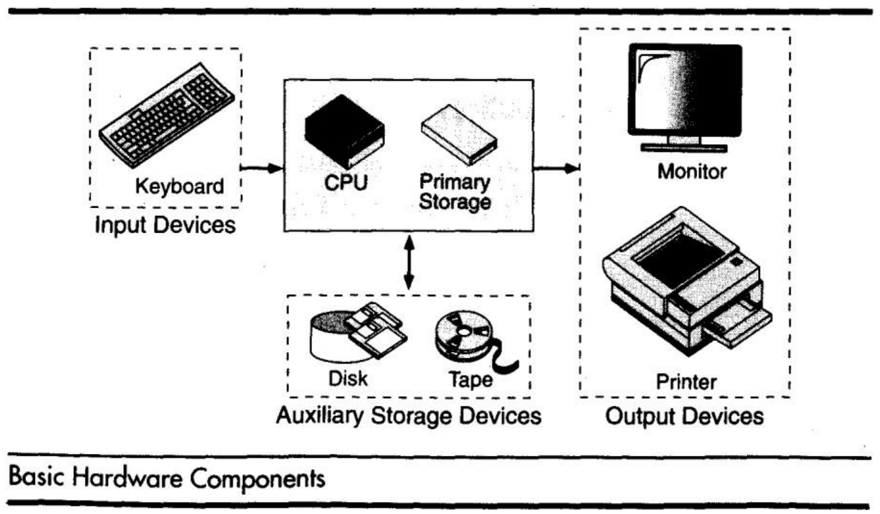
Computer Systems:

A computer is a system made of two major components: hardware and software. The computer hardware is the physical equipment. The software is the collection of programs (instructions) that allow the hardware to do its job.



Computer Hardware

The hardware component of the computer system consists of five parts: input devices, central processing unit (CPU) ,primary storage, output devices, and auxiliary storage devices.



Basic Hardware Components

The **input device** is usually a keyboard where programs and data are entered into the computers. Examples of other input devices include a mouse, a pen or stylus, a touch screen, or an audio input unit.

The **central processing unit (CPU**) is responsible for executing instructions such as arithmetic calculations,comparisons among data, and movement of data inside the system. Today's computers may have one ,two, or more CPUs .**Primary storage** ,also known as main memory, is a place where the programs and data are stored temporarily during processing. The data in primary storage are erased when we turn off a personal computer or when we log off from a time-sharing system.

The **output device** is usually a monitor or a printer to show output. If the output is shown on the monitor, we say we have a **soft copy**. If it is printed on the printer, we say we have a hard copy.

Auxiliary storage, also known as **secondary storage**, is used for both input and output. It is the place where the programs and data are stored permanently. When we turn off the computer, or programs and data remain in the secondary storage, ready for the next time we need them.

Computer Software

Computer software is divided in to two broad categories: system software and application software .System software manages the computer resources .It provides the interface between the hardware and the users. Application software, on the other hand is directly responsible for helping users solve their problems.

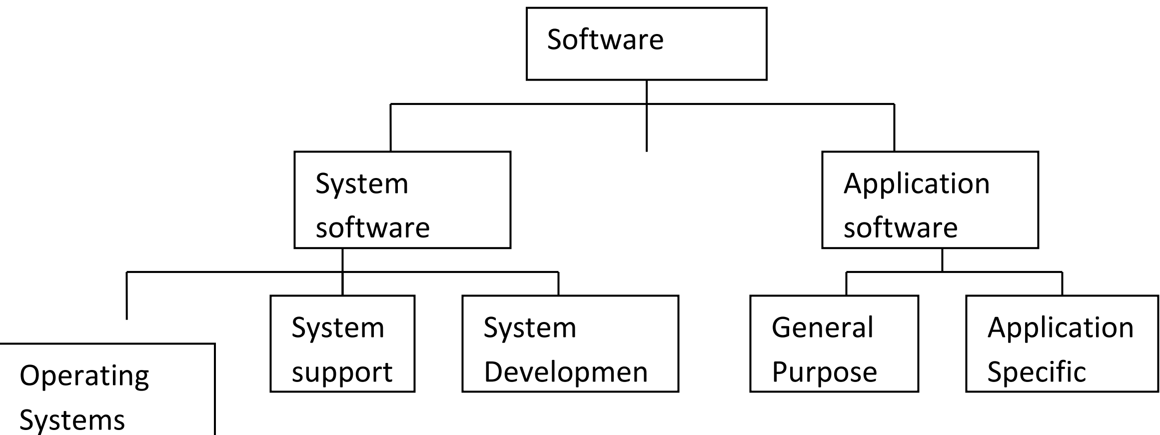


Fig: Types of software

System Software:

System software consists of programs that manage the hardware resources of a computer and perform required information processing tasks. These programs are divided into three classes: the operating system, system support, and system development.

The **operating system** provides services such as a user interface, file and database access, and interfaces to communication systems such as Internet protocols. The primary purpose of this software is to keep the system operating in an efficient manner while allowing the users access to the system.

System support software provides system utilities and other operating services. Examples of system utilities are sort programs and disk format programs. Operating services consists of programs that provide performance statistics for the operational staff and security monitors to protect the system and data.

The last system software category ,**system development software**, includes the language translators that convert programs into machine language for execution ,debugging tools to ensure that the programs are error free and computer –assisted software engineering(CASE) systems.

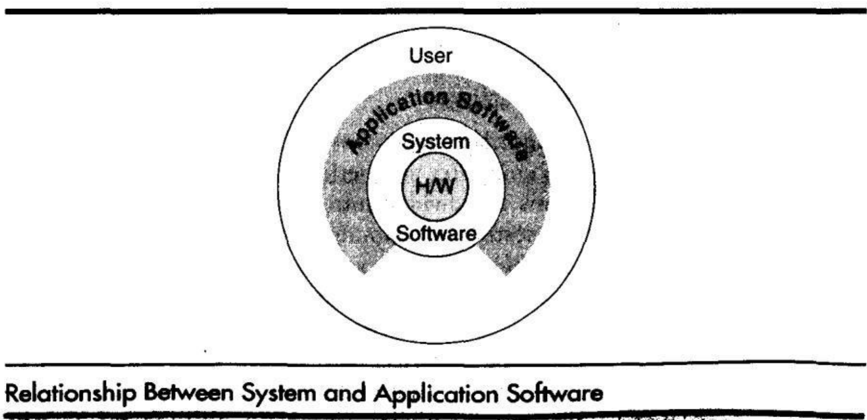
Application software

Application software is broken in to two classes :general-purpose software and application –specific software. **General purpose software** is purchased from a software developer and can be used for more than one application. Examples of general purpose software include word processors ,database management systems ,and computer aided design systems. They are labeled general purpose because they can solve a variety of user computing problems.

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

A general ledger system used by accountants and a material requirements planning system used by a manufacturing organization are examples of application-specific software. They can be used only for the task for which they were designed they cannot be used for other generalized tasks.

The relation ship between system and application software is shown in fig-2.In this figure, each circle represents an interface point .The inner core is hard ware. The user is represented by the out layer. To work with the system,the typical user uses some form of application software. The application software in turn interacts with the operating system ,which is apart of the system software layer. The system software provides the direct interaction with the hard ware. The opening at the bottom of the figure is the path followed by the user who interacts directly with the operating system when necessary.



Relationship Between System and Application Software

Computer Languages:

To write a program for a computer, we must use a **computer language.** Over the years computer languages have evolved from machine languages to natural languages.

1940's Machine level Languages

1950's Symbolic Languages

1960's High-Level Languages

Machine Languages

In the earliest days of computers, the only programming languages available were machine languages. Each computer has its own machine language, which is made of streams of 0's and 1's.

Instructions in machine language must be in streams of 0's and 1's because the internal circuits of a computer are made of switches transistors and other electronic devices that can be in one of two states: off or on. The off state is represented by 0 , the on state is represented by 1.

The only language understood by computer hardware is machine language.

Symbolic Languages:

In early 1950's Admiral Grace Hopper, A mathematician and naval officer developed the concept of a special computer program that would convert programs into machine language.

The early programming languages simply mirror to the machine languages using symbols of mnemonics to represent the various machine language instructions because they used symbols, these languages were known as symbolic languages.

Computer does not understand symbolic language it must be translated to the machine language. A special program called assembler translates symbolic code into machine language. Because symbolic languages had to be assembled into machine language they soon became known as assembly languages.

Symbolic language uses symbols or mnemonics to represent the various ,machine language instructions.

High Level Languages:

Symbolic languages greatly improved programming effificiency; they still required programmers to concentrate on the hardware that they were using. Working with symbolic languages was also very tedious because each machine instruction has to be individually coded. The desire to improve programmer efficiency and to change the focus from the computer to the problem being solved led to the development of high-level language.

High level languages are portable to many different computers, allowing the programmer to concentrate on the application problem at hand rather than the intricacies of the computer. High-level languages are designed to relieve the programmer from the details of the assembly language. High level languages share one thing with symbolic languages, They must be converted into machine language. The process of converting them is known as compilation.

The first widely used high-level languages, FORTRAN (FORmula TRANslation)was created by John Backus and an IBM team in 1957;it is still widely used today in scientific and engineering applications. After FORTRAN was COBOL(Common Business-Oriented Language). Admiral Hopper was played a key role in the development of the COBOL Business language.

C is a high-level language used for system software and new application code.

ALGORITHM:

Algorithms was developed by an Arab mathematician. It is chalked out step-by-step approach to solve a given problem. It is represented in an English like language and has some mathematical symbols like ->, >, <, = etc. To solve a given problem or to write a program you approach towards solution of the problem in a systematic, disciplined, non-adhoc, step-by-step way is called Algorithmic approach. Algorithm is a penned strategy(to write) to find a solution.

Example: Algorithm/pseudo code to add two numbers

Step 1: Start
Step 2:Read the two numbers in to a,b
Step 3: c=a+b
Step 4: write/print c
Step 5: Stop.

FLOW CHART :

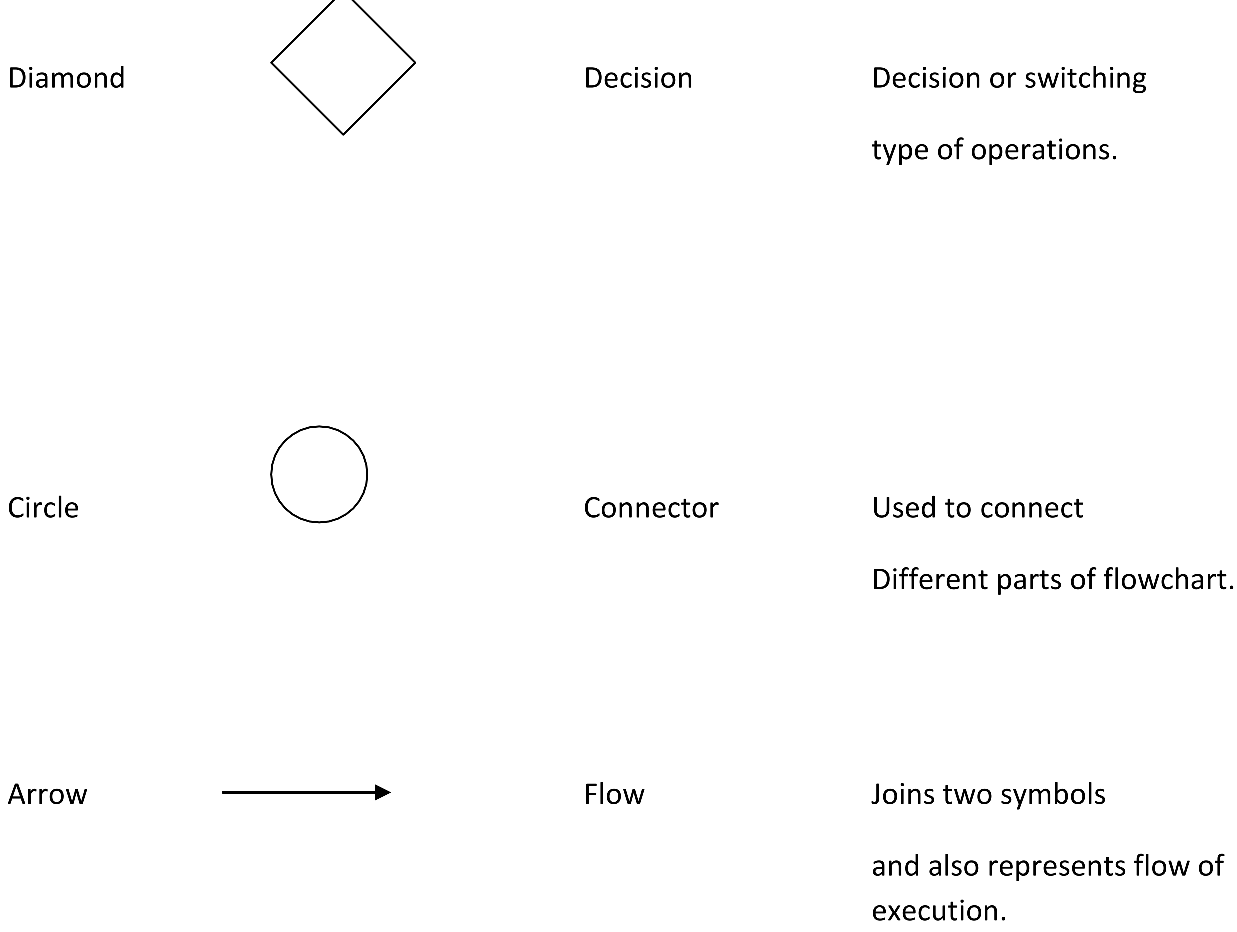
A Flow chart is a Graphical representation of an Algorithm or a portion of an Algorithm. Flow charts are drawn using certain special purpose symbols such as Rectangles, Diamonds, Ovals and small circles. These symbols are connected by arrows called flow lines.

