PROGRAMMING IN C (THEORY)

M.A./M.Sc. Mathematics (Final)

MM-503

Directorate of Distance Education
Maharshi Dayanand University
ROHTAK – 124 001

Copyright © 2004, Maharshi Dayanand University, ROHTAK
All Rights Reserved. No part of this publication may be reproduced or stored in a retrieval system or transmitted in any form or by any means; electronic, mechanical, photocopying, recording or otherwise, without the written permission of the copyright holder.

Maharshi Dayanand University
ROHTAK – 124 001

Contents

UNIT-I 5

1. Computer Basics and Programming
2. Introduction to C
3. Program Development
4. Functions
5. Constants and Variables
6. Elementary I/O Functions
7. Scalar Data Types
8. Introduction to Pointers
9. Control Flow
10. Looping

UNIT-II 48

1. Operators and Expressions
2. C Operators
3. Bit-Manipulation Operators

UNIT-III 73

1. Arrays and Pointers
2. Pointer Arithmetic
3. Strings
4. String-handling Functions
5. Multi-dimensional Arrays

UNIT-IV 103

1. Storage Classes
2. Dynamic Memory Allocation
3. Structures
4. Unions
5. Linked List
6. Passing Arguments to a Function

UNIT-V 131

1. C Preprocessor
2. Compiler Control Directives
3. Line Control
4. Input and Ouput
5. File Management in C
6. Selecting an I/O Method

M.A./M.Sc. Mathematics (Final)

PROGRAMMING IN C (THEORY)

MM-503

Max. Marks : 100

Time : 3 Hours

Note: Question paper will consist of three sections. Section I consisting of one question with ten parts covering whole of the syllabus of 2 marks each shall be compulsory. From Section II, 10 questions to be set selecting two questions from each unit. The candidate will be required to attempt any seven questions each of five marks. Section III, five questions to be set, one from each unit. The candidate will be required to attempt any three questions each of fifteen marks.

Unit I

An overview of programming. Programming language, Classification.

C Essentials-Program Development. Functions. Anatomy of a C function. Variables and Constant. Expressions. Assignment Statements. Formatting Source Files. Continuation Character. The Preprocessor.

Scalar Data Types-Declarations, Different Types of Integers. Different kinds of Integer Constants. Floating-Point Types. Initialization. Mixing Types. Explicit Conversions — Casts. Enumeration Types. The Void Data Type. Typedefs. Finding the Address of an object. Pointers.

Control Flow-Conditional Branching. The Switch Statement. Looping. Nested Loops. The break and continue Statements. The goto statement. Infinite Loops.

Unit II

Operators and Expressions — Precedence and Associativity. Unary Plus and Minus operators. Binary Arithmetic Operators. Arithmetic Assignment Operators. Increment and Decrement Operators. Comma Operator. Relational Operators. Logical Operators. Bit – Manipulation Operators. Bitwise Assignment Operators. Cast Operator. Size of Operators. Conditional Operators. Memory Operators.

Unit III (2 Questions)

Arrays and Pointers — Declaring an Array. Arrays and Memory. Initializing Arrays, Encryption and Decryption. Pointer Arithmetic. Passing Pointers as Function Arguments. Accessing Array Elements through Pointers. Passing Arrays as Function Arguments. Sorting Algorithms. Strings. Multidimensional Arrays. Arrays of Pointers. Pointers to Pointers.

Unit IV

Strong Classes — Fixed vs. Automatic Duration. Scope. Global variables. The register Specifier. ANSI rules for the syntax and Semantics of the storage – class keywords. Dynamic Memory Allocation.

Structures and Unions-Structures. Linked Lists. Unions. Enum Declarations.

Functions – Passing Arguments. Declarations and Calls. Pointers to Functions. Recursion. The main () Function. Complex Declarations.

Unit V

The C Preprocessor – Macro Substitution. Conditional Compilation. Include Facility. Line Control.

Input and Output-Streams, Buffering. The <Stdio. H> header File. Error Handling. Opening and Closing a File. Reading and Writing Data. Selecting an I/O Method. Unbuffered I/O Random Access. The standard library for input/output.

UNIT - I

1. Computer Basics and Programming

Computer is composed of three major components :

(i) Hardware (which we can touch upon)
(ii) Software (operations, instructions etc)
(iii) Liveware (users/Human beings who interact with H/W & S/W)

Software can be classified in two main categories

(a) System Software which is used to run computer system
(b) Application Software which is used to solve specific problem/applications

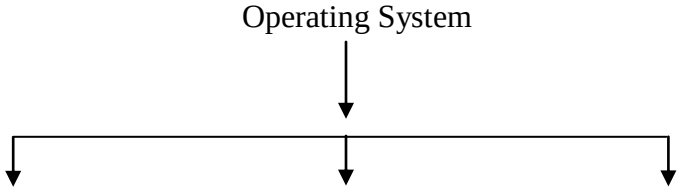
System software can be divided into the following types.

(i) Operating System which is a group of programs that acts as an interface between the user and machine and as a resource manager
(ii) Translators which convert the programs in any language into machine language.
(iii) Loaders/Linkers
(iv) System Utilities

Translators are of two types ;

(a) **Compilers :** These are of two types : Interactive (IDE), Non-interactive (e.g. FORTRAN compiler)
(b) **Interpreter** (e.g. BASIC Language interpreter)

Operating Systems can be categorised as follows

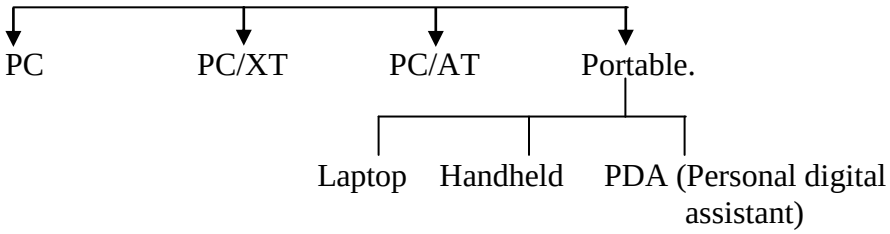


Single-user	Multi-user	Multi-tasking
(A single user can work on a single machine)	(More than one user can work on a single machine)	(More than one task can be performed on a single machine by single as well as more than one users)
e.g. MS-DOS CP/M (control program etc for Microcomputer)	e.g. UNIX (AT & T) LINUX VMS (Virtual Machine System etc	e.g. UNIX Windows 9x/2000 Windows NT etc

1.1. Classification of Computers : Computers used presently are called digital computers. Examples of digital computers are

PC (Personal Computers)
PC|XT (PC with Extended Technology)
PC|AT (PC with Advanced Technology)
Classification of digital computers is as follows :

(i) Microcomputers : All Pentium I, II, III, IV are in this category. These are of four types as shown below



(ii) Minicomputers : These have more than one terminals for different users – upto 32 approx. UNIX operating system is must for such computers

(iii) Mainframe Computers : These support more than 32 users, upto 128 users.
e.g. VAX 8350
One such at Punjab University Chandigarh (supports all researchers & student of the university)
These have VMS Operating System.

(iv) Super Computers : These are fastest computers having multiple processors. These are useful for some specific areas of applications such as

– Weather forecasts

– Fluid Dynamics

– Vector Processing

– Molecular Dynamics.

One such at Meteorological Deptt. at Delhi, is
CRAY XMP-14, which provides all weather bulletins.
(Seymour Cray - father of super computers)
others are
PARAM Supercomputer (Developed by C-DAC, Pune).
ANURAG Supercomputer (installed at BARC, Bombay).

(v) Ultra Computers : These are still to be created/designed but with good intelligence/knowledge like human beings.

1.2. An Overview of Programming : Although computers can perform complex and difficult operations, they need to be told exactly what to do and they must be instructed in a precise and limited language that they can

understand. These instructions are known as **software.** The machinery that actually executes the instructions is known as **hardware.** At the hardware level, computers understand only simple commands such as "copy this number", "add these three numbers", "compare these two numbers" etc. Such commands constitute the computer's instruction set and programs written in the computer's **machine language.** It is extremely tiresome to write programs in machine language because even the simplest tasks require many instructions. Moreover, in most machine languages, everything such as instructions, data, variables etc, are represented by **binary numbers.** Binary numbers are composed of zeros and ones, each digit is called a **bit** which is the short form of binary digit. These programs, consisting of a jumble of zeros and ones are difficult to write, read and maintain.

In early stages (1940s & 1950s), all programs were written in machine language or its improved version, known as **assembly language.** In assembly language, each instruction is identified by a short name rather than a number and variables can be identified by names rather than numbers. Programs written in assembly language require a special program, called an **assembler,** to translate assembly language instructions into machine instructions. These days, programs are written in assembly language only when execution speed is a high priority.

Presently, majority of programs are written in languages called **high-level languages** which were first developed in the 1950s and 1960s. High-level languages allow programmers to write programs in language more natural to them than the computer's restrictive language. Broadly speaking, programming languages can be viewed as lying along a spectrum with machine languages at one end and human languages, such as English, French, Russian etc, at the other end. High-level languages fall somewhere in between these extremes, usually closer to the machine language. High-level languages allow programmers to deal with complex objects without worrying about details of the particular computer on which the program is running. Of course, programming languages differ from human languages since they are designed specially to manipulate informations. They are much more limited and precise than human languages.

1.3. Programming Languages, Classification : There are many different languages which can be used for computer programming. Generally, the programming languages can be divided into two categories i.e. low-level language and high-level language (HLL). In the first category, we have machine language and assembly language. In the H.L.L. category, we have FORTRAN, BASIC, PASCAL, COBOL etc. A single instruction in HLL can produce many machine instructions. This greatly simplifies the task of writing complete, correct program. Every HLL requires a **compiler or interpreter** to translate instructions in the HLL into low-level instructions that the computer can execute. This process is known as **compilation** or

interpretation, depending upon how it is carried out. A compiler is similar to an assembler, but much more complex. Compilers translate the entire program into machine language before executing any of the instructions. Interpreters, on the other hand, proceed through a program by translating and then executing single instruction or small group of instructions. In either case, the translation is carried out automatically within the computer. It should be noted that compiler or interpreter is itself a computer program. It accepts a program written in a HLL as input and generates a corresponding machine language program as output. The original HLL program is called the **source program** and the resulting machine language program is called the **object program.** Every computer must have its own compiler or interpreter for a particular HLL. The farther a programming language is from a machine language, the more difficult it is for the compiler to perform its task.

A HLL offers three significant advantages over machine language. These are readability (simplicity), portability (i.e. machine independence) and maintainability. Despite these advantages, the significant drawback of HLL is reduced efficiency. This is due to the fact that when a compiler translates programs into machine language, it may not translate them into the most efficient machine code. Nevertheless, HLL are superior to machine and assembly languages in most instances. Also sophisticated compilers can perform tricks to gain efficiency that most assembly language programmers would never think of. The main reason for the superiority of HLL, however, is that most of the cost of software development lies in maintenance, where readability and portability are crucial. These three concepts can be taken care of using careful programming.

2. Introduction to C

The C language was first developed in 1972 by Dennis M. Ritchie at Bell Telephone Laboratories Inc. (now a part of AT & T) as a system programming language i.e. a language to write operating systems and system utilities. An operating system is a program that manages the resources of a computer besides functioning as an interface between the user and the machine. It controls the entire operations of the computer. The resources of a computer include the CPU, main memory, disks and other devices such as printer that are connected to the computer. Ritchie's intent in designing C was to give programmers a convenient means of accessing a machine's instruction set. This meant creating a language that was high-level enough to make programs readable and portable, but simple enough to map easily onto the underlying machine. We can call C as a middle-level language since it stands between the low-level and HLL categories i.e. it results in good programming efficiency as well as good machine efficiency. C is so flexible and enables compilers to produce such efficient machine code that in 1973, Ritchie and Ken Thompson rewrote most of UNIX operating systems in C.

C is characterised by the ability to write very concise source programs due to the large number of operators included within the language. It has a relatively small instruction set, though actual implementations include extensive library functions which enhance the basic instructions. Furthermore, the language encourages users to write additional library functions of their own. Thus the features and capabilities of the language can easily be extended by the user. C compilers are commonly available for computers of all sizes and C interpreters are becoming increasingly common. Another important characteristic of C is that its programs are highly portable, even more than those with high-level languages. Therefore, most C programs can be processed on many different computers with little or no alteration.

During the earlier stages, C was considered strictly a UNIX systems language. The version of C used on UNIX systems is known as PCC (portable C compiler). It is only in recent years that C has come to be viewed as a more general-purpose programming language. Throughout most of its history, the only formal specification for the C language was a document written by Ritchie entitled 'The C Reference Manual'. In 1977, Ritchie and Brian Kernigham expanded this document into a full-length book called 'The C programming Language' often referred to as the K & R standard. With the emergence of personal computers (PC) and the growing popularity of C, K & R and PCC standards were no longer satisfactory. At that time, there were many variants of C, each differing in little ways. Finally, the American National Standard Institute (ANSI) formed a committee to define an official version of the C language. In 1983, the committee met for the first time and they have been meeting four times a year since then. The final version of the C standard was satisfied as an ANSI standard in 1989. The ANSI standard for the C language is specified in a document entitled American National Standard for Information Systems – Programming Language C. The address to obtain copies of ANSI standard, is

American National Standard Institute,
1430 Broadway,
New, York, NY 10018.

The C programming language has acquired the reputation for being a mysterious and messy language that promotes bad programming habits. Part of the problem is that C gives special meanings to many punctuation characters, such as asterisks, plus sign, braces, angle brackets etc. which can be intimidating to the beginners. The other, more serious, complaint concerns the relative dearth of rules i.e. C does not have very strict rules (as in PASCAL, FORTRAN etc) to protect programmers from making accidental blunders. Also, C programmers have tremendous liberty to write unusual code. In many instances, this freedom allows them to write useful programs that would be difficult to write in other languages. However, the freedom may be abused by inexperienced programmers who delight in writing

needlessly tricky code. Thus we conclude that C is a powerful language but it requires self-restraint and discipline. Also it should be kept in mind that there is a huge difference between good programs and working programs. A good program not only works, but is easy to read and maintain. It is very possible to write good programs in C language.

2.1. Versions of C : The C language is the same whether it is used to program in UNIX operating system or DOS operating system. UNIX only adds a set of functions that helps us to utilize the power of the operating system. Depending on the compiler being used, we have different versions of C. Even for the same computer, there may be several compilers, each with its own specification. For example, for IBM PC or its compatibles, we have Microsoft C compiler and Quick C compiler from Microsoft Corporation, and Turbo C compiler from Borland International. Similarly, for machine running UNIX operating system, we have a version of C, known as ANSI C. The developers of Turbo C version of C language have added more features to ANSI version. Almost all the programs written according to ANSI standard can be run under Turbo C environment.

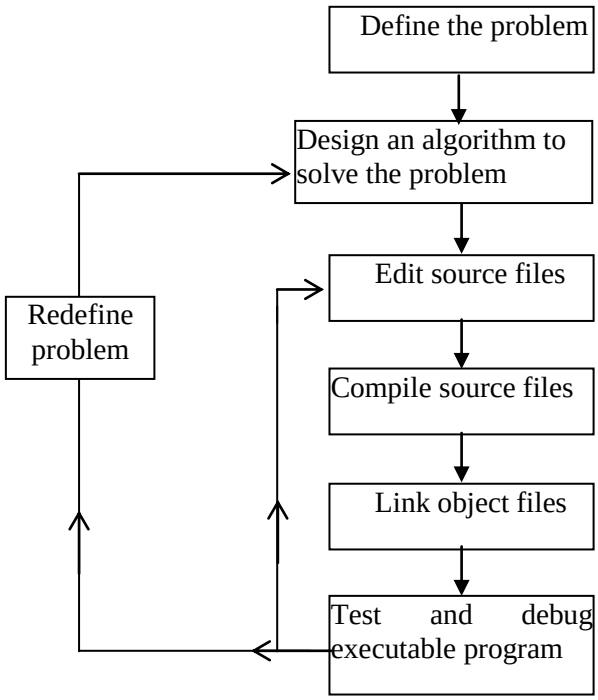
2.2. Remark : In the early 1980s, another high-level programming language, called C++, was developed by Bjarne Stroustrup at the Bell laboratories. C++ was originally called " C with classes". C++ is built upon C, and hence all standard C features are available within C++. However, C++ is not merely an extension of C. Rather, it incorporates several new fundamental concepts that form a basis for object-oriented programming. C++ is a superset of C, so programmers can use a C++ compiler to compile existing C programs, and then gradually evolve those programs to C++.

The most popular compilers for C++ are Turbo C++ and Borland C++, both produced by Borland International.

2.3. C-Essentials : One of the hardest parts about learning a programming language is that everything is interrelated. It often seems impossible to understand anything before we know everything. We shall describe the C essentials which we need to know to write our first program. First we discuss the development of a program.

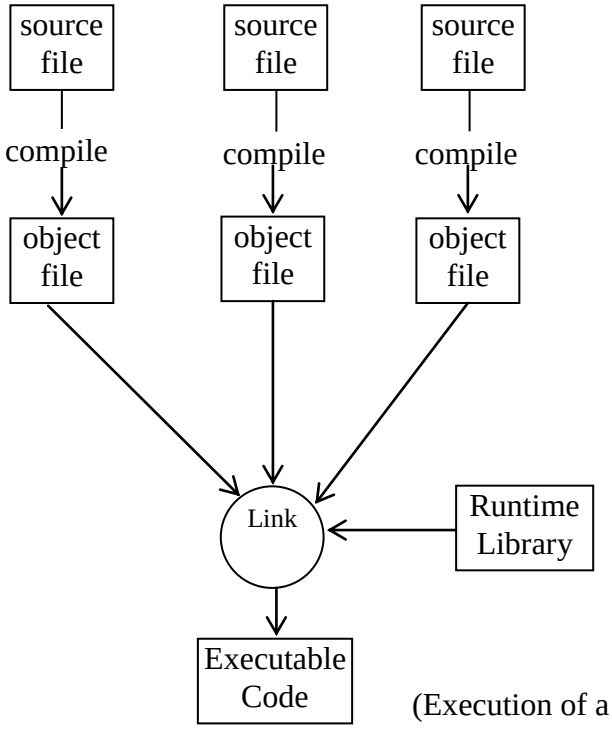
3. Program Development

Program development consists of a number of steps, as shown in the figure. The first step in developing a program is to clearly define the problem and design an algorithm to solve it. An algorithm is a well-defined set of rules to solve a particular problem in a finite number of steps. This stage of development process is extremely important and should be given special attention by the beginners. At present, we are concerned with later stages of software development that occur after one has defined the problem and designed an algorithm. Some of these stages vary from one computing environment to another. We should read the system documentation for our computer to find out how to execute programs in our particular environment.



The execution of a program written in C, involves the following three general steps.

1. Edit each source file.
2. Compile each source file to produce an object file.
3. Link the object files together to produce an executable program.



(Execution of a C program)

It should be noted that source and object code can be spread out in multiple files, but the executable code for a program generally lies in a single file. Although the above mentioned three steps remain the same irrespective of the operating system, system commands for implementing these steps and conventions for naming files may differ on different systems. The two most popular operating system (environments) are MS-DOS (for microcomputers) and UNIX (for minicomputers). The process of executing a C program is slightly different under DOS than under UNIX and hence we shall discuss the two separately. The above three steps collectively are sometimes termed as running C program.

3.1. Running C Programs in DOS :A C program must be converted into an executable file (i.e. a file with. EXE or .COM extension) to run it in DOS. First, the C program has to be typed in a file using an editor. This file is called the source file. Often, C compilers in DOS have an editor of their own. The actual compiler program comes in the following two forms

(i) As an IDE (Integrated Development Environment) : The IDE includes all facilities to develop and run programs, such as the editor, compiler, debugger and so on. In case of Turbo C and Turbo C++, the IDE is a program called TC. EXE. In case of Borland C++, the IDE is BC. EXE. So the IDE is invoked by typing a command such as tc or bc at the DOS prompt. After issuing one of the above two commands, press Alt-F to open the file menu and then choose the open command (or press F3 key). Type the file name of the program using C extension (such as hello. C, xyz. C, day 1.c) in the text box. Since the file does not exist, a blank edit window is opened with a blinking cursor in it. Type the program here and do not forget that programs are usually saved in files with a. c extension (with . cpp for C++). Choose save under the file menu to save the file (or press F2 key). Now, press Ctrl-F9 to run the program. Assuming that no typing mistake are committed, the screen will flicker for a moment. Actually, the program has been executed by the IDE. To view the output screen, press Alt-F5. Press any key to get back to the main IDE.

(ii) As a Command-line Compiler : This comes as a program called TCC. EXE in Turbo C and Turbo C++. The program is called BCC. EXE in Borland C++. These are programs that have to be invoked from the command line, and their purpose is to convert the C program into an executable file. For example, assuming that the file hello .C has been created using an editor, the command tcc hello.c typed at the DOS prompt will convert the C program hello/.c to an executable program called hello.exe.

In both these cases (IDE or command-line), an executable file such as hello.exe would have been generated. This program is stand alone executable i.e. it can run independently, without the help of any other file, the IDE or the compiler. To see this, we exit from it by pressing Alt-X and type

hello

at the DOS prompt. This will execute the program, and the message will be printed on the screen.

3.2. Running C Programs in UNIX : Every UNIX system has a C compiler called cc. On some UNIX systems, a C compiler called gcc exists. Both are command-line compilers. Their purpose is to convert C program into executable code. The program has to be typed into a file separately, using an editor such as vi, emacs, ed etc. Thereafter, the command

cc hello.c

will create an executable file called a. out by default. This file can be executed by typing a. out at the prompt.

3.3. Remark : The execution of a program actually takes place in two stages i.e. compiling and linking. After the program has been translated into the object code, it is linked with the other programs and functions that are required by the program. In addition to combining object files, the linker also links in functions from runtime library if necessary. Under UNIX, the linking is automatically done when cc command is issued. In DOS also, linking is generally automatic.

3.4. The Runtime Library : One of the reasons for C being such a small language, is that it defers many operations to a runtime library. The runtime library is a collection of object files. Each file contains the machine instructions for a function that performs one of a wide variety of services. The functions are divided into groups, such as I/0, memory management, mathematical operations, string manipulation etc. For each group there is a source file, called the header file, that contains information which we need to use these functions. By conventions, the names for header files end with a. h extension. For example, the standard group of I/O functions has an associated header file called stdio.h.

To include the header file in a program, we should insert the following statement in our source file

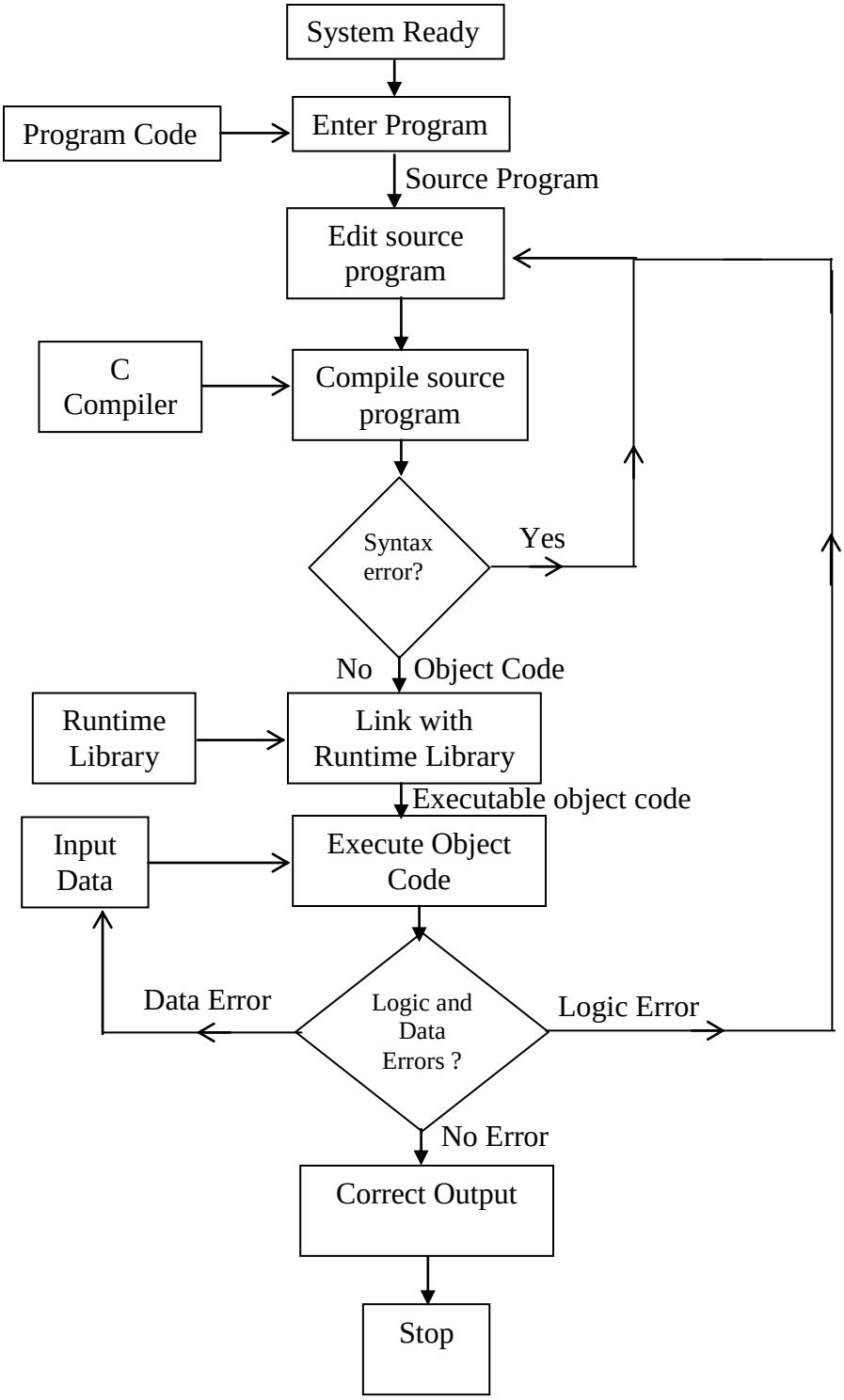
include < filename >

e.g.

include < stdio.h >

Usually, this would be one of the first lines in our source file. We shall discuss the # include directive and other preprocessor commands in more details later on.

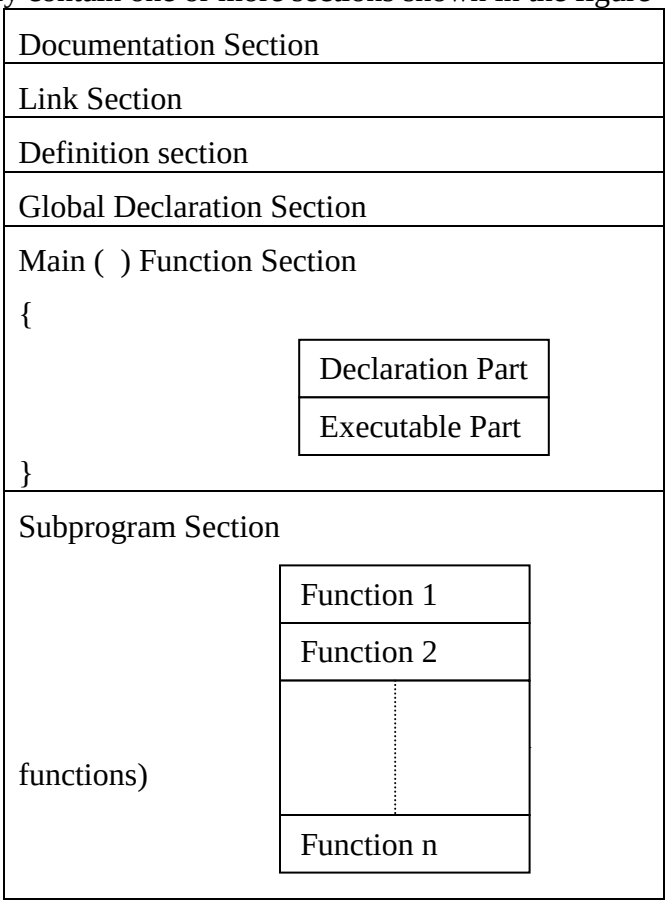
The complete process of compiling and running a C program is as shown the following figure.



(Process of compiling and running a C program)

3.5. Basic Structure of a C Program : C programs can be viewed as a group of building block called functions. A function is a subroutine that may include one or more statements designed to perform a specific task. To write

a C program, we first create functions and then put them together. A C program may contain one or more sections shown in the figure



The documentation section consists of a set of comment lines giving the name of the program, the author and other details which the programmer would like to use later. The link section provides instructions to the compiler to link functions from the system library. The definition section defines all symbolic constants. There are some variables that are used in more than one function. Such variables are called **global** variables and are declared in the global declaration section that is outside of all the functions.

Every C program must have one main ( ) function section. This section contains two parts, declaration part and executable part. The declaration part declares all the variables used in the executable part. There is at least one statement in the executable part. These two parts must appear between the opening and the closing braces. The program execution begins at the opening brace and ends at the closing brace. The closing brace of the main function section is the logical end of the program. All statements in the declaration and executable parts end with a semicolon (;)

The subprogram section contains all the user-defined functions that are called in the main function. User defined functions are generally placed immediately after the main function, although they may appear in any order. All sections, except the main function section may be absent when they are not required.

Unlike other programming languages such as COBOL, FORTRAN etc, C is a free form language i.e. the C compiler does not care, where on the line we begin typing. Still we should select one style and use it with total consistency to make the program easily readable. C program statements are written in lower case letters. Upper case letters are used only for symbolic constants. Braces group the program statements together and mark the beginning and the end of functions. A proper indentation of braces and statements would make a program easier to read and debug. The generous use of comments inside a program increases its readability and helps to understand the program logic. This is very important for testing and debugging the program. The comment has the form

/* comment */ .

3.6. Important Features of a C Program : The following points should be kept in mind when writing a C program.

(i) every C program requires a main ( ) function. Use of more than one main ( ) is illegal. The place main is where the program execution begins.

(ii) The execution of a function begins at the opening brace of the function and ends at the corresponding closing brace.

(iii) C programs are written in lower case letters. However, uppercase letters are used for symbolic names and output strings.

(iv) All the words in a program line must be separated from each other by at least one space, or a tab, or a punctuation mark.

(v) Every program statement must end with a semicolon.

(vi) All variables must be declared for their types before they are used in the program.

(vii) We must make sure to include header files using # include directive when the program refers to special names and functions that it does not define.

(viii) Compiler directives such as define and include are special instructions to the compiler to help it compile a program. They do not end with a semicolon.

(ix) The sign # of compiler directives must appear in the first column of the line.

(x) When braces are used to group statements, make sure that the opening brace has a corresponding closing brace.

(xi) C is a free-form language and therefore a proper form of indentation of various sections would improve legibility of the program.

(xii) Comments should be inserted very frequently. Use of appropriate comments in proper places increases readability and understandability of the program and helps users in testing and debugging. In commenting process, the symbols /* and */ should be matched appropriately. The matter between these symbols is not executed i.e. comments are non-executable statements.

3.7. C Character Set : The characters that can be used to form words numbers and expressions depend upon the computer on which the program is run. However, a subset of characters is available that can be used on most personal, micro, mini and mainframe computers C uses the uppercase letters A to Z, the lowercase letters a to z, the digits 0 to 9, and certain special characters listed below:

Character	Name
,	comma
.	period
;	semicolon
:	colon
?	question mark
‘	apostrophe
”	quotation mark
|	vertical bar
/	slash
\	backslash
~	tilde
_	underscore
$	dollar sign
#	pound sign or number sign or Hash
%	percent sign
&	ampersand
^	caret
*	asterisk
–	minus sign
+	plus sign
=	assignment sign
<	opening angle bracket or less than sign
>	closing angle bracket or greater than sign
(	left parenthesis
)	right parenthesis
[	left bracket
]	right bracket
{	left brace
}	right brace

Some combinations of these characters are also used. C also uses the following white spaces : blank space, horizontal tab, carriage return, new line, form feed.

3.8. Remark : Many non-English Keyboards do not support all the characters mentioned above. ANSI C introduces the concept of trigraph sequence to provide a way to enter certain characters that are not available on some keyboards. Each trigraph sequence consists of three characters i.e. two question marks followed by another character as shown below :

3.9. ANSI C Trigraph Sequences

Trigraph Sequence	Equivalent Character
?? =	# (number sign)
??(	[(left bracket)
??)	] (right bracket)
??<	{ (left brace)
??>	} (right brace)
??!	| (vertical bar)
??/	\ (back slash)
??'	^ (caret)
??-	~ (tilde)

4. Functions

The most important concept underlying high-level language is the notion of functions. In other languages, they may be called subroutines or procedures, the idea is the same. A C function is a collection of C language operations. Programs are developed with layers of functions. We can think of function names as abbreviations for long, possibly complicated sets of commands. We need only to define a function once, but we can invoke (call) it any number of times. This means that any set of operations that occurs more than once is a candidate for becoming a function.

For example, let us consider a simple function that calculates the square of a number

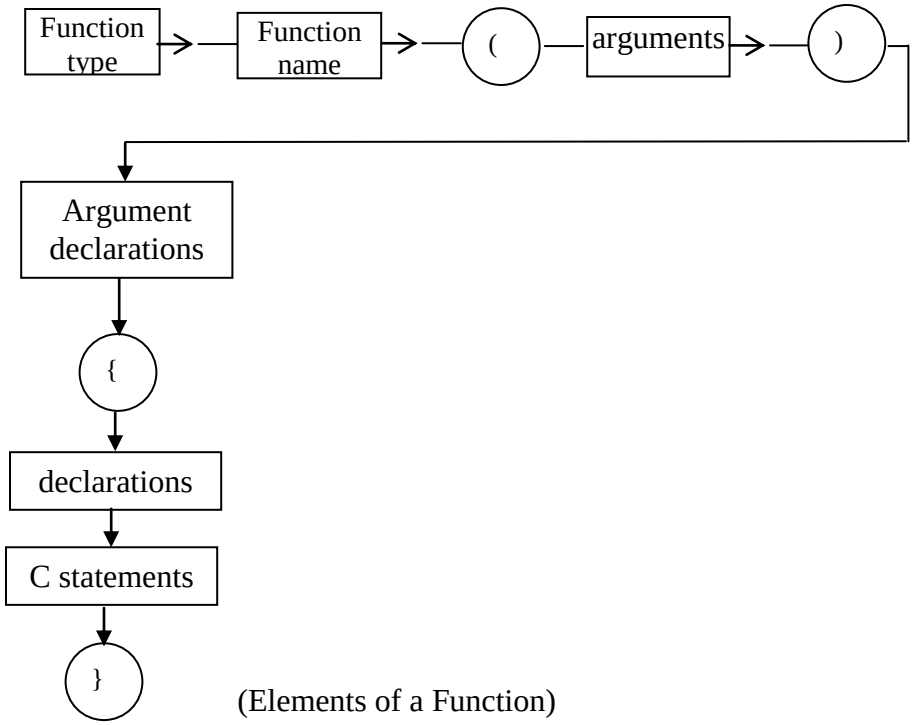
```
int   square (num)
int   num ;
{
      int   answer;
      answer = num * num;
      return answer;
}
```

A function is like a specialized machine that accepts data as input, processes it in a definite manner, and hands back the results.

For example, the square ( ) function takes a number as input and returns the square of the number as the result. Whenever we want to know the square of a number, we "call" the square function. The key to using functions successfully is to make them perform small pieces of a larger program. Ideally, however, each piece should be general enough so that it can be used in other programs as well.

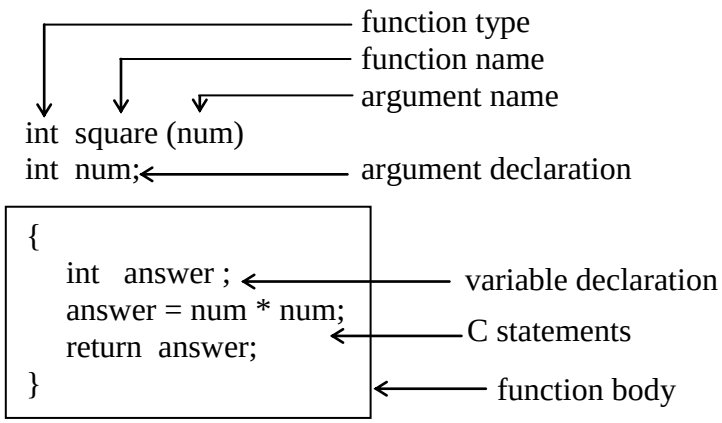
There are many runtime functions present in the runtime library to perform elementary tasks. For example, there is an fopen ( ) function that opens a file, an fgetc ( ) function that reads a character from a file, a scanf ( ) function that reads data entered from the keyboard, a printf ( ) function that prints text and fclose ( ) function that closes a file. The runtime library contains a powerful set of functions, so we should always check it before writing our own functions. One point worth stressing is that functions should be small, yet general. For example, the fopen ( ) function is written so that we can pass it any file name and it will open the corresponding file. In fact, fopen ( ) is even more general, allowing us to specify whether the file contains text or numeric data, and whether it is to be opened for read or write access. This is a good illustration of the principle that the best functions perform small autonomous tasks, but are written so that the tasks can be easily modified by changing the input.

4.1. Anatomy of a C Function : Functions are the building blocks of all C programs. The general layout of a function is shown in the figure in which some of the elements are optional. The required parts are the function name, the parentheses following the function name, and the left and right braces which denote the beginning and the end of the function body. The other elements are optional.



(Elements of a Function)

For example, let us consider the square ( ) function which we introduced earlier. The following figure identifies all of the function's components



(Anatomy of the square ( ) function)

Let us describe each line in detail. The first line has three parts. The first word, int, is a **reserved keyword** that stands for "integer". It signifies that the function is going to return an integer value. There are about thirty two keywords in C, each of which has a language-defined meaning. Keywords are always written in lowercase letters and are reserved by the C language, which means that we can not use them as names for variables. The second word, square, is the name of the function itself. This is what we use to call the function. We could have named the function anything, but it is best to use names that remind us of what the function actually does. The parentheses following the name of the function indicate that square is, infact, a function and not some other type of variable. num is the name of the **argument.** Arguments represent data that are passed from the calling function to the function being called. On the calling side, they are known as **actual arguments,** on called side, they are referred to as **formal arguments.** As with naming functions, we could give the argument any name we want, but num seems sufficiently descriptive.

Functions can take any number of arguments. For example, a function that computes x to the power y would take two arguments, separated by comma, i.e. int power (x, y). The second line of the square ( ) function is an argument declaration. The keyword, int, again signifies that the input is going to be an integer. The semicolon ending the line is a punctuation mark indicating the end of the statement or declaration.

The function body contains all the executable statement. This is where calculations are actually performed. The function body must begin with a left brace and end with a right brace. The line following the left brace is a declaration of the integer variable called answer. Program variables are names for data objects whose values can be used or changed. The declaration of answer follows the same format as the declaration of num, but it lies within the function body. This indicates that it is not an argument to the function.

Rather, it is a variable that the function is going to use to hold a value temporarily. Once the function finishes, answer becomes in accessible. All variables declared within a function body must be declared immediately after a left brace.

The next line is the first **executable statement** i.e. the first statement that actually performs a computation. It is called an **assignment statement** because it assigns the value on the right-hand side of the equal sign to the variable on the left-hand side. We read it as "Assign the value of num times num to answer". The symbol ∗ is an **operator** that represents multiplication and '=' is an operator that represents **assignment.**

The next statement is a return statement, which causes the function to return to its caller. The return statement may optionally return a value from the function, in this case answer.

Before proceeding further, we need to take a closer look of these function components – particularly variables, variable names, constants, expressions and assignment statements.

5. Constants and Variables

Two important programming tokens are constants and variables. A program must be able to store its input somewhere so that it can work with the data and process it into a form that is meaningful to the user. All the data processed by a program is stored either as a variable or a constant. The primary storage area a program uses is called a variable. Variables are memory locations in the computer's memory that holds data. Thus a variable achieves its variableness by representing a location or address in computer memory. Variables hold different kind of data, and the same variable might hold different values during the execution of a program. Contents of a variable vary and hence the name is so. Each variable in C has some characteristics associated with it, called its **storage class.**

Constants never change. Constants are numbers, characters or words/string that we put in a program. It is also possible to store a constant in a variable which we shall explain later on.

5.1. Keywords and Identifiers : Every C word is classified as either a keyword or an identifier. All keywords have fixed meanings and these meanings can not be changed. Keywords serve as basic building blocks for program statements. All keywords must be written in lowercase. List of C keywords is as follows :

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned

continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Some compilers may recognize some other keywords also.

Identifier generally refer to variables and function names. These are user defined names. Names may contain letters, numbers and the underscore character _, but must start with a letter or underscore, Names beginning with an underscore, however, are generally reserved for internal system variables.

The C language is **case sensitive** which means that it differentiate between lowercase and uppercase letters. So the names Total, total, TOTAL are all different. A name can not be the same as one of the reserved keywords. Also we should avoid using names that are used by the runtime library. There is no C-defined limit to the length of a name, although each compiler sets its own limit. The ANSI standard requires compilers to support names of at least 31 characters. Some older compilers impose an 8-character limit.

5.2. Expressions : An expression is a formula for computing a value. It is a combination of operands and operators arranged as per the syntax of the language. The operands may contain function references, variables, constants etc. The expressions in C may be divided into the following four types.

1) Arithmetic expressions
2) Relational expressions
3) Logical expressions
4) Conditional expressions

Each type of expression takes certain types of operands and uses a specific set of operators. Evaluation of every expression produces a value of specific type. Expressions are not statements, but may be components of statements. We shall discuss them in detail in unit-II.

5.3. Assignment Statements : A statement causes the computer to carry out some action. There are several different assignment operators in C, all of which are used to form assignment expressions which assign the value of an expression to an identifier. Assignment expressions are often referred to as assignment statements, since they are usually written as complete statements. However, assignment expressions can also be written as expressions that are included within other statements. The most commonly used assignment operator is '='. Assignment statements that make use of this operator are written in the form

Identifier = expression

Where 'identifier' generally represents a variable, and 'expression' represents a constant, a variable or a more complex expression.
e.g. disc = b ∗ b – 4.0 ∗ a ∗c

5.4. The main ( ) function : Having written and compiled the function square ( ), we still can not execute it. Every executable program must contain a special function called main ( ), which is where the program execution begins. The main function can call other functions. For example, to invoke square ( ) function, we write

```
main ( )
{
    extern  int   square ( );
    int   solution ;
    solution = square (7);
    exit (0);
}
```

This assigns the square of 7 to the variable named 'solution'. The rules governing a main ( ) function are the same as the rules for other functions. Note, however that we do not identify the function's data type and we do not declare any arguments. This is a convention that we adopt for now. Actually, main ( ) does return a value and it takes two arguments. We shall discuss this concept later on. The exit ( ) function is a runtime library routine that causes a program to end, returning control to the operating system. If the argument to exit ( ) is zero, it means that the program is ending normally without errors. Nonzero arguments indicate abnormal termination of the program. Calling exit ( ) from a main ( ) function is exactly the same as executing a return statement, i.e.

```
    exit (0);
```

is the same as

```
    return 0;
```

we should include either exit ( ) or return in every main ( ) function. Under ANSI features, we need to include the stdlib.h header file wherever we call the exit ( ) function.

We have declared two names in main ( ). The first is the function square ( ), which we are going to call. The special keyword 'extern' indicates that square ( ) is defined elsewhere, possibly in another source file. The other variable 'solution' is an integer that we use to store the value returned by square ( ). The next statement actually invokes the square ( ) function. It is an assignment statement, with the right-hand side of the statement being the function invocation. The argument 7 is placed in parenthesis to indicate that it is the value being passed as an actual argument to square ( ). We recall that in square ( ), the name of this passed argument is num. The square ( )

function then computes the square of num and returns it. The return value 49 gets assigned to solution in the main ( ) function.

6. Elementary I/O Functions

We now have a working program, but it is not practically useful since we have no way to see the answer. Further, if we want to find the square of other numbers, we would have to edit the source file, change the argument to square ( ), and then recompile, relink, and reexecute the program. It would be better if we could dynamically specify which number we want to square while using the above program. For this purpose, we discuss standard I/0 functions.

In microcomputers (Pcs), the common input device is the keyboard and the output device is the computer screen, called the monitor. This subsystem comprising keyboard and monitor is referred to as a **console**. To make C language as compact as possible, Dennis Ritchie has omitted the I/0 statements and has put them in extra overhead. At present, we describe only the console I/0 functions that perform the input from a standard input device and output to a standard output device.

There are many library functions available for console I/0. These functions are divided into two categories : unformatted and formatted. The only difference between these functions is that the formatted functions permit the I/0 to be formatted as per requirements :

6.1. Unformatted I/O Function : In this category, we have functions for performing I/O of one character at a time, known as character I/O functions and functions that perform I/O of one string at a time, known as string I/O functions.

For character input, the functions available are getchar ( ), getche ( ) and getch ( ). All these three functions return a character that has been recently, typed. In the first, an enter key is required, in second and third, no enter key is required. Further, in the third, the typed character is not seen on the computer screen. This function is used to hide the input. This input is usually the password used for system security. For character output, the function available is putchar ( ).

For string input, the function is gets ( ) and for string output the function puts ( ) is used. The length of the string is limited by the declaration of the string variable. Both these functions require the enter key.

6.2. Formatted I/O Functions : In this category, we have scanf ( ) function for input and printf ( ) function for output. The working of these functions is described below.

6.3. scanf ( ) Function : This function allows us to input data in a fixed format. The general form of this function is scanf ("format string", list of addresses of variables); The format string (or control string) has the format specifiers that begin with the conversion character '%' and are separated by space or comma. The address of a variable (i.e. pointer to variable) is the integer value data arguments preceded by the ampersand (address of) operator

'&'. On execution, the input data must be supplied strictly according to the specified format string. Commonly used scanf format specifiers (codes) are as follows :

Code	Meaning
%c	read a single character
%d	read a decimal integer
%e	read a floating point value (exponential notation)
%f	read a floating point value (exponential notation)

Code	Meaning
%g	read a floating point value (exponential notation)
%h	read a short integer
%i	read a decimal hexadecimal or octal integer
%o	read an octal integer
%s	read a string
%u	read an unsigned decimal integer
%x	read a hexadecimal integer
%[..]	read a string of word (s)

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

h-for short integer

l-for long integers or double

L-for long double

Sometimes an integer number w, known as field width of the number to be read, is used as prefix. Thus we use % wd for input of integer number e.g. to input 561238, 375 we use %6d, %3d respectively. The following general points should be kept in mind while using a scanf ( ) function.

(i) All function arguments, except the control string, must be pointers to variables.

(ii) Format specifications contained in the control string should match the arguments in order.

(iii) Input data items must be separated by spaces and must match the variables receiving the input in the same order.

(iv) The reading will be terminated when scanf ( ) encounters a mismatch of data or a character that is not valid for the value being read.

(v) When searching for a value, scanf ( ) ignores line boundaries and simply looks for the next appropriate character.

(vi) Any unread data items in a line will be considered as a part of the data input line to the next scanf( ) call.

(vii) When the field width specifier w is used, it should be large enough to contain the input data size.

If large value of w is used for small data, the input value is right justified in the field. If left justified data is needed, we use hyphen '-' after character %, before w.

6.4. printf ( ) Function : This function allows to output the data in a fixed format. Its general form is

printf (″format string″, list of variables);

where the format string contains the format specifiers, the sequences that begin with character '\' (back slash), known as escape sequences, and the text to be output alongwith the output data. All the format specifiers used with scanf ( ) function are valid for printf ( ) function. Thus, printf ( ) is the mirror image of scanf ( ).

Commonly used escape sequences are listed below :

Escape sequence	Name	Effect
\a	alert (or beep)	Produces an audible or visible alert signal
\b	backspace	Moves the cursor one space left
\f	form feed	Moves the cursor to the next logical page.
\n	newline	Prints a newline i.e. the subsequent output starts from newline
\r	carriage return	Prints a carriage return
\t	horizontal tab	Prints a horizontal tab i.e. moves over to the next eight-space-wide field.
\v	vertical tab	Prints a vertical tab
\'	single quote	insert character' in the output
\"	double quote	insert character " in the output
\\	backslash	insert character \ in the output
\?	Question mark	insert character ? in the output

In addition to the escape sequences listed above, C also supports escape character sequences of the form

\ octal-number

and \ hex-number

which translates into the character represented by the octal or hexadecimal number. Such syntax is most frequently used to represent the null character as ′\0′. This is exactly equivalent to the numeric constant zero (0).
Since we require the output in more specified format, so we use the format specification of the form

% w.p type-specifier

where w is an integer number that specifies the total number of columns for the output value and p is another integer number that specifies the number of digits to the right of the decimal point (of a real number) or the number of characters to be printed from a string. Both w and p are optional.

Printf ( ) never supplies a newline automatically and therefore multiple printf ( ) statements can be used to build one line of output. A newline is introduced by the newline character ′\n′.

6.5. Continuation Character : To span a quoted string over more than one line, we must use the continuation character, which is a backslash. Prior to the ANSI standard, the continuation character could only be used to continue character strings. The standard extended this notion so that we can now stretch variable names over multiple lines.
For example, the following program uses the continuation character to print a long string.

```
main ( )
{
    printf (′′This string is too long to fit on one line, so \
        we need to use the continuation\ character.′′);
}
```

6.6 The Preprocessor : We may consider the C preprocessor as a separate program that runs before the actual compiler. It is automatically executed when we compile a program, so we need not explicitly invoke it. The preprocessor has instructions for the compiler and these instructions are known as preprocessor commands or directives. Each directive begins with the character ‘#’ (pound sign or Hash). Although, these directives are not part of the C language, but they expand the scope of the C programming environment. Unlike C statements, a preprocessor directive ends with a newline, not a semicolon.

We shall discuss the preprocessor in detail in unit-V. Here, we take a close look at the # include directive, already mentioned in connection with header file, and a new preprocessor command called # define directive.

(i) # include directive : The preprocessor # include directive instructs the compiler to read source text from another file as well as the file it is currently compiling. This enables us to insert the contents of one file into another file before compilation begins, although the original file is not actually altered. The # include command has two forms :

```
# include   <filename>
```

and `# include  ″filename″`

If the filename is surrounded by angle brackets, the preprocessor looks in a special place designated by the operating system. This is where all system include files, such as the header files for the runtime library, are kept. If the filename is surrounded by double quotes, the preprocessor looks in the directory containing the source file. If it can not find the include file there, it searches for the file as if it had been enclosed in angle brackets. By convention, the names of include files usually end with .h extension.

When we compile the program, the preprocessor replaces the # include directive with the contents of the specified file

(ii) # define directive : Just as it is possible to associate a name with a memory location by declaring a variable, it is also possible to associate a name with a constant. We do this by using a preprocessor directive called # define. To avoid confusion between constants and variables, it is common practice to use all uppercase letters for constant names and lowercase letters for variable names. The general form of # define directive is # define identifier string where identifier is the name used for constant and string is a character string that is substituted for the identifier each time it is encountered in the source file. The identifier is called a **macroname** and the replacement process is called **macrosubstitution.**

Naming constants has two important benefits. First, it enables us to give descriptive name to a non-descriptive number. For example, let us consider

```
# define   MAX_PAGE_WIDTH     70
```

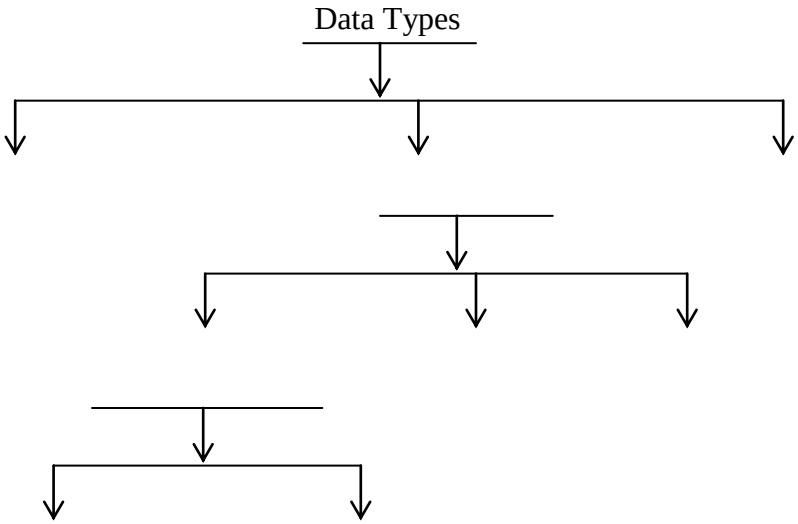
In our program, we can use MAX_PAGE_WIDTH, which means something, instead of ‘70’ which does not tell us much. Creative naming of constants can make a program much easier to read. The other advantage of constant names is that they make a program easier to change. For example, the maximum page width parameter might appear many times in a large text formatting program. Suppose we want to change the maximum width from 70 to 80. If, instead of using a constant name, we used the constant 70, we will need to change 70 to 80 wherever it appears and hope that we are changing the right 70’s. If we use a constant name, we need only change the definition as follows

```
# define    MAX_PAGE_WIDTH     80
```

and recompile.

7. Scalar Data Types

The most important features of modern programming languages is their ability to divide data into different types. The C language offers a small but useful set of data types. There are eight different types of integers and two types of floating-point objects (three with the ANSI standard). In addition, integer constants can be written in decimal, octal, or hexadecimal notation. These types, i.e. integers and floating points, are called **arithmetic types** or sometimes primary data types. Together with pointers and enumerated types, they are known as **scalar types** because all of the values lie along a linear scale. That is, any scalar value is either less than, equal to, or greater than any other scalar value. In addition to scalar types, there are **aggregate types**, which are built by combining one or more scalar types. Aggregate types, which include arrays, structures, and unions, are useful for organizing locally related variables into physically adjacent groups. There is one more type called **void** which is neither scalar nor aggregate. Here, we shall describe scalar variables and constants and void types. We shall discuss the aggregate types in unit-III and IV. The classification of data types is shown in the following figure.



7.1. Declarations : Every variable must be declared before it is used. A declaration provides the compiler with information about how many bytes should be allocated and how those bytes should be interpreted. Observe that zeros and ones are called bits and computer memories are arranged into 8-bit multiples called bytes (ANSI standard).

A declaration consists of data type, followed by one or more variable names, ending with a semicolon. We have the following nine reserved words for scalar data types.

char	double	short	signed
int			
float	enum	long	unsigned

The first five i.e. char, int, float, double, enum are basic types. The remaining four i.e. long short, signed, unsigned are qualifiers that modify a basic type in some way. We can treat basic types as nouns and the qualifiers as adjectives. All the declarations in a block must appear before any executable statements. It is better to group declarations of the same type together for easy reference. For declaring single-character names we have a convention borrowed from FORTAN . The names i, j, k, m, n are generally used for integer counters, temporary variables x, y, z are used for floating point temporary variables and C is used for temporary character variables. The use of single-character names *l* and o is avoided as they may be confused with digit 1 and 0.

Just as we can declare the data type of a variable, we can also declare the type of value returned by a function. Unlike other variables, functions have a default return type (int) if we do not explicitly give them a return type. To make program more readable, it is better to declare the return type. For the functions which return no value, we use void type.

7.2. Different Types of Integers : On all machines, the reserved word int is treated as an integer in the sense that it can not hold fractional values, but it is least descriptive as it has different sizes on different machines. Different compilers allocate different bytes for an int, ranging from one to four bytes. As per ANSI standard, we assume that an int is four bytes.

If we do not care how many bytes are allocated, we can use int. If the size matters, we use the size qualifiers i.e. short or long. On most machines, a short int is two bytes and a long int is four bytes For example, to declare m as a short int and n as a long int, we write

```
short int m;
long int n;
```

The compiler would allocate at least two bytes for m and at least four for n. The compiler is smart enough to infer int even if we leave it out i.e. the above statements can be written as

```
short m;
long n;
```

In special cases, such as counting of things, we require non-negative numbers. To declare an integer variable non-negative only, we use the unsigned qualifier. Note again that the statements

```
unsigned int p;
```

and

```
unsigned p;
```

are identical.

In most cases, variables are signed by default so the keyword signed is superfluous, but ANSI standard recognises the keyword signed and it may be used for character data type.

The number of bits used to represent an integer type determine the range of values that can be stored in that type. In case of an ordinary int (or a short int or a long int), the leftmost bit (the most significant bit) is reserved for the sign. With an unsigned int all of the bits are used to represent the numeric value. Thus, an unsigned int can be approximately twice as large as an ordinary int.

Most programming languages make a distinction between numeric and character data. In reality, characters are stored in the computer as numbers. Every character has a unique numeric code. According to ASCII (American Standard Code for Information Interchange), all character values lie within the range 0 to 255, which means that a character can be represented by a single byte (However, to allow for programming in language other than English, ANSI supports multibyte character).

In C, there is a data type called char, but it is really just a one byte integer value that can be used to hold either characters or numbers. For example, after making the declaration

```
char   C;
```

we can make either of the following assignments

```
C = ′A′ ;

C = 65;
```

In both the cases, the decimal value 65 is loaded into the variable C since 65 is the ASCII code for the letter ‘A’. It should be noted that character constants are enclosed in single quotes. The quotes tell the compiler to get the numeric code value of the character. Since char are treated as small integers, we can perform arithmetic operations on them. For example in the expressions

```
int   j;

j = ′A′ + ′B′;
```

j gets the value 131, since ‘A’ and ‘B’ have values 65 and 66.

In the ASCII character set, character codes are ordered alphabetically. The upper-case letters have codes from 65 to 90 and lowercase letters from 97 to 122. This guides that we can have a function that changes a character from uppercase to lowercase, as

```
char   to_lower (ch)
char   ch;
{
  return   ch + 32;
}
```

Equivalently, C runtime library contains two functions called to upper ( ) and to lower ( ) that change a character’s case. It is better to use these functions

than using the above statements. The following table shows the sizes and ranges of integer types.

Type	size in bytes	Value Range
int	4	-2^{31} to $2^{31}-1$
short int	2	-2^{15} to $2^{15}-1$
long int	4	-2^{31} to $2^{31}-1$
unsigned short int	2	0 to $2^{16}-1$
unsigned long int	4	0 to $2^{32}-1$
signed char	1	-2^{7} to $2^{7}-1$
unsigned char	1	0 to $2^{8}-1$

7.3. Different kinds of Integer Constants : The constants representing decimal numbers are called **decimal constants**. We can also write **octal** and **hexadecimal** constants. An octal constant is written by preceding the octal value with the digit zero. A hexadecimal constant is written by proceeding the hexadecimal value with a zero and an x or X. The following table shows some decimal constants and their octal and hexadecimal equivalents

Decimal	Octal	Hexadecimal
5	005	0x5
8	010	0x8
15	017	0xf
–16	–020	–0x10
22	026	0x16
87	0127	0x57
–187	–0273	–0xBB
255	0377	0xff

We remind that scanf( ) and printf ( ) functions have format specifiers 0 and x for octal and hexadecimal numbers respectively. The following program reads a hexadecimal number (with or without ox prefix) from the terminal and prints its decimal and octal equivalents.

```
/* This program prints the decimal and octal equivalents of a hexadecimal constant */

# include < stdio.h>

main ( )
```

```
{
        int   num ;
        printf (′′Enter a hexadecimal constant : ′′) ;
        scanf (′′%x′′, & num);
        printf (′′The decimal equivalent of % x is : %d\n′′, num, num);
        printf (′′The octal quivalent of % x is : %O\n′′, num, num);
        exit (0) ;
}
```

C supports some special backslash character constants that are used in output functions. For example, the symbol '\n' stands for newline character. We have already discussed complete list of such constants (escape character sequences).

7.4. Floating-Point Types : Integers are inadequate for representing very large numbers and fractions. For this, we need floating-point types. There are two ways to write floating-point constants. In the first, we place a decimal point in a number. e.g. 3568.0, 86., 0.23, .5, 0.000000003 etc. To declare a variable capable of holding such values, we use the 'float' or 'double' keyword. The word double stands for double-precision, because on many machines it is capable of representing about twice as much **precision** as a float. The precision refers to the number of decimal places that can be represented. On many machines, a double also takes up twice as much memory. Generally, a float requires four bytes and a double requires eight bytes.

The ANSI supports an additional floating-point type called 'long double'. This is a new type which provides even greater range and precision than double.
The following function takes a double value as an argument and represents a temperature in Fahrenheit and converts it to Celsius.

```
/* This converts a float value from Fahrenheit to Celsius */
double   fahrenheit_to_celsius (temp_fahrenheit)
double  temp_fahrenheit;
{
        double   temp_celsius;
        temp_celsius = (temp_fahrenheit – 32.0)/1.8;
        return   temp_celsius;
}
```

The following function computes the area of a circle with given radius.

```
        /* This calculates the area of a circle whose radius is given */

# define    PI   3.14159
float   area_of_circle (radius)
float   radius;
{
        float   area ;
        area = PI * radius * radius ;
        return   area;
}
```

Note that we have used the constant name PI which is more meaningful than the string of digits 3.14159. The second way of writing floating-point constants is the scientific notation which is useful for lengthy floating-point values. In this notation, a value consists of two parts : a number called the **mantissa** followed by a power of 10 called the **characteristic** or exponent. The letter e or E, which stands for exponent, is used to separate the two parts. For example,

4e3 represents $4*10^3$ i.e. 4000.

$-5.4e-3$ represents $-5.4*10^{-3}$ i.e. -0.0054

Here, it should be noted that the exponent value must be an integer.