(or)

The diagrammatic representation of way to solve the given problem is called flow chart.

The following are the most common symbols used in Drawing flowcharts:

Oval		Terminal	start/stop/begin/end.
Parallelogram		Input/output	Making data available For processing(input) or recording of the process information(output).
Rectangle		Process	Any processing to be Done .A process changes or moves data.An assignment operation.



Diamond		Decision	Decision or switching type of operations.
Circle		Connector	Used to connect Different parts of flowchart.
Arrow		Flow	Joins two symbols and also represents flow of execution.

INTRODUCTION TO ‘C’ LANGUAGE:

C language facilitates a very efficient approach to the development and implementation of computer programs. The History of C started in 1972 at the Bell Laboratories, USA where Dennis M. Ritchie proposed this language. In 1983 the American National Standards Institute (ANSI) established committee whose goal was to produce “an unambiguous and machine independent definition of the language C “ while still retaining it’s spirit .

C is the programming language most frequently associated with UNIX. Since the 1970s, the bulk of the UNIX operating system and its applications have been written in C. Because the C language does not directly rely on any specific hardware architecture, UNIX was one of the first portable operating systems. In other words, the majority of the code that makes up UNIX does not know and does not care which computer it is actually running on. Machine-specific features are isolated in a few modules within the UNIX kernel, which makes it easy for you to modify them when you are porting to a different hardware architecture.

C was first designed by Dennis Ritchie for use with UNIX on DEC PDP-11 computers. The language evolved from Martin Richard's BCPL, and one of its earlier forms was the B language, which was written by Ken Thompson for the DEC PDP-7. The first book on C was *The C Programming Language* by Brian Kernighan and Dennis Ritchie, published in 1978.

In 1983, the American National Standards Institute (ANSI) established a committee to standardize the definition of C. The resulting standard is known as *ANSI C,* and it is the recognized standard for the language, grammar, and a core set of libraries. The syntax is slightly different from the original C language, which is frequently called K&R for Kernighan and Ritchie. There is also an ISO (International Standards Organization) standard that is very similar to the ANSI standard.

It appears that there will be yet another ANSI C standard officially dated 1999 or in the early 2000 years; it is currently known as "C9X."

BASIC STRUCTURE OF C LANGUAGE:

The program written in C language follows this basic structure. The sequence of sections should be as they are in the basic structure. A C program should have one or more sections but the sequence of sections is to be followed.

1. Documentation section
2. Linking section
3. Definition section
4. Global declaration section
5. Main function section

    {

    Declaration section

    Executable section

    }

6. Sub program or function section

1. DOCUMENTATION SECTION : comes first and is used to document the use of logic or reasons in your program. It can be used to write the program's objective, developer and logic details. The documentation is done in C language with /* and */ . Whatever is written between these two are called comments.

2. LINKING SECTION : This section tells the compiler to link the certain occurrences of keywords or functions in your program to the header files specified in this section.

e.g. #include <stdio.h>

3. DEFINITION SECTION : It is used to declare some constants and assign them some value.

e.g. #define MAX 25

Here #define is a compiler directive which tells the compiler whenever MAX is found in the program replace it with 25.

4. GLOBAL DECLARATION SECTION : Here the variables which are used through out the program (including main and other functions) are declared so as to make them global(i.e accessible to all parts of program)

e.g. int i; (before main())

5. MAIN FUNCTION SECTION : It tells the compiler where to start the execution from

```
main()
{
    point from execution starts
}
```

main function has two sections

1. declaration section : In this the variables and their data types are declared.

2. Executable section : This has the part of program which actually performs the task we need.

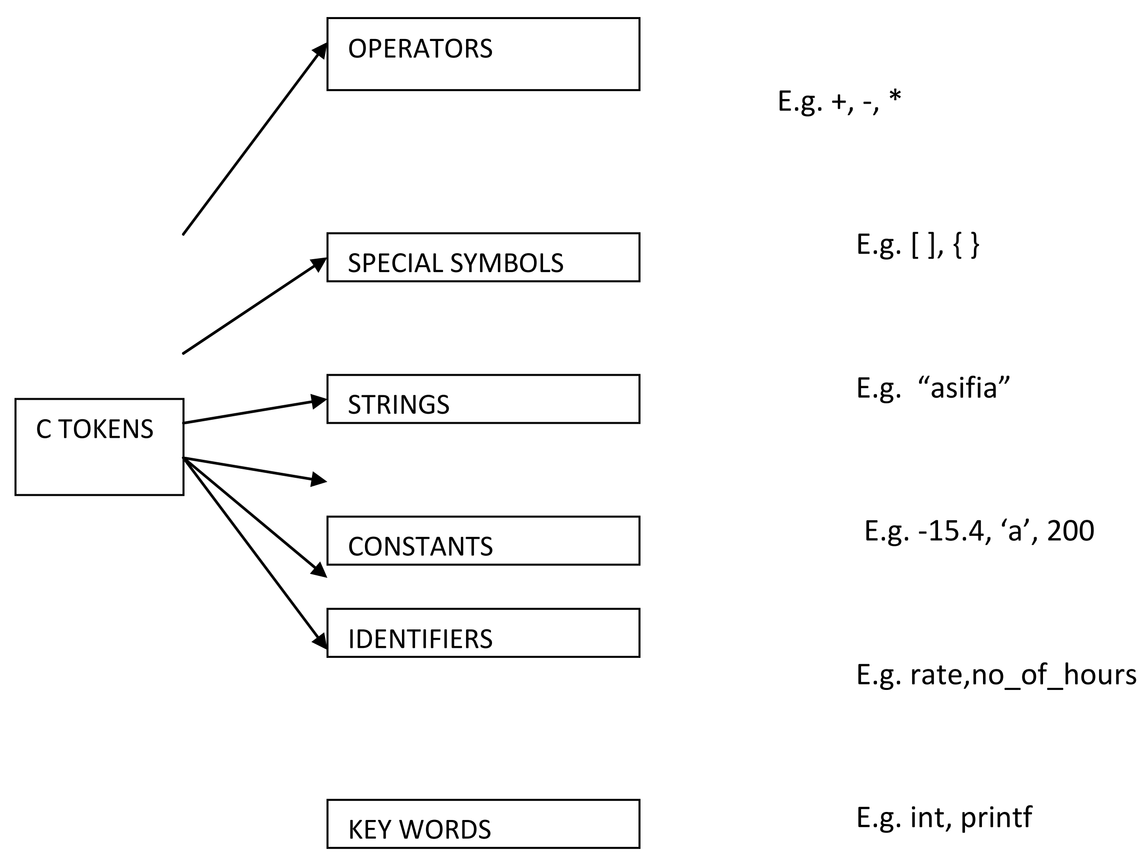
6. SUB PROGRAM OR FUNCTION SECTION : This has all the sub programs or the functions which our program needs.

SIMPLE ‘C’ PROGRAM:

```
/* simple program in c */
#include<stdio.h>
main()
{
printf("welcome to c programming");
} /* End of main */
```

C-TOKENS :

Tokens are individual words and punctuations marks in English language sentence. The smallest individual units are known as C tokens.



A C program can be divided into these tokens. A C program contains minimum 3 c tokens no matter what the size of the program is.

KEYWORDS :

There are certain words, called keywords (reserved words) that have a predefined meaning in 'C' language. These keywords are only to be used for their intended purpose and not as identifiers.

The following table shows the standard 'C' keywords

Auto	Break	Case	Char	Const	Continue
Default	Do	Double	Else	Enum	Extern
Float	For	Goto	If	Int	Long
Register	Return	Short	Signed	Sizeof	Static
Struct	Switch	Typedef	Union	Unsigned	void
Volatile	While				

IDENTIFIERS :

Names of the variables and other program elements such as functions, array,etc,are known as identifiers.

There are few rules that govern the way variable are named(identifiers).

1. Identifiers can be named from the combination of A-Z, a-z, 0-9, _(Underscore).

2. The first alphabet of the identifier should be either an alphabet or an underscore. digit are not allowed.

3. It should not be a keyword.

Eg: name,ptr,sum

After naming a variable we need to declare it to compiler of what data type it is .

The format of declaring a variable is

Data-type id1, id2,.... idn;

where data type could be float, int, char or any of the data types.

id1, id2, id3 are the names of variable we use. In case of single variable no commas are required.

eg   float a, b, c;

int   e, f, grand total;

char present_or_absent;

ASSIGNING VALUES :

When we name and declare variables we need to assign value to the variable. In some cases we assign value to the variable directly like

a=10;

in our program.

In some cases we need to assign values to variable after the user has given input for that.

eg we ask user to enter any no and input it

```
/* write a program to show assigning of values to variables */

#include<stdio.h>

main()

{

        int a;

        float b;

        printf("Enter any number\n");

        b=190.5;

        scanf("%d",&a);

        printf("user entered %d", a);
```

```
        printf("B's values is %f", b);
}
```

CONSTANTS :

A quantity that does not vary during the execution of a program is known as a constant supports two types of constants namely Numeric constants and character constants.

NUMERIC CONSTANTS:

1. Example for an integer constant is 786,-127
2. Long constant is written with a terminal ‘l’or ‘L’,for example 1234567899L is a Long constant.
3. Unsigned constants are written with a terminal ‘u’ or ‘U’,and the suffix ‘ul’ and ‘UL’ indicates unsigned long. for example 123456789u is a Unsigned constant and 1234567891ul is an unsigned long constant.
4. The advantage of declaring an unsigned constant is to increase the range of storage.
5. Floating point constants contain a decimal point or an exponent or both. For Eg : 123.4,1e-2,1.4E-4,etc.The suffixes f or F indicate a float constant while the absence of f or F indicate the double, l or L indicate long double.

CHARACTER CONSTANTS:

A character constant is written as one character with in single quotes such as ‘a’. The value of a character constant is the numerical value of the character in the machines character set. certain character constants can be represented by escape sequences like ‘\n’. These sequences look like two characters but represent only one.

The following are the some of the examples of escape sequences:

Escape sequence	Description
\a	Alert
\b	Backspace
\f	Form feed
\n	New Line
\r	Carriage return
\t	Horizontal Tab
\v	Vertical Tab

String constants or string literal is a sequence of zero or more characters surrounded by a double quote. Example , “ I am a little boy”. quotes are not a part of the string.

To distinguish between a character constant and a string that contains a single character ex: ‘a’ is not same as “a”. ‘a’ is an integer used to produce the numeric value of letter a in the machine character set, while “a” is an array of characters containing one character and a ‘\0’ as a string in C is an array of characters terminated by NULL.

There is one another kind of constant i.e Enumeration constant , it is a list of constant integer values.

Ex.: enum color { RED, Green, BLUE }

The first name in the enum has the value 0 and the next 1 and so on unless explicit values are specified.

If not all values specified , unspecified values continue the progression from the last specified value. For example

Enum months { JAN=1, FEB,MAR, …, DEC -

Where the value of FEB is 2 and MAR is 3 and so on.

Enumerations provide a convenient way to associate constant values with names.

VARIABLES :

A quantity that can vary during the execution of a program is known as a variable. To identify a quantity we name the variable for example if we are calculating a sum of two numbers we will name the variable that will hold the value of sum of two numbers as 'sum'.