7.5. Initialization : A declaration allocates memory for a variable, but it does not necessarily store an initial value at the location (fixed duration variables are an exception which we shall discuss in unit – IV). Because we often want a variable to start with a particular value, the C language provides a special syntax for initializing a value. Essentially, we just include an assignment expression after the variable name in a declaration. For example, the expression

```
char  ch = ′A′;
```

allocates one byte for ch, and also assigns the character ‘A’ to it. Thus the initialization is just a shorthand for combining a declaration statement and an assignment statement. For example, the above expression is exactly the same as

```
char  ch ;
ch = ′A′;
```

When multiple variables are being declared in a single statement, initialization, for example, is done in the following way

```
int   i = 10,   j = 5;
```

Also the righthand side of the assignment symbol can have an expression. For example, the following statements

```
int   i = 10,   j = 5;
int   k = i/j;
```

will initialize the value of k to 2.

7.6. Mixing Types : C language allows us to mix arithmetic types in expressions with few restrictions. To make sense out of an expression with mixed types, C performs conversions automatically. These implicit conversions make the programmer's job easier but put a greater burden on the compiler. This can be dangerous since the compiler may make conversions that are unexpected. For example, the following expression

```
2.0 +1/2
```

does not evaluate 2.5 as we might expect. Instead it gives 2.0. However, if we use 1./2 or 1/2. , then we get 2.5. The implicit conversions are also called quiet conversions or automatic conversions. In general, most implicit conversions are invisible.

When a char or short int appears in an expression, it is converted to an int. Unsigned char and unsigned short are converted to int if the int can represent their value; otherwise they are converted to unsigned int. These conversions are called **integral widening conversions** or **integral promotions**.

In an arithmetic expression, objects are converted to conform to the conversion rules of the operator. Each operator has its own rules for operand type agreement, but most binary operators require both operands to have the same type. If the types differ, the compiler converts one of the operands to agree with the other. To decide which operand to convert, the compiler follows the hierarchy of C scalar data type given below (from lower to higher) int $\rightarrow$ unsigned $\rightarrow$ long int$\rightarrow$ unsigned long int $\rightarrow$ float $\rightarrow$ double $\rightarrow$ long double. Thus, if the operands are of different types, the 'lower' type is automatically converted to the 'higher' type before the operation proceeds.

In certain situations, arguments to functions are converted. We shall discuss this type of conversion later on.

In assignment statements, the value on the R.H.S. of the assignment sign is converted to the data type of the variable on the L.H.S. These are called **assignment conversions.** However, the following changes are introduced during the final assignment.

(a) float to int causes truncation of the fractional part.
(b) Double to float causes rounding of digits.
(c) Long int to int causes dropping of the excess higher order bits.

7.7. Explicit Conversions-Casts : In C, it is also possible to explicitly convert a value to different type. Explicit conversion is called **casting** (or

local conversion) and is performed with a construct called a **cast**. To cast an expression, we enter the target data type enclosed in parentheses directly before the expression. For example,

```
j = (float) 5;
```

converts the integer 5 to a float before assigning it to j. Of course, if j is an integer, the compiler would implicitly convert the value back to an integer before making an assignment. Let us consider another example,

```
int   j = 2,  k = 3;
float   f;
f = k/j;
```

Here, it appears that f gets assigned the value 1.5. However, f is actually assigned the value 1.0. This is because the expression

```
k/j
```

contains only ints, so they are not promoted to floating point type. Thus the value 1.5 is truncated to the integer value 1, since the result of an integer expression is always an integer. Then the value 1 is converted to 1.0 as it is being assigned to a floating-point variable. One way to avoid this problem is to cast either, or both, of the integer variables to float. For example, let us write

```
f = (float) j/k;
```

This explicitly converts j to a float. Then the implicit conversion rules come into play i.e. k is automatically converted to float. The result of an expression containing two floats is a float and so f gets assigned the value 1.5.

7.8. Enumeration Types : In addition to integer, floating-point, and pointer type, the scalar types also include enumeration type. This type enables us to declare variables and the set of named constants that can be legally stored in the variable. An enumeration is a user-defined type with values ranging over a finite set of identifiers called **enumeration constants** (or enumerators). The compiler reports an error if we assign a value other than those in this set. The general form of the enumeration type is enum variable {list} where ‘enum’ is keyword for declaring enumeration type, ‘variable’ is the name of enum variable, and ‘list’ is the list of constant names separated by comma. Note that ‘list’ is enclose in braces. For example, enum colour {red, blue, green, yellow}, defines colour as enumeration variable which can be assigned one of the four constant values : red, blue, green and yellow. Internally, the C compiler treats an enum type (such as colour) as an integer itself. Constant names in an enum declaration receive a default integer value based on their position in the enumeration list. The default values start at zero and go up by

one with each new constant name. For example, in the declaration of colour, the identifiers red, blue, green and yellow represent the integer values 0, 1, 2 and 3 respectively. We can override these default values by specifying other values. If we specify a value, all subsequent default values begin at one more than the last define value. For example, if the definition of colour is enum colour (red = 10, blue, green = –5, yellow); then the statements

```
colour  c;
c = blue;
```

will assign the value 11 to c. If we use

```
c = yellow ;
```

c will get the value –4.

Note that the assigned values need not be in ascending order, but for readability, it is better to write then in ascending order.

7.9. The void Data Types : The void data type also known as **empty** data type is useful in many situations. It has two important purposes. The first is to indicate that a function does not return a value and performs only some printing or house keeping operations. For example, we may see a function definition as

```
void  func (int x, int y)
{
   ……..
}
```

which indicates that the function does not return any useful value. So we can not assign the returned value to a variable. Similarly, the statement

```
int   func (void)
{
  ……..
}
```

indicates that the function does not take any argument. The other purpose of void is to declare a generic pointer which we shall discuss in unit-IV.

7.10. Typedefs : C provides a facility called “type definition” that allows users to define an identifier that would represent an existing data type. The user-defined data type identifier can later be used to declare variables. The general form of the typedef statement is

```
typedef  type  identifier ;
```

where ‘type’ refers to an existing data type and ‘identifier’ refers to the new name given to the data type. It should be noted that the new type is new only in name, but not in data type. Thus typedef can not create a new type, it only tells the compiler to recognize the ‘identifier’ as synonymous of ‘type’. For example in the statements

```
typedef  int  units ;
typedef  float  marks;
```

‘units’ symbolizes int and ‘marks’ symbolizes float. They can be used to declare variables as

```
units  batch 1, batch 2;
marks  group 1 [20] , groups 2[20]
```

Here, batch 1 and batch 2 are declared as int variables, group 1[20] and group 2[20] are declared as 20 element floating point array variables. Conventionally, typedef identifiers may be capitalized so that they are not confused with variable names. Thus in the above examples, we may use UNITS and MARKS.

The main advantage of typedef is that we can create meaningful data type names for increasing the readability of the program. Typedefs are specially useful for abstracting global types that can be used throughout a program. We shall describe this concept in unit-IV.

7.11. Finding the Address of an Object : As mentioned already, every variable has a unique address that identifies its storage location in memory. For some applications, it is useful to access the variable through its address rather than through its name. To obtain the address of a variable, we use the ‘&’ (ampersand) operator. When reading an expression, the ampersand operator is translated as “address of”. For example, suppose j is a long int whose address is 2486. The statement

```
count = & j;
```

stores the address value 2486 to the variable count. We read it as “Assign the address of j to count”. The following program prints the value of the variable called j and the address of j

```
# include  < stdio.h>
main ( )
{
  int   j = 5;
  printf (′′The value of j is : %d\n′′, j);
  printf (′′The address of j is : % p\n′′, &j)’
```

```
    exit (0);
}
```

The output is

```
The value of j is : 5
The address of j is : 2486
```

The particular address listed above (i.e. 2486) is arbitrary. The address may be different for different program and different computers.

It should be noted that printf ( ) requires a special format specifier (% p) to print address value.

Some compilers print an address with %d, %O and % x specifiers

8. Introduction to Pointers

Pointers are regarded as one of the most difficult topics in C. Although they may appear as little confusing for a beginner, they are most powerful tools and are used frequently with aggregate types, such as arrays and structures. In fact, real power of C lies in the proper use of pointers. Here, we consider only the introductory study of pointers and we shall discuss them in detail in unit-III.

Since addresses of variables are not guaranteed to be represented in the same fashion as integers, therefore to store addresses, we need a special type of variable, called a **pointer variable.** So a pointer variable is another variable that holds the address of the given variable to be accessed. Thus pointers provide a way of accessing a variable without referring to variable directly. To declare a pointer variable, we precede the variable name with an asterisk (∗). The declaration has the form

```
type ∗pt_name;
```

Where ‘type’ is the pre-defined or user-defined data type and indicates that the pointer will point to the variables of that specific data type. For example, the statement

```
int ∗ x;
```

declares the variable x as a pointer variable that points to an integer data type. It should be noted that the type ‘int’ refers to the data type of the variable being pointed to by x and not the type of the value of the pointer. Similarly, the statement

```
float ∗ y;
```

declares y as a pointer to a floating-point variable.

8.1. Initializing a Pointer : Once a pointer variable has been declared, it can be made to point to a variable by using an assignment statement as

```
x = & pay;
```

which causes x to point to 'pay' i.e. x none contains the address of pay. This is known as pointer initialization. A pointer should be initialized before use. We should also ensure that pointer variables always point to the corresponding type of data. For example,

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

will give wrong results.

Further, assigning an absolute address to a pointer variable is prohibited. Thus

```
int   * x;
x = 237;
```

is invalid. A pointer variable can be initialized in its declaration itself. For example

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

is valid. It declares x as an integer variable and p as a pointer variable and then initializes p to the address of x. Note carefully that this is an initialization of p, not of *p. Further, the target variable x is declared first. Thus, the statement

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

is not valid.

8.2. Dereferencing a Pointer (Accessing a variable through its pointer): The asterisk, in addition to being used in pointer declarations, is also used to dereference a pointer i.e. to get the value stored at the pointer address. Here, the unary operator * is termed as dereference operator or indirection operator, as it allows to access the value of a variable indirectly. Let us consider the following statements

```
int   pay,  *p,  j;
pay = 5000;
p = & pay;
j = *p;
```

The first line declares pay and j as integer variables and p as a pointer variable pointing to an integer. The second line assigns the value 5000 to pay and the third line assigns the address of pay to the pointer variable p. The fourth line contains the indirection operator ∗. When the operator ∗ is placed before a pointer variable in an expression, on the R.H.S. of the equal sign, the pointer returns the value of the variable of which the pointer value is the address. In this case, ∗p returns the value of the variable pay, because p is the address of pay. The ∗ can be remembered as 'value at address'. Thus the value of j would be 5000. The two statements

```
p = & pay;
j = ∗p ;
```

are equivalent to

```
j = ∗& pay;
```

which in turn is equivalent to

```
j = pay;
```

Hence it should be noted that the character '∗' in the declaration statement tells the compiler that variable is a pointer variable, while accessing the variable (i.e. dereferencing a pointer) it tells the compiler that the value to be accessed is the value stored in storage location whose address (memory location) is held in a pointer variable.

8.3. Advantages of Using Pointers : Some of the advantages of using pointers are as follows :

(i) A pointer enables us to access a variable that is defined outside the function
(ii) Pointers are more efficient in handling the data tables.
(iii) Pointers reduce the length and complexity of a program
(iv) Pointer increase the execution speed
(v) The use of pointer array to character strings results in saving of data storage space in memory.
(vi) The use of pointers enables us to return more than one value from a function.

The most common uses of pointers are :

(i) Accessing array elements
(ii) Passing arguments to functions by reference (i.e. the arguments can be modified)
(iii) Passing arrays and strings to functions
(iv) Creating complex data structures such as linked lists, binary trees, graphs etc
(v) Obtaining memory from the system dynamically.

We shall discuss these uses in the subsequent units.

9. Control Flow (Control Structure)

We have observed that a C program is a set of statements which are normally executed sequentially in the order in which they appear. This happens when no options or no repetitions of certain calculations are necessary. Most programming problems are not so simple. In fact, the great power of programming languages lies in their ability to instruct the computer to perform the same task repeatedly, or to perform a different task if parameter change. In HLL, this goal is achieved by using control flow statements that allow us to alter the sequential flow. Control flow statements fall into two general categories

(i) conditional branching (conditional execution)

(ii) looping (iteration)

The ability to decide whether or not to execute a sequence of statements based on the value of an expression, results in conditional branching. Looping is the ability to perform the same set of operations repeatedly until a special condition is met. Various statements falling under the above said two general categories are as follows :

(i) Conditional Branching
(a) if statement
(b) if-else statement
(c) else if construct
(d) switch statement
(ii) Looping
(a) while statement
(b) do-while statement
(c) for statement

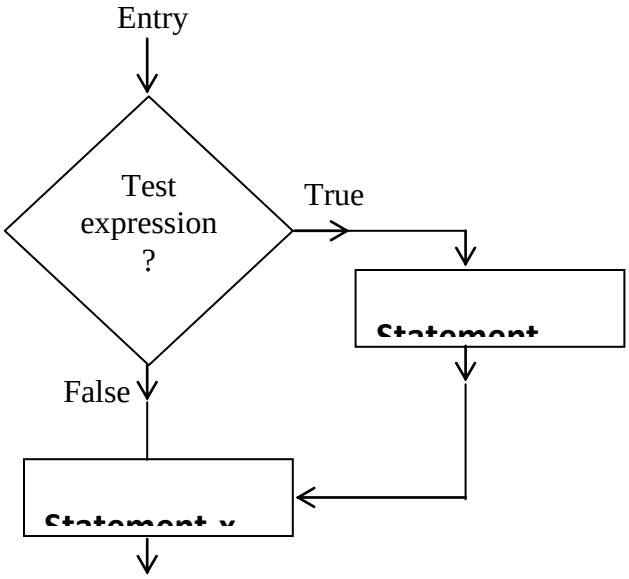
Further, for jumping out of a loop, we have the following three statements
(a) break statement
(b) continue statement
(c) goto statement

Let us first discuss the statements which result in conditional branching.

9.1. The if Statement : The if statement is a powerful decision making statement and is used to control the flow of execution of statements. It is a two-way decision statement and is used in conjunction with an expression. The syntax of the if statement is as follow :

```
if (test expression)
{
  statement-block
}
statement –x
```

Here, the statement-block is a set of statements. Such set of statements enclosed within a pair of braces, is called a **compound statement.** (A function body is, thus, just a compound statements). When the statement-block is not a compound statement i.e. it is a single statement, then the use of braces is optional. The if statement allows the computer to evaluate the test expression first and then depending on whether the value of the expression (relation or condition) is true (non-zero) or false (zero), it transfers the control to a particular statement. If the test expression is true, the statement-block will be executed; otherwise the statement-block will be skipped and the execution will jump to the statement-x. It should be noted that when condition is true, both the statement-block and the statement-x are executed in sequence. The working of the if statement is depicted in the following figure



A common use of the if statement is to test the validity of data. The following program demonstrates the use of if statement.

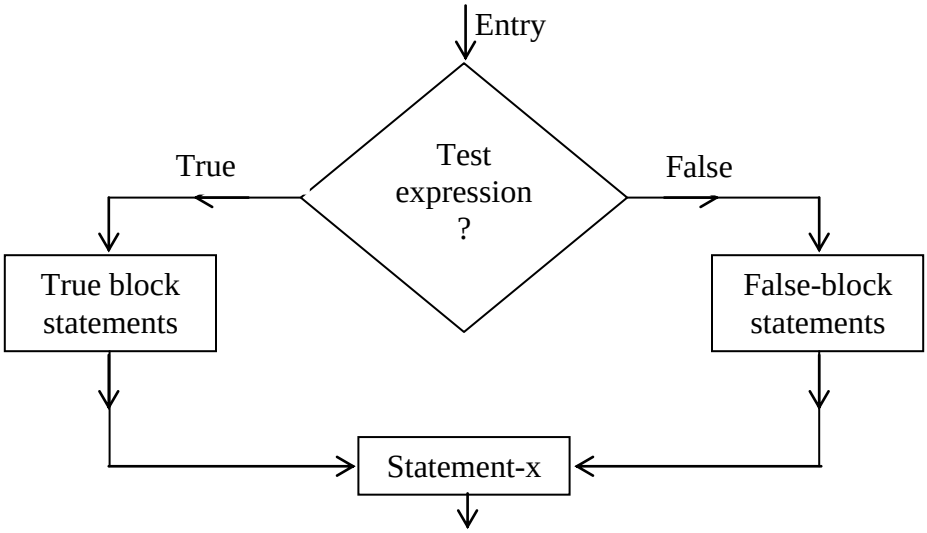
```
/* This program demonstrates the use of if statement */
main ( )
{
        float   number;
        printf (''\n Enter any number : '');
        scanf (''% f '', & number);
        if (number > 0)
        printf (''Number entered is positive \n'');
}
```

If statements can be nested i.e. an if statement can be contained within another if statement. The inner if statement will be executed if the condition of the outer if statement evaluates a true (non-zero) value.

9.2. The if-else Statement : The if-else statement is an extension of the if statement. The if statement executes a single statement-block, when the test expression evaluates a non-zero value. It does nothing when it evaluates a zero value. In an if-else statement, one statement-block is executed if the test condition evaluates to a non-zero value and another statement-block is executed if the test condition evaluates to a zero value. The syntax of if-else statement is

```
if (test expression)
      {
            True-block statements
      }
else
{
      False-block statements
 }
 statement-x
```

Here, either true-block or false-block will be executed and in both cases, the control is transferred subsequently to statement-x.
The working of if-else statement is shown below :

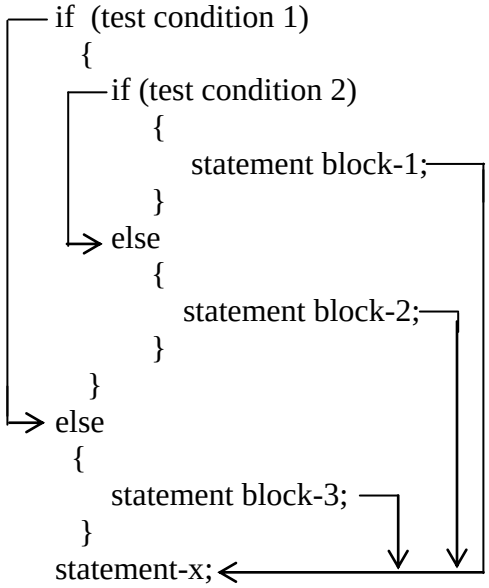


To following programs illustrates the if-else statement.

```
/* This program demonstrates the use of if-else statement */
main ( )
{
      float  number;
```

```
        printf (″\n Enter any number : ″);
            scanf (″%f ″, & number);
        if (number >0)
            printf (″\n Number entered is positive\n″);
        else
            printf (″\n Number entered is negative \n″);
}
```

When a series of decisions are involved, we can use more than one if-else statement in nested form as follows :



If the condition 1 is false, the statement block-3 will be executed otherwise it continue to perform the second test. If the condition 2 is true, the statement block-1 will be evaluated, otherwise statement block-2 will be evaluated and then the control is transferred to the statement-x.

The indentations after if and else are included for increasing readability, not for functionality.

9.3. The else if Construct (Ladder) : When we have a series of if-else statements in nested form, the indentation gets deeper and the program becomes difficult to read and write. For such situations, we have another way of putting ifs together for multipath decisions. This may is called else if construct (or ladder) where multipath decision is a chain of ifs in which the statement associated with each else is an if. Its general form is as follows :

```
if (condition 1)
{
    statement block-1
```

```
 }
else if (condition 2)
{
     statement block-2
}
else if (condition 3)
{
     statement block – 3
}
…….
…….
else if (condition—n)
{
      statement block-n
}
else
{
      statement block-s
}
statement-x;
```

The conditions are evaluated in order. If any condition is true, then the statement block associated with it is executed and then the control is transferred to the statement-x, skipping the rest of the ladder. When all the n conditions become false, then the last else part handles the 'none of the above' or default case, resulting in execution of statement block-s, and than the statement-x The following program demonstrates the use of else-if construct, where the operator '= =' stands for 'equal to'.

```
/* This program illustrates the use of else-if construct */
main ( )
{
        int   day;
        printf (″\n Enter day of week as number : ″);
        scanf (″%d″& day);
        if (day = = 0)
```

```
                printf (′′\n Today is Sunday \n′′);
        else if (day = = 1)
                printf (′′\n Today is Monday \n′′);
        else if (day = = 2)
                printf (′′\n Today is Tuesday \n′′);
        else if (day = = 3)
                printf (′′\n Today is Wednesday \n′′);
        else if (day = = 4)
                printf (′′\n Today is Thursday \n′′);
        else if (day = = 5)
                printf (′′\n Today is Friday \n′′);
        else if (day = = 6)
                printf (′′\n Today is Saturday \n′′);
        else
                printf(′′\n Wrong input \n′′);
}
```

9.4. The Switch Statement : The switch statement allows us to specify an unlimited number of execution paths based on the value of a single expression. It provides a better alternative to the else if construct. It has more flexibility and a clearer format than the else if construct. It is particularly useful when a variable is to be compared with different constants, and in case it is equal to one of them, a set of statements is executed.

However, the switch statement relies heavily on another statement, called the **break statement.** The break statement transfers the control out of the block where it is used. The syntax of the break statement is

```
break;
```

This statement is also used in conjunction with a for, a while, and a do-while statement which we shall discuss later on. The syntax of the switch statement is

```
switch (expression)
 {
  case value 1 :
                statement block-1
                break;
```

```
    case value 2;
                statement block-2
                break;
                ……
                ……
    case value n:
                statement block-n
                break;
    default :
                default-block
                break;
}
statement-x;
```

The ‘expression’ is an integer expression or characters. Value 1, value 2,……. are constants or constant expressions (evaluable to an integer constant) and are known as **case labels.** During the execution of the switch statement, the value of the ‘expression’ is successively compared against the case values value 1, value 2,…… If a case is found whose value matches with the value of the ‘expression’, the control is transferred to that appropriate case, the statements of that block are executed, and then the break statement transfers the control out of the switch statement i.e. to statement-x. If the value of the ‘expression’ does not match any of the case values, control goes to the ‘default’ keyword, which is usually at the end of the switch statement. The last break after the default case is not necessary but it may be used for the sake of consistency. The use of the default case is optional. If it is not present, no action takes place if all matches fail and the control goes to the statement-x.

Here, it should be noted that there is no need to put braces around the individual statement blocks lying after the case labels and the case labels end with a colon (:). Also no two case labels should have the same value. However, if we want that a same statement-block is to be executed for more than one case labels, then we put these cases one after the other, and then the specific statement-block i.e., we use

```
switch (expression)
{
  …….
  ……
  case value 5 :
  case value 6 :
```

```
                    statement-block
                    break;
                ……
                ……
            }
```

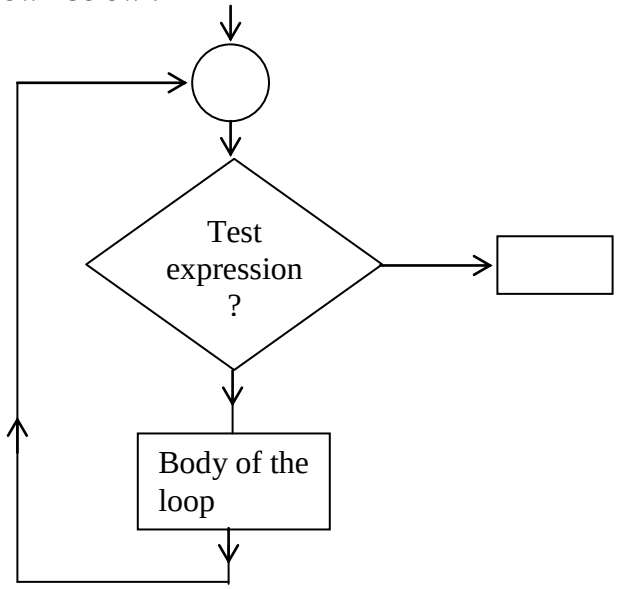
The following program domonstrates the use of switch statement.

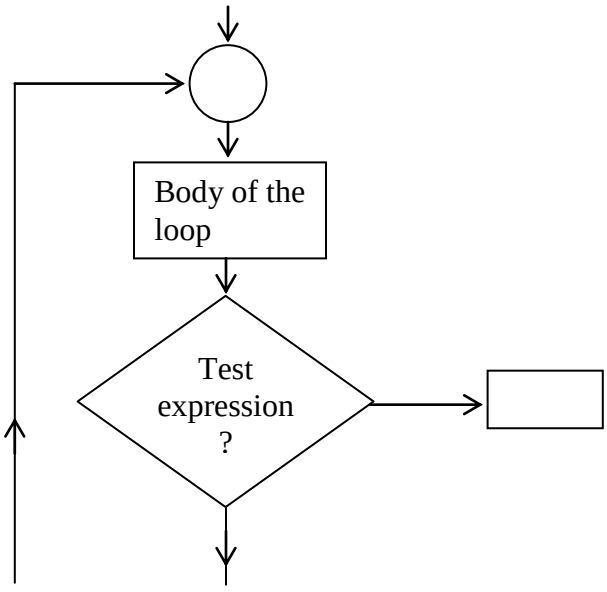
```
/* This program demonstrates the use of switch statement */
main ( )
{
    int  day ;
    printf (″\n Enter day of week as number : ″);
    scanf (″%d″, & day);
    switch (day)
    {
        case 0 : printf (″\n Today is Sunday\n″);
            break ;
        case 1 : printf (″\n Today is Monday\n″);
            break ;
        case 2 : printf (″\n Today is Tuesday\n″);
            break ;
        case 3 : printf (″\n Today is Wednesday\n″);
            break ;
        case 4 : printf (″\n Today is Thursday\n″);
            break ;
        case 5 : printf (″\n Today is Friday\n″);
            break ;
        case 6 : printf (″\n Today is Saturday\n″);
            break ;
        default : printf (″\n Wrong input \n″);
            break;
    }
}
```

10. Looping

In C, loops cause a section of the program to be executed repeatedly while an expression is true. When the expression becomes false, the loop terminates

and the control transfers to the statement following the loop. A loop contains two parts, one is known as the **control statement** and the other is the **body of the loop**. The control statement tests certain conditions and then directs the repeated execution of the statements contained in the body of the loop. Depending on the position of the control statement, a loop may be classified either as the **entry-controlled loop (top-tested loop)** or as the **exit-controlled loop (bottom tested loop)**. In the entry-controlled loop, the control conditions are tested before the start of the loop execution. If the conditions are not satisfied, the body of the loop will not be executed. In case of an exist-controlled loop, the test is performed at the end of the body of the loop and therefore the body is executed unconditionally for the first time. The two types of loops are shown below :





Generally, the test expression (condition) will transfer the control out of the loop after performing the desired number of loop executions. In case, due to some reason, it does not do so, the control sets up an **infinite loop** and the body of the loop is executed again and again.

A looping process generally includes the following four steps
(i) Setting and initialization of a counter.
(ii) Execution of the statements in the loop.
(iii) Test for a specific condition for execution of the loop.
(iv) Incrementing the counter.

The test may be either to determine whether the loop has been repeated the specified number of times or to determine whether a particular condition has been met.

In C, looping is performed by using while, do-while, and for statements and as a result, we have three types of loops named after these statements, i.e., while loop, do-while loop, and for loop respectively. Let us discuss these three statements in detail.

10.1 The while statement : The while statement is the simplest of all the looping structures in C. This statement is suited for problems where it is not known in advance that how many times a statement or a statement-block will be executed. The syntax of a while statement is

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

The semantics of the while statement are as follows. First, the test expression is evaluated. If it is a nonzero value (i.e. true), body of the loop is executed and then the program control returns to the top of the while statement and the process is repeated. This continues indefinitely until the test expression evaluates to zero (i.e. false), and then the control is transferred out of the loop i.e. to the statement immediately following the loop.
We observe that the while is an entry-controlled loop statement. For example, suppose we want to calculate the sum of squares of all integers between 1 and 10. The program loop for this, is as follows :

```
…….
…….
sum = 0;
n = 1;
while (n < = 10)
```

```
{
    sum = sum + n* n;
    n = n + 1;
}
```

Because the incrementation operation occurs so frequently, C has a special increment operator '+ +'. Thus the statement n = n +1; is equivalent to n ++; The following program which calculates the average marks of a number of students, illustrates the use of while statement. We type the marks one by one, and at the end of the list, we enter a number that can not possibly be valid marks, for example, –1.

```
/* Program to find the average of marks of a
* number of students */
  # include  < stdio.h>
  void  main ( )
  {
/* i is the input variable, sum is the sum of the marks, num is
  * the number of marks */
        int   i, num = 0;
        float   sum = 0, average ;
        printf (''Input the marks, –1 at the end. \n'');
        scanf (''%d'', & i);
        while (i ! = –1)                    /* i is not equal to
–1*/
        {
            sum + = i ;                     /* sum = sum +i
*/
            num ++;                         /* num = num +1
*/
            scanf (''%d'', &i);
        }
        average = sum/num ;
        printf (''The average is %.3f '', average);
  }
```

The first scanf statement inputs the first marks and stores it in i, so that the statements inside the loop can have some valid data to work on. The scanf

inside the loop reads the rest of the numbers one by one. When –1 is read, the condition in the while statement ($i \neq -1$) evaluates to false and the while loop terminates and control transfers to next statement.

10.2. The do-while Statement : In the while statement, the test condition is at the top of the loop. This means that if the condition is false at the very first attempt, the body of the loop will not be executed at all. But there are certain situations where we need to execute the body of the loop at least once. Such situations can be handled with the help of the do-while statement. It has the form

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

On reaching the do statement, the program proceeds to evaluate the body of the loop first. At the end of the loop, the test expression in the while statement is evaluated. If the condition is true, the program continues to evaluate the body of the loop once again. This process continues till the condition becomes false and the loop is terminated. Here, we observe that the body of the loop is always executed at least once. Also, since the test expression is evaluated at the bottom of the loop, the do-while statement provides an exit controlled loop. Also note the semicolon following the while statement at the bottom. The following program illustrates the use of do-while statement.

```
/* Program to illustrate the do-while loop */
# include >stdio.h>
void  main ( )
{
        char   inchar;
        do
        {
                printf (''Input y or n : '');
                scanf (''%C'', & inchar);
        }
        while (inchar ! = 'y' && inchar ! = 'n');
        if (inchar = = 'y')
```

```
                printf (''You pressed y. \n'');
        else
                printf (''You pressed n.\n'');
    }
```

The user is prompted to press y on r. In reality, the user can press any key other than y or n. In such a case, the message must be shown again and the user should be allowed to enter one of the two keys.
The do-while loop in the above program, first prompts the user and then evaluates the condition

(inchar ! = 'y' && inchar ! = 'n')

It continues executing the body of the loop and prompts the user while the condition is true (i.e. while the input character is neither a y nor an n).

10.3. The for Statement : This statement is suited for problems where the number of times a statement-block will be executed is known in advance. The for loop is entry-controlled loop that provides a more concise loop control structure. The for loop finds the maximum applications in C. The syntax of the for statement is

```
for (expr 1; expr 2; expr 3)
{
    body of the loop
}
```

where expr 1, expr 2, expr 3 are expressions. Generally expr 1 and expr 3 are assignments or function calls and expr 2 is a relational expression. The expr 1 and expr 3 parts can have more than one statement separated by a comma. The expr 1 is executed once, and expr 2 and expr 3 parts are executed repeatedly. The for statement operates as follows :

(i) First, expr 1 part is evaluated. This is usually an assignment expression such as i = 1, day = 0 that initializes one or more variables. So, this part can be named as initialization of the loop control variables.
(ii) Then expr 2 is evaluated. This is the conditional part of the statement i.e. it is a test-condition for the control variable
(iii) If expr 2 is false, program control exits the for statement and flows to the next statement of the program. If expr 2 is true, body of the loop is executed.
(iv) After execution of the body of the loop, expr 3 part is evaluated. In this part, control variables are incremented (updated) using assignment statements such as i = i +1, day++ and then the new values of the control variables are again tested in expr 2. If the condition is true, the

body of the loop is again executed. This process continues till the condition becomes false.