DATA TYPES :

To represent different types of data in C program we need different data types. A data type is essential to identify the storage representation and the type of operations that can be performed on that data. C supports four different classes of data types namely

1. Basic Data types
2. Derives data types
3. User defined data types
4. Pointer data types

BASIC DATA TYPES:

All arithmetic operations such as Addition , subtraction etc are possible on basic data types.

E.g.: int a,b;

Char c;

The following table shows the Storage size and Range of basic data types:

TYPE	LENGTH	RANGE
Unsigned char	8 bits	0 to 255
Char	8 bits	-128 to 127
Short int	16 bits	-32768 to 32767
Unsigned int	32 bits	0 to 4,294,967,295

Int	32 bits	-2,147,483,648 to 2,147,483,648
Unsigned long	32 bits	0 to 4,294,967,295
Enum	16 bits	-2,147,483,648 to 2,147,483,648
Long	32 bits	-2,147,483,648 to 2,147,483,648
Float	32 bits	3.4*10E-38 to 3.4*10E38
Double	64 bits	1.7*10E-308 to 1.7*10E308
Long double	80 bits	3.4*10E-4932 to 1.1*10E4932

DERIVED DATA TYPES:

Derived datatypes are used in 'C' to store a set of data values. Arrays and Structures are examples for derived data types.

Ex: int a[10];

Char name[20];

USER DEFINED DATATYPES:

C Provides a facility called typedef for creating new data type names defined by the user. For Example ,the declaration ,

typedef int Integer;

makes the name Integer a synonym of int.Now the type Integer can be used in declarations ,casts,etc,like,

Integer num1,num2;

Which will be treated by the C compiler as the declaration of num1,num2as int variables.

"typedef" ia more useful with structures and pointers.

POINTER DATA TYPES:

Pointer data type is necessary to store the address of a variable.

INPUT AND OUTPUT STATEMENTS :

The simplest of input operator is getchar to read a single character from the input device.

```
varname=getchar();
```

you need to declare varname.

The simplest of output operator is putchar to output a single character on the output device.

```
putchar(varname)
```

The getchar() is used only for one input and is not formatted. Formatted input refers to an input data that has been arranged in a particular format, for that we have scanf.

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

Control string specifies field format in which data is to be entered.

arg1, arg2... argn specifies address of location or variable where data is stored.

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

%d used for integers

%f floats

%l long

%c character

for formatted output you use printf

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

```
/* program to exhibit i/o */

#include<stdio.h>
```

```
main()
{
    int a,b;
    float c;
    printf("Enter any number");
    a=getchar();
    printf("the char is ");
    putchar(a);
    printf("Exhibiting the use of scanf");
    printf("Enter three numbers");
    scanf("%d%d%f",&a,&b,&c);
    printf("%d%d%f",a,b,c);
}
```

Errors in C

Error is an illegal operation performed by the user which results in abnormal working of the program. Programming errors often remain undetected until the program is compiled or executed. Some of the errors inhibit the program from getting compiled or executed. Thus errors should be removed before compiling and executing.
The most common errors can be broadly classified as follows.

Syntax errors: Errors that occur when you **violate the rules** of writing C/C++ syntax are known as syntax errors. This compiler error indicates something that must be fixed before the code can be compiled. All these errors are detected by compiler and thus are known as compile-time errors.
Most frequent syntax errors are:

- Missing Parenthesis (})
- Printing the value of variable without declaring it
- Missing semicolon like this:

Run-time Errors : Errors which occur during program execution(run-time) after successful compilation are called run-time errors. One of the most common run-time error is division by zero also known as Division error. These types of error are hard to find as the compiler doesn't point to the line at which the error occurs.
Linker Errors: These error occurs when after compilation we link the different object files with main's object using *Ctrl+F9* key(RUN). These are errors generated when the executable of the program cannot be generated. This may be due to wrong function prototyping, incorrect header files. One of the most common

linker error is writing **Main()** instead of **main()**.
Logical Errors : On compilation and execution of a program, desired output is not obtained when certain input values are given. These types of errors which provide incorrect output but appears to be error free are called logical errors. These are one of the most common errors done by beginners of programming. These errors solely depend on the logical thinking of the programmer and are easy to detect if we follow the line of execution and determine why the program takes that path of execution.
Semantic errors : This error occurs when the statements written in the program are not meaningful to the compiler.

UNIT - II

OPERATORS :

An operator is a symbol that tells the compiler to perform certain mathematical or logical manipulations. They form expressions.

C operators can be classified as

1. Arithmetic operators
2. Relational operators
3. Logical operators
4. Assignment operators
5. Increment or Decrement operators
6. Conditional operator

7. Bit wise operators

8. Special operators

1. ARITHMETIC OPERATORS : All basic arithmetic operators are present in C.

operator	meaning
+	add
-	subtract
*	multiplication
/	division
%	modulo division(remainder)

An arithmetic operation involving only real operands(or integer operands) is called real arithmetic(or integer arithmetic). If a combination of arithmetic and real is called mixed mode arithmetic.

2. RELATIONAL OPERATORS : We often compare two quantities and depending on their relation take certain decisions for that comparison we use relational operators.

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

It is the form of

ae-1 relational operator ae-2

3. LOGICAL OPERATORS : An expression of this kind which combines two or more relational expressions is termed as a logical expressions or a compound relational expression. The operators and truth values are

op-1	op-2	op-1 && op-2	op-1 || op-2
non-zero	non-zero	1	1
non-zero	0	0	1
0	non-zero	0	1
0	0	0	0

op-1	!op-1
non-zero	zero
zero	non-zero

5. ASSIGNMENT OPERATORS : They are used to assign the result of an expression to a variable. The assignment operator is '='.

v op=exp

v is variable

op binary operator

exp expression

op= short hand assignment operator

short hand assignment operators

use of simple assignment operators	use of short hand assignment operators
a=a+1	a+=1
a=a-1	a-=1
a=a%b	a%=b

6. INCREMENT AND DECREMENT OPERATORS :

++ and == are called increment and decrement operators used to add or subtract.

Both are unary and as follows

++m or m++

--m or m--

The difference between ++m and m++ is

if m=5; y=++m then it is equal to m=5;m++;y=m;

if m=5; y=m++ then it is equal to m=5;y=m;m++;

7. CONDITIONAL OPERATOR : A ternary operator pair "?:" is available in C to construct conditional expressions of the form

exp1 ? exp2 : exp3;

It work as

if exp1 is true then exp2 else exp3

8. BIT WISE OPERATORS : C supports special operators known as bit wise operators for manipulation of data at bit level. They are not applied to float or double.

operator	meaning
&	Bitwise AND

Operator	Meaning
|	Bitwise OR
^	Bitwise exclusive OR
<<	left shift
>>	right shift
~	one's complement

9. SPECIAL OPERATORS : These operators which do not fit in any of the above classification are ,(comma), sizeof, Pointer operators(& and *) and member selection operators (. and ->). The comma operator is used to link related expressions together.

sizeof operator is used to know the sizeof operand.

```
/* programs to exhibit the use of operators */

#include<stdio.h>
main()
{
   int sum, mul, modu;
   float sub, divi;
   int i,j;
   float l, m;
  printf("Enter two integers ");
  scanf("%d%d",&i,&j);
  printf("Enter two real numbers");
  scanf("%f%f",&l,&m);
  sum=i+j;
  mul=i*j;
```

```
  modu=i%j;
  sub=l-m;
  divi=l/m;
  printf("sum is %d", sum);
  printf("mul is %d", mul);
  printf("Remainder is %d", modu);
  printf("subtraction of float is %f", sub);
  printf("division of float is %f", divi);
}
/* program to implement relational and logical */
#include<stdio.h>
main()
{
        int i, j, k;
        printf("Enter any three numbers ");
        scanf("%d%d%d", &i, &j, &k);
        if((i<j)&&(j<k))
                printf("k is largest");
        else if i<j || j>k
                {
                        if i<j && j >k
                                printf("j is largest");
                        else
                                printf("j is not largest of all");
```

```
			}
}
/* program to implement increment and decrement operators */
#include<stdio.h>
main()
{
	int i;
	printf("Enter a number");
	scanf("%d", &i);
	i++;
	printf("after incrementing %d ", i);
	i--;
	printf("after decrement %d", i);
}
/* program using ternary operator and assignment */
#include<stdio.h>
main()
{
	int i,j,large;
	printf("Enter two numbers ");
	scanf("%d%d",&i,&j);
	large=(i>j)?i:j;
	printf("largest of two is  %d",large);
}
```

Operator Precedence and Associativity in C

<u>Operator precedence</u> determines which operator is performed first in an expression with more than one operators with different precedence.

<u>Operators Associativity</u> is used when two operators of same precedence appear in an expression. Associativity can be either **L**eft **t**o **R**ight or **R**ight **t**o **L**eft.

For example: ‘*’ and ‘/’ have same precedence and their associativity is **L**eft **t**o **R**ight, so the expression “100 / 10 * 10” is treated as “(100 / 10) * 10”.

<table>
<tr><th>Operator</th><th>Description</th><th>Associativity</th></tr>
<tr><td>( )
[ ]
.
->
++ —</td><td>Parentheses (function call) (see Note 1)
Brackets (array subscript)
Member selection via object name
Member selection via pointer
Postfix increment/decrement (see Note 2)</td><td>left-to-right</td></tr>
<tr><td>++ —
+ –
! ~
(type)
*
&
sizeof</td><td>Prefix increment/decrement
Unary plus/minus
Logical negation/bitwise complement
Cast (convert value to temporary value of type)
Dereference
Address (of operand)
Determine size in bytes on this implementation</td><td>right-to-left</td></tr>
<tr><td>* / %</td><td>Multiplication/division/modulus</td><td>left-to-right</td></tr>
<tr><td>+ –</td><td>Addition/subtraction</td><td>left-to-right</td></tr>
<tr><td><< >></td><td>Bitwise shift left, Bitwise shift right</td><td>left-to-right</td></tr>
<tr><td>< <=
> >=</td><td>Relational less than/less than or equal to
Relational greater than/greater than or equal to</td><td>left-to-right</td></tr>
<tr><td>== !=</td><td>Relational is equal to/is not equal to</td><td>left-to-right</td></tr>
<tr><td>&</td><td>Bitwise AND</td><td>left-to-right</td></tr>
<tr><td>^</td><td>Bitwise exclusive OR</td><td>left-to-right</td></tr>
<tr><td>|</td><td>Bitwise inclusive OR</td><td>left-to-right</td></tr>
<tr><td>&&</td><td>Logical AND</td><td>left-to-right</td></tr>
<tr><td>| |</td><td>Logical OR</td><td>left-to-right</td></tr>
<tr><td>? :</td><td>Ternary conditional</td><td>right-to-left</td></tr>
<tr><td>=
+= -=
*= /=
%= &=
^= |=
<<= >>=</td><td>Assignment
Addition/subtraction assignment
Multiplication/division assignment
Modulus/bitwise AND assignment
Bitwise exclusive/inclusive OR assignment
Bitwise shift left/right assignment</td><td>right-to-left</td></tr>
<tr><td>,</td><td>Comma (separate expressions)</td><td>left-to-right</td></tr>
</table>

SELECTION STATEMENTS(DECISION MAKING):

IF AND SWITCH STATEMENTS :

We have a number of situations where we may have to change the order of execution of statements based on certain conditions or repeat a group of statements until certain specified conditions are met.

The if statement is a two way decision statement and is used in conjunction with an expression. It takes the following form

If(test expression)

If the test expression is true then the statement block after if is executed otherwise it is not executed

```
if (test expression)

{

    statement block;


}

statement–x ;
```

only statement–x is executed.

```
/* program for if */

#include<stdio.h>

main()

{

    int a,b;

    printf("Enter two numbers");
```

```
	scanf("%d%d",&a,&b):
	if a>b
		printf(" a is greater");
	if b>a
		printf("b is greater");
}
```

The if –else statement:

If your have another set of statement to be executed if condition is false then if-else is used

```
if (test expression)
{
	statement block1;
}
else
{
	statement block2;
}
statement –x ;
```

```
/* program for if-else */
#include<stdio.h>
main()
{
	int a,b;
```

```
	printf("Enter two numbers");
	scanf("%d%d",&a,&b):
	if a>b
		printf(" a is greater")
	else
		printf("b is greater");
}
```

Nesting of if..else statement :

If more than one if else statement

```
if(text cond1)
{
	if (test expression2
	{
		statement block1;
	}
	else
	{
		statement block 2;
	}
}
else
{
	statement block2;
```

```
}

statement-x ;
```

if else ladder

```
    if(condition1)
        statement1;
    else if(condition2)
        statement 2;
    else if(condition3)
        statement n;
    else
        default statement.
    statement-x;
```

The nesting of if-else depends upon the conditions with which we have to deal.

THE SWITCH STATEMENT:

If for suppose we have more than one valid choices to choose from then we can use switch statement in place of if statements.

```
switch(expression)
    {.
        case value-1
```

```
                    block-1
                     break;
            case value-2
                    block-2
                     break;
                --------
                --------
            default:
                    default block;
                    break;
        }
        statement–x
```

In case of

```
        if(cond1)
            {
            statement-1
            }
        if (cond2)
            {
                statement 2
            }
```

```
/* program to implement switch */
#include<stdio.h>
main()
{
            int marks,index;
           char grade[10];
           printf("Enter your marks");
           scanf("%d",&marks);
           index=marks/10;
           switch(index)
           {
                      case 10 :
                      case 9:
                      case 8:
                      case 7:
                      case 6: grade="first";
                                 break;
                      case 5 : grade="second";
                                 break;
                      case 4 : grade="third";
                                 break;
                      default : grade ="fail";
                                 break;
           }
```

```
	printf("%s",grade);
}
```

LOOPING :

Some times we require a set of statements to be executed repeatedly until a condition is met.

We have two types of looping structures. One in which condition is tested before entering the statement block called entry control.

The other in which condition is checked at exit called exit controlled loop.

WHILE STATEMENT :

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

It is an entry controlled loop. The condition is evaluated and if it is true then body of loop is executed. After execution of body the condition is once again evaluated and if is true body is executed once again. This goes on until test condition becomes false.

```
/* program for while  */
#include<stdio.h>
main()
{
	int count,n;
	float x,y;
	printf("Enter the values of x and n");
```

```
        scanf("%f%d",&x,&n);
        y=1.0;
        count=1;
        while(count<=n)
        {
                y=y*x;
                count++;
        }
        printf("x=%f; n=%d; x to power n = %f",x,n,y);
}
```

DO WHILE STATEMENT :

The while loop does not allow body to be executed if test condition is false. The do while is an exit controlled loop and its body is executed at least once.

```
        do
        {
                body
        }while(test condition);
```

```
/* printing multiplication table */
#include<stdio.h>
#define COL 10
#define ROW 12
main()
```

```
{
        int row,col,y;
        row=1;
        do
        {
                col=1;
                do
                {
                        y=row*col;
                        printf("%d",y);
                        col=col+1;
                }while(col<=COL);
                printf("\n");
                row=row+1;
        }while(row<=ROW);
}
```

THE FOR LOOP :

It is also an entry control loop that provides a more concise structure

```
for(initialization; test control; increment)
{
        body of loop
}
```

```
/* program of for loop */

#include<stdio.h>
main()

{

	long int p;

	int n;

	double q;

	printf("2 to power n ");

	p=1;

	for(n=0;n<21;++n)

	{

		if(n==0)

			p=1;

		else

			p=p*2;

		q=1.0/(double)p;

		printf("%101d%10d",p,n);

	}

}
```

UNCONDITIONAL STATEMENTS:

BREAK STATEMENT:

This is a simple statement. It only makes sense if it occurs in the body of a `switch`, `do`, `while` or `for` statement. When it is executed the control of flow jumps to the statement immediately following the body of the statement containing the `break`. Its use is widespread in `switch` statements, where it is more or less essential to get the control .

The use of the break within loops is of dubious legitimacy. It has its moments, but is really only justifiable when exceptional circumstances have happened and the loop has to be abandoned. It would be nice if more than one loop could be abandoned with a single break but that isn't how it works. Here is an example.

```
#include  <stdio.h>
#include <stdlib.h>
main(){
     int i;

     for(i = 0; i < 10000; i++){
            if(getchar() == 's')
                  break;
            printf("%d\n", i);
     }
     exit(EXIT_SUCCESS);
}
```

It reads a single character from the program's input before printing the next in a sequence of numbers. If an „s" is typed, the break causes an exit from the loop.

If you want to exit from more than one level of loop, the break is the wrong thing to use.

CONTINUE STATEMENT:

This statement has only a limited number of uses. The rules for its use are the same as for break, with the exception that it doesn't apply to switch statements. Executing a continue starts the next iteration of the smallest enclosing do, while or for statement immediately. The use of continue is largely restricted to the top of loops, where a decision has to be made whether or not to execute the rest of the body of the loop. In this example it ensures that division by zero (which gives undefined behaviour) doesn't happen.

```
#include  <stdio.h>
#include <stdlib.h>
main(){
     int i;

     for(i = -10; i < 10; i++){
            if(i == 0)
                  continue;
            printf("%f\n", 15.0/i);
            /*
             * Lots of other statements .....
             */
     }
     exit(EXIT_SUCCESS);
```

}

The `continue` can be used in other parts of a loop, too, where it may occasionally help to simplify the logic of the code and improve readability. `continue` has no special meaning to a `switch` statement, where `break` does have. Inside a `switch`, `continue` is only valid if there is a loop that encloses the `switch`, in which case the next iteration of the loop will be started.

There is an important difference between loops written with `while` and `for`. In a `while`, a continue will go immediately to the test of the controlling expression. The same thing in a `for` will do two things: first the update expression is evaluated, then the controlling expression is evaluated.

GOTO AND LABELS:

In C, it is used to escape from multiple nested loops, or to go to an error handling exit at the end of a function. You will need a *label* when you use a `goto`; this example shows both.

```
goto L1;
/* whatever you like here */
L1: /* anything else */
```

A label is an identifier followed by a colon. Labels have their own „name space‟ so they can't clash with the names of variables or functions. The name space only exists for the function containing the label, so label names can be re-used in different functions. The label can be used before it is declared, too, simply by mentioning it in a `goto` statement.

Labels must be part of a full statement, even if it's an empty one. This usually only matters when you're trying to put a label at the end of a compound statement—like this.

```
label_at_end: ; /* empty statement */
}
```

The `goto` works in an obvious way, jumping to the labelled statements. Because the name of the label is only visible inside its own function, you can't jump from one function to another one.

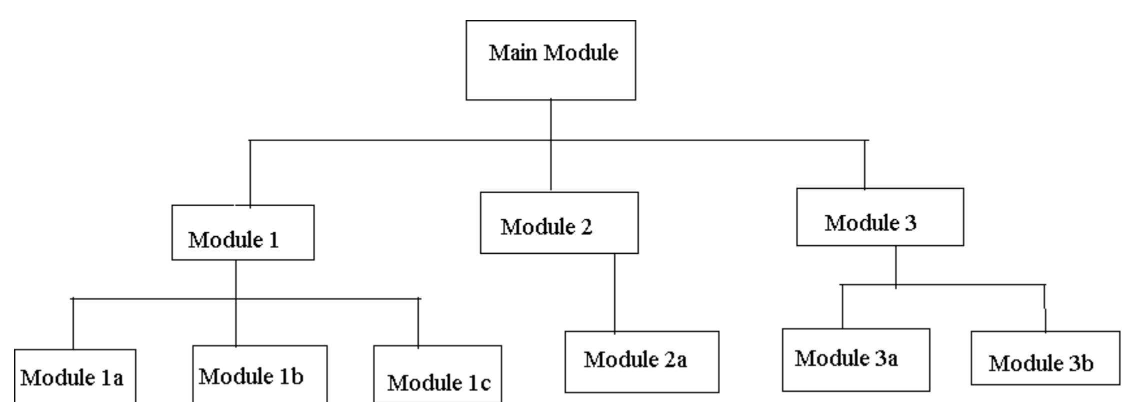
It's hard to give rigid rules about the use of gotos but, as with the do, continue and the break (except in switch statements), over-use should be avoided. More than one goto every 3–5 functions is a symptom that should be viewed with deep suspicion.

FUNCTIONS-DESIGNING STRUCTURED PROGRAMS:

The planning for large programs consists of first understanding the problem as a whole, second breaking it into simpler, understandable parts. We call each of these parts of a program a **module** and the process of subdividing a problem into manageable parts **top-down design**.

The principles of top-down design and structured programming dictate that a program should be divided into a main module and its related modules. Each module is in turn divided into sub-modules until the resulting modules are intrinsic; that is, until they are implicitly understood without further division.

Top-down design is usually done using a visual representation of the modules known as a structure chart. The structure chart shows the relation between each module and its sub-modules. The structure chart is read top-down, left-right. First we read Main Module. Main Module represents our entire set of code to solve the problem.



Structure Chart

Moving down and left, we then read Module 1. On the same level with Module 1 are Module 2 and Module 3. The Main Module consists of three sub-modules. At this level, however we are dealing only with Module 1. Module 1 is further subdivided into three

modules, Module 1a, Module 1b, and Module 1c. To write the code for Module 1, we need to write code for its three sub-modules.

The Main Module is known as a calling module because it has sub-modules. Each of the sub-modules is known as a called module. But because Modules 1, 2, and 3 also have sub-modules, they are also calling modules; they are both called and calling modules.

Communication between modules in a structure chart is allowed only through a calling module. If Module 1 needs to send data to Module 2, the data must be passed through the calling module, Main Module. No communication can take place directly between modules that do not have a calling-called relationship.

How can Module 1a send data to Module 3b?

It first sends data to Module 1, which in turn sends it to the Main Module, which passes it to Module 3, and then on to Module 3b.

The technique used to pass data to a function is known as parameter passing. The parameters are contained in a list that is a definition of the data passed to the function by the caller. The list serves as a formal declaration of the data types and names.

FUNCTIONS :

A function is a self contained program segment that carries out some specific well defined tasks.

Advantages of functions:

1. Write your code as collections of small functions to make your program modular

2.Structured programming

3.Code easier to debug

4.Easier modification

5.Reusable in other programs

Function Definition :

type func_name(parameter list)

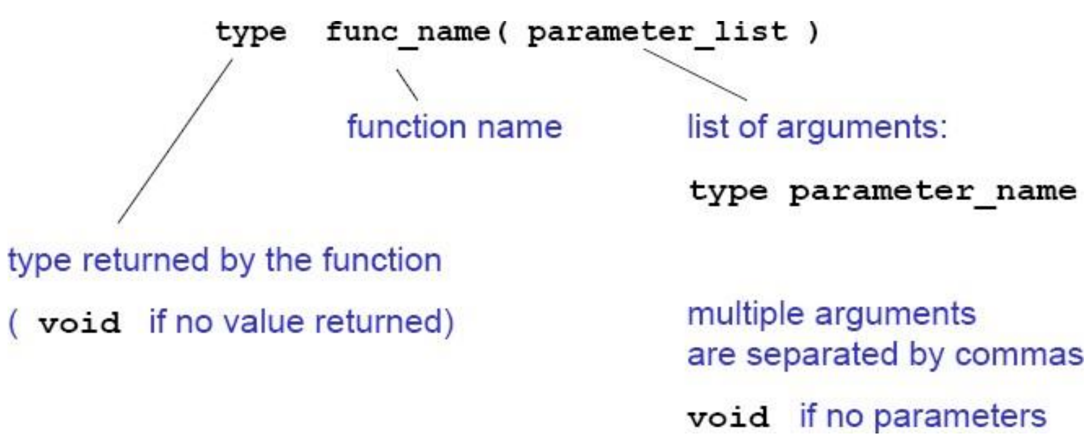
{

declarations;

statements;

}

FUNCTION HEADER :



Examples:

```
int fact(int n)
void error_message(int errorcode)
double initial_value(void)
int main(void)
```

Usage:

```
a = fact(13);
error_message(2);
x=initial_value();
```

FUNCTION PROTOTYPES

If a function is not defined before it is used, it must be declared by specifying the return type and the types of the parameters.

double sqrt(double);

Tells the compiler that the function **sqrt()** takes an argument of type double and returns a double. This means, incidentally, that variables will be cast to the correct type; so **sqrt(4)** will return the correct value even though **4** is **int** not **double**. These function prototypes are placed at the top of the program, or in a separate header file, **file.h**, included as

#include "file.h"

Variable names in the argument list of a function declaration are optional:

void f (char, int);

void f (char c, int i); /*equivalent but makes code more readable */

If all functions are defined before they are used, no prototypes are needed. In this case, **main()** is the last function of the program.

SCOPE RULES FOR FUNCTIONS :

Variables defined within a function (including main) are local to this function and no other function has direct access to them. The only way to pass variables to function is as parameters. The only way to pass (a single) variable back to the calling

function is via the return statement

Ex:

```
int func (int n)

{

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

return n;

}//b not defined locally!
```

```
int main (void)

{

int a = 2, b = 1, c;

c = func(a);

return 0;

}
```

FUNCTION CALLS :

When a function is called, expressions in the parameter list are evaluated (in no particular order!) and results are transformed to the required type. Parameters are copied to local variables for the function and function body is executed when return is encountered, the function is terminated and the result (specified in the return statement) is passed to the calling function (for example main).