From the above discussion, we observe that the for loop can be expressed as

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

Also, the statement

```
for (expr 1; expr 2; expr 3)
{
        body of the loop
}
```

in terms of a while statement, can be put as

```
expr 1;
while (expr 2)
{
        body of the loop
        expr 3;
}
```

In a for statement, any of the three parts i.e. expr 1, expr 2, expr 3 can be omitted, but semicolons separating these parts must remain. expr 2 is really omitted and in general, omission of any part is avoided. Thus, the following six statements are equivalent.

(1)
```
for (i = 0; i < = 10 ; i ++)
{
  ……..
  ……..
}
```

(2)
```
i = 0;
for (; i < = 10; i ++)
{
  ……
  ……
}
```
Initialization is done prior to the for statement

(3)
```
for (i = 0; i < = 10;)
{
```
Incrementation is done in the body of for loop

```
    ……..
    ……..
    i ++ ;
}
```

(4)
```
for (i = 0; ; i ++)
{
    …….
    …….
  If ( i > 10) break ;
}
```

Termination of the for statement is done from within the body of the loop.

(We can also use goto statement instead of break statement)

(5)
```
i = 0;
for (; i < = 10 ;)
{
    ……..
    ……..
    i + + ;
}
```

(6)
```
i = 0;
for (; ;)

{
    ……
    ……
    if (i > 10) break;
    i + + ;
}
```

Just as it is possible to omit one or more of the expressions in a for loop, it is also possible to omit the body of the for loop. His is useful when the loop's work is being performed by the expressions of the for statement. For example, we can set up time delay loops using the null statement as follows :

```
for  (i = 1000; i > 0 ; i = i –1)
    ;          /* Null Statement */
```

This loop is executed 1000 times without producing any output. It simply causes a time delay. Here, we observe that the for statement allows negative incrementation. It should be noted that if we use

```
for (i = 100; i > 0; i = i –1);
```

then the semi colon at the end will be considered as a null statement. But we should never use such syntax as it is potentially misleading. The following program which finds the average of n real numbers, illustrates the use of for statement.

```
/* Program to find average of n real numbers */

# include < stdio.h>
main ( )
{
        int   i, n ;
        float   number, sum = 0, average;
        printf (''\n Enter value of n : '');
        scanf (''%d'', &n);
        for (i = 1; i < = n ; i ++)
        {
                printf (''\n Enter real number %d: '', i);
                scanf (''%f '', & number);
                sum + = number ;
        }
        average = sum/n;
        printf (''\n Average of numbers = %.2f\n'', average );
}
```

10.4. Nested Loops : Like if statements, it is possible to nest looping statements. The main point to remember with nested loops is that the inner loops must finish before the outer loops can resume iterating. Proper indentation must be used for nesting of loops.

The following program which prints a multiplication table upto 10, demonstrates the nesting of loops.

```
/* This program prints a multiplication table using nested loop */
# include < stdio.h>
main ( )
{
```

```
    int   j,  k ;
    printf (″ 1      2      3      4      5      6      7      8
9       10  \n″);
    printf (″ ……..…… \n″);
    for ( j = 1; j < = 10; j ++) /∗ outer loop ∗/
{
        printf (″%5d |″, j);
        for (k = 1; k < = 10; k ++) /∗ inner loop ∗/
            printf (″%5d″, j∗k);
        printf (″\n″);
    }
    exit (0) ;
}
```

the output of this program would be as follows :

	1	2	3	4	5	6	7	8	9	10
1|	1	2	3	4	5	6	7	8	9	10
2|	2	4	6	8	10	12	14	16	18	20
3|	3	6	9	12	15	18	21	24	27	30
4|	4	8	12	16	20	24	28	32	36	40
5|	5	10	15	20	25	30	35	40	45	50
6|	6	12	18	24	30	36	42	48	54	60
7|	7	14	21	28	35	42	49	56	63	70
8|	8	16	24	32	40	48	56	64	72	80
9|	9	18	27	36	45	54	63	72	81	90
10|	10	20	30	40	50	60	70	80	90	100

For each value of j, the program first prints j, then loops through ten values of k, printing j∗k for each iteration, and then prints a newline.

Now, we discuss the last part of the control structure i.e. the statements for jumping out of a loop. For this, we have break, continue, and goto statements.

10.5. The break and continue statements : We have already seen the break statement in switch statement where it prevents program flow from falling through to the next case value. In other words, we can say that the break

statement prematurely terminates the switch statement, causing control to flow to the next statement after switch. The break statement serves the similar purpose when used within a looping statement. However, it should be noted that the break statement is best suited for a switch statement.

The following program which calculates the average of a set of positive numbers, illustrates the use of break in a loop.

```
/* Program to compute average of a set of positive numbers */
    # include < stdio.h>
    main ( )
    {
       int  i ;
            float   x, sum = 0, average;
            printf (''Enter numbers, NEGATIVE number
                at the end.\n'');
            for (i = 1; i < = 100; i ++)
            {
                    scanf (''%f '', & x);
                    if (x < 0)
                            break ;
                    sum + = x;
            }
            average = sum/(float) (i –1);
            printf (''\n'');
            printf ('' Total numbers = % d\n'', i –1);
            printf ('' Sum = %f\n'', sum);
            printf ('' Average = %f\n'', average);
    }
```

In the above program, the loop is written to read 100 values. However, if we want to use less number of values, we must enter a negative value in the last, to mark the end of the input, resulting in break of loop.

The **Continue statement** provides a means for returning to the top of a loop earlier than normal. It is particularly useful for a situation where we want that from a given statement onwards, the rest of the statements upto the last

statement of the loop are to be by passed. Thus, it tells the compiler, " Skip the following statements and **continue** with the next iteration". The syntax of the continue statement is

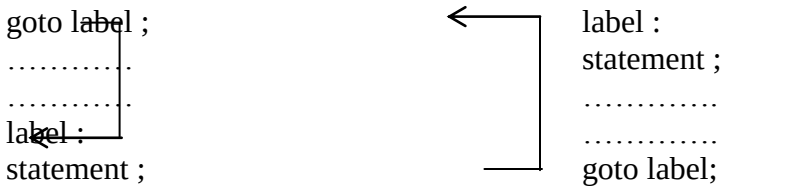
```
continue;
```

Let us consider a program which evaluates the square root of a series of numbers and prints the result. The process stops when a number 9999 is typed in. In case, the series contains a negative number, the process of evaluation of square root is bypassed using a continue statement. The program also prints a message saying that the number is negative and keeps an account of negative and positive items

```
/* Program to illustrate use of break and continue statements */
# include < stdio.h >
# include < math.h >        /* for sqrt ( ) function */
main ( )
{
        int   count = 0 negative = 0;
        double   number, sqroot ;
        printf (''Enter 9999 to & top\n'');
        while (count < = 100)
        {
                printf (''Enter a number : '');
                scanf (''%lf '', & number);
                if (number = = 9999)
                        break ;   /* Exit from the loop */
                if (number < 0)
                {
                        printf (''Number is negative \n \n'');
                        negative ++ ;
                        continue ;   /* skip rest of the loop */
                }
                sqroot = sqrt (number);
                printf (''Number = % lf \n, square root
                                = % lf \n\n'',number,sqroot);
```

```
            count ++ ;
        }
            printf (''\n Negative items = %d \n'', negative);
            printf ('' Number of items done = %d \n'', count);
        printf (''END OF DATA \n'');
    }
```

10.6. The goto Statement : It is a good practice to avoid using goto statement; still a situation may arise where we are forced to use it. The goto statement is used to branch unconditionally from one point to another in the program. The destination spot is identified by a **statement label,** which is just a name followed by a colon. The label is placed immediately before the statement where the control is to be transferred.
The syntax of goto and label statements are



(Forward jump) (Backward jump)

Thus, the label : can be anywhere in the program either before or after the goto label;
During the execution, when a statement like

goto total;

is met, the control will jump unconditionally to the statement immediately flowing the label total :
Further, we observe that in case of forward jump, some statements will be skipped and in case of backward jump, a loop will be formed and some statements will be executed repeatedly. The goto statement is useful particularly when we want to transfer control from deeply nested statements, such as jumping out of two or more loops at once. In such case break statement can not be used directly since it exits only from the innermost loop.

The following program illustrates the use of goto.

```
/* Program to illustrate use of goto statement */
# include < stdio.h>
# include < math.h >
 main ( )
{
```

```
        int   num ;
        printf (′′Enter any number.\n′′);
        scanf (′′ % d′′, & num);
        if (num < 0)
                goto bad_val;
        else
        {
                printf (′′Square root of number is % f ′′, sqrt
(num));
                goto end;
        }
        bad_val :
        printf(′′Error : Negative value.\n′′);
        exit (1);          /* abnormal exit */
        end : exit (0) ;
    }
```

However, this program can be written in a much better way without using goto. We have used goto just for illustration.

10.7. Infinite Loops : An infinite loop is a loop that does not contain a terminating condition or a loop in which the terminating condition is never reached. In most instances, infinite loops are produced by bugs in the program. For example, in the following statement

```
for (i = 0; i < 10; i ++)
{
……….
……….
     i = 1
}
```

the loop will never finish because i is assigned the value 1 on each iteration.
In certain situations, we may require an infinite loop. There are many ways to write infinite loops, but the two most common are

```
for ( ; ; )
 {
      body of the loop
}
```

when test expression (expr 2) is not present, it is taken as permanently true

and

```
while (1)        | The condition is permanently true.
{
        body of the loop
}
```

Also, if we use a got statement at the end of a loop and the label : prior to the beginning of the, loop, then due to backward jump, an infinite loop is set. Again note that such infinite loops should be avoided.
To get out of an infinite loop, we need to abort the program manually. On most systems, we can abort a program using Ctrl-C.

UNIT - II

1. Operators and Expressions

Operators are the verbs of a programming language that enable us to calculate values. C supports a rich set of operators which is one of its distinguishing characteristics. We have already seen a number of C operators, such as = +, –, *, & etc. An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations. Operators operate on certain data types called operands, and they form a part of the mathematical or logical expressions. An expression is a combination of variables, constants and operators written according to the syntax of the language. In C, every expression evaluates to a value. It consists of one or more operands and zero or more operators linked together to compute a value. Thus.

a + 2

is a legal expression. The variable a all by itself is also an expression and so is the constant 2, since they both represent a value.

Based on data types, we have the following four important types of expressions.

(i) Constant expressions that contain only constant values e.g.

3

′a′

2+3 *13/7.0

(ii) Integral expressions that, after all implicit and explicit type conversions, produce a result that has one of the integer types. e.g. if i and j are integers, the following are integral expressions

i

i *j

i/j+5

j –’a’

5 +(int) 3.0

(iii) Float expressions which, after all implicit and explicit type conversions, produce a result that has one of the floating-point types.

e.g. if x is a float or double, the following are float expressions

5.0

x

x+5

x/y*3

5.0–3

5+(float) 3

(iv) Pointer expressions that evaluate to an address value. These include expressions containing pointer variables, the "address of" operator (&), string literals, and array names.

e. if i is an int and p is a pointer, the following are pointer expressions

&i

p

p+1

″xyz″

In c, operators can be classified into various categories based on their utility and action. A list of operator types is given below.

(i) Arithmetic operators

(ii) Relational operators

(iii) Logical operators

(iv) Assignment operators

(v) Increment and decrement operators

(vi) Conditional operators

(vii) Bitwise operators

(viii) Comma operator

(ix) Special operators.

Expressions are named on the basis of these specific categories of operators.

1.1. Precedence and Associativity : All operators have two important properties called precedence and associativity. Both properties affect how operands are attached to operators. The **precedence** is used to determine how an expression involving more than one operator is evaluated. There are distinct levels of precedence and an operator belongs to one of these levels. The operators at the higher level of precedence are evaluated first, regardless of the order in which they appear. The operators of the same precedence level are evaluated either from left to right or from right to left, depending on the level. This is known as the **associativity** property of an operator. Left to right associativity means that the compiler starts on the left of the expression and works right. Similarly, right to left associativity means that the compiler starts on the right of the expression and works left. Thus, grouping or binding of operands with operators occurs in either of the two ways, depending on the operator.

For example, the two expressions

3 + 4 ∗ 5

$4 * 5 + 3$

both evaluate 23 as the operand 4 is grouped with multiplication operator rather than addition operator because multiplication operator has higher precedence. Also both the operators are left to right associative. Again, consider

$a = b = c * d$

Here, c *d would be calculated first due to higher precedence of * and then the value will be assigned to b, then b is assigned to a, due to right to left associativity of the assignment operator. The following table lists all c operators in order of precedence alongwith indicating their associativity. Rank 1 indicates the highest precedence level.

Category	**Operator**	**Operation**	**Precedence**	**Associativity**
Primary	( )	Function call	1	L→R (left to right)
	[ ]	Array elements		
	–>	C indirect component selector		
	: :	C scope access/resolution		
	.	C Direct component selector		
Unary	cast operator	Explicit conversion or local conversion	2	R→L (right to left)
	sizeof	Returns size of operand, in bytes		
	!	Logical negation (NOT)		
	~	Bitwise 1's complement		
	+	Unary plus'		
	–	Unary minus		
	+ +	Preincrement or postincrement		
	– –	Predecrement or postdecrement		
	&	Address of		
	*	Pointer reference or indirection or dereference		
Member access	.*	Dereference	3	L→R
	→*	Dereference		
Multipli-cative	*	Multiply	4	L→R

	/	Divide		
	%	Remainder or modulus		
Additive	+	Binary plus (addition)	5	L→R
	–	Binary minus (subtraction)		
Shift	<<	Shift left	6	L→R
	>>	Shift right		
Category	**Operator**	**Operation**	**Precedence**	**Associativity**
Relational	<	Less than	7	L→R
	< =	Less than or equal to		
	>	Greater than		
	> =	Greater than or equal to		
Equality	= =	Equal to	8	L→R
	!=	Not equal to		
Bitwise AND	&	Bitwise AND	9	L→R
Bitwise XOR	^	Bitwise exclusive OR	10	L→R
Bitwise OR	|	Bitwise inclusive OR	11	L→R
Logical AND	&&	Logical AND	12	L→R
Logical OR	||	Logical OR	13	L→R
Conditional	? :	Conditional expression	14	R→L
Assignment	=	Simple assignment	15	R→L
	* =	Assign product		
	/ =	Assign quotient		
	% =	Assign remainder (modulus)		
	+ =	Assign sum		
	– =	Assign difference		
	& =	Assign bitwise AND		
	^ =	Assign bitwise XOR		
	| =	Assign bitwise OR		
	<< =	Assign left shift		
	>> =	Assign right shift		
Comma	,	Evaluate	16	L→R

1.2. Remarks : (i) When we write an operator which consists of more than one character, such as > =, there should be no intervening spaces between the characters. Thus > = is illegal.

(ii) We should use parentheses to specify a particular grouping order, since compiler groups operands and operators first that appear within the parentheses. Parentheses serve a valuable stylistic function by making an expression more readable and ensuring that the expression is evaluated correctly. They also enable us to understand an expression without referring to the precedence table. In case of nested parentheses, the compiler groups the expression enclosed by the innermost parentheses first.
(iii) Another important property of expressions is the **order of evaluation** of operands of an operator. The precedence and associativity have little to do with this property. For most operators, the compiler is free to evaluate the operands (or sub-expressions) in any order it pleases.
For example, consider the statement

x = i ∗ i ++ ;

Here we have two operands i and i++ for the operator ∗.
If i = 3 and the left-hand operand is evaluated first, then

x = 3∗3 = 9

If right-hand operand is evaluated first, then

x = 4∗ 3 = 12

This increment, decrement, and the assignment operators cause such **side effects** i.e. they not only result in a value but also change the value of a variable. Statements of the above type are not portable and should be avoided. To prevent side effect bugs, we should remember that if we use a side effect operator in an expression, we should not use the affected variable anywhere else in the expression. Thus, the above statement should be broken into two statements as

x = i ∗ i;

+ + i;

2. C Operators

Now, let us discuss the C operators in detail.

2.1. Unary Plus and Minus Operators : The plus and minus operators are called unary operators when they take only one operand. The operand may be any integer or floating-point value. The type of the result is the type of the operand after integral promotions. The unary plus sign is an ANSI feature not found in older compilers. It does not have any effect except to promote small integer types. The unary minus operator returns the negation of its argument.
These operators have the following operation

Operator	Symbol	Form	Operation
Unary plus	+	+ x	Value of operand
Unary minus	–	–x	Negation of x

The unary minus operator should not be confused with the binary subtraction operator. Even though they use the same symbol, they are different. For example

```
i = 5 – – x;
```

is interpreted as

```
i = (5–(–x));
```

The first dash is a subtraction operator, the second is a unary minus sign. Further note that the space between the two dashes prevent them from being interpreted as a decrement operator.

2.2. Binary Arithmetic Operators : Operators of this category are listed below.

Operator	Symbol	Form	Operation
Multiplication	*	x * y	x times y
Division	/	x/y	x divided by y
Remainder	%	x % y	Remainder of x divided by y
Addition	+	x+y	x plus y
Subtraction	–	x–y	x minus y

All of the arithmetic operators bind from left to right.

We are very much familiar with these operators, except the remainder (%) operator. The multiplication, division and remainder operators are called **multiplicative operators** and have a higher precedence than the **additive operators** i.e. addition and subtraction. The multiplicative operators accept the operands of integral or floating-point type only whereas the operands of the additive operators can be those whose type is integral, floating point or pointer. However, the remainder (modulus or modulo division) operator can not be used on floating-point data.

When both operands of the division operator are of the same sign during integer division, the result is truncated towards zero (i.e. fractional part is truncated). If one of them is negative, the direction of truncation is implementation dependent.

Thus 5/6 = 0 and –5/–6 = 0

but –5/6, 5/–6 may be zero or –1 (machine dependent) i.e. the compiler is free to round off the result either up or down. Similarly, during modulo division, we may have

–7%2 = 1 or –1

8% –5 = 3 or –3

On some machines, during modulo divisions, the sign of the result is the sign of the first operand i.e. the dividend.

Thus

–9% 2 = –1

–9% –2 = –1

9% –2 = 1

Obviously, we should avoid division and remainder operations with negative numbers since the result can vary from one compiler to another.

If the sign of the remainder is important for our program's operations, we should use the runtime library div( ) function, which computes the quotient and the remainder of its two arguments. The sign of both results is determined in a guaranteed and portable manner.

As an illustration of the use of the remainder operator, let us consider the following program which convert a given number of days into months and days.

```
/* Program to convert days into months and days */
# include <stdio.h>
main ( )
{
        int   months, days;
              printf (″ Enter Number of Days\n″);
              scanf (″%d″, & days);
              months = days/30;
              days = days % 30;
              printf (″Months = %d,  Days = %d″, months,
days);
              return 0;
 }
```

Here, the statement

months = days/30

truncates the decimal part and assigns the integer part to months. Similarly, the statement

days = days% 30

assigns the remainder part of the division to days.

Most frequent application of the modulus operator is to perform some action at regular intervals. This is illustrated in the following program which reads a line of input and prints it out, inserting a newline after every seven characters.

```
/* Program to break a line into newlines of 7 characters*/
# include <stdio.h>
main ( )
{
        int   c, i = 0;
        printf (″Enter any string to be broken:″);
        while (( c = getchar ()) ! =′\n′)
        {
                if (i%7 = = 0)
                printf (″\n″);
                putchar (c);
                i ++;
        }
    exit (0);
}
```

The output of this program is as follows :

```
Earth moves round the Sun continuously…..
Earth m
oves ro
und the
Sun con
tinuous
ly……
```

2.3. Remarks. (i) C does not have an operator for exponentiation. However, a runtime library function pow(x, y) exists in math.h which returns x^y.
(ii) Expressions involving arithmetic operators are called arithmetic expressions.

2.4. Arithmetic Assignment Operators : For this category, we have the operators given in the following table

Operator	Symbol	Form	Operation
assign	=	x = y	assign the value of y to x
add-assign	+ =	x + = y	assign the value of x + y to x i.e. x = x + y
subtract-assign	– =	x – = y	assign the value of x –y to x i.e. x = x–y
multiply-assign	∗ =	x ∗ = y	assign the value of x ∗ y to x i.e. x = x ∗y
divide-assign	/ =	x / = y	assign the value of x/y to x i.e. x = x/y
remainder-assign	% =	x % = y	assign the value of x% y to x i.e. x = x%y

All the assignment operators have right to left associativity.

As already mentioned, the assign operator (=) causes the value of the right-hand operand (or expression) to be written into the memory location of the left-hand operand. The left-hand operand, sometimes called **lvalue**, must refer to a memory location.

During multi-assignment statements, the right most operand gets assigned to the leftmost operand. In the process, each assignment may cause implicit conversions. For example, the following statement

```
int   i;
float  x;
x = i = 5.6;
```

assigns the truncated value 5 to both i and x, whereas the statement

```
i = x = 5.6;
```

assigns 5.6 to x and 5 to i.

Other five assignment operators in the above table are obtained by combining the assignment with each of the arithmetic operators. That is why they are called arithmetic assignment operators. They are sometimes termed as **shorthand assignment operators.** Their functioning is well explained in the table itself. The use of arithmetic assignment operators has three advantages

(i) The contents of the left-hand side are not repeated which helps in avoiding spelling mistakes and making code more readable and concise. For example, the expression

```
actualvalue (3 ∗ i–2) = actualvalue (3∗ i–2) + alpha ;
```

is better written as

actualvalue (3∗i–2) + = aplha;

This concept becomes even more important when referencing structure and union members.

(ii) Efficiency of the code gets increased due to the fact that some computers have special machine instructions to perform arithmetic-assign combinations. A good compiler will usually rewrite an expression for us to take advantage of this feature.

(iii) If the left-hand value contains side effects, since as the left-hand value is not repeated, the side effects occur only once. This feature has special significance for the case of arrays.
Further, due to relatively low precedency of assign operators, we observe that the following two expressions are not the same

i = i ∗ 2 +3;

i ∗ = 2+3;

Here, we have three operators i.e. =, +, ∗ in which + has higher precedency than = and ∗ has higher precedency than +. So the compiler interprets the first expression as

i = ((i ∗2) +3)

and the second expression as

i ∗ = (2+3) i.e. i = (i∗(2 +3))

So, we should use such expressions carefully. Some more examples of expressions using arithmetic assignment operators are tabulated as follows, where we use the statements

int i = 2, j = 3, k = 4;

float x = 1.0, y = 1.5;

Expression	**Equivalent form**	**Result**
i + = j–k ∗ x+y	i = (i +(j–(k∗x)+y))	2
j – = i/k + x–y	j =(j–((i/k) + x–y))	3
k/ = x +y∗j	k = (k/(x+(y∗j)))	0
y ∗ = x+k –j∗i	y = (y∗(x+k–(j∗i)))	–1.5

k % = j∗x–i k (k%((j∗x)–i))
0

i – = x + =k/= j–y i = (i –(x = x +(k = k/(j–y)))) –1

2.5. Increment and Decrement Operators : We have two versions of each of these operators as shown in the following table.

Operator	Symbol	Form	Operation
Postfix increment	+ +	i ++	get value of i, then increment i
Prefix increment	+ +	++i	increment i, then get value of i
Postfix decrement	– –	i ––	get value of i, then decrement i
Prefix decrement	– –	– – i	decrement i, then get value of i

From the table, we observe that if the operator comes after the variable, it is termed as **postfix** operator and if it comes before the variable, it is called a **prefix** operator.

The postfix increment or decrement operators fetch the current value of the variable and store a copy of it in a temporary location. The compiler then increments or decrements the variable. The temporary copy, which has the variable's value before it was modified, is used in the expression. For example, consider the following program

```
# include <stdio.h>
main ( )
{
        int  i = 2, j = 2;
    printf (″ i :%d\t j: %d\n″, i ++, j ––);
    printf (″ i: %d\t j : %d\n″, i, j);
    return 0;
}
```

The output is

i : 2 j : 2

i : 3 j : 1

Thus, in the first printf call, the initial values of i and j are used, after that they are incremented and decremented respectively.

In case of prefix increment and decrement operators, operands are modified before they fetch the values. Thus in the above program, if we replace i++, j–– by ++i, ––j, then the output is

i : 3 j = 1

i : 3 j : 1

As far as the side effect is concerned, the two versions are equivalent i.e. both versions produce the same side effect.

Further, we observe that the increment and decrement operators are unary operators and thus have a very high precedency. For these operators, the operand must be a scalar lvalue, so it is illegal to increment or decrement a constant or a structure. However, we can increment or decrement a pointer variable but the meaning of adding one to a pointer is totally different from that of adding one to an arithmetic value, which we shall see in Unit–III.

Some examples using increment and decrement operators are listed below, where we use the statement

int i = 0, j = 1, k = 2;

Expression	**Equivalent form**	**Result**
i + + – + + j +k ++	(i+ +) – (+ +j) + (k ++)	0
+ +i + j+ + + + +k	(+ +i) + (j + +) + (+ +k)	5
i – –+ j + + – – –k∗2	(i – –) + (j + +) –((– –k)∗2)	–1
j + + ∗ – – k – + + i	((j + +) ∗ (– –k)) –(+ + i)	0
i + = – – j + k + +	i = (i +((– – j) + (k + +)))	2
j – = i – – – – – k	j = (j –((i– –) – (– – k)))	2
j + + + k Misleading	(j + +) +k or j + (+ + k)	
k + + ∗ – – k Machine dependent	(k + +) ∗ (– – k)	(2 or 1)

2.6. Comma Operator : The comma operator can be used to link the related expressions together. It allows to evaluate two or more distinct expressions wherever a single expression is allowed. A comma-linked list of expressions

is evaluated left to right and the value of the right-most expression is the value of the combined expression. For example, the statement

value = (x = 2, y = 3, x +y);

first assigns the value 2 to x, then assigns 3 to y and finally assign 5 (i. e. 2+3) to value. Since comma operator has the lowest precedence of all operators, the parentheses are necessary.

Although the comma operator is legal in a number of situations, it leads to confusing code in many of them. Therefore, by convention, the comma operator is used primarily in the first and last expressions of a for statement. For example,

```
for ( i = 0, j = 100; j – i > 0; i + +, j – –)
```

Here, both i and j are initialized before the loop is entered. After each iteration, i is incremented and j is decremented. This statement is equivalent to the following while loop.

```
i = 0;
j = 100;
while (j –i >0)
{
    ……..
    ……..
    i + + ;
    j – –;
}
```

which can also be written as

```
i = 0, j = 100;
while (j – i >0)
{
    ……
    ……..
    i + +, j – – ;
}
```

The comma operator can also be used in while statement, such as while (i > 0, i ! = 10)

and for exchanging values, such as

a = b, b = c, c = d, d = a;

2.7. Relational Operators : Operators of this category are as follows:

Operator	Symbol	Form	Result
Greater than	>	a > b	1 if a is greater than b (true); else 0 (false)
Less than	<	a < b	1 if a is less than b; else 0
Greater than or equal to	> =	a > =b	1 if a is greater than or equal to b; else 0
Less than or equal to	< =	a < = b	1 if a is less than or equal to b; else 0
Equal to	= =	a = = b	1 if a is equal to b; else 0
Not equal to	! =	a ! = b	1 if a is not equal to b; else 0

Like the arithmetic operators, the relational operators are binary operators. These operators are also called **comparison operators.** Expressions involving relational operators are called relational expressions or Boolean expressions (after the name of mathematician and logician George Boole who reduced logic to a propositional calculus involving only true and false values). In C, the value of a relational expression is either one or zero. It is one if the specific relation is true and zero if the relation is false.

It should be noted that all the relational operators have lower precedence than the arithmetic operators, therefore when arithmetic expressions are used on either side of a relational operator, the arithmetic expressions will be evaluated first and then the results compared. Thus, the expression

a + b * c < x/y

is treated as

(a + (b *c)) < (x/y)

Among the relational operators given in the above table, the first four i.e. >, <, > =, < = have the same precedence. The = = and ! = operators have lower precedence.
Relational operators are used in decision statements such as if, while, for etc to decide the course of action of a running program.

The following examples illustrate how relational expressions are evaluated.

Expression	value
1 > 0	1
0 > 1	0
0 = = 0	1
1 > = –1	1
1 ! = –1	1
1 ! = 1	0

2∗3 < = 2+3 0

The following examples illustrate how the compiler analysis the complex relational expressions, where we use the statements

int i = 0, j = 1, k = –1;

float x = 1.5, y = 2.5;

Expression	**Equivalent form**	**Result**
i > j	i > j	0
j/k < y	(j/k) < y	1
i + j∗k > = y/x	(i +(j∗k)) > =(y/x)	0
i < = j > = k	(i < = j) > = k	1
k < =(y–x) = = j	(k < = (y–x)) = = j	1
x < = k/j ! = i	x < = (k/j)) ! = i	0
–y +j = = x >k > i	((–y) +j) = = ((x>k) >i)	0
x + =(y > = k)	x = (x + (y > =k))	2.5
+ + i = = j ! = (y–x) > k	(((+ + i) = = j) ! = (y–x)) > k	1
y – = (j/k > i)	y = (y–((j/k) > i))	Machine dependent (1.5 or 2.5)

2.8. Logical Operators : In addition to the relational operators, C has the following three logical operators

Operator	**Symbol**	**Form**	**Result**
logical AND	& &	a && b	1 if a and b are non zero; else 0
logical OR	||	a ||b	1 if a or b is non zero ; else 0
logical NOT	!	!a	1 if a is zero; else 0

We know that in algebra, the expression

a < b < c

is true if b is greater than a and less than C. On the contrary, this expression has a very different meaning in C as it is evaluated as

(a < b) < c

The sub-expression (a < b) is evaluated first and results in either 1 or 0. So in C, the expression is true if a is less than b and c is greater than 1 or if a is not less than b and c is greater than zero. To obtain the algebraic meaning, we must rewrite the expression using logical operators.

The logical AND operator (&&) and the logical OR operator (||) evaluate the truth and falseness of pairs of expressions. The AND operator returns TRUE only if both expressions are TRUE. The OR operator returns TRUE if either expression is TRUE. Thus, the above expression for testing whether b is greater than a and less than c, should be written as

(a < b) & & (b < c)

More examples of such expressions are

a > b && c = = 10

age > 25 && salary < 10000

i < 0 || i > 10

a < m || a < n

An expression of such type which combines two (or more) relational expressions, using && or || operator, is termed as a **logical expression or a compound relational expression.**

The logical negation (NOT) operator (!) takes only one operand. If the operand is TRUE, the result is FALSE and if the operand is false, the result is TRUE. We have already observed that, in C, TRUE is equivalent to any non zero value and FALSE is equivalent to zero. Thus, we have the following logical tables for each logical operator alongwith the numerical equivalent. It should be noted that the operators return 1 for TRUE and 0 for FALSE.

Operand	Operator	Operand	Result
Zero	&&	Zero	0
Non zero	&&	Zero	0
Zero	&&	Non zero	0
Non zero	&&	Nonzero	1
Zero	||	Zero	0
Non zero	||	Zero	1
Zero	||	Non zero	1
Non zero	||	Nonzero	1
Operand	!	Zero	1
Not needed	!	Non zero	0

The operands for the logical operators may be integers or floating-point type. Thus, the expressions

```
2 && –3
1.5 && 5
```

result in 1 as both operands are non zero. Logical operators (and the comma and conditional operators) are the only operators for which the order of evaluation of the operands is defined. For these operators, the compiler must evaluate operands from left to right. Moreover, the compiler has freedom of not to evaluate an operand if it is unnecessary. For example, consider the following expression

```
if ((a ! = 0) && (b+c = = 25.0))
```

Here, if a = 0, then the second part i.e. b + c = 25.0 will not be evaluated. This freedom to compiler can have unexpected consequences when one of the expressions contains side effects. For example, consider, the expression.

```
if ((a < 10) && (b = = c ++))
```

In this case, c is incremented only when a < 10. If a is not less than 10, the 2nd sub-expression will not be evaluated. This may or may not be what we intended. Thus we should avoid using side effect operators in relational expressions.

Generally, a logical expression is used as the conditional part of a looping statement or in an if statement. Linking expressions with the logical AND operator is equivalent to using nested if statements. Thus, the expression

```
if ((a < b) && (b > c))
{
        statement-block
}
```

is functionally equal to

```
if (a < b)
        if (b > c)
        {
                statement-block
        }
```

This is true as long as there is no else part. However, the following if-else statement

```
if ((a < b) && (b > c))
{
```

```
            statement block-1
        }
        else
        {
            statement block-2
        }
```

is not the same as

```
        if (a < b)
            if (b >c)
            {
                statement block–1
            }
        else
        {
            statement block-2
        }
```

To get the same functionality, we should write

```
        if (a < b)
            if (b > c)
            {
                statement block–1
            }
            else
            {
                statement block–2
        }
      else
        {
            statement block–2
        }
```

We prefer the logical expression versions of the above statements since they are more readable and more maintainable. However, in cases where logical

expression become too long to understand, it is better to break it up into nested expressions.

The following examples illustrate the use of logical and relational operators. Here, it should be noted that the logical NOT operator has a higher precedence than others. The logical AND operator has higher precedence than logical OR operator. Both the logical AND and OR operators have lower precedence than the relational and the arithmetic operators. In these examples, we use the statements

```
int   i = 0, j = 1,        k = –1;
float   x = 1.5, y = 2.5;
```

Expression	Equivalent form	Result
i && j	(i) && (j)	0
i < j && x < y	(i < j) && (x<y)	1
j + k || !i	(j +k) ||(!i)	1
i ∗j < y ∗k ||x	((i ∗j) <(y∗k)) || x	1
x + j || i ∗y	(x +j) || (i∗y)	1
y∗ i || !j &&x	((y∗ i) || (!j)) &&x	0
x∗2 && y || j/k	((x∗2) &&y) ||(j/k)	1
! i && j +k || ! y	((! i) && (j+k)) || (! y)	0
! y || ! x || j–k	((! y) || (! x) || (j–k)	1
i < = 10 && k > =1&&y	((i < =10) && (k > =1) && y	0
(y > x) + ! i && + + k	((y > x) + (! i)) && (+ + k)	0
i ++ && x || k ++	((i ++) && x) || (k ++)	1

3. Bit-Manipulation Operators

C has the distinction of supporting special operators known as **bitwise operators** for manipulation of data at bit level. These operators operate on each bit of data and they are used for testing, complementing or shifting bits to the left or right. The operands for these operators must be of integer type. The following table lists the bitwise operators and their meaning.

Operator	Symbol	Form	Result
left shift	<<	a << b	a shifted left b bits
right shift	>>	a >> b	a shifted right b bits
bitwise AND	&	a & b	a bitwise AND ed with b
bitwise inclusive OR	|	a|b	a bitwise OR ed with b
bitwise exclusive OR (XOR)	^	a ^b	a bitwise exclusive OR ed with b
bitwise 1's complement	~	~a	bitwise complement of a

These operators can be classified into the following three categories

(i) Bitwise shift operators (<<, >>)

(ii) Bitwise logical operators (&, |, ^)

(iii) Bitwise |'s complement operator (~)

To discuss these operators in detail, we assume, for convenience, that an integer occupies 16 bits (2 bytes). To represent negative numbers, most computers use **two's complement notation.** First of all, we explain this notation.

3.1. Two's Complement Notation : We observe that in a 16-bit int (short int), each bit has a value of 2 to the power n, where n represents the position of the bit. Thus, we have

2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}

For example, the decimal value 13 would be represented by setting bits 0, 2 and 3 as

0000 0000 0000 1101 ($2^3 + 2^2 + 2^0 = 8 + 4+1 = 13$)

(The space after every four bits is used only for clarity. Actually, the integers occupy continuous bits)

In two's complement notation, the leftmost bit i.e. the most significant bit is a sign bit. If it is set to one, the number is negative; if it is zero, the number is positive. To negate a binary number, we first complement all the bits (i.e. change 0's to 1's and 1's to 0's) and add 1 to it. Thus to get –13, we first complement the bits to get

1111 1111 1111 0010

then we add 1 to obtain

1111 1111 1111 0011

This represents –13 in two's complement notation.

To see the magnitude of this negative number, we take two's complement again (2's complement of a 2's complement is the original number itself). For this, we find

0000 0000 0000 1100

and finally

0000 0000 0000 1101

3.2. Remarks (i) Some machines use the less popular notation, called one's complement, in which we simply complement the bits to negate a number, without adding one. This notation seems simpler, but has several drawbacks, one of which is that there are two representations for zero i.e.

0000 0000 0000 0000

and

1111 1111 1111 1111

In two's complement notation, there is only one representation for zero because in that case, after complementing the bits we add one, which makes all the bits zero again.

(ii) One of the interesting features of two's complement notation is that −1 is represented by all bits being set to one. It also follows that the largest positive number that can be represented occurs when all but the sign bit are set. This value is $2^{(n-1)} - 1$ where n is the number of bits. The largest negative value is $-2^{(n-1)}$.

Now, we discuss the bit-manipulation operators in detail.

3.2. Bitwise Shift Operators : The two shift operators, << and >>, enable us to shift the bits of an object a specified number of places to the left or the right. These operators are used in the following forms.

operand << n (left shift)

operand >> n (right shift)

where operand is an integer (or integer expression) that is to be shifted and n is the number of bit positions to be shifted.

The left-shift operation causes all the bits in the operand to be shifted to the left by n positions. The leftmost n bits in the binary model of the operand will be lost and the rightmost n bit positions that are vacated will be filled with zeros.

Similarly, in case of right-shift operation, all the bits in the operand get shifted to the right by n positions. The rightmost n bits will be lost. The leftmost n

bit positions that are vacated, will be filled with zeros if the operand is an unsigned integer. If the operand to be shifted is signed, then the operation is machine dependent.

Here, it should be noted that n should not be negative and it should not exceed the number of bits used to represent the operand.

Thus, when operand is unsigned (i.e. + ve), the shifting takes place as illustrated in the following examples.

Expression	**Binary Model of Operand**	**Binary Model of the Result**	**Result Value**
13 << 3	0000 0000 0000 1101	0000 0000 0110 1000	104
13 >> 2	0000 0000 0000 1101	0000 0000 0000 0011	3
127 >> 4	0000 0000 0111 1111	0000 0000 0000 0111	7
16 << 10	0000 0000 0001 0000	0100 0000 0000 0000	2^{14}
1 << 15	0000 0000 0000 0001	1000 0000 0000 0000	-2^{15}

We observe that shifting to the left is equivalent to multiplying by powers of 2. i.e.

a << b is equivalent to $a * 2^b$

Shifting non-negative integers to the right is equivalent to dividing by powers of 2 i.e.

a >> b is equivalent to $a/2^b$

Thus the shift operators are often used for multiplication and division by powers of 2.
When a negative value is shifted to the right, the vacant bits on the left can be filled with ones or zeros, depending on the implementation.
This is illustrated in the following example.

Expression	**Binary Model of Operand**	**Binary Model of the Result**	**Result Value**
–5 >> 2	1111 1111 1111 1011	0011 1111 1111 1110	$2^{13}-1$
–5 >> 2	1111 1111 1111 1011	1111 1111 1111 1110	–2

The first version, in which the vacant bits are filled with zeros, is called a **logical shift.** The second version retains the arithmetic value and is, therefore, called an **arithmetic shift.** The ANSI standard does not specify

whether a compiler should perform a logical shift or an arithmetic shift for signed objects. Generally, we should avoid shifting signed objects.

3.3. Bitwise Logical Operators : There are three bitwise logical operators i.e. bitwise AND (&), bitwise OR(|) and bitwise exclusive OR (^). These operators are similar to logical operators, except that they operate on every bit in the operands.

While using bitwise operators, it is better to express the operands in hexadecimal notation, where we observe that hexadecimal numbers act as a shorthand for binary numbers. Every hexadecimal digit represents four bits of binary representation. We have the following conversion table for converting from binary to hexadecimal and vice-versa.

Decimal	Octal	Hex	Binary
0	0	0	0000
1	1	1	0001
2	2	2	0010
3	3	3	0011
4	4	4	0100
5	5	5	0101
6	6	6	0110
7	7	7	0111
8	10	8	1000
9	11	9	1001
10	12	A	1010
11	13	B	1011
12	14	C	1100
13	15	D	1101
14	16	E	1110
15	17	F	1111

Let us describe the bitwise logical operators with the help of examples and using the above conversion table.

(a) Bitwise AND : The bitwise AND operator (&) compares each bit of the left operand with the corresponding bit in the right operand. If both bits are one, a one is placed at that bit position in the result, otherwise a zero is placed

at that bit position. For example, the following table illustrates the use of the bitwise AND operator.

Expression	Hexadecimal value	Binary Representation
9581	0x 256 D	0010 0101 0110 1101
5347	0x 14 E3	0001 0100 1110 0011
9581 & 5347	0x 0461	0000 0100 0110 0001

We observe that the decimal value of the result is

$$4 \times 16^2 + 6 \times 16^1 + 1 \times 16^0 = 1024 + 96+1 = 1121$$

(b) Bitwise OR : The bitwise inclusive OR operator (|) places a one in a bit position of the result if at least one of the bits in the two operands has the value 1, otherwise a zero is placed at that bit position. The use of bitwise OR is illustrated as follows.

Expression	Hexadecimal value	Binary Representation
9581	0x 256 D	0010 0101 0110 1101
5347	0x 14 E3	0001 0100 1110 0011
9581 | 5347	0x 35EF	0011 0101 1110 1111

The decimal value of the result is $3\times 16^3 + 5 \times 16^2 + 14 \times 16^1 + 15 = 13807$

(c) Bitwise Exclusive OR : The bitwise exclusive OR (XOR) operator (^) sets a bit in the result's bit position if either operand (but not both) has a bit set at that position. Thus, this operator places a one in a bit position whenever the corresponding bits in the two operands differ. This is illustrated in the following table.

Expression	Hex. Value	Binary Representation
9581	0x 256 D	0010 0101 0110 1101
5347	0x 14E3	0001 0100 1110 0011
9581 ^ 5347	0x 318 E	0011 0001 1000 1110

Decimal equivalent of the result is 12686.

3.4. Bitwise Complement Operator : The bitwise complement operator (~) is a unary operator. It gives the value obtained by complementing each bit of the operand. This is illustrated as follows.

Expression	Hex. Value	Binary Representation

5347	0x 14 E3	0001 0100 1110 0011
~5347	0x EB1C	1110 1011 0001 1100

The bit-manipulation operators are frequently used for extracting data from an integer field that holds multiple pieces of information. This programming technique is known as **masking.** The operand (a constant or variable) that is used to perform marking is called the **mask.** In this process, we can extract desired bit from (or transform desired bit in) a variable by using bitwise operators.

For example, let us consider a program which reads an integer and prints the value of a specified bit in the integer. The bits are numbered from 0, starting from the right. For instance, to find the value of the second bit of an integer i, it is necessary to shift i to the right by two bits, and take the least significant digit. The program is as follows :

```
/* program for fishing the nth bit */
# include < stdio.h >
void  main ( )
{
int   i,  n;       /* i is the input integer and n is the bit
                   * position to be extracted */
   int   bit;     /* bit is the value of the bit extracted (0 or 1) */
   printf (''Input an integer :'');
   scanf (''%d'', &i);
   printf (''Bit position to be extracted:'');
   scanf (''%d'', &n);
   bit = (i >>n) & 1;
   printf(''The bit is%d'', bit);
}
```

In this program, the statement

```
bit = (i >> n) & 1;
```

first shifts i to the right by n bits and then masks (clears) all bits of i except the least significant bit, giving us the value of the bit which we wanted. Here, the operand 1 acts as a mask..

Masking is particularly useful for compressing information. For example, suppose that we have a test consisting of 32 questions having yes/no answers. Such each question has only two possible answers, we can store the answer to

each in a single bit. The answers for the entire test can be stored in a 32-bit int, as shown in the following code.

```
# include <stdio.h>
long  get_answers ( )
{
        long  answer = 0;
        int   i;
        char  c;
        for (i = 0; i < = 31; i ++)
        {
                scanf (″%c″, & c);
                if (c = = ′y’ || c = = ‘Y’)
                answers | = 1 << i;
}
printf (″Answers entered = (%lx)″, answer);
return  answer ;
}
```

Note particularly how the correct bit is set for each yes answer. With each iteration through the for loop, i is incremented, so the expression

```
1 << i
```

moves the set bit one position to the left, as shown below

value of i	**value of 1 << i**
0	0000 0000 0000 0000 0000 0000 0000 0001
1	0000 0000 0000 0000 0000 0000 0000 0010
2	0000 0000 0000 0000 0000 0000 0000 0100
3	0000 0000 0000 0000 0000 0000 0000 1000
4	0000 0000 0000 0000 0000 0000 0001 0000

……
……

when this expression is OR ed with answer, all the bits that have the answer ‘y’ or ‘Y’ are set. For example, suppose that the test answers are

ynyy nyyy nyyn ynny yyny ynnn yynn ynyn

The bit pattern of answer (with high-order bits on the left) will be as follows.

0101 0011 0001 1011 1001 0110 1110 1101

This illustrates the use of the bitwise OR to set one or more bits in an object. Having arranged the bits in answer, we need a procedure for comparing answer with the correct answers (actual answers). This is done with the help of exclusive OR operator, as follows.

```
/* Suppose that the correct answers are :
* nyyy ynny yyny nnyy ynnn nyny ynyn yynn
* 0111 1001 1101 0011 1000 0101 1010 1100
*/
# define CORRECT_ANSWERS ox79D385AC
double grade_test (answers)
long int answers ;
{
    extern int count_bits ( );
    long wrong_bits ;
    double grade;
    wrong_bits = answer ^CORRECT_ANSWERS;
    grade = 100* ((32 – count_bits (wrong_bits))/32.0);
    return grade;
 }
```

Here, the exclusive OR operator (^) compares answers with CORRECT ANSWERS and sets a bit in wrong_bits only when the operands differ. Hence, wrong_bits has bits set for each wrong answer. To find the grade, we subtract the number of wrong answers from the total to get the number of right answers. Then we divide the number of right answer by the total and finally multiply by 100. For example, in the present case, there are 11 wrong answers, the expression becomes

100 *((32.0–11)/32.0)

i.e. 100* (21.0/32.0)

which gives a grade of 66.

We still need to write a count_bits ( ) function which counts the number of bits set in wrong_answer. This function is similar to get_answers ( ), but instead of using the OR operator to set bits, we use the AND operator to read bits. This is done as follows:

```
int  count_bits (long_num)
long  int  long_num;
{
        int   i,  count = 0;
        for (i = 0; i < = 31; i ++)
        if (long_num &(1 << i))
        ++ count;
        return  count;
}
```

Now, we can invoke all these functions from the main ( ) function to form an executable program. Thus, the program becomes as follows:

```
/* grade.c : A test consisting of 32 questions having Y/N answers */
# include <stdio.h>
main ( )
{
        extern  double  grade_test ( );
        extern  long  int  get_answers ( );
        double  grade;
        printf (″Enter the answers : \n″);
        grade = grade_test (get_answers( ));
        printf (″The grade is % 3.0f\n″, grade);
        exit (0);
 }
```

We observe that the argument to grade_test ( ) is itself a function. It is functionally equivalent to

```
value = get_answers ( );
grade = grade_ test (value);
```

But in the nested version, there is no need to declare a temporary variable ‘value’.

Here, the format specifier %3.0f directs printf ( ) to output at least three digits of the result and to round the decimal digits.

If the program is named as grade.c, an execution with 5 incorrect answers, would be as follows

```
grade.c
Enter the answers :
        ynyy nyny nnyn nnnn yynn ynny ynyy yyyn
The grade is 84 .
```

It should be noted that this program works only when there are exactly 32 questions and answers. However, we can modify this program for any number of questions and answers. For more than 32 questions, we need to use an array.

3.5. Bitwise Assignment Operators : We have observed that the arithmetic assignment operators are formed with the combination of arithmetic operators and the assignment operator. Analogous to arithmetic assignment operators, we have the following bitwise assignment operators which are due to the combination of bitwise operators and the assignment operator.

Operator	**Symbol**	**Form**	**Operation**
assign left shift	<< =	a << = b	assign a << b to a
assign right shift	>> =	a >> = b	assign a >> b to a
assign bitwise AND	& =	a & = b	assign a & b to a
assign bitwise OR	| =	a | = b	assign a | b to a
assign bitwise XOR	^ =	a ^ = b	assign a ^ b to a

We have already used one such operator i.e. '1 =' in the previous example, in the statement

```
answers |  = 1 << i;
```

which is the same as

```
answers = answers |(1 << i);
```

Similarly, the statement

```
x  >> = 5;
```

is equivalent to

```
x = x >> 5,
```

In these operators, note that the bitwise operators have the higher precedency.

3.6. Cast Operator : We have already introduced this operator in the case of explicit conversion, where we forced a type conversion in a way that is

different from the automatic (implicit) conversion. The operator is as follows :

Operator	Symbol	Form	Operation
cast	(type)	(type) a	converts a to type

Here, a may be a constant, variable or an expression and type is one of the standard c data type. For example, 11/4 gives 2, whereas (float) 11/4 gives 2.75, where 11 is converted to 11.0. It should be noted that the cast operator has very high precedency.

Another use of the cast operator is to convert function arguments. For example, let us consider a program that prints the powers of 2 upto 2^{31}. The runtime library function pow ( ) will do the job, but it expects its arguments to be of type double. If our variables are integers, we need to cast them to double before we pass them as arguments, as shown below.

```
/* Program to print powers of 2 upto 2^31 */
# include <stdio.h>
# include <math.h>
main ( )
{
        int   i;
        long   j;
        for (i = 0; i < 32; i + +)
        {
                j = (int) pow (2.0, (double) i);
                printf( ″%3d \t  \t%15lu \n″, i, j );
        }
        exit (0);
 }
```

Here, if we pass i without casting it to double, the program will fail, since the pow ( ) function is expecting a double object and it interprets whatever object is passed as if it were a double. However, the ANSI standard supports a new syntax for declaring the types of arguments, called prototyping, which we shall discuss in unit-IV.

Further, the value returned by pow ( ) function is a double, so we cast it to int before assigning it to j. This cast is actually unnecessary since the compiler

automatically converts right-hand expressions of an assignment. It has been used only just for illustration.

The most frequent and important uses of cast operator involve pointers and data initialization which we shall discuss later on.

3.7. Sizeof Operator : This operator has the following syntax.

Operator	Symbol	Form	Operation
sizeof	sizeof	sizeof (t) or sizeof x	Return the size in bytes, of data type t or expression x.

The size of operator is a compile time operator. When used with an operand, it returns the number of bytes the operand occupies. The operand may be an expression or a data type. The size of operator is generally used to determine the lengths of arrays and structures when their sizes are not known to the programmer. It is also used to allocate memory space dynamically to variables during execution of a program.

When the operand is an expression, the expression itself is not evaluated, the compiler determines only what type the result would be. Any side effect in the expression, therefore, will not have an effect. The result type of the sizeof operator is either int, long, unsigned int, or unsigned long, depending on our compiler. The ANSI standard requires it to be unsigned. It is bad practice to assume the size of a particular data type since its size may vary from compiler to compiler. The only result that is guaranteed is the size of a char, which is always 1

The following program prints the sizes of the basic data types.

```
# include <stdio.h>
void  main ( )
{
        printf (′′TYPE\t\t SIZE\n\n′′);
        printf(“char \t \t % d \n”, size of (char));
        printf (“short \t \t % d \n”, size of (short));
        printf (“int \t \t % d \n”, size of (int));
        printf (“float \t \t % d \n”, size of (float));
        printf (“long \t \t % d \n”, size of (long));
        printf(“ double \t \t % d \n”, size of (double));
}
```

The output of this program when compiled on UNIX system, is as follows

TYPE	SIZE
char	1
short	2
int	4
float	4
long	4
double	8

On Turbo C, the output is

TYPE	SIZE
char	1
short	2
int	2
float	4
long	4
double	8

3.8. Conditional Operator: The C language has an unusual operator which is useful for making two-way decisions. This operator is a combination of the question mark (?) and the colon (:) and takes three operands. This operator is called the conditional operator or **ternary operator.** The syntax of this operator is as follows.

Operator	Symbol	Form	Operation
conditional	? :	a ? b : c	if a is non zero result is b; otherwise result is c.

This operator is just a shorthand for a common type of if-else branch. The if-else expression

```
if (a < b)
        c = a;
else
        c = b;
```

can be written as

```
c = ((a < b) ? a : b);
```

Here, the first operand is the test condition. It must have scalar type. The second and third operands represent the final value of the expression. Only one of them is selected, depending on the value of the first operand. The second and third operands can be of any data type, but the conversion rules apply.

The conditional operator is difficult to read and should be used with care. However, in certain situations, it is found to be handy. For example, the evaluation of the following function

$y = 2x + 3$ for $x \leq 2$

$y = 3.5\ x + 5$ for $x > 2$

can be written as

```
y = (x > 2) ? (3.5* x + 5) : (2*x+ 3) ;
```

Further, the conditional operator may be nested for evaluating more complex assignment decisions. For example, let us consider the weekly salary of a salesman who is selling some domestic products. If x is the number of products sold in a week, his weekly salary is given by

$$\text{salary} = \begin{cases} 5x + 100 & \text{for } x < 50 \\ 400 & \text{for } x = 50 \\ 5.5\ x + 150 & \text{for } x > 50 \end{cases}$$

This can be written as

```
salary (x ! = 50)? ((x < 50)? (5 * x + 100) : (5.5*x + 150)) : 400;
```

Using if-else statements, this is equivalent to

```
if (x < = 50)
        if (x < 50)
                salary = 5* x + 100;
        else
                salary = 400;
else
        salary = 5.5 * x + 150;
```

We observe that the version using the conditional operator is better since the code becomes more concise and perhaps, more efficient in that case. However, when more than one nesting is there, it is better to use if statements since in such cases, the readability of conditional operator becomes poor.

3.9. Memory Operators : There are several operators which enable us to access and dereference memory locations. Such operators are called memory operators. We have already discussed two such unary operators i.e. the address of (&) operator and the dereference (*) operator, in unit-I. All the memory operators are listed in the following table.

Operator	Symbol	Form	Operation

address of	&	& a	get the address of a
dereference	*	*a	get the value of object stored at address a
array elements	[ ]	x[3]	get the value of array element 3.
dot	.	x·y	get the value of member y in structure x.
right-arrow	→	p→y	get the value of member y in the structure pointed to by p.

The last three operators in the above table will be used when studying arrays and structures.

3.10. Example. Enclose the following expressions in parentheses the way a C compiler would evaluate them.

Solution :

Given expression	**Required form**
(i) a = b * c = = 2;	a = ((b*c) = = 2);
(ii) a = f(x) && a > 100;	a = ((f(x) && (a > 100));
(iii) a = = b && x ! = y;	(a = = b) && (x ! = y);
(iv) a = b + = 2 + f(2);	a = (b + =(2 +(f(2))));
(v) a = s·f + x·y;	a = ((s·f) + (x ·y));
(vi) a = b > > 2 + 4;	a = (b >>(2+4));
(vii) a = b && a > z ?x = y : z;	a = ((b && (a > z)) ?x = y : z);
(viii) a = * + + *p;	a = (* (+ + (*p)));
(ix) a = b^c &d;	a = (b^(c&d));
(x) a = b + + && c || d ++;	a = ((b ++) &&c) ||(d ++));

3.11 Example. Write a function called circular shift (a, n) which takes a, which is an unsigned long int, and shifts it left n positions, where the high-order bits are reintroduced as low_order bits.

Solution. The required function is

circular_shift (a, n) = (a <<n) /(a>> (32–n));

For example, if the binary representation of a is

0001 0110 0011 1010 0111 0010 1110 0101

then the call

circular_shift (a, 5)

returns a long int whose binary representation is

1100 0111 0100 1110 0101 1100 1010 0010

In this particular case, the function becomes

circular_shift (a, 5) = (a <<5)/(a >> 27);

3.12. Remark. A similar function for circular shift to the right is

circular_shift (a, n) = (a >>n)/(a << (32–n));

3.13. Example. Write a function that reads a number in binary form and converts it to hexadecimal form.

Hint. Add zeros to the left of binary form so that number of digits of binary form is divisible by 4. Read four characters at a time say a, b, c, d and then convert it into single decimal value x i.e. use

scanf (″ % 1d% 1d% 1d% 1d″, &a, &b, &c, &d);

x = 8∗ a + 4 ∗ b + 2 ∗ c + d;

Then convert x into hexadecimal form using switch statement. Repeat the process (number of binary bits)/4 times.

3.14. Example. Write a function called pack ( ) that accepts four chars and packs them into a long int.

Solution. Required function is as follows

```
long   int   pack (a, b, c, d)
char  a, b, c, d;
{
        long   int   abcd;
        abcd = ((a ∗ 256 + b) ∗ 256 +c) ∗ 256 + d;
        return   abcd;
}
```

3.15. Example. Using shift operators, determine the largest int value that your computer can store.

Hint. Use infinite loop to shift integral value 1 to the left by 1 place in each iteration. If x is the last value written before your computer shows overflow, then largest integer value represented by the computer is 2∗x–1.

UNIT-III

1. Arrays and Pointers

We have already introduced pointers as one of the scalar data types. Here, we discuss them in more detail and introduce an aggregate type called an array. Arrays and pointers are closely related in C. Together, they represent some of the most powerful features of C which account for its popularity.

When we want to store more than one value at a time in a single variable, we require an array. An array is a group of related data items that share a common name. It is a collection of identically typed variables stored contiguously (adjacently, touching each other) in memory. Each variable in an array is called an **element** and can be accessed by giving the array name alongwith an index expression called a **subscript.** Thus, an array can be considered as a subscripted variable. A subscript value 0(zero) identifies the initial element, a value 1 identifies the next element and so on.

The most basic purpose of arrays is to store large amounts of related data that share the same data type. For example, we can define an array name 'salary' to represent a set of salaries of a group of employees. A particular value is indicated by writing the subscript in brackets after the array name. Thus salary [0] represents the salary of first employee and salary [5] that of 6^{th} employee and so on.

1.1. Declaring an Array : Like any other variable, arrays must be declared before use. The general form of array declaration is

```
type  variable-name [size];
```

where 'type' specifies the type of elements that will be contained in the array, such as char, int, float etc and 'size' indicates the maximum number of elements that can be stored inside the array. For example, suppose that we want to analyze the temperature fluctuation throughout a year. If the` average temperature for each day is represented by daily_temp, then the required array for temperatures can be written as

```
int   daily_temp [365];
```

This array contains 365 integer elements. We can then enter the temperature of each day with assignment statements, such as

```
daily_temp [0] = 35;

daily_temp [1] = 42;

daily_temp [2] = 37;
```

The objects on the left side of the assignment expression are called **array element references** since they reference a single array element. We note that the highest subscript is always one less than the array's size, therefore, for this array, the last element is daily_temp [364].

It is important to note the difference between an array declaration and an array element reference. Though they look the same, they have different functions. In an array declaration, the subscript defines the size of the array whereas in an array element reference, the subscript determines which element of the array is to be accessed. Thus in the declaration statement

```
int  ar[6];
```

the subscript 6 specifies the number of elements in the array whereas in the statement

```
ar[3] = 25;
```

the subscript 3 specifies the particular element to access.

Let us write a program that gives us the average temperature for a year. To simplify the program, let us assume that we have already assigned temperature values for every element in the array daily_temp [ ]. The program is as follows.

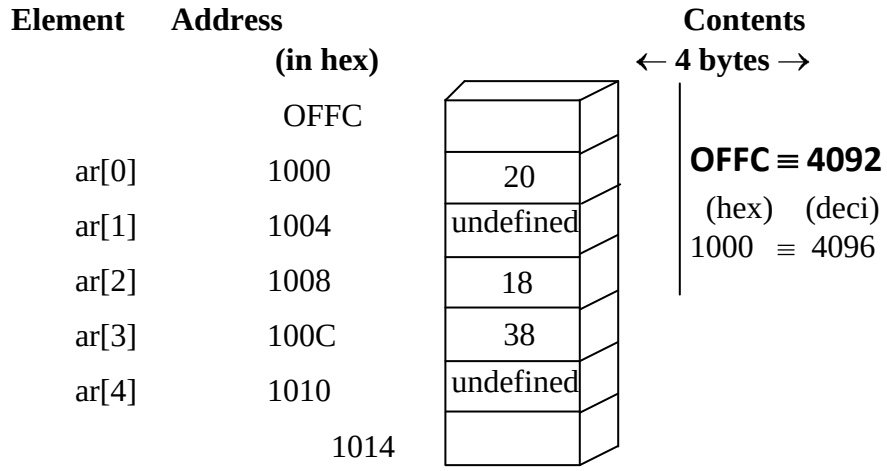
```
/* Program to calculate average temperature for the year */
# include <stdio.h>
# define DAYS_IN_YEAR 365
main ( )
{
        int   i, sum = 0;
            int   daily_temp [DAYS_IN_YEAR];
            /* Assign values to daily_temp [ ] here.*/
        for (i = 0; i < DAYS_IN_YEAR ; + + i)
      sum + = daily_temp [i];
printf (''Average temperature for the year is : %d\n'',
            sum/DAYS_IN_YEAR);
            exit (0);
}
```

It should be noted that the subscripts of an array can be integer constants, integer variables, or expressions that yield integers. C performs no bounds checking and, therefore, care should be taken to ensure that the array subscripts are within the declared limits. However, any reference to the arrays outside the declared limits would not necessarily cause an error but it gives unpredictable results.