Ex:

```
int fact (int n)

{

int i, product = 1;

for (i = 2; i <= n; ++i)

product *= i;

return product;

}

int main (void)

{

int i = 12;

printf("%d",fact(i));

return 0;

}
```

TYPES OF FUNCTIONS:

USER DEFINED FUNCTIONS:

Every program must have a main function to indicate where the program has to begin its execution. While it is possible to code any program utilizing only main function, it leads to a number of problems. The program may become too large and complex and as a result task of debugging, testing and maintaining becomes difficult. If a program is divided into functional parts, then each part may be independently coded and later combined into a single unit, these subprograms called “functions” are much easier to understand debug and test.

There are times when some types of operation for calculation is repeated many times at many points through out a program. For instance, we might use the factorial of a number at several points in the program. In such situations, we may repeat the program statements whenever they are needed. Another approach is to design a function that can be called and used whenever required.

The advantages of using functions are

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

A function is a self-contained block of code that performs a particular task. Once a function has been designed and packed it can be treated as a “black box” that takes some data from main program and returns a value. The inner details of the program are invisible to the rest of program.

The form of the C functions.

```
function-name ( argument list )
argument declaration;
{
        local variable declarations;
        executable statement 1;
```

```
        executabie      statement 2;
        - - - - - - - - - -
        - - - - - - - - - -
        - - - - - - - - - -
        return (expression) ;
    }
```

The return statement is the mechanism for returning a value to the calling function. All functions by default returns int type data. we can force a function to return a particular type of data by using a type specifier in the header.

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

STANDARD LIBRARY FUNCTIONS AND HEADER FILES:

C functions can be classified into two categories, namely, library functions and user-defined functions .Main is the example of user-defined functions.Printf and scanf belong to the category of library functions. The main difference between these two categories is that library functions are not required to be written by us where as a user-defined function has to be developed by the user at the time of writing a program.However, a user-defined function can later become a part of the c program library.

ANSI C Standard Library Functions :

The C language is accompanied by a number of library functions that perform various tasks.The ANSI committee has standardized header files which contain these functions.

Some of the Header files are:

<ctype.h> character testing and conversion functions.

<math.h> Mathematical functions

<stdio.h> standard I/O library functions

<stdlib.h> Utility functions such as string conversion routines memory allocation routines , random number generator,etc.

<string.h> string Manipulation functions

<time.h> Time Manipulation functions

MATH.H

The math library contains functions for performing common mathematical operations. Some of the functions are :

abs : returns the absolute value of an integer x

cos : returns the cosine of x, where x is in radians

exp: returns "e" raised to a given power

fabs: returns the absolute value of a float x

log: returns the logarithm to base e

log10: returns the logarithm to base 10

pow : returns a given number raised to another number

sin : returns the sine of x, where x is in radians

sqrt : returns the square root of x

tan : returns the tangent of x, where x is in radians

ceil : The ceiling function rounds a number with a decimal part up to the next highest integer (written mathematically as $\lceil x \rceil$)

floor : The floor function rounds a number with a decimal part down to the next lowest

integer (written mathematically as $\lfloor x \rfloor$)

STRING.H

There are many functions for manipulating strings. Some of the more useful are:

strcat : Concatenates (i.e., adds) two strings

strcmp: Compares two strings

strcpy: Copies a string

strlen: Calculates the length of a string (not including the null)

strstr: Finds a string within another string

strtok: Divides a string into tokens (i.e., parts)

STDIO.H

Printf: Formatted printing to stdout

Scanf: Formatted input from stdin

Fprintf: Formatted printing to a file

Fscanf: Formatted input from a file

Getc: Get a character from a stream (e.g, stdin or a file)

putc: Write a character to a stream (e.g, stdout or a file)

fgets: Get a string from a stream

fputs: Write a string to a stream

fopen: Open a file

fclose: Close a file

STDLIB.H

Atof: Convert a string to a double (not a float)

Atoi: Convert a string to an int

Exit: Terminate a program, return an integer value

free: Release allocated memory

malloc: Allocate memory

rand: Generate a pseudo-random number

system: Execute an external command

TIME.H

This library contains several functions for getting the current date and time.

Time: Get the system time

Ctime: Convert the result from time() to something meaningful

The following table contains some more header files:

<assert.h>	Contains the assert macro, used to assist with detecting logical errors and other types of bug in debugging versions of a program.
<complex.h>	A set of functions for manipulating complex numbers. (New with **C99**)
<ctype.h>	Contains functions used to classify characters by their types or to convert between upper and lower case in a way that is independent of the used character set (typically ASCII or one of its extensions, although implementations utilizing EBCDIC are also known).
<errno.h>	For testing error codes reported by library functions.
<fenv.h>	For controlling floating-point environment. (New with **C99**)
<float.h>	Contains defined constants specifying the implementation-specific properties of the floating-point library, such as the minimum difference between two different floating-point numbers (_EPSILON), the maximum number of digits of accuracy (_DIG) and the range of numbers which can be represented (_MIN, _MAX).
<inttypes.h>	For precise conversion between integer types. (New with **C99**)

<iso646.h>	For programming in ISO 646 variant character sets. (New with **NA1**)
<limits.h>	Contains defined constants specifying the implementation-specific properties of the integer types, such as the range of numbers which can be represented (_MIN, _MAX).
<locale.h>	For setlocale() and related constants. This is used to choose an appropriate locale.
<math.h>	For computing common mathematical functions
<setjmp.h>	Declares the macros setjmp and longjmp, which are used for non-local exits
<signal.h>	For controlling various exceptional conditions
<stdarg.h>	For accessing a varying number of arguments passed to functions.
<stdbool.h>	For a boolean data type. (New with **C99**)
<stdint.h>	For defining various integer types. (New with **C99**)
<stddef.h>	For defining several useful types and macros.
<stdio.h>	Provides the core input and output capabilities of the C language. This file includes the venerable `printf` function.
<stdlib.h>	For performing a variety of operations, including conversion, pseudo-random numbers, memory allocation, process control, environment, signalling, searching,

	and sorting.
<**string.h**>	For manipulating several kinds of strings.
<**tgmath.h**>	For type-generic mathematical functions. (New with **C99**)
<**time.h**>	For converting between various time and date formats.
<**wchar.h**>	For manipulating wide streams and several kinds of strings using wide characters - key to supporting a range of languages. (New with **NA1**)
<**wctype.h**>	For classifying wide characters. (New with **NA1**)

CATEGORIES OF FUNCTIONS :

A function depending on whether arguments are present or not and whether a value is returned or not may belong to.

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

1. Functions with no arguments and no return values :

When a function has no arguments, it does not receive any data from calling function. In effect, there is no data transfer between calling function and called function.

```
#include<stdio.h>

main()

{
```

```
        printline();
        value();
        printline();
}
printline()
{
        int i;
        for(i=1;i<=35;i++0)
                printf("%c","-");
                printf("\n");
}
value()
{
        int year, period;
        float inrate,sum,principal;
        printf("Enter Principal, Rate,Period");
        scanf("%f%f%f",&principal,&inrate,&period);
        sum=principal;
        year=1;
        while(year<=period)
        {
                sum=sum*(1+inrate);
                year=year+1;
        }
```

```
    printf("%f%f%d%f",principal,inrate,period,sum);
}
```

2. Arguments but no return values :

The function takes argument but does not return value.

The actual (sent through main) and formal(declared in header section) should match in number, type and order.

In case actual arguments are more than formal arguments, extra actual arguments are discarded. On other hand unmatched formal arguments are initialized to some garbage values.

```
#include<stdio.h>
main()
{
    float prin,inrate;
    int period;
    printf("Enter principal amount, interest");
    printf("rate and period\n");
    scanf("%f%f%d",&principal,&inrate,&period);
    printline('z');
    value(principal, inrate, peiod);
    printline('c');
}
printline(ch)
char ch;
{
    int i;
```

```
        for(i=1;i<=52;i++)
                printf("%c",ch);
                printf("\n");
}
```

PARAMETER PASSING TECHNIQUES:

Parameter passing mechanism in „C" is of two types.

1. Call by Value
2. Call by Reference.

The process of passing the actual value of variables is known as Call by Value.

The process of calling a function using pointers to pass the addresses of variables is known as Call by Reference. The function which is called by reference can change the value of the variable used in the call.

Example of Call by Value:

```
#include <stdio.h>
void swap(int,int);
main()
{
        int a,b;
        printf("Enter the Values of a and b:");
        scanf("%d%d",&a,&b);
        printf("Before Swapping \n");
        printf("a = %d \t b = %d", a,b);
        swap(a,b);
        printf("After Swapping \n");
        printf("a = %d \t b = %d", a,b);
}


void swap(int a, int b)
{
        int temp;
        temp = a;
        a = b;
        b = temp;
}
```

Example of Call by Reference:

```
#include<stdio.h>
main()
{
        int a,b;
        a = 10;
        b = 20;
        swap (&a, &b);
        printf("After Swapping \n");
        printf("a = %d \t b = %d", a,b);
}
void swap(int *x, int *y)
{
        int temp;
        temp = *x;
        *x = *y;
        *y = temp;
}
```

STORAGE CLASSES :

In ‘C’ a variable can have any one of four Storage Classes.

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

SCOPE :

The Scope of variable determines over what parts of the program a variable is actually available for use.

LONGEVITY :

Longevity refers to period during which a variable retains a given value during execution of a program (alive). So Longevity has a direct effect on utility of a given variable.

The variables may also be broadly categorized depending on place of their declaration as internal(local) or external(global). Internal variables are those which are declared within a particular function, while external variables are declared outside of any function.

AUTOMATIC VARIABLES :

They are declared inside a function in which they are to be utilized. They are created when 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. Because of this property, automatic variables are also referred to as local or internal variables.

By default declaration is automatic. One important feature of automatic variables is that their value cannot be changed accidentally by what happens in some other function in the program.

```
#include<stdio.h>

main()

{

        int m=1000;

        func2();

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

}



func1()

{

        int m=10;

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

}
```

```
func2()

{

        int m=100;

        func1();

        printf("%d",m);

}
```

First, any variable local to main will normally live throughout the whole program, although it is active only in main.

Secondly, during recursion, nested variables are unique auto variables, a situation similar to function nested auto variables with identical names.

EXTERNAL VARIABLES :

Variables that are both alive and active throughout entire program are known as external variables. They are also known as Global Variables. In case a local and global have same name local variable will have precedence over global one in function where it is declared.

```
#include<stdio.h>

int x;

main()

{

        x=10;

        printf("%d",x);

        printf("x=%d",fun1());

        printf("x=%d",fun2());
```

```
        printf("x=%d",fun3());



}
fun1()
{
        x=x+10;
        return(x);
}
fun2()
{
        int x;
        x=1;
        return(x);
}
fun3()
{
x=x+10;
return(x);
}
```

An extern within a function provides the type information to just that one function.

STATIC VARIABLES :

The value of Static Variable persists until the end of program. A variable can be declared Static using Keyword Static like Internal & External Static Variables are differentiated depending whether they are declared inside or outside of auto variables, except that they remain alive throughout the remainder of program.

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

REGISTER VARIABLES :

We can tell the Compiler that a variable should be kept in one of the machines registers, instead of keeping in the memory. Since a register access is much faster than a memory access, keeping frequently accessed variables in register will lead to faster execution

Syntax:

```
register int Count.
```

ARRAYS :

An array is a group of related data items that share a common name.

Ex:- Students

The complete set of students are represented using an array name students. A particular value is indicated by writing a number called index number or subscript in brackets after array name. The complete set of value is referred to as an array, the individual values are called elements.

ONE – DIMENSIONAL ARRAYS :

A list of items can be given one variable index is called single subscripted variable or a one-dimensional array.

The subscript value starts from 0. If we want 5 elements the declaration will be

int number[5];

The elements will be number[0], number[1], number[2], number[3], number[4]

There will not be number[5]

Declaration of One - Dimensional Arrays :

Type variable – name [sizes];

Type – data type of all elements Ex: int, float etc.,

Variable – name – is an identifier

Size – is the maximum no of elements that can be stored.

Ex:- float avg[50]

This array is of type float. Its name is avg. and it can contains 50 elements only. The range starting from 0 – 49 elements.

Initialization of Arrays :

Initialization of elements of arrays can be done in same way as ordinary variables are done when they are declared.

Type array name[size] = {List of Value};

Ex:- int number[3]={0,0,0};

If the number of values in the list is less than number of elements then only that elements will be initialized. The remaining elements will be set to zero automatically.