1.2. Arrays and Memory : Here, we discuss how arrays are stored in memory. For this, let us consider an array ar, which is declared and assigned values by the following statements

```
int   ar [5]; /* declaration*/
ar[0] = 20;
ar[2] = 18;
ar[3] = ar[0] + ar[2];
```

If we assume that our machine allocates four bytes for an int, then the storage for this array is as shown below



We have shown the array starting at address 1000, but it could start anywhere in memory. Here, it should be noted that ar[1] and ar[4] have undefined values and thus their values are unpredictable. The contents of these memory locations are whatever is left over from the previous program execution. Such undefined values are often termed as **garbage** or **trash** (nonsense) and they produce most troublesome bugs because they cause different results each time the program is executed. To avoid such bugs, we can initialize arrays.

We can find the size in bytes of an array by using the size of operator. For instance, the expression

```
sizeof (ar)
```

evaluates to 20 since the array ar consists of five 4-byte ints. However, to get the size of a particular element, we use subscript such as

```
sizeof (ar[2])
```

which evaluates to 4.

1.3. Initializing Arrays : The initialization of the elements of arrays is similar to the ordinary variables when they are declared. The general form of initialization of arrays is

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

The values in the list are separated by commas. For example, the statement

static int ar[5] = {1, 0, 3·5, 5, 2};

will declare the array ar of size 5 and assign

1, 0, 3, 5, 2 to ar[0], ar[1], ar[2], ar[3], ar [4]

respectively. If the number of values in the list is less than the number of elements in the array, then only that many elements will be initialized. The remaining elements will be set to zero automatically. However, if we enter more initialization values than the number of elements in the array, the compiler reports an error. Thus, the statement

static int ar [5] = {1, 0, 3·5};

gives the initial values 1, 0, 3, 0, 0 to ar[0], ar[1], ar[2],

ar[3], ar[4] respectively.

Further, the statement

static int ar[5];

initializes all the elements of the array to zero.

When we initialize all the elements in the array, we may omit the array size. In such case, the compiler automatically finds out how many elements are in the array based on how many initial values are present. Thus, the statement

static char name [4] = {′g′, ′i′, ′t′, ′a′};

which declares the name to be array of four characters, initialized with the string "gita", can be written as

static char name [ ] = {′g′, ′i′, ′t′, ′a′};

We have used the keyword 'static' before type declaration. This declares the variable as a **static variable.** We shall discuss such variable and other class of variables, when studying storage classes, in Unit-IV.

1.4. Remarks: (i) Prior to ANSI standard, automatic arrays could not be initialized. Only external and static arrays could be initialized. However, the ANSI standard permits arrays with auto storage class to be initialized. Thus, when an ANSI compiler is used, the Keyword static can be omitted. Since ANSI compilers accept both the versions, we use the storage class static in our programs so that they can be run under both the old (K & R) and new (ANSI) compilers.

(ii) Initialization of arrays in C suffers two drawbacks. First of the two is that there is no convenient way to initialize only selected elements. Secondly, there is no shortcut method for initializing a large number of array elements like the one available in ForTran.

1.5. Encryption and Decryption : Computers are used to store private informations and so it is needed to make them secure against intruders. On large computer systems, each file has a protection status that controls who can access the file and what they can do to it. These measures, such as passwords, provide various levels of protection, but are not sufficient for total security. A more powerful security technique is to encode files. Every character is translated into a code character so that the file seems to be meaningless to someone who does not know the code.

As an illustration of encoding technique, we have the following encoding function which uses an array.

```
/* This function returns a coded value for a character*/
# define   INVALID_VAL   –1
char  encode (ch)
char  ch;
{
        static  unsigned  char  encoder [128] = [127, 123,
    119, 115,  111,    107,    103,    99,    95,    91,    87,    83,
     79,        75,    71,    67,    63,    59,    55,    51,    47,
43,
        39,    35,    31,    27,    23,    19,    15,    11,    7,
      3,
    126,122,   118,    114,    110,    106,    102,    98,    94,    90,
     86,        82,    78,    74,    70,    66,    62,    58,    54,
       50,
     46,        42,    38,    34,    30,    26,    22,    18,    14,
       10,
      6,         2,          125,    121,    117,    113,    109,
      105,    101,    97,
     93,        89,    85,    81,    77,    73,    69,    65,    61,
       57,
     53,        49,    45,    41,    37,    33,    29,    25,    21,
       17,
     13,         9,          5,          1,          124,
      120,    116,    112,    108,    104,
    100, 96,   92,    88,    84,    80,    76,    72,    68,    64,
```

```
        60,     56,     52,     48,     44,     40,     36,     32,     28,
          24,
        20,     16,     12,     8,      4,      0};
    /* test for invalid character */
    if (ch > 127)
                        return   INVALID_Val;
    else
                        return    encoder [ch]; /* returns coded character
*/
}
```

Thus, we have set up 128-element array initialized with numbers from 0 to 127, written by using a simple pattern. In actual cases, the pattern should be harder to follow which results in greater security. Each element must have a unique value. Real encoders use an algorithm to create the translation array.

Having initialized the array, we test the input argument to make sure that it is a valid character (Note that unsigned char objects have a range from 0 to 255). If ch>127, it is not a printable character so we return –1 to signify an input error. If ch ≤ 127, we use it as a subscript expression and return the element referenced by that subscript. For every value of ch satisfying 0 ≤ ch ≤ 127, there is a unique translation code. For example, if ch = 0, the function returns 127, if ch = 1, we get 123 and so on.

To see its functioning, let us consider the following program which invokes the function encode ( ).

```
# include <stdio.h>
main ( )
{
      char C[6];
            int   i;
            C[0]  = encode (′R′)
       C[1} = encode (′0′)
       C[2] = encode (′h′)
       C[3] = encode (′t′)
       C[4] = encode (′a′)
```

```
        C[5] = encode (′k′)
        for (i = 0; i < 6; + + i)
                print (′′%d \t′′, C[i]);
        exit (0);
}
```

If our computer uses the ASCII representation of characters, the output of the above (refer to the ASCII codes table on page 594-595 of the book of Peter A. Dornell) is as follows :

5 @ \ , x P

R	o	h	t	a	k							
				ASCII →	82	111	104	116	97	107		
					Position in							
				encoder →	83	112	105	117	98	108		
								Value in				
								encoder →	53			
64	92	44	120	80								
							ASCII →	5	@	\	,	x P

Thus “Rohtak” is coded as “5 @ \ , x P”.

Therefore, anyone trying to read a file which contains these encoded characters will be totally confused. The authorized reader is provided a decoder which has a reverse translation table to translate the file back to its original form.

2. Pointer Arithmetic

The C language allows arithmetic operations to be performed on pointer variables. Valid operations are

(i) Addition of an integer to a pointer
(ii) Subtraction of an integer from a pointer
(iii) Subtraction of one pointer from another of the same type.
(iv) Comparison of two pointers of the same type.

We note again that, when dereferenced, a pointer to

a char, short, int, float, long, double accesses respectively 1, 2, 4 (2 in Turbo C), 4, 4, 8 bytes of memory.
Thus, if p is a pointer, then the expression

```
p + 2
```

is valid and refers to two objects after the object that p points to. Since p holds an address, performing arithmetic on p generates a new address value. However, rather than simply adding 2 to p, the compiler multiplies 2 by the size of the object that p points to. This process is called **scaling** and the size of the data type is called the **scale factor.** For example, suppose that the address value held by p is 1000. If p is declared as a pointer to a 4-byte long int, the 2 in p+2 is multiplied by 4. Thus the value of p+2 is 1008. Similarly if p is declared as a pointer to a char, p+2 is equal to 1002. Hence in every case, p +2 always means 2 objects after p, regardless of the type of object that p points to. The subtraction of an integer from a pointer has similar effect.

If we subtract two pointers of the same type, the operation yields an integral value that represents the number of objects between the two pointers. If the first pointer represents a lower address than the second pointer, the result is negative. For example,

```
& ar[3] – & ar[1] gives 2
```

and

```
& ar[1] – & ar[3] gives –2
```

We can compare the pointers of the same type using the relational operators. The comparison can test for either equality or inequality. Thus if p and q are pointers, then the expressions p > q, p = = q, p ! = q are allowed, where p and q point to variables of same data type. Moreover, a pointer variable can be compared with zero, usually called a **null pointer.** A null pointer is any pointer assigned the integral value zero. Null pointers are particularly useful in control-flow statements since the zero-valued pointer evaluates to false, whereas all other pointer values evaluate to true. For example, the following while loop continues iterating until p is a null pointer.

```
char *p;
…….
…….
while (p)
{
  ……..
  ……../* iterates until p is a null pointer */
}
```

The use of null pointers will be observed when we discuss arrays of pointers.

2.1. Passing Pointers as Function Arguments

Generally, the compiler complains if we try to mix different types of pointers. The one exception to this rule occurs when we pass pointers as arguments. In

the absence of function prototyping (which we shall discuss in Unit-IV), the compiler does not check to make sure that the type of the actual argument is the same as the type of the formal argument. If the types are different, strange results can be obtained.

The following program illustrates what can happen it we pass a pointer to one type but declare it as a pointer to a different type on the receiving end.

```
/* Program to illustrate an important concept in C
 *  language while passing pointers as function
* arguments */
        # include <stdio.h>
        void  clr (p)
        long  * p;
        {
        *p = 0;                    /* store a zero at location p */
        }
        main ( )
        {
                static  short  ar[3] = {2, 3, 5};
                        clr (& ar[1]);      /* clear element 1 of ar [ ] */
                        printf (''ar[0] = %d\n ar[1] = % d\n ar[2] = %
d\n'',
                 ar[0], ar[1], ar[2]);
                exit (0);
}
```

Here, first we assign the values 2, 3, 5 to ar[0], ar[1], ar[2] respectively. Then we send the address of element 1 to the clr ( ) function which makes the element equal to zero. So the resulting values of ar[0], ar[1], ar[2] should be 2, 0, 5 respectively. But, actually, the output is

```
ar[0] = 2
ar[1] = 0
ar[2] = 0
```

This strange behaviour is due to the fact that we have declared pointer p in the clr ( ) function as a pointer to a long integer. When zero is assigned to the

address of p, four bytes are zeroed. Since ar[1] is a short int i.e. only two bytes long, so two extra bytes get cleared. As array elements are stored in contiguous memory locations, the two extra bytes which have been cleared, belong to ar[2]. Thus ar[2] has also been cleared.

2.2. Accessing Array Elements Through Pointers

We have already observed that one way to access array elements is to enter the array name followed by a subscript i.e. using array indexing. Another way is through pointers. The pointer accessing method is much faster than array indexing. We note that the declarations

```
short  ar[3];

short  *p;
```

create an array of three variables of type short, which are ar[0], ar[1], ar[2], and a variable named p that is a pointer to a short. Using the "address of" operator (&), we can now make the assignment

```
p = & ar[0];
```

which assigns the address of array element 0 to p. By dereferencing p as

```
* p
```

we get the value of element ar [0]

The expressions ar [0] and *p refer to the same memory location until the value of p is changed. Further, due to scaled nature of pointer arithmetic, the expression

```
*(p+2)
```

refers to the same memory contents as

```
ar[2]
```

Infact, for any integer expression e,

```
* (p +e)
```

is the same as

```
ar[e]
```

This provides an important relationship between arrays and pointers. Thus, adding an integer to a pointer that points to the beginning of an array, and then dereferencing that expression, is the same as using the integer as a subscript value to the array.

Another important relationship is that an array name that is not followed by a subscript is interpreted as a pointer to the initial element of the array. Thus, the expression

```
ar
```

and

& ar[0]

are exactly the same.

Combining these two relationships, we conclude that

ar[n]

is the same as

∗(ar+n)

this relationship is unique to the C language and is one of its most important features. When the C compiler sees an array name, it translates it into a pointer to the initial element (base element) of the array. Then the compiler interprets the subscript as an offset from the base address position. For example, the compiler interprets the expression ar[2] as a pointer to the base element of ar, plus an offset of 2 elements. The offset of 2 means skip two elements. Thus the two expressions.

ar[2]

and

∗(ar+2)

are equivalent. In both cases, ar is a pointer to the initial element of the array and 2 is an offset that tells the compiler to add two to the pointer value.

Due to the above relationship, pointer variables and array names can be used interchangeably to reference array elements. Here, it should be noted that the values of pointer variables can be changed whereas array names cannot be changed. This is because an array name by itself is not a variable, it refers to the address of the array variable and we are aware of the fact that we cannot change the address of variables. From this we conclude that a naked array name (without a subscript or indirection operator) cannot appear on the left-hand side of an assignment statement. This important difference between pointers and arrays will be observed when we describe character strings.

The following program finds out the smallest in an array of n elements using pointers.

```
# include <stdio.h>
main ( )
{
        int   i, n, small, ∗ ptr, a[50] ;
        printf (′′Enter the size of the array′′);
        scanf (′′%d′′, &n);
```

```
        printf(''\n Enter array elements \n'');
        for (i = 0; i < n; i + +)
                scanf (''%d'', & a [i]);
        ptr = a;                    /* ar ptr = & a[0]; */
        small = * ptr;    /* assigns contents of a[0] to small */
        ptr + +;                    /* pointer points to next element
                * in the array i.e. a[1]*/
        /* loop n–1 times to search for smallest element
         * in the array */
            for (i = 1; i < n; i + +)
            {
                if(small > * ptr)
                small = *ptr;
                ptr + +;  /* pointer is incremented to a[i + 1] */
            }
            printf (''\n smallest element is % 6d\n'', small);
    }
```

The output of this program is as follows:

```
Enter the size of the array 7
Enter array elements
10  –17  0  23  5  –42 34
Smallest element is –42
```

2.3. Passing Arrays as Function Arguments : When a function requires to manipulate an entire array, it is necessary to pass the entire array to the function during a single call rather than passing one element at a time. This can be achieved through the fact that the name of the array points to the base address of the array. To pass an array to a called function, it is sufficient to list the name of the array, without any subscripts, and the size of the array as arguments. For example, the call

```
largest (ar, n);
```

will pass all the elements contained in the array are of size n. The called function expecting this call must be appropriately defined. The largest ( ) function header may of the form :

```
float  largest (array, size)
float  array [ ];
int   size;
```

The function largest ( ) is defined to take two arguments, i.e., the array name and the size of the array to specify the number of elements in the array. The declaration of the formal argument 'array' is made as follows.

```
float  array [ ]; /* or float * array; */
```

Here, the pair of brackets informs the compiler that the argument 'array' is an array of numbers. It is not necessary to specify the size of the 'array' here.

To illustrate the concept, let us consider the following program to find the largest value in an array of elements.

```
main ( )
{
        float   largest ( );
        static  float  value [5] = {1.5, –3.75, 0.8, 4.89, 2.36};
        printf (″%f\n″, largest (value, 5));
}
float   largest (ar, n)
float   ar [ ];
int   n;
{
        int   i;
        float   max;
        max = ar[0];
        for (i = 1; i < n; i + + )
        if (max < ar [i])
                        max = ar[i];
        return (max);
}
```

In this program, when the function call largest (value, 5) is made by the main ( ) function, the values of all elements of the array 'value' are passed to the corresponding elements of the array ar in the called function. The 'largest'

function finds the largest value in the array and returns the result to the main ( ) function.

We should remember one major distinction when dealing with array arguments. If a function changes the values of an array elements, then these changes will be made to the original array that passed to the function. When an entire array is passed as an argument, the contents of the array are not copied into the formal parameter array, instead, information about the addresses of array elements are passed on to the function. Therefore, any changes introduced to the array elements are truly reflected in the original array in the calling function. However, this does not apply when an individual element is passed on as argument.

To further illustrate the concept of passing arrays as function arguments, let us write a program to calculate the standard deviation of an array of values. We know that S. D. of a set of n values is given by

$$\text{S. D.} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\bar{x} - x_i)^2}, \text{ where } \bar{x} \text{ is the mean of the values.}$$

In our program, we have three functions. The function main ( ) reads the elements of the array value [ ] from the terminal and calls the function std_dev ( ) to print the standard deviation of the array elements, std_dev ( ), in turn, calls another function mean ( ) to supply the average value of the array elements. The program is as follows.

```
/* Program to calculate S.D. */
# include <stdio.h>
# include <math.h>
# define   SIZE   10
main ( )
{
        float   value [SIZE], std_dev ( );
        int   i;
        printf (''Enter % d float values\n'', SIZE);
        for (i = 0; i < SIZE; i + +)
                scanf (''%f'', & value [i]);
        printf (''S.D. is % f\n'', std_dev (value, SIZE));
}
float  std_dev (ar, n)
```

```
float  ar [ ];
int   n;
{
        int   i;
        float   mean ( ), x, sum = 0. 0;
        x = mean (ar, n);
                for (i = 0; i < n; i + +)
                                sum + =(x–ar[i])∗ (x–ar[i]);
                return (sqrt(sum/(float)n));
}
float  mean (ar, n)
float  ar[ ];
int   n;
{
        int   i;
        float  sum = 0.0;
        for (i = 0; i < n; i + + )
                                sum = sum + ar[i];
        return(sum/(float)n);
}
```

The output is of the form

```
Enter 10 float values
……
S. D. is…
```

2.4. Sorting Algorithms : One of the common programming operation and a classic application of arrays is sorting a list of objects into alphabetical or numerical order. There are many sorting algorithms, and the mathematical analysis for deciding which one is most efficient is a subject of long discussion. Here, we discuss one of the simpler algorithms, called a **bubble sort**, which sorts the array in ascending (descending) order. In a bubble sort, we compare adjacent elements, starting with the first two, and interchange them if the first is larger than the second. After comparing the first two

elements, we compare the second and third, then third and fourth and so on until me reach the end of the array. Comparing all the adjacent pairs is termed as a **pass.** If in the first pass, we need to interchange any of the pairs, we need to make another pass. We keep making passes until the array is in sorted order. To see the actual process, we have added a few printf ( ) statements that show the current status of the array before each pass. The process is as follows :

```
/* sorting of an array of ints in ascending order
* using the bubble sort algorithm */
# define   TRUE    1
# define   FALSE   0
# include <stdio.h>
void   bubble_sort(list, list_size)
int   list [ ], list_size;
{
        int   i, j, temp, sorted = FALSE;
        while (! sorted)
        {
                sorted = TRUE;   /* Suppose list is sorted */
                /* print loop-not part of bubble sort algorithm */
                for (j = 0; j < list_size ; j + + )
                        printf(''%d\t'', list [j]);
                printf(''\n'');
                /* End of printf loop */
                for (i = 0; i < list_size –1; i + +)
                {
                        if (list [i] > list [i + 1])
                        {
                                /* At least one element is out of order */
```

```
                                                    sorted = FALSE;
                                                    temp = list [i];
                                                    list [i] = list [i +1];
                                                    list [i +1] = temp;
                                                }
                                            } /* end of for loop */
                                        } /* end of while loop */
                                    }
```

The bubble_sort ( ) function with 10–element array (say), is called by the main ( ) function as shown in the following program.

```
main ( )
{
    int i;
    static int list [ ] = {7, 53, 26, 15, 110, 66, 87, 42, 81, 5};
    bubble_sort (list, size of (list)/sizeof (list [0]));
        exit (0);
}
```

Note have we pass the number of elements in the array using the 'sizeof ' operator. We have obtained the number of elements in the array by dividing the size of the array by the size of each element. This is a useful technique in C because it is portable. We can add new elements to the array, and the size of the array elements can vary, but we never need to change the function call.

The bubble sort is a simple algorithm that illustrates array manipulation but it is not very efficient as it needs temporary space for exchanging data, and has excessive data movement, especially if the size of the data is large. Pointer can be used to perform the same task more efficiently since in that case, instead of data exchange, pointers are exchanged. The standard runtime library contains a much more efficient sorting function called qsort ( ), which we shall discuss in Unit-IV.

The output of the above program is as follows:

```
        7       53      26      15      110     66      87      42      81
    5
7       26      15      53      66      87      42      81      5       110
7       15      26      53      66      42      81      5       87      110
```

```
7    15   26   53   42   66   5    81   87   110
7    15   26   42   53   5    66   81   87   110
7    15   26   42   5    53   66   81   87   110
7    15   26   5    42   53   66   81   87   110
7    15   5    26   42   53   66   81   87   110
7    5    15   26   42   53   66   81   87   110
5    7    15   26   42   53   66   81   87   110 .
```

3. Strings

One of the most common uses of arrays is to store strings of characters. A string is a sequence of characters (an array of characters) terminated by a **null character**. A null character is a character with a numeric value of zero. In C, it is represented by the escape sequence '\0'. A string constant, sometimes called a string literal, is any series of characters enclosed in double quotes. It has a data type of **array of char,** and each character in the string takes up one byte. Further, the compiler automatically supplies null character at the end of the string.

Character strings are commonly used to build meaningful and readable programs. The common operations performed on character strings are :

(i) Reading and writing strings

(ii) Combining strings together

(iii) Copying one string to another

(iv) Comparing strings for equality

(v) Extracting a portion of a string

We shall discuss these operations in detail and develop programs that involve these operations.

3.1. Declaring and Initializing Strings: A string variable is any valid C variable name and is always declared as an array of type char. The general form of declaration of a string variable is

```
char string_name [size];
```

The 'size' determines the number of characters in the 'string name'. For example

```
char days [20];
char name [50];
```

The array is one element longer than the number of characters in the string to accommodate the trailing null character. Thus, 'size' should be equal to the maximum number of characters in the string plus one.

Character arrays may be initialized when they are declared. C permits a character array to be initialized in either of the following two forms

```
static char city [10] = ″New Delhi″
```

static char city [10] = {′N′, ′e′, ′w′, ′ ′, ′D′, ′e′, ′l′, ′h′, ′i′, ′\0′};

Note that when we initialize a character array by listing its elements, we must supply explicitly the null terminator. We prefer the first form of initialization. When we specify an array size, we must allocate enough characters to hold the string. For example, in the initialization,

static char str [20] = ′′Mathematics′′;

the first 12 elements are initialized and the remaining 8 elements receive the default initial value zero. We may also initialize a character array without specifying the number of elements. Thus, in the initialization

static char str [ ] = ′′some text′′;

str [ ] is ten characters in length.

In the above declarations, we may omit the word static when we use ANSI compilers.

3.2. Reading Strings from Terminal: (i) Reading Words : The scanf( ) function can be used with %s format specification to read in a string of characters. Thus, we can use

```
char name[20];
scanf(′′%s′′, name);
```

The problem with the scanf( ) function is that it terminates its input on the first white space it finds (A white space may be blanks, tabs, carriage return, form feed and new lines). Thus, if we type

New Delhi

then only the string "New" will be read into the array address since the string is terminated due to blank space after 'New'.

It should be noted that unlike previous scanf ( ) calls, in the case of character arrays, the ampersand (&) is not required before the variable 'name'. The scanf ( ) function automatically terminates the string that is read with a null character and so the character array should be large enough to hold the input string plus the null character.

To read the entire line "New Delhi", we may use two character arrays of appropriate sizes. Thus, the statement

scanf(′′%s %s, adr1, adr2);

with the line of text

New Delhi

will assign the string "New" to adr1 and "Delhi" to adr2.

It should be noted that the string 'left hand' is treated as two words while the string 'left-hand' as one word.

The following program reads a series of words from a terminal using scanf ( ) function.

```
# include < stdio.h>
main ( )
{
        char  word1[50], word2[50], word3[50], word4[50];
        printf (′′Enter text : \n′′);
        scanf(′′%s %s′′, word1, word2);
        scanf(′′%s %s′′, word 3, word 4);
        printf(′′\n′′);
        printf(′′word1 = % s\n word 2 = %s\n′′, word1, word2);
         printf(′′word 3 = % s\n word4 = % s\n′′, word3, word4);
}
```

Output is as follows :

```
Enter text :
Rohtak Road, New Delhi
word 1 = Rohtak
word 2 = Road,
word 3 = New
word 4 = Delhi
Enter text:
Rohtak–Road, New Delhi, India
word 1 = Rohtak–Road,
word 2 = New
word 3 = Delhi
word 4 = India
```

(ii) Reading a Line of Text :- In many text processing applications, we need to read an entire line of text from the terminal. It is not possible to use scanf ( ) to read a line since it terminates reading as soon as a space is encountered in input. The function getchar( ) which we have already used to read a single character, can be used repeatedly to read successive single characters from the input and place them into a character array. Thus, an entire line of text can be read and stored in an array. The reading is terminated when the newline character('\n') is entered and the null character is then inserted at the end of the string.

Let us consider the following program to read a line of text containing a series of words from the terminal

```
# include <stdio.h>
main ( )
{
        char  line [81], character;
        int   i;
        i = 0;
        printf (′′Enter text Press <Return>at end \n′′);
        do
        {
                character = getchar ( );

                line [i] = character;
                i + + ;
        }
        while (character ! = ′\n′);
        i = i – 1;
        line [i] = ′\0′;
        printf(′′\n %s\n′′, line);
}
```

Output is as follows :

```
Enter text. Press < Return > at end
Maharshi Dayanand University, Rohtak
Maharshi Dayanand University, Rohtak
```

The above program reads a line of text (upto a maximum of 80 characters) into the string ‘line’. Every time a character is read, it is assigned to its location in the string ‘line’ and then tested for newline character. When a newline character is read (indicating the end of line), the reading loop is terminated and the newline character is replaced by the null character to indicate the end of the character string.

When the loop is exited, the value of the index i is one number higher than the last character position in the string, since it has been incremented after

assigning the new character to the string. Therefore, the index value i–1 gives the position where the null character is to be stored.

C does not provide operators that work on strings directly. For instance, we cannot assign one string to another directly. Thus, the assignment statements

```
string = ″abc″;
string1 = string2;
```

are not valid. If we want to copy the characters in string 2 into string1, we may do so on a character-by-character basis.

The following program copies one string to another and counts the number of characters copied.

```
/* Program for copying one string into another */
# include <stdio.h>
main ( )
{
    char   str1[80], str2[80];
    int   i;
    printf (″Enter a string: \n″);
    printf (″?″);
    scanf (″%s″, str 2);
    for (i = 0; str 2[i]! = ′\0′ ; i + +)
        str1[i] = str2[i];
        str1[i] = ′\0′;
        printf (″\n″);
        printf (″%s\n″, str 1);
    printf (″Number of characters = %d\n″, i);
}
```

Output is as follows :

```
Enter a string:
? Amritsar
Amritsar
Number of characters = 8
```

Enter a string:

? Bhattacharya

Bhattacharya

Number of characters = 12

In the above program, we have used for loop to copy the characters contained in str2 into str1. The loop is terminated when the null character is met.

3.3. Writing String to Screen : We observe that we use printf ( ) function with %s format to print strings to the screen. The %s format can be used to display an array of characters that is terminated by the null characters. Thus the statement

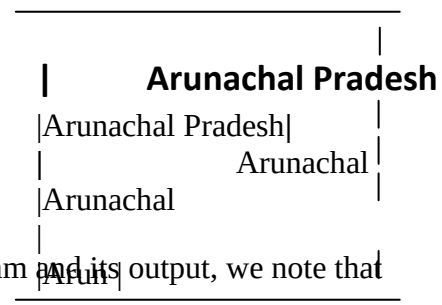
printf(″%s″, name);

displays the entire contents of the array 'name'. We can also specify the precision with which the array is displayed. For example, the specification %10.5s indicates that the first five characters are to be printed in a field width of 10 columns. Also, if we include the minus sign in the specification (e.g. %–10.5s), the string will be printed left justified.

Let us consider the following program which stores the string "Arunachal Pradesh" in the array 'state' and displays the string under various format specifications.

```
#include <stdio.h>
main ( )
{
    static  char  state [20] = ″Arunachal Pradesh″;
    printf(″\n\n″)'
    printf(″————————————\n″);
    printf(″|%20s|\n″, state);
    printf(″|% 10s|\n″, state);
    printf(″|% 20.9s|\n″, state);
    printf (″|% –20.9s | \n″, state);
    printf(″|%20.0s|\n″, state);
    printf(″| %.4s|\n″, state);
   printf(″|%s|\n″, state);
   printf(″----------\n″);
}
```

Output is as follows:



From the program and its output, we note that

(i) when the field width is less than the length of the string, the entire string is printed.

(ii) The integer value after the decimal point specifies the number of characters to be printed. Thus, when zero occurs after decimal point, nothing is printed.

(iii) The minus sign in the specification causes the string to be printed left-justified.

3.4.Strings Versus Chars :- It is important to observe the difference between string constants and character constants. In the following two declarations, one byte is allocated for ch but two bytes are allocated for the string "a" (one extra byte for the terminating null character), plus additional memory is allocated for the pointer ps.

```
char  ch = ′a′;

char  ∗ps = ′′a′′;
```

Note that an implementation-defined number of bytes are allocated for the pointer ps.

It is legal to assign a character constant through a dereferenced pointer, such as

```
∗p = ′a′;
```

But it is illegal to assign a string to a dereferenced char pointer, such as

```
∗ p = ′′a′′;     /∗ illegal∗/
```

Since a string is interpreted as a pointer to a char and a dereferenced pointer has the type of the object that it points to, this last assignment attempts to assign a pointer value to a char variable. This is illegal.

By the same reasoning, it is legal to assign a string to a pointer (without dereferencing it), but it is illegal to assign a character constant to a pointer, such as

```
p = ′′a′′;

p = ′a′;    /∗ illegal ∗/
```

The last assignment attempts to assign a char value to a pointer variable. This is illegal. Further, the crucial observation to be made is that initializations and assignments are not symmetrical. Thus we can write

```
char *p = ″string″;
```

but not

```
*p = ″string″;
```

Note that this is true for assignments and initializations of all data types, not just character arrays. For example,

```
float f;
float *pf = &f ;    /* legal */
*pf = &f;    /*illegal*/
```

3.5. Combining Strings Together :

We have observed that we cannot assign one string to another directly. Similarly, we cannot join two strings together by the simple arithmetic addition. Thus, the statements such as

```
str 3 = str1 + str2;
str2 = str1 + ″hello″;
```

are not valid. The characters from str1 and str2 should be copied into str 3 one after the other. The size of the array str3 should be large enough to hold the total characters. The process of combining two strings together is called **concatenation.**

Let us consider a program in which we concatenate three strings into one string. In this program, the names of students of a class are stored in three arrays, namely first_name, second_name and last_name. These three parts are concatenated into one string to be called name. The program is as follows:

```
# include <stdio.h>
main ( )
{
    int   i, j, k;
    static   char  first_name [10] = {″VISWANATH″};
    static  char  second_name [10] = {″PRATAP″};
    static  char  last_name [10] = {″SINGH″};
    char name [30];
    /* copy first_name into name */
    for (i = 0; first_name [i] = ′\0′; i + + )
        name [i] = first_name [i];
    name [i] = ′  ′; /*End first_name with a space */
    /*copy second_name into name */
```

```
        for(j = 0; second_name [j]! = '10'; j + +)
                name [i + j + 1] = second_name [j];
        name [i + j + 1] = '  '; /* End second_name with a space */
        /* copy last_name into name */
        for (k = 0; last_name [k] ! = '\0'; k + +)
                name [i + j + k + 2] = last_name [k];
        name [i + j + k + 2] = '\0'; /* End name with a null char */
        printf(''\n\n'');
        printf(''%s\n'', name);
}
```

Output is as follows:

VISWANATH PRATAP SINGH

In the above programme, we have used three for loops. In the first loop, the characters contained in the first_name are copied into the variable 'name' until the null character is reached. The null character is not copied, instead it is replaced by a space. Similarly, the second_name is copied into 'name', starting from the column just after the space, and the same process for last_name. At the end, we place a null character to terminate the concatenated string 'name'.

3.6. Remarks : (i) The ANSI standard states that two adjacent string literals will be concatenated into a single null-terminated string. For example, the statement

```
printf(''Mharshi''  ''Dayanand'' ''University\n'');
```

is treated as if it had been written

```
printf(''Maharshi Dayanand University \n'');
```

Note that the terminating null characters of the strings are not included in the concatenated string. This feature is particularly useful with regards to macros that expand to string literals, which we shall describe in Unit-V.

(ii) string concatenation can be used to break up long strings that would otherwise require the continuation character. For example, the statement

```
printf(''This is a very long string that\
cannot be put in one line\n'');
```

can be written as

```
printf(''This is a very long string that''
```

```
″cannot be put in one line\n″);
```

The second version i.e. string concatenation (combined with the fact that the compiler ignores the spaces between the items) gives us greater formatting flexibility.

3.7. Comparison of Two Strings: C does not permit the comparison of two strings directly. Thus the statements, such as

```
if(str1 = = str2)

if(str2 = = ″ABC″)
```

are not valid. It is therefore necessary to compare the two strings to be tested, character by character. The comparison is done until there is a mismatch or one of the strings terminates into a null character, whichever occurs first. The following segment of a program illustrates the comparison of str1 and str2.

```
……
……
i = 0;
while (str 1[i] = = str2[i] && str1[i] ! = '\0' && str2[i] ! = '\0')
        i = i + 1;
if(str1[i] = = ′\0′ && str2[i] = = ′\0′)
        printf(″strings are equal\n″);
else
        printf(″strings are not equal\n″);
…….
…….
```

4. String-Handling Functions

In addition to printf( )and scanf( ), the C runtime library contains a large number of functions that manipulate strings. Out of these string-handling functions, the most commonly used are the following :

Function	**Operation**
strcat( )	concatenates two strings
strcmp( )	compares two strings
strcpy( )	copies one string over another
strlen( )	finds the length of a string

We discuss briefly how these functions can be used in the processing of strings, which illustrates some of the concepts behind arrays and pointers. To

use these functions, the header file string.h must be included in the program with the statement

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

4.1. The String Length Function : The simplest string function is strlen( ), which returns the number of characters in a string, not including the trailing null character. Using arrays, strlen( ) can be written as

```
int   strlen (str)
char  str [ ];
{
        int   i = 0;

    while (str [i])
    + + i;
    return  i;
}
```

We test each element, one by one, until we reach the null character. If str[i] is the null character, it will have a value of zero, making the while condition false. Any other value of str[i] makes the while condition true. When null character is reached, the while loop terminates and the function returns i, which is the last subscript value and hence the length of the string. The above function can be written by using a for loop instead of a while loop, as

```
int   strlen (str)
char  str[ ];
{
        int   i;
        for (i = 0; str[i]; + + i)
            ;      /*Null statement*/
        return  i;
}
```

The pointer version of strlen ( ) is as follows:

```
int   strlen (str)
char  * str;
{
```

```
        int  i ;
        for (i = 0; * str + +; i + +)
            ;   /* Null statement */
        return  i;
    }
```

Here, the expression

```
* str + +
```

illustrates a common idiom in C. Since the + + operator has the same precedence as the * operator, associativity rules come into effect. Both operators bind from right to left, so the expression tells the compiler to

(i) evaluate the post increment operator. Since + + is a post-increment operator, the compiler passes str to the next operator but makes a note to increment str after the entire expression is complete.
(ii) Evaluate the indirection (*) operator, applied to str.
(iii) Complete the expression by incrementing str.

4.2. Remark : In the above discussion, we have explained the working of strlen ( ) function. When we include the string.h header file, it works automatically and provides the result. For example,

```
char str [ ] = ″Rohtak″
printf(″%d\n″, strlen (str)) ;
```

will output 6.

4.3. Strcat ( ) Function : This function joins two strings together. It has the form

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

where str1 and str2 are character arrays i.e. strings. When the function strcat ( ) is executed, str2 is appended to str1. It does so by removing the null character at the end of str1 and placing str2 from there. Here, we must make sure that the size of str1, to which str2 is appended, is large enough to accommodate the final string.

The strcat ( ) function may also be used to append a string constant to a string variable. Thus

```
strcat (feel, ″GOOD″);
```

is valid, where ‘feel’ is a string variable. C permits nesting of strcat ( ) functions. Thus, the statement

```
strcat (strcat(str1, str2), str3);
```

is allowed which concatenates the three strings. The resultant string is stored in str1, so it should be of enough large size.

4.4. Strcmp ( ) function : This function has the form

strcmp (str1, str2);

where str1 and str2 may be string variables or string constants. The strcmp ( ) function compares two strings and has a value 0 if they are equal. If they are not equal, it has a numeric difference between the first nonmatching characters in the strings. Our major concern is to determine whether the strings are equal; if not, which is alphabetically above. The value of the mismatch is rarely important. For example, the statement

strcmp(″their″, ″there″);

will return a value –9 which is the numeric difference between ASCII codes of i and r. i.e. i minus r in ASCII code is –9. If the value is negative, str1 is alphabetically above str 2. More examples of strcmp ( ) are

strcmp (name1, name2);

strcmp(name1, ″Mohan″);

strcmp(″Ram″, ″Rom″);

4.5. Strcpy ( ) Function :

This function works just like a string-assignment operator. It has the form

strcpy (str1, str2);

which assigns the contents of str 2 to str1. Here, str2 may be a character array variable or a string constant. For example, the statement

strcpy (city, ″ROHTAK″);

will assign the string "ROHTAK" to the string variable 'city'.

Again, we note that the size of str1 should be large enough to receive the contents of str 2.

4.6. Remark : We have already observed that the string-handling functions work on character by character basis. Thus, for example, the strcpy ( ) function in the actual form, can be put as

```
void  strcpy (str1, str2)
char  str1 [  ], str2 [  ];
{
    int   i;
    for (i = 0; str 2[i]; + + i)
```

```
        str 1 [i] = str 2[i];
    str1[+ + i] = '\0';
}
```

Here, we explicitly append a null character because the loop ends before the terminating null character is copied. Also note that we should use the prefix increment operator. Using pointers, we can rewrite the above function as

```
void  strcpy (str1, str2)
char  * str1, * str 2;
{
    int   i;
    for (i = 0; *(str 2 + i); + + i)
        *(str1 + i) = *(str 2 + i);
    str1 [+ + i] = '\0';
}
```

A more superior version that runs faster on most computers is as follows.

```
void  strcpy (str1, str2)
char  * str1, * str2;
{
    while(*str1 + + = * str 2 + +)
        ;   /* Null statement */
}
```

This version utilizes just about all the shortcuts that C provides. Here, instead of adding an offset to the string pointer, we have just incremented it with the post increment operator. The result of the assignment is used as the test condition for the while loop. We note that even an assignment expression has a value. If * str1 equals zero (which it will on the terminating null character), the entire assignment expression will equal zero and the while loop will end. By using the assignment expression as a test condition, no extra statement is needed to assign the terminating null character. Further note that we have used a post fix increment operator instead of a prefix operator, since if we do otherwise (i.e. if we use *(+ + str2), the function will not work because it would always skip the initial element.

4.7. Pattern Matching : Here, we discuss a pattern matching function which is not a part of many C libraries, but is very common and useful. In ANSI C runtime library, this function is called strstr ( ) but we call it pat_match (

).This function accepts two arguments, both pointer to character strings. It then searches the first string for an occurrence of the second string. If it is successful, it returns the byte position of the occurrence, if it is unsuccessful, it returns –1. For example if the first string is "Everybody complains about the system but nobody ever does anything to improve it" and the second string is "the system", the function would return 26 because "the system" starts at element 26 of the first string. The function is as follows.

```
 int  pat_match (str1, str2)
char  str1 [ ], str2 [ ];
{
        int   i, j ;
        for (i = 0; i < strlen(str1); + + i)
        {
                for (j = 0;(j < strlen (str2) && (str 2[j]= =
                                        str1 [i + j])); j + +)
                        if (j = = strlen (str 2))
                                return  i;
         }
         return –1;
    }
```

We have two loops, one nested within the other. The outer loop increments i until it reaches the end of str1. The inner loop compares the current character in str1 with the first character in str2. If they are equal, it tests the next character in each string. The loop ends either when the characters in the two strings no longer match or when there are no more characters in str2. If the loop ends because there are no characters left, the strings match and we return i, which is the byte position in str1. If the loop ends because the strings do not match, we jump back to the outer loop and test the next character in str1. If we reach the end of str1 without a match, we return –1.

Here, the value –1 is a convenient failure indicator. In most C library functions, –1 or 0 is used as a failure signal. In the present case, we cannot use 0 for failure because 0 will be returned if the pattern match is successful on the initial element of str 2. Thus, if the pattern match is successful, a non-negative number will be returned.

In the above form, the pat_match function calls strlen ( ) with each iteration of the for loop. This is a waste of computer cycles since the string length never changes. We can remove this drawback by storing the string length in a variable, as follows.

```
pat_match (str1, str2)
char  str1 [ ], str 2 [ ];
{
        int   i, j;
        int   length1 = strlen (str1);
        int   length 2 = strlen (str 2);
        for (i = 0; i <length 1; + +i)
        {
                for (j = 0; j < length2; j + + )
                        if(str2[j] ! = str1[i + j])
                                break;
                if(j = = length 2)
                        return  i;
        }
        return –1;
}
```

Even more efficient version of this function, using pointers instead of arrays, is shown below.

```
pat_match(str1, str2)
char  * str1, *str 2;
{
      char  *p, *q, *substr;
      /* Iterate for each char position in str1 */
      for (substr = str1; * substr; substr + + )
     {
            p = substr;
            q = str 2;
             /* check if str 2 matches at this char position */
             while (*q)
                    if(*q + +! = * p + + )
                            goto no_match;
            /* When every char in str2 matched, then return
```

```
                    * the number of chars between the original start
                    * of str1 and the current char position by
                    * using pointer subtraction
                    */
                        return  substr-str1;
                    /* When while loop could not match str2 */
                        no_match :  ; /*Null statement, since a
                                    * label prefixes a statement */
                }
                return –1;
            }
```

The following program reads in a string and a pattern to be matched and then uses the pat_match function.

```
# include <stdio.h>
main ( )
{
    char   first_string [100], pattern [100];
    int   pos;
    printf(″Enter str:″);
    gets (first_string);
    printf (″Enter pattern to be matched:″);
    gets(pattern);
    pos = pat_match(first_string, pattern);
    if(pos = = –1)
        printf(″Pattern not matched.\n″);
    else
        printf(″Pattern matched at\ position %d\n″, pos);
   exit (0);
}
```

Execution of the program would be as follows:

```
Enter str: M. D. University is a good university.
Enter pattern to be matched : university
Pattern matched at position 27
```

In the above program, we cannot use scanf ( ) and %s because scanf ( ) stops assigning characters to the array as soon as a space character is encountered. If the first_string or pattern contains a space, the program would not work. Therefore, we have used another runtime library function gets ( ) which reads a string from terminal (including spaces) and assigns the string to a character array. The gets ( ) function takes one argument, which is a pointer to the character array. It reads characters from the terminal until a linefeed or end-of-line is encountered. Further, we should make sure that our character array is large enough to hold the longest possible input string.

Common string-handling functions in the standard library are listed below.

Function	**Operation**
strcpy( )	Copies a string to an array
strncpy( )	Copies a portion of a string to an array
strcat( )	Appends one string to another
strcmp( )	Compares two strings
strncmp( )	Compares two strings upto a specified number of characters
strchr( )	Finds the first occurrence of a specified character in a string
strcoll( )	Compares two strings based on an implementation-defined collating sequence
strcspn( )	Computes the length of a string that does not contain specified characters
strerror( )	Maps an error number with a textual error message
strlen( )	Computes the length of a string
strpbrk( )	Finds the first occurrence of any specified characters in a string
strrchr( )	Finds the last occurrence of any specified characters in a string
strspn ( )	Computes the length of a string that contains only specified characters
strstr( )	Finds the first occurrence of one string embedded in another
strtok( )	Breaks a string into a sequence of tokens.
strxfrm( )	Transforms a string so that it is a suitable as an argument to strcmp( )

5. Multidimensional Arrays

So far we have discussed the array variables that can store a list of values. There are situations where we need to store and manipulate multidimensional data structures. Most frequent cases are of matrices and tables, which are two-dimensional arrays.

A multidimensional array is an **array of arrays** and is declared with consecutive pair of brackets. The general form of a multidimensional array is

```
type  array_name [s1] [s2] [s3]…[sn];
```

where si is the size of the ith dimension. Thus, the statement

```
int   x[3] [5];   /* two dimensional array */
```

declares x as a 3-element array of 5-element arrays.

Similarly,

```
char  x[2] [3] [5];  /* three dimensional array */
```

declares x as a 2-element array of 3-element arrays of 5-element arrays.

Although a multidimensional array is stored as a one-dimensional sequence of elements, we can treat it as an array of arrays. For example, let us consider the following 5×5 magic square, which is termed so because the rows, columns, and diagonals all have the same sum.

17	24	1	8	15
23	5	7	14	16
4	6	13	20	22
10	12	19	21	3
11	18	25	2	9

For storing of this square in an array, we make the following declaration

```
static  int   magic [5] [5] = { { 17, 24, 1, 8, 15},
                                {23, 5, 7, 14, 16},
                                {4, 6, 13, 20, 22},
                                {10, 12, 19, 21, 3},
                                {11, 18, 25,  2,  9}
                              };
```

Note that each row of values is enclosed by braces. To access an element in a multidimensional array, we specify as many subscripts as are necessary. For example, the elements of a two dimensional array ar can be accessed by the expression

```
ar [i] [j]
```

where i and j refer to the row number and column number respectively. Multidimensional arrays are stored in row-major order, which means that the last subscript varies fastest. For example, the array declared as

```
int   ar [2] [3] = { {1, 3, 5},
                     {2, 4, 6}
                   };
```

is stored as shown below

Element	**Address (in hex)**	**Contents ← 4 bytes →**
	OFFC	
ar[0][0]	1000	1
ar[0][1]	1004	3
ar[0][2]	1008	5
ar[1][0]	100C	2
ar[1][1]	1010	4
ar[1][2]	1014	6
	1018	

Further, the array reference

```
ar[1][2]
```

is interpreted as

```
*(ar[1] +2)
```

which can be further expressed as

```
*(*(ar+1)+2)
```

we observe that ar is an array of arrays. When *(ar+1) is evaluated, the 1 is scaled to the size of the object, which in this case is a 3-element array of ints(which we consider four bytes long), and then 2 is scaled to the size of an int as

```
*(int *) ( (char*) ar +(1*3*4) + (2*4))
```

We have used the (char*) cast to turn off scaling because we have already made the scaling explicit. The (int *) cast ensures that we get all four bytes of the integer when we dereference the address value.

After doing the arithmetic, the expression becomes

```
*(int*) ((char *) ar + 20)
```

The value 20 has already been scaled so it rep resents the number of bytes to skip. Thus, if ar starts at address 1000(hex value), as shown in the diagram, ar[1][2] refers to the int that begins at address 1014, which is 6.

In multidimensional array, if we specify fewer subscripts than there are dimensions, the result is a pointer to the base type of the array. Thus, for the above discussed two dimensional array, if we make the reference

ar[1]

then, it is same as

& ar[1][0]

and the result is a pointer to an int.

The ANSI standard places no limits on the number of dimensions of an array. The exact limit is determined by the compiler. However, as per standard, they should support at least six dimensions.

5.1. Initializing a Multidimensional Array: Like the one-dimensional arrays, multi-dimensional arrays may be initialized by specifying the elements in row major order in which we enclose each row in braces. We have already considered one such example in case of 5×5 magic square.

When initializing such arrays, if there are less initializers, the extra elements in the row are initialized to zero. For example, let us consider

```
static int table [5][3] = { {1, 0, 2},
                            {3},
                            {4, 5, 6}
                          };
```

This declares an array of 5 rows and three columns, but only the first three rows are initialized, and only the first element of the second row is initialized. All other elements are initialized to zero. Thus, the declaration produces the following array

1	0	2
3	0	0
4	5	6
0	0	0
0	0	0

If we do not use the inner braces i.e. static int table[5] [3] = {1, 0, 2, 3, 4, 5, 6};
then the result is as follows:

1	0	2
3	4	5

6 0 0
0 0 0
0 0 0

which is very misleading. To enhance readability and clarity, we should always enclose each row of initializers in its own set of braces.

Similar to the case of one-dimensional arrays, if we omit the size specification of a multidimensional array, the compiler automatically determines the size based on the number of initializers present. In the case of multidimensional arrays, however, we should remember that we are really declaring an array of arrays, which means that we may only omit the first size specification, but we must specify the other sizes. For example

```
static  int   ar1[ ] [3] [2] = { { {0, 0}, {0, 1}, {1, 2}},
                                 { {1, 1}, {2, 1}, {2, 2}}
                               };
```

initializes a 2×3×2 array because there are twelve initializers. Each element in the array ar 1 is itself a 3×2 array. If we had added another initializer, the compiler would allocate space for a 3×3×2 array, initializing the extra elements to zero. On the other hand, the initialization

```
static  int   ar 2[ ] [ ] = {0, 1, 3, 2, 4, 6};
```

is illegal since the compiler cannot understand whether to create 2×3 array or 3×2 array. However,, if we specify the size of each array other than the first, the declaration becomes obvious.

5.2. Passing Multidimensional Arrays as Arguments : For passing a multidimensional array as an argument, we pass the array name as we did for the case of single-dimensional array. The value passed is a pointer to the initial element of the array, but in this case, the initial element is itself an array. On the receiving side, we must declare the argument appropriately, as shown in the following example.

```
function 1 ( )
{
     int   ar[3] [4] [5];
     ……..
     ……..
     function 2(ar);
     ……..
     ……..
}
function 2(received_arg)
int   received_arg [ ] [4] [5];
{
```

```
    ……
    ……
}
```

Again note that we have omitted the size of the array being passed, but we must specify the size of each element in the array. Most compilers do not check bounds, so it does not matter whether we specify the first size For example, the compiler would interpret the declaration of received_arg as if it had been written as

```
int (* received_arg) [4] [5];
```

Another way to pass multidimensional arrays is to explicitly pass a pointer to the first element, and pass the dimensions of the array as additional arguments. In our example, what gets passed is actually a **pointer to a pointer to a pointer to an int.** Thus, we have

```
function 1 ( )
{
    int ar [3] [4] [5];
    ……..
    ……..
    function 2(ar, 3, 4, 5);
    …….
    …….
}
function 2(received_arg, dim1, dim2, dim3)
int  *** received_arg;
int  dim 1, dim 2, dim 3;
      {
      ……..
      ……..
      }
```

One advantage of this approach is that we need not know ahead of time the shape of the multidimensional array. However, the drawback of this approach is that we need to perform the indexing arithmetic manually to access an element. For example, to access ar [i] [j] [k] in function 2( ), we would need to write

```
*((int*) received_arg + i *dim 3* dim 2 + j*dun 2 + k)
```

Again note that we need to cast received_arg to a pointer to an int because we are performing our own scaling. Although this method increases the programmer's work, it gives more flexibility to function 2( ) since it can accept three-dimensional arrays of any size and shape. Moreover, it is possible to define a macro that simplifies the indexing expression. We shall discuss the complex macros in Unit-V.

5.3. Arrays of Pointers: An array of pointers is similar to an array of any predefined data type. As a pointer variable always contains an address, an array of pointers is a collection of addresses. These can be addresses of ordinary isolated variables or of array elements. The elements of an array of pointers are stored in the memory just like the elements of any other kind of array. All rules that apply to other arrays also apply to the array of pointers.
To visualize the situations where it is useful to employ an array of pointers, let us consider the following declaration

```
char * ar_of_ptr[6];
```

Here, the variable ar_or_ptr [ ] is a 6-element array of pointers to characters and not a pointer to a 6-element array of characters. This is because the array element operator [ ] has higher precedence than the dereferencing operator *. We shall discuss complex declarations of this type in more detail in Unit-IV.

So far the pointers have not been assigned any value, so they point to random addresses in memory. But we can make assignments such as

```
char * ar_of_ptr[6];

char C0 = 'a';

char C1 = 'b';

ar_of_ptr [0] = & C0;

ar_of_ptr [1] = & C1;
```

Here, the compiler first allocates two bytes somewhere in memory for the variables C0 and C1. Then it assigns the addresses of these variables to ar_of_ptr[0] and ar_of_ptr [1]. The storage relationship is shown in the figure

Element	**Address (in hex)**	**Memory ← 1 byte→**
	IFFF	
C0	2000	'a'
C1	2001	'b'
	2002	

Element	Address (in hex)	Memory ←4bytes→
	OFFC	
ar_of_ptr[0]	1000	2000
ar_of_ptr[1]	1004	2001
ar_of_ptr[2]	1008	undefined
ar_of_ptr[3]	100C	undefined
ar_of_ptr[4]	1010	undefined
ar_of_ptr[5]	1014	undefined
	1018	

The addresses in the figure are arbitrary. The only thing that is guaranteed is that ar_of_ptr [0] and ar_of_ptr[1] will contain the addresses of C0 and C1 and that C0 and C1 will be initialized to ‘a’ and ‘b’ respectively.

To further illustration the concept of arrays of pointers, let us consider the following example

```
# include <stdio.h>
void  main ( )
{
      int   ar[5] = {1, 2, 3, 4, 5}, i, * p;
      for (p = ar = 4, i = 0; i < 5; i ++)
            printf (″% i″, p[–i]);
}
```

Here, the integer pointer p is initially assigned the address of the 5^{th} element of the array. The variable i is initialized to 0. The expression p[–i] is the same as the expression ∗(p–i). Hence, when i is 0, p[–i] refers to the 5^{th} element, when i is 1, p[–i] = ∗(p–1) refers to the 4^{th} element, and so on. The for loop is executed until all the elements of the array are printed. The output of the program is

```
5  4  3  2  1
```

Arrays of pointers are frequently used to access arrays of strings. In such cases, they are termed as **arrays of pointer to strings.** An array of pointers to

strings is an array whose elements are pointers to the base addresses of the string. Such array is declared and initialized in the following manner.

```
static char * pet [ ] = {″Cat″, ″Dog″, ″Lion ″};
```

Here, pet[0], pet[1], pet[2] are pointers to the base addresses of the strings "Cat", "Dog", "Lion" respectively.

To illustrate the concept, let us consider the following function which takes an integer (1 to 12) representing a month as its input and prints the name of the month.

```
include <stdio.h>

void print_month (m)

int m;

{

    static char * month [13] = {"Badmonth″, ″January″,

        ″February″, ″March″, ″April″, ″May″, ″June″,

        ″July″, ″August″, ″September″, ″October″,

        ″November″, ″December″};

    if (m > 12)

    {

        printf (″Illegal month value.\n″);

        exit (1);

    }

    printf (″%s\n″, month [m]);

}
```

Here, the variable month is a 13-element array of pointers to chars. We could have declared a 12-element array without using the initial element initialized to "Badmonth" and then change the printf ( ) statement to

```
printf (″%s\n″, month [m–1]);
```

we prefer the first version because it is more straightforward.

The print_month ( ) function would be more useful if, instead of printing the month, it returned it. The calling function then could do the required job. To write this version, we need to declare a function that returns a pointer to a char, as follows

```
# include <stdio.h>

char * month_text (m)

int m;

{
```

```
    static char * month [13] = {……
    ……
    ……....};
    if (m > 12)
    {
        printf (″Illegal month value.\n″);
        exit (1);
     }
     return month [m];
}
```

5.4. Pointers to Pointers : We know that a pointer is a variable that holds the address of another variable. This concept can be further extended. We can have a variable that holds an address of a variable that in turn holds an address of another variable. This type of variable is known as pointer to a pointer. The underlying concept is known as **double indirection.** This concept can be further extended to higher levels. Thus, a pointer to a pointer is a construct and is used frequently in sophisticated programs

To declare a pointer to a pointer, we precede the variable name with two successive asterisks. Thus

```
int ** p;
```

declares p to be a pointer to a pointer to an int. We have already observed one such variable i.e.

```
int *** received_arg; /*pointer to pointer to pointer to int*/
```

when discussing multidimensional arrays passed as arguments.

To dereference the pointer to pointer and to access the int, we need to use two asterisks. Thus, the statement

```
i = ** p;
```

assigns an integer to i.

For further illustration, let us consider the statements

```
int r = 5;
int * q = & r;
int ** p = & q;
```

Here, both q and p are pointers but q contains the address of an int, whereas p contains the address of a pointer to an int. The storage pattern is as shown below.

Variable	Address (in hex)	Contents ←4bytes→
r	99C	5
q	1004	99C
p	100C	1004

We can assign values to r in three ways as follows :

r = 10;

∗ q = 10;

∗∗ p = 10;

UNIT-IV

1. Storage Classes

Most large programs are written by teams of programmers. Every member of the team works independently on a piece of the large program. After finishing the job, all the pieces are linked together to form the complete program. For such process to work, there should be a mechanism to ensure that variables declared by one programmer do not conflict with unrelated variables of the same name declared by another programmer. Further, there is usually some data that needs to be shared between different source files. So, there must be a mechanism which ensures that some variables declared in different files do refer to the same memory locations and that the computer interpret those locations in a consistent manner.

In C, we define whether a variable is to be shared, and which portions of the program can share it, by designating its **scope.** Scope is the technical term which denotes the region of the C source text in which a name's declaration is active. Another property of variables is **duration,** which describes the lifetime of a variable's memory space. Variables with **fixed duration** are guaranteed to retain their value even after their scope is exited. There is no such guarantee for variables with **automatic duration.** The scope and duration of a variable is collectively termed as **storage class.**

Let us consider the following program segment

```
void  function ( )
{
    int   i;
    static  int   ar [  ] = { 1, 3, 5};
    …….
    …….
}
```

Here, we have two variables i and ar. Both have **black scope** as they are declared within a block. They can be seen or referenced only by the statements within the block. Such variables with block scope are often termed as **local variables.** The variables with block scope, by default, have automatic duration. Thus, the variable i has automatic duration. However, the variable ar has fixed duration because it is declared with the 'static' keyword. This means that i has memory allocated to it automatically and may have a new address each time the block is entered. The variable ar, on the other hand, has memory allocated for it just once and keep its original address for the duration of the program.

1.1. Remark. For discussing fixed and automatic variables, we use the term fixed and not static so as not to confuse the concept with the keyword. The

'static' keyword does give the variable static duration, but it also has scoping implications not usually associated with static variables.

1.2. Fixed Vs. Automatic Duration : From the above discussion as well as the names themselves we observe that a fixed variable is one which is stationary, whereas an automatic variable is one whose memory storage is automatically allocated during program execution. This means that a fixed variable has memory allocated for it at program start-up time and the variable is associated with a single memory location until the end of the program. An automatic variable has memory allocated for it whenever its scope is entered. The automatic variable refers to that memory address only as long as code within the scope is being executed. Once the scope of the automatic variable is exited, the compiler is free to assign that memory location to the next automatic variable it sees. If the scope is re-entered, a new address is allocated for the variable.

Local variables are automatic by default, but we can make them fixed by using the keyword 'static' in the declaration. The 'auto' keyword explicitly makes a variable automatic, but we rarely use it since it is superfluous.

The difference between fixed and automatic variables is especially important for initialized variables. Fixed variables are initialized only once, whereas automatic variables are initialized each time their block is re-entered. Let us consider the following program

```
void  next ( )
{
        int   i = 1;
        static  int   j = 1;
        i + + ;
        j + +;
        printf (″ i : %d \t j : % d \n″, i, j);
}
main ( )
{
        next ( );
         next ( );
        next ( );
}
```

Here, the next ( ) function increments two variables i and j, both initialized to 1. i has automatic duration by default, while j has fixed duration because of the 'static' keyword. The output of the program is

```
i : 2     j : 2
i : 2     j : 3
i : 2     j : 4
```

When next ( ) is called by main ( ) second time, memory for i is reallocated and i is re-initialized to 1. On the other hand, j has still maintained its memory address and is not re-initialized. Thus, the value of i well always be 2 and j will increase by 1 with each call.

Another important difference between automatic and fixed variables is that automatic variables are not initialized by default whereas fixed variables get a default initial value of zero. Thus, in the above program, if we replace the statements

```
int   i = 1;
static  int   j = 1;
```

by the statements without initializing the variables, i.e.

```
int   i;
static  int   j;
```

the output of the program will be

```
i : 23057        j : 1
i : 23057        j : 2
i : 23057        j : 3
```

The value of i are random since the variable is never initialized. With each call of next ( ), same or different garbage value is received by i, depending upon the mode of calling.

One more difference between initializing variables with fixed and automatic duration is the kinds of expressions that may be used as an initializer. For scalar variables with automatic duration, the initializer may be any expression so long as all of the variables in the expression have been previously declared. For example, all of the following declarations are legal

```
{
    int   i = 1, j = 2;
    int   n = i + j ;
    float   x = 3.56 * 4.3;
    ……….
    ……….
```

The following statements are illegal.

```
{
    int   n = i + j; /* i and j have not yet been declared */
    int   i = 1, j = 2; /* it is too late to declare i and j */
    ……….
    ……….
```

The rules for initializing variables with fixed duration are more strict. The initialization must be a constant expression, which means that it may not contain variable names. For example,

```
int   i = 5 * 3+12; /* valid */

int   j = i; /* not valid */
```

1.3. Scope : We have already mentioned that the scope of a variable determines the region over which we can access the variable by name. There are four types of scope : program, file, function, and block. Let us describe them in detail.

(i) Program Scope : It signifies that the variable is active among different source files that make up the entire executable program. Thus, variables with program scope are visible to routines in other files as well as their own file. Such variables are often termed as **global variables.** To create a global variable, we declare it outside a function without the 'static' keyword.

(ii) File Scope : File scope signifies that the variable is active from its declaration point to the end of the source file. Thus the variable having file scope is active throughout the file. So, if a file contains more than one functions, all of the functions following the declaration are able to use the variable. To give a variable file scope, we declare it outside a function with 'static' keyword. Variables with file scope are particularly useful when we have a number of functions that operate on a shared data structure, but we do not want to make the data available to other functions. A file that contains this group of functions is often called a **module.** The linked-list functions, which we shall discuss later on, illustrate a good use of a variable with file scope.

(iii) Function Scope : It signifies that the name is active from the beginning to the end of the function. The only names that have function scope are 'goto' labels. Labels are active from the beginning to the end of a function. This means that labels must be unique within a function. However, different functions may use the same label names without creating any conflict.