Ex:- float total[5]= {0.0,15.75,-10};

The size may be omitted. In such cases, Compiler allocates enough space for all initialized elements.

int counter[ ]= {1,1,1,1};

```
/* Program Showing one dimensional array */

#include<stdio.h>
```

```
main()

{

int i;

float x[10],value,total;

printf("Enter 10 real numbers\n");

        for(i=0;i<10;i++)

        {

                scanf("%f",&value);

                x[i]=value;

        }

        total=0;

        for(i=0;i<10;i++)

                total=total+x[i]

        for(i=0;i<10;i++)

                printf("x*%2d+=%5.2f\n",I+1,x*I+);

                printf("total=%0.2f",total);

}
```

TWO – DIMENSIONAL ARRAYS:

To store tables we need two dimensional arrays. Each table consists of rows and columns. Two dimensional arrays are declare as

type array name [row-size][col-size];

/* Write a program Showing 2-DIMENSIONAL ARRAY */

```
/* SHOWING MULTIPLICATION TABLE */
#include<stdio.h>
#include<math.h>
#define ROWS 5
#define COLS 5
main()
{
        int row,cols,prod[ROWS][COLS];
        int i,j;

        printf("Multiplication table");
        for(j=1;j< =COLS;j++)
                printf("%d",j);
        for(i=0;i<ROWS;i++)
        {
                row = i+1;
                printf("%2d|",row);
                for(j=1;j < = COLS;j++)
                {
                        COLS=j;
                        prod[i][j]= row * cols;
                        printf("%4d",prod*i+*j+);
                }
        }
```

}

INITIALIZING TWO DIMENSIONAL ARRAYS:

They can be initialized by following their declaration with a list of initial values enclosed in braces.

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

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

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

When all elements are to be initialized to zero, following short-cut method may be used.

int m[3][5] = {{0},{0},{0}};

STRINGS(CHARACTER ARRAYS) :

A String is an array of characters. Any group of characters (except double quote sign)defined between double quotes is a constant string.

Ex: “C is a great programming language”.

If we want to include double quotes.

Ex: “\”C is great \” is norm of programmers “.

Declaring and initializing strings :-

A string variable is any valid C variable name and is always declared as an array.

char string name [size];

size determines number of characters in the string name. When the compiler assigns a character string to a character array, it automatically supplies a null character ('\0') at end of String. Therefore, size should be equal to maximum number of character in String plus one.

String can be initialized when declared as

1. char city*10+= "NEW YORK';
2. char city[10]= ,'N','E','W',' ','Y','O','R','K','/0'-;
3.

C also permits us to initializing a String without specifying size.

Ex:- char Strings* += ,'G','O','O','D','\0'-;

READING STRINGS FROM USER:

%s format with scanf can be used for reading String.

char address[15];

scanf("%s",address);

The problem with scanf function is that it terminates its input on first white space it finds. So scanf works fine as long as there are no spaces in between the text.

Reading a line of text :

If we are required to read a line of text we use getchar(). which reads a single characters. Repeatedly to read successive single characters from input and place in character array.

```
/* Program to read String using scanf & getchar */

#include<stdio.h>

main()
```

```
{
    char line[80],ano_line[80],character;
    int c;
    c=0;
    printf("Enter String using scanf to read \n");
    scanf("%s", line);
    printf("Using getchar enter new line\n");
    do
    {
        character = getchar();
        ano_line[c] = character;
        c++;
    - while(character !='\n');
    c=c-1;
    ano_line*c+='\0';
}
```

STRING INPUT/OUTPUT FUNCTIONS:

C provides two basic ways to read and write strings.First we can read and write strings with the formatted input/output functions,scanf/fscanf and prinf/fprinf.Second we can use a special set of strin only functions ,get string(gets/fgets)and put string(puts/fputs).

Formatted string Input/Output:

Formatted String Input:scanf/fscanf:

Declarations:

int fscanf(FILE ****stream*, const char ****format***, ...);**
int scanf(const char ****format*, ...);**

The ..scanf functions provide a means to input formatted information from a stream

fscanf reads formatted input from a stream

scanf reads formatted input from stdin

These functions take input in a manner that is specified by the format argument and store each input field into the following arguments in a left to right fashion.

Each input field is specified in the format string with a conversion specifier which specifies how the input is to be stored in the appropriate variable. Other characters in the format string specify characters that must be matched from the input, but are not stored in any of the following arguments. If the input does not match then the function stops scanning and returns. A whitespace character may match with any whitespace character (space, tab, carriage return, new line, vertical tab, or formfeed) or the next incompatible character.

Formatted String Output:printf/fprintf:

Declarations:

int fprintf(FILE ****stream*, const char ****format***, ...);**
int printf(const char ****format*, ...);**

The ..printf functions provide a means to output formatted information to a stream.

fprintf sends formatted output to a stream

printf sends formatted output to stdout

These functions take the format string specified by the *format* argument and apply each following argument to the format specifiers in the string in a left to right fashion. Each character in the format string is copied to the stream except for conversion characters which specify a format specifier.

String Input/Output

In addition to the Formatted string functions,C has two sets of string functions that read and write strings without reformatting any data.These functions convert text file lines to strings and strings to text file lines.

gets():

Declaration:

```
char *gets(char *str);
```

Reads a line from **`stdin`** and stores it into the string pointed to by *str*. It stops when either the newline character is read or when the end-of-file is reached, whichever comes first. The newline character is not copied to the string. A null character is appended to the end of the string.

On success a pointer to the string is returned. On error a null pointer is returned. If the end-of-file occurs before any characters have been read, the string remains unchanged.

puts:

Declaration:

```
int puts(const char *str);
```

Writes a string to **`stdout`** up to but not including the null character. A newline character is appended to the output.

On success a nonnegative value is returned. On error **`EOF`** is returned.

STRING HANDLING/MANIPULATION FUNCTIONS:

strcat( )	Concatenates two Strings
strcmp( )	Compares two Strings
strcpy( )	Copies one String Over another

strlen( )   Finds length of String

strcat() function:

This function adds two strings together.

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

strcat(string1,string2);

string1 = VERY

string2 = FOOLISH

strcat(string1,string2);

string1=VERY FOOLISH

string2 = FOOLISH

Strncat: Append n characters from string2 to stringl.

char *strncat(const char *string1, char *string2, size_t n);

strcmp() function :

This function compares two strings identified by arguments and has a value 0 if they are equal. If they are not, it has the numeric difference between the first non-matching characters in the Strings.

Syntax: int strcmp (const char *string1,const char *string2);

strcmp(string1,string2);

Ex:- strcmp(name1,name2);

strcmp(name1,"John");

strcmp("ROM","Ram");

Strncmp: Compare first n characters of two strings.

int strncmp(const char *string1, char *string2, size_t n);

strcpy() function :

It works almost as a string assignment operators. It takes the form

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

strcpy(string1,string2);

string2 can be array or a constant.

Strncpy: Copy first n characters of string2 to stringl .

char *strncpy(const char *string1,const char *string2, size_t n);

strlen() function :

Counts and returns the number of characters in a string.

Syntax:int strlen(const char *string);

n= strlen(string);

n→ integer variable which receives the value of length of string.

```
/* Illustration of string-handling */

#include<stdio.h>
#include<string.h>
main()
{
	char s1[20],s2[20],s3[20];
	int  X,L1,L2,L3;
	printf("Enter two string constants\n");
	scanf("%s %s",s1,s2);
	X=strcmp(s1,s2);
	if (X!=0)
	{
		printf("Strings are not equal\n");
		strcat(s1,s2);
```

```
}
else
printf("Strings are equal \n");
strcpy(s3,s1);
L1=strlen(s1);

L2=strlen(s2);
L3=strlen(s3);
printf("s1=%s\t length=%d chars \n",s1,L1);
printf("s2=%s\t length=%d chars \n",s2,L2);
printf("s3=%s\t length=%d chars \n",s3,L3);
}
```

UNIT – III

RECURSIVE FUNCTIONS:

Recursion is a repetitive process in which a function calls itself (or) A function is called recursive if it calls itself either directly or indirectly. In C, all functions can be used recursively.

Example: Fibonacci Numbers

A recursive function for Fibonacci numbers (0,1,1,2,3,5,8,13...)

/* Function with recursion*/

```
int fibonacci(int n)

{

if (n <= 1)

return  n;

else

return (fibonacci(n-1) + fibonacci(n-2));

}
```

With recursion 1.4×10^{9} function calls needed to find the 43rd Fibonacci number(which has the value 433494437) .If possible, it is better to write iterative functions.

```
int factorial (int n) /* iterative version */

{

for ( ; n > 1; --n)

product *= n;

return product;

}
```

POINTERS:

One of the powerful features of C is ability to access the memory variables by their memory address. This can be done by using Pointers. The real power of C lies in the proper use of Pointers.

A pointer is a variable that can store an address of a variable (i.e., 112300).We say that a pointer points to a variable that is stored at that address. A pointer itself usually occupies 4 bytes of memory (then it can address cells from 0 to 232-1).

Advantages of Pointers:

1. A pointer enables us to access a variable that is defined out side the function.
2. Pointers are more efficient in handling the data tables.
3. Pointers reduce the length and complexity of a program.
4. They increase the execution speed.

Definition :

A variable that holds a physical memory address is called a pointer variable or Pointer.

Declaration :

Datatype * Variable-name;

```
Eg:-    int *ad;        /* pointer to int */
        char *s;        /* pointer to char */
        float *fp;      /* pointer to float */
        char **s;       /* pointer to variable that is a pointer to char */
```

A pointer is a variable that contains an address which is a location of another variable in memory.

Consider the Statement

p=&i;

Here „&" is called address of a variable.
„p" contains the address of a variable i.

The operator & returns the memory address of variable on which it is operated, this is called Referencing.

The * operator is called an indirection operator or dereferencing operator which is used to display the contents of the Pointer Variable.

Consider the following Statements :

```
int *p,x;
x =5;
p= &x;
```

Assume that x is stored at the memory address 2000. Then the output for the following printf statements is :

	Output
Printf("The Value of x is %d",x);	5
Printf("The Address of x is %u",&x);	2000
Printf("The Address of x is %u",p);	2000
Printf("The Value of x is %d",*p);	5
Printf("The Value of x is %d",*(&x));	5

POINTERS WITH ARRAYS :

When an array is declared, elements of array are stored in contiguous locations. The address of the first element of an array is called its base address.

Consider the array

2000	2002	2004	2006	2008
a[0]	a[1]	a[2]	a[3]	a[4]

The name of the array is called its base address.

i.e., a and k& a[20] are equal.

Now both a and a[0] points to location 2000. If we declare p as an integer pointer, then we can make the pointer P to point to the array a by following assignment

```
P = a;
```

We can access every value of array a by moving P from one element to another.

i.e.,

P	points to 0th element
P+1	points to 1st element
P+2	points to 2nd element
P+3	points to 3rd element
P +4	points to 4th element

Reading and Printing an array using Pointers :

```
main()
{
        int *a,i;
        printf("Enter five elements:");
        for (i=0;i<5;i++)
                scanf("%d",a+i);
                printf("The array elements are:");
                for (i=0;i<5;i++)
                        printf("%d", *(a+i));
}
```

In one dimensional array, a[i] element is referred by

(a+i) is the address of ith element.

* (a+i) is the value at the ith element.

In two-dimensional array, a[i][j] element is represented as

((a+i)+j)

STRUCTURES :

A Structure is a collection of elements of dissimilar data types. Structures provide the ability to create user defined data types and also to represent real world data.

Suppose if we want to store the information about a book, we need to store its name (String), its price (float) and number of pages in it(int). We have to store the above three items as a group then we can use a structure variable which collectively store the information as a book.

Structures can be used to store the real world data like employee, student, person etc.

Declaration :

The declaration of the Structure starts with a Key Word called Struct and ends with ; . The Structure elements can be any built in types.

```
struct <Structure name>
{
        Structure element 1;
        Structure element 2;
                -
                -
                -
        Structure element n;
};
```

Then the Structure Variables are declared as

```
        struct < Structure name > <Var1,Var2>;

        Eg:-    struct emp
                {
                        int empno.
                        char ename[20];
                        float sal;
                };
        struct emp  e1,e2,e3;
```

The above Structure can also be declared as :

```
struct emp
{
        int empno;
        char ename[20];
        float sal;
}e1,e2,e3;
```

(or)

```
struct
{
int empno;
char ename[20];
float sal;
}e1,e2,e3;
```

Initialization :

Structure Variables can also be initialised where they are declared.

```
struct emp
{
        int empno;
        char ename[20];
        float sal;
};
struct emp e1 = { 123,"Kumar",5000.00};
```

To access the Structure elements we use the .(dot) operator.
To refer empno we should use e1.empno
To refer sal we whould use e1.sal

Structure elements are stored in contiguous memory locations as shown below. The above Structure occupies totally 26 bytes.

e1.empno	E1.ename	E1.sal
123	Kumar	5000.00
2000	2002	2022

1. Program to illustrate the usage of a Structure.

```
main()
{
    struct emp
    {
            int empno;
            char ename[20];
            float sal;
    };
    struct emp e;
    printf (" Enter Employee number: \n");
    scanf("%d",&e.empno);
    printf (" Enter Employee Name: \n");
    scanf("%s",&e.empname);
    printf (" Enter the Salary: \n");
    scanf("%f",&e.sal);
    printf (" Employee No = %d", e.empno);
    printf ("\n Emp Name = %s", e.empname);
    printf ("\n Salary = %f", e.sal);
}
```

/* Program to read Student Details and Calculate total and average using structures */

```
#include<stdio.h>
main()
{
        struct stud
        {
                int rno;
                char sname[20];
```

```
		int m1,m2,m3;
	};
	struct stud s;
	int tot;
	float avg;

	printf("Enter the student roll number: \n");
	scanf("%d",&s.rno);
	printf("Enter the student name: \n");
	scanf("%s",&s.sname);
	printf("Enter the three subjects marks: \n");
	scanf("%d%d%d",&s.m1,&s.m2,&s.m3);

	tot = s.m1 + s.m2 +s.m3;
	avg = tot/3.0;



	printf("Roll Number : %d\n",s.rno);
	printf("Student Name: %s\n",s.sname);
	printf("Total Marks : %d\n",tot);
	printf("Average : %f\n",avg);
}
```

/* Program to read Item Details and Calculate Total Amount of Items*/

```
#include<stdio.h>
main()
{
	struct item
	{
		int itemno;
		char itemname[20];
		float rate;
		int qty;
	};
	struct item i;
	float tot_amt;

	printf("Enter the Item Number \n");
	scanf("%d",&i.itemno);
	printf("Enter the Item Name \n");
	scanf("%s",&i.itemname);
	printf("Enter the Rate of the Item \n");
	scanf("%f",&i.rate);
	printf("Enter the number of %s purchased ",i.itemname);
```

```
        scanf("%d",&i.qty);

        tot_amt = i.rate * i.qty;

        printf("Item Number: %d\n",i.itemno);
        printf("Item Name: %s\n",i.itemname);
        printf("Rate: %f\n",i.rate);
        printf("Number of Items: %d\n",i.qty);
        printf("Total Amount: %f",tot_amt);
}
```

ARRAY OF STRUCTURES :

To store more number of Structures, we can use array of Structures. In array of Structures all elements of the array are stored in adjacent memory location.

/* Program to illustrate the usage of Array of Structures.

```
main()
{
        struct book
        {
                char name[20];
                float price;
                int pages;
        };
        struct book b[10];
        int i;
        for (i=0;i<10;i++)
        {
                print("\n Enter Name, Price and Pages");
                scanf("%s%f%d", b[i].name,&b[i].price,&b[i].pages);
        }
        for (i i=0;i<10;i++)
                printf(" \n%s%f%d", b[i].name,b[i].price,b[i].pages);
}
```

UNIONS :

Union, similar to Structures, are collection of elements of different data types. However, the members within a union all share the same storage area within the computer‟s memory, whereas each member within a Structure is assigned its own unique Storage area.

Structure enables us to treat a member of different variables stored at different places in memory, a union enables us to treat the same space in memory as a number of different variables. That is, a union offers a way for a section of memory to be treated as a variable of one type on one occasion and as a different variable of a different type on another occasion.

Unions are used to conserve memory. Unions are useful for applications involving multiple members, where values need not be assigned to all of the members at any given time. An attempt to access the wrong type of information will produce meaningless results.

The union elements are accessed exactly the same way in which the structure elements are accessed using dot(.) operator.

The difference between the structure and union in the memory representation is as follows.

```
struct cx
{
        int i;
        char ch[2];
};
struct cx s1;
```

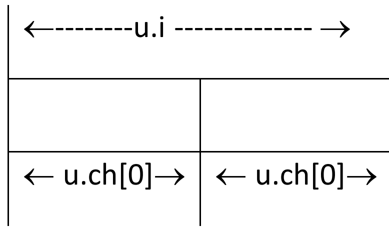
The Variable s1 will occupy 4 bytes in memory and is represented as

← ---------- s.i---------→		← s.ch[0]→	← s.ch[1]→
4002	4003	4004	4005

}

The above datatype, if declared as union then

```
union ex
{
        int i;
        char ch[2];
}
union ex u;
```



/* Program to illustrate the use of unions */

```
main()
{
    union example
    {
        int i;
        char ch[2];
    };
```

```
union exaple u;

u.i = 412;

print("\n u.i = %d",u.i);
print("\n u.ch*0+ = %d",u.ch*0+);
print("\n u.ch*1+ = %d",u.ch*1+);

u.ch[0] = 50; /* Assign a new value */

print("\n\n u.i = %d",u.i);
print("\nu.ch*0+=%d",u.ch[0]);
print("\n u.ch*1+ =%d",u.ch*1+);
}
```

Output :

```
u.i = 512
u.ch[0] = 0
u.ch[1] =2
u.i = 562
u.ch[0] = 50
u.ch[1] =2
```

A union may be a member of a structure, and a structure may be a member of a union. And also structures and unions may be freely mixed with arrays.

/* Program to illustrate union with in a Structure */

```
main()
{

    union id
    {

        char color;
        int size;

    };

    struct {

        char supplier[20];
        float cost;
        union id desc;

    }pant, shirt;
    printf("%d\n", sizeof(union id));
    shirt.desc.color = 'w';
    printf("%c %d\n", shirt.desc.color, shirt.desc.size);
    shirt.desc.size = 12;
    printf("%c %d\n", shirt.desc.color, shirt.desc.size);
}
```

UNIT IV

to search an element in a given array, there are two popular algorithms available:

1. Linear Search
2. Binary Search

Linear Search

Linear search is a very basic and simple search algorithm. In Linear search, we search an element or value in a given array by traversing the array from the starting, till the desired element or value is found.

It compares the element to be searched with all the elements present in the array and when the element is **matched** successfully, it returns the index of the element in the array, else it return `-1`.

Linear Search is applied on unsorted or unordered lists, when there are fewer elements in a list.

It has a time complexity of **O(n)**, which means the time is linearly dependent on the number of elements, which is not bad, but not that good too.

```
int linearSearch(int values[], int target, int n)
{
for(int i = 0; i < n; i++)
{
if (values[i] == target)
{
return i;
}
}
return -1;
}
```

Binary Search

Binary Search is used with sorted array or list. In binary search, we follow the following steps:

1. We start by comparing the element to be searched with the element in the middle of the list/array.
2. If we get a match, we return the index of the middle element.
3. If we do not get a match, we check whether the element to be searched is less or greater than in value than the middle element.
4. If the element/number to be searched is greater in value than the middle number, then we pick the elements on the right side of the middle element(as the list/array is sorted, hence on the right, we will have all the numbers greater than the middle number), and start again from the step 1.
5. If the element/number to be searched is lesser in value than the middle number, then we pick the elements on the left side of the middle element, and start again from the step 1.

Binary Search is useful when there are large number of elements in an array and they are sorted.

It has a time complexity of **O(log n)** which is a very good time complexity.

```
int binarySearch(int values[], int len, int target)
{
    int max = (len - 1);
```

```
    int min = 0;

    int guess;  // this will hold the index of middle elements
    int step = 0;  // to find out in how many steps we completed the search

    while(max >= min)
    {
        guess = (max + min) / 2;
        // we made the first guess, incrementing step by 1
        step++;

        if(values[guess] ==  target)
        {
            printf("Number of steps required for search: %d \n", step);
            return guess;
        }
        else if(values[guess] >  target)
        {
            // target would be in the left half
            max = (guess - 1);
        }
        else
        {
            // target would be in the right half
            min = (guess + 1);
        }
    }

    // We reach here when element is not
    // present in array
    return -1;
}

int main(void)
{
    int values[] = {13, 21, 54, 81, 90};
    int n = sizeof(values) / sizeof(values[0]);
    int target = 81;
    int result = binarySearch(values, n, target);
    if(result == -1)
    {
        printf("Element is not present in the given array.");
    }
    else
    {
        printf("Element is present at index: %d", result);
    }
    return 0;
}
```

Different Sorting Algorithms

There are many different techniques available for sorting, differentiated by their efficiency and space requirements. Following are some sorting techniques.

1. Bubble Sort
2. Insertion Sort
3. Selection Sort
4. Quick Sort
5. Merge Sort

6. Heap Sort

Bubble Sort Algorithm

Bubble Sort is a simple algorithm which is used to sort a given set of `n` elements provided in form of an array with `n` number of elements. Bubble Sort compares all the element one by one and sort them based on their values.

If the given array has to be sorted in ascending order, then bubble sort will start by comparing the first element of the array with the second element, if the first element is greater than the second element, it will **swap** both the elements, and then move on to compare the second and the third element, and so on.

If we have total `n` elements, then we need to repeat this process for `n-1` times.

It is known as **bubble sort**, because with every complete iteration the largest element in the given array, bubbles up towards the last place or the highest index, just like a water bubble rises up to the water surface.

Sorting takes place by stepping through all the elements one-by-one and comparing it with the adjacent element and swapping them if required.

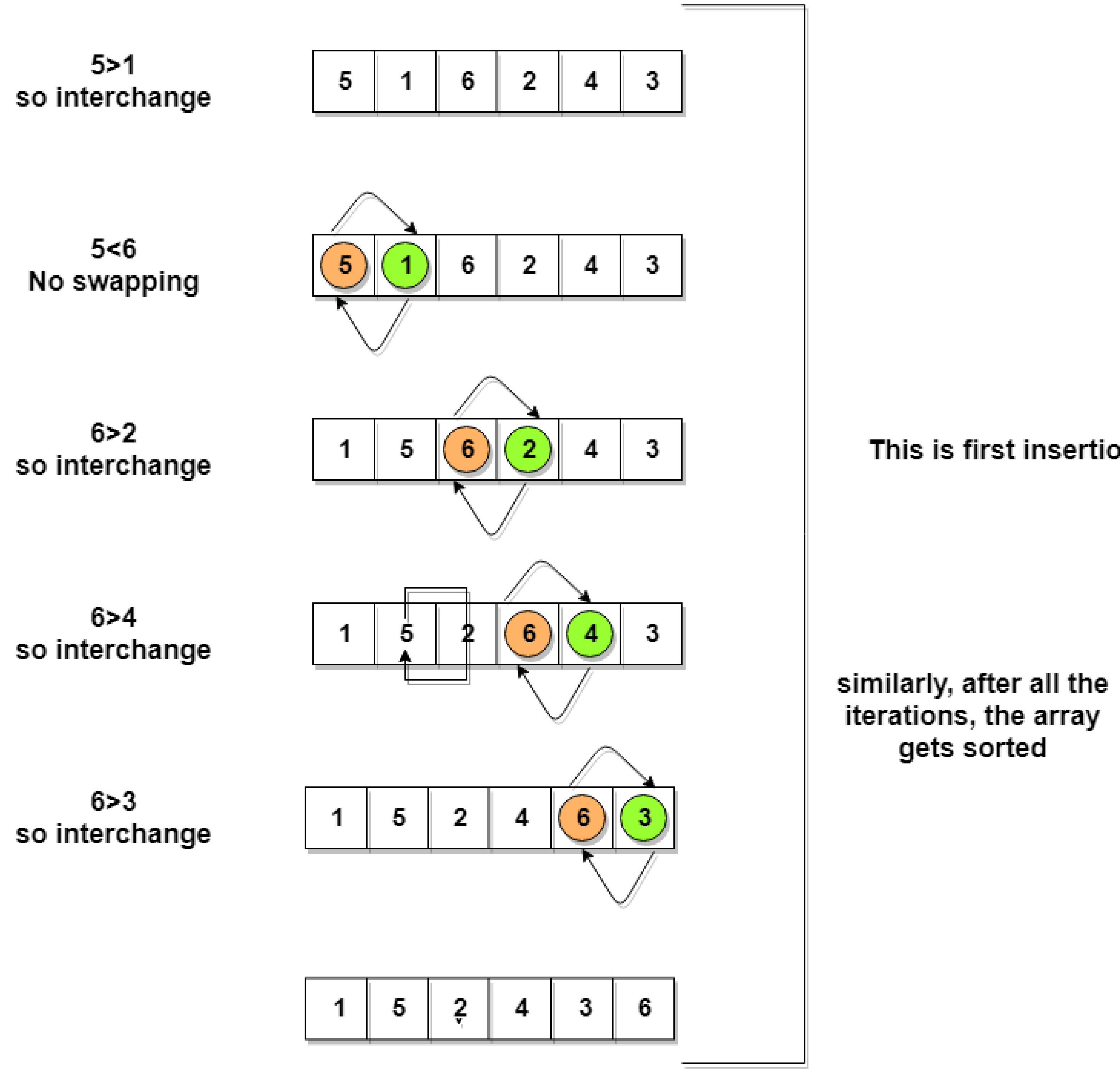
Implementing Bubble Sort Algorithm

Following are the steps involved in bubble sort(for sorting a given array in ascending order):

1. Starting with the first element(index = 0), compare the current element with the next element of the array.
2. If the current element is greater than the next element of the array, swap them.
3. If the current element is less than the next element, move to the next element. **Repeat Step 1**.

Let's consider an array with values `{5, 1, 6, 2, 4, 3}`

Below, we have a pictorial representation of how bubble sort will sort the given array.



So as we can see in the representation above, after the first iteration, `6` is placed at the last index, which is the correct position for it.

Similarly after the second iteration, `5` will be at the second last index, and so on.

Time to write the code for bubble sort:

```
// below we have a simple C program for bubble sort
#include <stdio.h>

void bubbleSort(int arr[], int n)
{
    int i, j, temp;
```

```
    for(i = 0; i < n; i++)
    {
        for(j = 0; j < n-i-1; j++)
        {
            if( arr[j] > arr[j+1])
            {
                // swap the elements
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }

    // print the sorted array
    printf("Sorted Array: ");
    for(i = 0; i < n; i++)
    {
        printf("%d  ", arr[i]);
    }
}

int main()
{
    int arr[100], i, n, step, temp;
    // ask user for number of elements to be sorted
    printf("Enter the number of elements to be sorted: ");
    scanf("%d", &n);
    // input elements if the array
    for(i = 0; i < n; i++)
    {
        printf("Enter element no. %d: ", i+1);
        scanf("%d", &arr[i]);
    }
    // call the function bubbleSort
    bubbleSort(arr, n);

    return 0;
}
```

Hence the **time complexity** of Bubble Sort is **$O(n^2)$**.

The main advantage of Bubble Sort is the simplicity of the algorithm.

The **space complexity** for Bubble Sort is **O(1)**

Following are the Time and Space complexity for the Bubble Sort algorithm.

- Worst Case Time Complexity [Big-O]: **$O(n^2)$**
- Best Case Time Complexity [Big-omega]: **O(n)**
- Average Time Complexity [Big-theta]: **$O(n^2)$**
- Space Complexity: **O(1)**

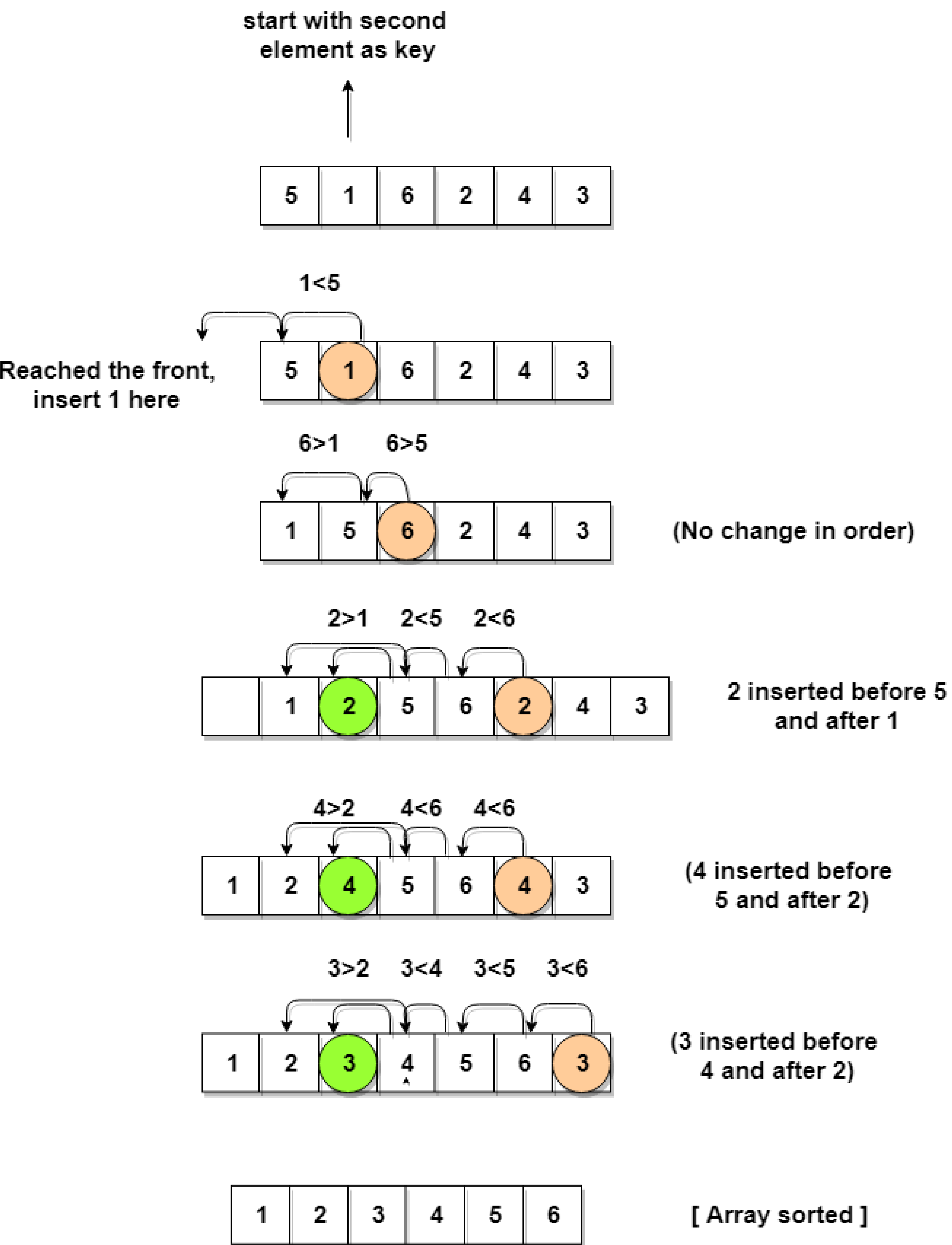
Insertion Sort

Following are the steps involved in insertion sort:

1. We start by making the second element of the given array, i.e. element at index `1`, the `key`. The `key` element here is the new card that we need to add to our existing sorted set of cards(remember the example with cards above).
2. We compare the `key` element with the element(s) before it, in this case, element at index `0`:
   - If the `key` element is less than the first element, we insert the `key` element before the first element.
   - If the `key` element is greater than the first element, then we insert it after the first element.
3. Then, we make the third element of the array as `key` and will compare it with elements to it's left and insert it at the right position.
4. And we go on repeating this, until the array is sorted.

Let's consider an array with values `{5, 1, 6, 2, 4, 3}`

Below, we have a pictorial representation of how bubble sort will sort the given array.



As you can see in the diagram above, after picking a `key`, we start iterating over the elements to the left of the `key`.

We continue to move towards left if the elements are greater than the `key` element and stop when we find the element which is less than the `key` element.

And, insert the `key` element after the element which is less than the `key` element.

Implementing Insertion Sort Algorithm

Below we have a simple implementation of Insertion sort in C++ language.

```
#include <stdlib.h>
#include <iostream>

using namespace std;

//member functions declaration
void insertionSort(int arr[], int length);
void printArray(int array[], int size);

// main function
int main()
{
	int array[5] = {5, 1, 6, 2, 4, 3};
	// calling insertion sort function to sort the array
	insertionSort(array, 6);
	return 0;
}

void insertionSort(int arr[], int length)
{
	int i, j, key;
	for (i = 1; i < length; i++)
	{
		j = i;
		while (j > 0 && arr[j - 1] > arr[j])
		{
			key = arr[j];
			arr[j] = arr[j - 1];
			arr[j - 1] = key;
			j--;
		}
	}
	cout << "Sorted Array: ";
	// print the sorted array
	printArray(arr, length);
}

// function to print the given array
void printArray(int array[], int size)
{
	int j;
	for (j = 0; j < size; j++)
	{
		cout <<" "<< array[j];
	}
	cout << endl;
}
```

Sorted Array: 1 2 3 4 5 6

Worst Case Time Complexity [Big-O]: **O(n^2)**

Best Case Time Complexity [Big-omega]: **O(n)**

Average Time Complexity [Big-theta]: **O(n^2)**

Space Complexity: **O(1)**

Quick Sort Algorithm

Quick Sort is based on the concept of **Divide and Conquer**. It is also called **partition-exchange sort**. This algorithm divides the list into three main parts:

1. Elements less than the **Pivot** element
2. Pivot element(Central element)
3. Elements greater than the pivot element

Pivot element can be any element from the array, it can be the first element, the last element or any random element. In this tutorial, we will take the rightmost element or the last element as **pivot**.

1. After selecting an element as **pivot**, which is the last index of the array in our case, we divide the array for the first time.
2. In quick sort, we call this **partitioning**. It is not simple breaking down of array into 2 subarrays, but in case of partitioning, the array elements are so positioned that all the elements smaller than the **pivot** will be on the left side of the pivot and all the elements greater than the pivot will be on the right side of it.
3. And the **pivot** element will be at its final **sorted** position.
4. The elements to the left and right, may not be sorted.
5. Then we pick subarrays, elements on the left of **pivot** and elements on the right of **pivot**, and we perform **partitioning** on them by choosing a **pivot** in the subarrays

Implementing Quick Sort Algorithm

Below we have a simple C program implementing the Quick sort algorithm:

```
// simple C program for Quick Sort
# include <stdio.h>

// to swap two numbers
void swap(int* a, int* b)
{
    int t = *a;
    *a = *b;
    *b = t;
}

/*
    a[] is the array, p is starting index, that is 0,
    and r is the last index of array.
*/
void quicksort(int a[], int p, int r)
```

```
{
    if(p < r)
    {
        int q;
        q = partition(a, p, r);
        quicksort(a, p, q);
        quicksort(a, q+1, r);
    }
}

int partition (int a[], int low, int high)
{
    int pivot = arr[high];  // selecting last element as pivot
    int i = (low - 1);  // index of smaller element

    for (int j = low; j <= high- 1; j++)
    {
        // If current element is smaller than or equal to pivot
        if (arr[j] <= pivot)
        {
            i++;    // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

// function to print the array
void printArray(int a[], int size)
{
    int i;
    for (i=0; i < size; i++)
    {
        printf("%d ", a[i]);
    }
    printf("\n");
}

int main()
{
    int arr[] = {9, 7, 5, 11, 12, 2, 14, 3, 10, 6};
    int n = sizeof(arr)/sizeof(arr[0]);

    // call quickSort function
    quickSort(arr, 0, n-1);

    printf("Sorted array: n");
    printArray(arr, n);
    return 0;
}
```

Complexity Analysis of Quick Sort

Worst Case Time Complexity [Big-O]: **O(n^2)**

Best Case Time Complexity [Big-omega]: **O(n*log n)**

Average Time Complexity [Big-theta]: **O(n*log n)**

Space Complexity: **O(n*log n)**

1. Define keyword, constant and variable.
2. Define relational operator?
3. Write detailed notes on C data types.
4. Write an algorithm, flowchart and C program to find the sum of numbers from 1 to n.
5. Discuss about the following operators in C language with example.
   a. Bitwise operators
   b. Increment and decrement operators
   c. Logical operators
6. Define algorithm. Write algorithm for finding factorial of a number.
7. Draw the flowchart to find the greatest of three numbers.
8. Write an algorithm and flowchart to find whether the given number is prime or not.
9. Write and explain syntax of —for loop.
10. Distinguish between while and do-while statements.
11. Define an array. How to initialize one-dimensional array? Explain with suitable examples.
12. What are the advantages of functions?
13. What is recursion? What are the advantages and Disadvantages of recursion?
14. Define pointer. How can you declare it?
15. Write a program to find the average marks obtained by a class of 50 students in a test.
16. Explain the following string handling functions with example: a.strcpy() b. strcmp() c. strcat() d.strlen() e. strncat()
17. Define array of structures.
18. Write some of the differences between Structure and Union?
19. What are the various searching algorithms? Explain
20. What are the various sorting algorithms? Write a program for insertion sort.