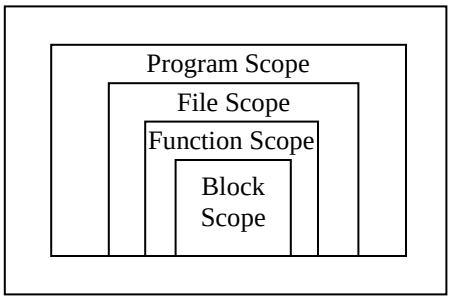
(iv) Block Scope : Block scope signifies that the variable is active from its declaration point to the end point of the block in which it is declared. A block is any series of statements enclosed in braces. This includes compound statements as well as function bodies.

We note that a variable with block scope cannot be accessed outside its block. This limitation actually turns into an advantage since it allows us to write sections of code without worrying about whether our variable names conflict with names used in the other parts of the program. Also it reduces the complexity of the program by making it more readable and maintainable, since it is known in advance that the variable's use is limited to a small region.

From the above discussion, we conclude that the scope of a variable, in general, is determined by the location of its declaration. Variables declared within a block have block scope; variables declared outside a black have file

scope if the 'static' keyword is present, or program scope if 'static' keyword is not present ; only 'goto' labels have function scope.

The four scopes are hierarchically arranged in the following figure



Hierarchy of Active Regions (Scopes)

A variable with program scope is also active within all files, functions and blocks that make up the program. Similarly, a variable with file scope is also active within all functions and blocks in the file, but is not active in other parts of the program. The block scope is the most limiting case.

The following program segment shows variables with all four types of scope.

```
int   i, /* Program scope */
static  int  j ; /* File scope */
function (k) /* Program scope */
int   k; /* Block scope */
{
      int   m; /* Block scope */
      start : /* Function scope */
      ……..
      ……..
```

It should be noted that function parameters have block scope. They are treated as if they are the first declarations in the top-level block.

C language allows us to give two variables the same name, provided they have different scopes. Further, it is also possible for variables with the same to have different scopes that overlap. In such case, the variable with the smaller scope temporarily hides the other variable i.e. the variable with smaller scope will be effective in the region of overlap. We should try to avoid such variables.

1.4. Remark: One of the most confusing aspect about storage-class declarations in C is that the 'static' keyword seems to have two effects depending on where it appears. In a declaration within a block, static gives a variable fixed duration instead of automatic duration. Outside a function, on the other hand, static has nothing to do with duration. Rather, it controls the scope of a variable, giving its file scope instead of program scope. One way of reconciling these dual meanings is to think of static as satisfying both file

scoping and fixed duration. Within a block, the stricter block scoping rules override static's file scoping, so fixed duration is the only manifest result. Outside a function, duration is already fixed, so file scoping is the only manifest result.

1.5. Global Variables : Variables declared outside all functions are said to have global scope. Such variables are called global variables. We should try to avoid global variables as much as possible. Since they increase the complexity of the program, the program becomes hard to maintain. Global variables also create the potential for conflicts between modules. Two programmers working on separate parts of a large program may choose the same name for different global variables. The problem would not surface until the entire program is linked together and at that time it may be difficult to handle the situation.

When we need to share data among different routines, it is usually better to pass the data directly, or pass pointers to a shared memory area. The one advantage of global variables is that they produce faster code. However, in most cases, the increase in execution speed comes at the cost of a significant decrease in maintainability.

Since global names must be recognised not only by the compiler but also by the linker or binder, their naming rules are a little different. The ANSI standard guarantees only that the first six characters of a global name will be recognised.

Although, we are not restrained from adding additional characters to make the name more meaningful, but we should make sure that first six characters are unique. Further, a compiler may suspend the case-sensitivity rule for global names. This is an unfortunate restriction, but it is necessary to support older systems.

The following program illustrates the concept of global variables.

```
# include <stdio.h>
int   x;
main ( )
{
        x = 10
        printf (″x = %d\n″,x);
        printf (″x = %d\n″, function 1( ));
        printf (″x = %d\n″, function 2( ));
        printf (″x = %d\n″, function 3( ));
}
function 1 ( )
```

```
{
        x = x+5;
        return (x);
}
function 2( )
{
        int   x;
        x = 1;
        return (x);
}
function 3( )
{
        x = x+10;
        return (x);
}
```

Output of the program is

```
x = 10
x = 15
x = 1
x = 25
```

We observe that x is used in all functions, but none except function 2( ), has a definition for x. Because x has been declared **above** all the functions, it is available to each function without having to pass x as a function argument. Thus, once a variable has been declared as global variable, any function can use it and change its value. Then, subsequent functions can reference only that new value. Because of this property, we should try to use global variables only for tables or for variables shared between functions when it is inconvenient to pass them as parameters.

Another aspect of global variable is that it is accessible only from the point of declaration to the end of the program.

1.6. Definitions and Allusions (External Declarations): Till now, we have assumed that every declaration of a variable causes the compiler to allocate memory for the variable. However, memory allocation is produced by only one type of declaration, called a **definition.** Global variables permit a second

type of declaration, which we call an **allusion**. An allusion looks just like a definition, but instead of allocating memory for a variable, it informs the compiler that a variable of the specified type exists but is defined elsewhere. For this purpose, we declare the variable or function with the storage class '**extern**'. For example,

```
main ( )
{
      extern  int   f( );     /* Allusion of f( ) */
      extern  float   g( );    /* Allusion of g( ) */
      …….
      …….
}
int   f ( )    /* definition of f( ) */
{
       …….
       …….
}
float  g( )    /* definition of g( ) */
{
       extern int   y ;    /* Allusion of y*/
       ……..
       ……..
}
int   y;   /* definition of y*/
```

The 'extern' keyword tells the compiler that the variables or functions are defined elsewhere. The purpose of the allusion is to enable the compiler to perform type checking. For any global variable or function, there may be any number of allusions but only one definition among the source files making up the program. The rules for creating definitions and allusions are one of the least standardized features of C language because they involve not just the C compiler, but the linker and loader as well. Here, we describe only the ANSI rules

To define a global variable according to ANSI stand, we need to make a declaration with an initializer outside a function. The presence or absence of the 'extern' keyword has no effect. Let us consider the following program segment

```
int   i = 0;        /* global definition */
```

```
extern  float  x = 1.0;    /* global definition */
function ( )
{
     int   j = 0; /* local definition */
     extern  int   i;    /* Allusion to global variable */
     …….
     …….
```

The above code defines two global variables, one local variable and alludes to one global variable.

If we omit an initializer, the compiler produces either an allusion (if extern is specified) or a **tentative definition** (if extern is not present). A tentative definition is a declaration that can become either a definition or an allusion depending on what the remainder of the source file contains. If no real definition for the variable occurs (i.e. one with an initializer) in the remainder of the source file, the tentative definition becomes the real definition, initialized to zero. Otherwise, if there is a real definition in the source file, the tentative definition becomes an allusion. Let us consider the following :

```
int   i ; /* Tentative definition */
int   j;  /* Tentative definition */
function ( )
{
          ……..
          ……..
}
int   i = 1;
```

Here, since real definition of i follows, therefore, the tentative definition of i becomes an allusion. However, there is no real definition of j, so the tentative definition of j becomes the real definition, initialized to zero. We should put all allusions in a header file which can be included in other source files. This ensures that all source files use consistent allusions. Any change to a declaration in a header file is automatically propagated to all source files those include that header file.

1.7. The Register Specifier (Register Variable) : We can tell the compiler that a variable should be kept in one of the machine's registers, instead of keeping in the memory where normal variables are stored. This is done by using the 'register' keyword. Variables with this keyword are called register variables. For example

```
register  int  count ;
```

may store the integer variable count in register. The 'register' keyword is designed to help the compiler to decide which variables to store in registers. However, it is only a hint, not a directive and the compiler is free to ignore it.

Since a register access is much faster than a memory access, keeping the frequently accessed variables, such as loop control variables, in the register will lead to a faster execution of programs. Although, ANSI standard does not restrict its application to any particular data type, most compilers allow only int. or char variables to be placed in the register.

The degree of support for 'register' varies widely from one compiler to another. Some good compilers store all variables defined as 'register' in a register until all the computer's registers are filled. Other compilers ignore 'register' altogether. When the registers are filled, C automatically convert 'register' variables into non-register variables.

1.8. Storage Class Modifiers : According to ANSI features, in addition to the **storage-class specifiers** (auto, static, extern, register) discussed above, there are following two **storage-class modifiers.**

(i) The Const Storage-Class Modifier : We may like the value of certain variable to remain constant during the execution of a program. We can do so by declaring the variable with the keyword 'const' (borrowed from C++ language) at the time of initialization. For example,

```
const  int  class_size = 50;
```

tells the compiler that the value of the int variable class_size must not be modified by the program. However, it can be used on the right-hand side of an assignment statement like any other variable.

(ii) The Volatile Storage-Class Modifier : The 'volatile' keyword, which is not supported by older compilers, informs the compiler that the variable can be modified at any time in ways unknown to the C compiler (i.e. by some external sources from outside the program). For example,

```
volatile  int  date ;
```

tells the compiler that the value of the int variable date may be altered by some external factors even if it does not appear on the left hand side of an assignment statement. When a variable is declared as 'volatile', the compiler will examine the value of the variable each time it is encountered to see whether any external alteration has changed the value.

It may be noted that the value of a variable declared as 'volatile' can be modified by its own program as well. If we want that the value must not be modified by the program while it may be altered by some other source, then we may declare the variable as both 'const' and 'volatile' as shown below.

```
volatile  const  int   police = 100;
```

1.9. ANSI Rules for the Syntax and Semantics of the Storage Class Keywords : So for we have described the semantics of storage classes i.e.

have they affect variables. But we have passed over some of the details about syntax i.e. how storage-classes are specified. Here, we summarize the ANSI rules for the syntax and semantics of the storage-class keywords.

We observe that there are four storage-class specifiers i.e. auto, static, extern, register and two storage-class modifiers i.e. const and volatile. Any of the storage-class keyword may appear before or after the type name in a declaration, but by convention they come before the type name. The semantics of each keyword depends to some extent on the location of the declaration. Omitting a storage class specifier also has a meaning, as described below.

(i) Auto : The 'auto' keyword which makes a variable automatic, is legal only for variables with block scope. Since this is the default anyway, 'auto' is somewhat superfluous and is rarely used.

(ii) Static : The 'static' keyword may be applied to declarations both within and outside a function (except for function arguments), but the meaning differs in the two cases. In declarations within a function, 'static' causes the variable to have fixed duration (lifetime) instead of the default automatic duration For variables declared outside a function, the 'static' keyword gives the variable file scope (visibility) instead of program scope.

(iii) extern : The 'extern' keyword can be used for declarations both within and outside a function (except for function arguments). For variables declared within a function, 'extern' signifies a global allusion, for declarations outside a function, it denotes a global definition. In this case, the meaning is the same whether we use 'extern' or not.

(iv) register : The 'register' specifier may be used only for variables declared within a function. It makes the variable automatic but also passes a hint to the compiler to store the variable in a register whenever possible. We should use the 'register' keyword for automatic variables that are accessed frequently. Compilers support this feature at various levels. Some do not support it at all, while others support as many as 20 concurrent register assignments.

(v) No Storage-Class Specifier Present : (i.e. Omitting a Storage Class Specifier) For variables with block scope, omitting a storage class specifier is the same as specifying 'auto'. For variables declared outside a function, omitting a storage class specifier is the same as specifying 'extern'. It causes the compiler to produce a global definition.

(vi) Const : The 'const' modifier guarantees that we cannot change the value of the variable during the execution of the program.

(vii) Volatile : Declaring a variable with the volatile modifier causes the compiler to turn off certain optimizations. This is particularly useful for device registers and other data segments that can change without the knowledge of the compiler.

The following table summarizes the scope and duration (visibility and lifetime

or active and alive) semantics of each storage class specifier.

Place where declared / Storage class specifier	Outside a Function	Within a Function	Function Arguments
Auto or register	Not Allowed	Scope: block duration : automatic	Scope : block duration : automatic
Static	Scope: file duration : fixed	Scope: block duration : fixed	Not Allowed
extern	Scope: program duration : fixed	Scope: block duration : fixed	Not allowed
No storage class specifier present	Scope: program duration : fixed	Scope: block duration : dynamic	Scope : block duration : automatic

Semantics of storage-class specifiers .

The syntax for storage-class specifiers is rather loose, allowing some declarations that have little or no meaning. For example, it is legal to declare a variable with both 'register' and 'volatile', although it is unclear how a compiler would interpret it. The only real syntactic restriction is that a declaration may include at most one storage-class specifier. But either or both modifiers can be used. For example, the following is perfectly legal.

```
{
extern const volatile int date;
…….
…….
```

It is an allusion to a variable of type int that is bot 'const' and 'volatile'.

2. Dynamic Memory Allocation

Most often we face situations in programming where the data is dynamic in nature i.e. the number of data items keep changing during execution of the program. We observe that C requires the number of elements in an array to be specified at compile time. But we may not be able to do so always. Our initial judgement of size, if it is wrong, may cause failure of the program or wastage of memory space. Many HLL permit a programmer to specify an array's size at run time. Such languages have the ability to calculate and assign, during execution, the memory space required by the variables in a program. The process of allocating memory at run time is known as **dynamic memory allocation.** C provides such facility through four runtime library functions. These functions, also known as **memory management functions,** can be used for allocating and freeing memory during program execution. They help us to build complex application programs that use the available

memory intelligently i.e. they enable us to allocate memory on the fly. These four dynamic memory allocation functions are as follows:

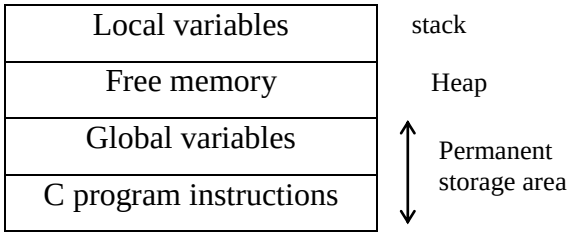
(i) malloc ( ) : It allocates a specified number of bytes in memory and returns a pointer to the beginning of the allocated block.

(ii) calloc ( ) : It is similar to malloc ( ), but initializes the allocated bytes to zero. The function also allows us to allocate memory for more than one object at a time.

(iii) realloc ( ) : It changes the size of a previously allocated block

(iv) free ( ) : It frees up memory that was previously allocated with malloc ( ), calloc ( ), or realloc ( ).

2.1. Memory Allocation Process : Before discussing the above functions, let us look at the memory allocation process associated with a C program. The conceptual view of storage of a C program in memory, is shown in fig.



(Storage of a C Program)

The program instructions and global as well as static variables are stored in a region known as **permanent storage area** and the local variables are stored in separate region called **stack.** The memory space that is located between these two regions, is available for dynamic memory allocation during execution of a program. This free memory region is called the **heap.** When program is executed, the size of the heap keeps changing due to creation and death of variables that are local to functions and blocks. Therefore, it is possible to encounter memory overflow during dynamic allocation process. In such situations, the memory allocation functions, mentioned above, return | a NULL pointer when they fail to locate enough memory requested.

Now, let us discuss the four memory management functions.

2.2. Allocating a Block of Memory : A block of memory can be allocated using the function malloc ( ). This function reserves a block of memory of specified size and returns a pointer of type void. This means that we can assign it to any type of pointer. It takes the following form.

ptr = (cast-type ∗) malloc (byte-size);

where ptr is a pointer of type ‘cast-type’.

The argument to malloc ( ) is the size in bytes of the block of memory to be allocated. malloc ( ) returns a pointer to the beginning of the allocated block. For example, on successful execution of the statement

p = (int ∗) malloc (100∗size of (int));

a memory space equivalent to "100 times the size of an int" bytes is reserved and the address of the first byte of the memory allocated is assigned to the pointer p of type of int. Similarly, the statement

q = (char ∗) malloc (20);

allocates 20 bytes of space for the pointer q of type char. It should be noted that the storage space allocated dynamically has no name and therefore its contents can be accessed only through a pointer. Further, the malloc ( ) function allocates a block of contiguous bytes. The allocation can fail if the space in the heap is not sufficient to satisfy the request. If it fails, it returns a NULL. We should therefore check whether the allocation is successful before using the memory pointer.

The following program uses a table of integers whose size will be specified interactively at run time. It tests for availability of memory space of required size. If it is available, then the required space is allocated and the address of the first byte of the space allocated is displayed

```
/∗ Program to illustrate use of malloc ( ) function ∗/
# include <stdio.h>
# include <stdlib.h>
# define NULL 0
main ( )
{
        int  ∗p, ∗ table, size ;
        printf (″\n what is the size of table ?″);
        scanf (″%d″, size);
        printf(″\n″);
        /∗ Memory allocation ∗/
        if ((table = (int ∗) malloc (size ∗ size of (int))
                                        = = NULL)
        {
                printf (″No space available \n″);
                exit (1);
        }
```

```
        printf (''\n Address of the first byte is %u\n}, table);
        /* Reading table values */
        printf(''\n Input table values \n'');
        for (p = table; p < table + size ; p ++)
              scanf (''%d'', p);
        /* Printing table values in reverse order */
        for (p = table + size –1; p > = table ; p – –)
            printf (''%d is stored at address %u) \n, *p, p);
        exit (0);
    }
```

The output is as follows :

```
What is the size of the table ? 4
Address of the first byte is 2432
Input table values
25      26      27      28
28 is stored at address 2444
27 is stored at address 2440
26 is stored at address 2436
25 is stored at address 2432
```

Here, we have assumed that int occupies four bytes.

2.3. Allocating Multiple Blocks of Memory : We have another memory allocation function calloc ( ) which is normally used for requesting memory space at run time for storing derived data types such as arrays and structures. While malloc ( ) allocates a single block of storage space, calloc ( ) allocates multiple blocks of storage, each of the same size, and then sets all bytes to zero.

The general form of calloc ( ) is

```
ptr = (cast-type *) calloc (n, element-size);
```

This allocates contiguous space for n blocks, each of size 'element-size' bytes. All bytes are initialized to zero and a pointer to the first byte of the allocated region is returned. If there is not enough space, a NULL pointer is returned.

2.4. Altering the Size of a Block : At some later stage, we may realize that the previously allocated memory space is too short or too large. In both the

cases, we can change the memory size already allocated with the help of the function realloc ( ). This process is called the **reallocation of memory.** Thus, if the original allocation was done by the statement

ptr = malloc (size);

then reallocation of space may be done by the statement

ptr = realloc (ptr, newsize);

This function allocates a new memory space of size 'newsize' to the pointer variable ptr and returns a pointer to the first byte of the new memory block. The 'newsize' may be larger or smaller than the 'size'. It should be noted that the new memory block may not begin at the same place as the old one. In case, it is not able to find additional space in the same region, it will create the same in an entirely new region and move the contents of the old block into the new block. The function guarantees that the old data will remain intact.

2.5. Releasing the Used Space : The compile-time storage of a variable is allocated or released by the system in accordance with its storage class. With the dynamic run-time allocation, it is our responsibility to release the space when it is not required. It becomes particularly important when the storage is limited. A block of memory is released using the free ( ) function as

free (ptr);

where ptr is a pointer to a memory block which has already been created by malloc ( ), calloc ( ) or realloc ( ). Use of an invalid pointer in the call may create problems and cause system crash.

The following program illustrates the use of malloc ( ), realloc ( ) and free ( ) function

```
# include <stdio.h>
# include <stdlib.h>
# define   NULL   0
main ( )
{
    char  * buffer ;
    /* Allocating memory */
    if (buffer = (char *) malloc (10)) = = NULL)
    {
        printf (″malloc failed.\n″);
        exit (1);
    }
```

```
    printf (′′Buffer of size % d created.\n′′, –m size (buffer));
    strcpy (buffer, ′′HYDERABAD′′);
    printf (′′\n Buffer contains : % s\n′′, buffer);
    /∗ Reallocating memory ∗/
   if ((buffer = (char ∗) realloc (buffer, 20)) = = NULL)
   {
        printf(′′Reallocation failed.\n′′);
        exit (1);
   }
   printf(′′\n Buffer size modified.\n′′);
   printf(′′\n Buffer still contains: % s\n′′, buffer);
  strcpy (buffer, ′′SECUNDERABAD′′);
  printf(′′\n Buffer now contains : % s\n′′, buffer);
    /∗ Freeing memory ∗/
   free (buffer);
   exit (0);
}
```

The output is as follows

```
Buffer of size 10 created.
Buffer contains : HYDERABAD
Buffer size modified.
Buffer still contains : HYDERABAD
Buffer now contains : SECUNDERABAD
```

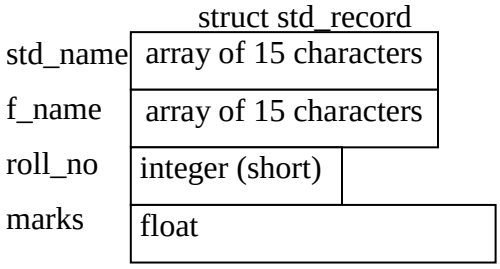
3. Structures

C supports a constructed data type known as structure, which is a method for packing data of different types. A structure is a convenient tool for handling a group of logically related data items. For example, it can be used to represent a set of attributes, such as student name, Father name, roll No. marks. We can define a structure to hold this information as follows

```
struct   std_record
{
        char  std_name [15];
        char  f_name [15];
        short  int  roll_no;
        float  marks;
};
```

The keyword struct declares a structure to hold the details of four fields, namely std_name, fname, roll_no and marks. These fields are called structure elements or members. Each member may belong to a different type of data. std_record is the name of the structure and is called the structure tag. The tag name may be used subsequently to declare variables that have the tag's structure.

Note that the above declaration has not declared any variables. It simply describe a format called template to represent information as :-



we can declare structure variables using the tag name anywhere in the program. For example

```
struct  std_record  student1, student2, student3;
```

declares student1, student2, student3 as variables of type struck std_record. Each one of these variables has four members as specified by the template. We know that members of a structure themselves are not variables. They do not occupy any memory until they are associated with the structure variables such as student. The use of tag name is optional and it is also allowed to combine both the template declaration and variable declaration in one statement. e.g.

```
struct
{
    char  std_name [15];
    char  f_name [15];
    short  int  roll_no;
    float  marks;
}  student1, student2, student3;
```

This declaration does not include a tag name later use in declarations.

3.1. Note : (i) The template is terminated with a semi colon.

(ii) For entire declaration, each member is declared independently for its name and type in a separate statement inside the template.

(iii) The tag name (i.e. std_record) can be used to declare structure variables of its type.

(iv) Structure definitions may also appear before the main ( ), along with # define. In such cases the definition is global.

3.2. Giving values to members :- As mentioned earlier that the members themselves are not variables. They should be linked to the structure variables in order to make them meaningful member. For example, the word Roll_no has no meaning, but roll_no of student1 has meaning. The link between a member and variable is established using the member operator '·', which is also known as 'dot operator' or period operator'. For example

```
student · roll_no
```

is a variable representing the roll_no of student1 and can be treated as any other ordinary variable. To assign the values we use the functions as follows

```
strcpy (student1 std_name,″Shamsher″);
strcpy (student1 · f_name, ″Randhir_Singh″);
student1 · roll_no = 8;
student1 · marks = 94:0;
```

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

```
scanf (″%s\n″, student 1 std_name);
scanf (″%d\n″, & roll_no);
```

3.3. Structure Initialization : Like other data type, a structure variable can be initialized. However, a structure must be declared as static if it is to be initialized inside a function. For example

```
main ( )
{
      static  struct
     {
         char  name [15];
         char  f_name [15];
         int   roll_no;
         float   marks;
         }
         student = {′′Shamsher′′, ′′Randhbir_Singh′′, 8, 94·0};
         ……..
         ……..
         ……..
 }
```

To initialize more than one variable we may write

```
main ( )
{
      struct  std_record
     {
              char  std_name [15];
              char  f_name [15];
              int   roll_no;
              float  marks;
        };
         static  struct  std_record  student1
   = {′′Ram′′, ′′Amit′′, 10, 63·0};
         static  struct  std_record  student 2
   = {′′mohan′′, ′′Ravi′′, 12, 76·50};
         ……
         ……
         ……
```

```
}
```

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

```
struct  marks
{
        int   subject 1;
        int   subject 2;
        int   subject 3;
};
main ( )
{
        static  struct  marks student [3] = {{95, 46, 72},
    {60, 46, 56}, {57, 35, 40}}
```

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

```
student [0] · subject 1 = 95;
student [0] · subject 2 = 46;
……….
```

3.5. Arrays within Structures : C also permits the use of array as structure members. We have already used arrays of characters inside a structure. We can also use single or multidimensional arrays of type int or float inside a structure. For example

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

Here, the member subject contains three elements, subject [0], subject [1] and subject [2]. These elements can be accessed using appropriate subscript. For example, the name

```
student [1] · subject [2];
```

would refer to the marks obtained in the third subject by the second student.

3.6. Structures within Structures (Nested structures): Structures within a structure means nesting of structures. Consider the declaration

```
struct  salary
{
        char  name [20];
        char  department [10]
        struct
        {
                int   dearness;
                int   house_rent;
                int   city;
        } allowance;
}employee;
```

where the salary structure contains a member named allowance which itself is a structure with three members, these members can be referred as

```
employee · allowance · dearness
employee · allowance · house_rent
employee · allowance · city
```

An inner structure can have more than one variable e.g.

```
struct  salary
{
  ……
  struct
  {
  ……
  }allowance, arrears;
}employee [100];
```

These can be accessed as
employee[1]·allowance·dearness
employee[2]·allowance · dearness
……
employee[1]· arrears· dearness….
……

we can also use tag names to define inner structures. For example

```
struct  pay
{
        int   dearness;
        int   house_rent;
        int   city;
};
struct  salary
{
        char  name [20];
        char  department [10];
        struct  pay allowance;
        struct  pay arrears;
};
struct  salary employee [100];
```

where pay template is defined outside the salary template and is used to define the structure of allowance and arrears inside the salary structure. It is also permissible to nest more than one type of structures.

3.7. Passing structures as function argument : There are three methods by which the values of a structure can be transferred from one function to another.

The first method is to pass each member of structure as an actual argument of the function call. The actual arguments are then treated independently like ordinary variables. This is the most elementary method and inefficient when the structure size is large.

The second method involves passing of a copy of the entire structure to the called function. Since the function is working on a copy of the structure, any changes to structure members within the function are not reflected in the original structure (in the calling function). It is, therefore, necessary for the function to return the entire structure back to the calling function. All compilers may not support this method of passing the entire structure as a parameter. The general format of sending a copy of a structure to the called function is

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

The called function takes the following form

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

The third approach is called pointers to pass the structure as an argument. In this case, the address location of the structure is passed to the called function. The function can access indirectly the entire structure and work on it. This method is more efficient as compared to others. For example :

```
struct  store
{
        char  name [20];
        float   price;
        int   quantity;
};
main ( )
{
        struct  store update ( );
        float mul ( ), p_increment, value;
        static  struct store item = {''LUX'', 25-75, 12};
        int q_increment;
        printf (''\n Input increment values'');
        scanf (%f%d, & p_increment, & q_increment);
        item = update (item, p_increment, q_increment);
        printf (''update values of item \n\n);
        printf(''Name : %s\n'', item·name);
        printf(''Price : %f\n'', item·price);
        printf(''Quantity : %d\n'', item·quantity);
        value = mul(item);
        printf (''\n value of the item = %f\n'', value);
```

```
}
struct  store update (product, p, q)
struct  store product;
float   p;
int   q;
{
        product · price + = p;
        product · quantity + = q;
        return (product);
}
 float  mul (stock)
 struct  store stock;
{
        return  (stock·price * stock·quantity);
 }
```

Here the function update receives a copy of the structure variable item as one of its parameters. Note that both the functions update and parameter product are declared as type struct store, because the function uses the parameter product to receive the structure variable item and also to return the updated values of item.

The function mul is of type float because it returns the product of price and quantity. However the parameter stock, receives the structure variable item, is declared as type struct store.

The entire structure return by update can be copied into a structure of identical type. The statement

```
item = update (item, p_increment, q_increment);
```

replaces the old values of item by the new ones.

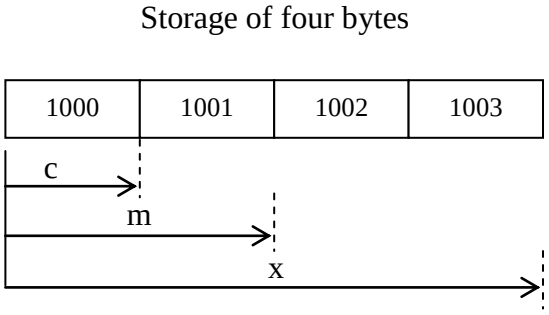
4. Unions

Unions follow the same syntax as structures. However, there is major distinction between them in terms of storage. In structures each member has its own storage location, whereas all members of a union use the same location. This implies that, although a union may contain many members of

different types but it can handle only one member at a time. A union can be declared using the keyword ‘union’ as follows

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

This declares a variable code of type union item. The union contains three members, each with a different data type and we can use only one of them at a time. This is due to the fact that only one location is allocated for a union variable, irrespective of its size.



The compiler allocates a piece of storage that is large enough to hold the largest variable in the union. In the above declaration, x is the largest member (require 4 bytes) and all the three variables share the same address.

To access a union member, we can use the same syntax that we use for structure members. That is,

code·m or code·x or code·c

are all valid member variables. A union creates a storage location that can be used by any one of its members at a time. When a different member is assigned a new value, the new value supercedes the previous member’s value. So During accessing, we should make sure that we are accessing the member whose value is currently stored. For example,

```
code ·m = 820;
code·x = 5114·73;
printf (′′%d′′, code·m);
```

would produce erroneous output.

Unions may be used in all places where a structure is allowed. The notation for accessing a union member, which is nested inside a structure remains the same as for the nested structures.

4.1. Enum Declaration : C supports a feature known as "type definition" that allows users to define an identifier that would represent an existing data type. Another user-defined data type is enumerated data type, defined as follows

enum identifier {value1, value2,… valuen};

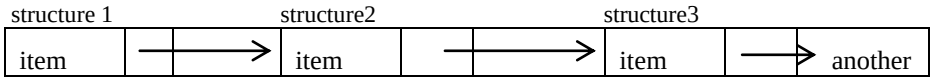
The "identifier" is a user-defined enumerated data type which can be used to declare variables that can have one of the values enclosed within the braces (known as enumeration constants)

More about this we have already discussed in unit Ist.

5. Linked Lists

We know that a list refers to a set of organized sequentially. An array is an example of list. One major problem with the arrays is that the size of an array must be specified precisely at the beginning, this may be a difficult test in many practical applications.

A completely different way to represent a list is to make each item in the list part of a structure that also contains a "link" to the structure containing the next item. This type of list is called a linked list because it is a list whose order is given by links from one item to the next (as below)



Each structure of the list is called a node and consists of two fields, one containing the item and the other containing the address of the next item (a pointer to the next item) in the list. Therefor a linked list is a collection of structures ordered not by their physical placement in memory (like an array) but by logical links that are stored as a part o the data in the structure itself.

The link is in the form of a pointer to another structure of the same type. The general form of a node is as follows :

```
struct  tag-name
{
type member1;
type member2;
……
……
struct  tag-name *next;
```

```
};
```

Let us consider a structure

```
struct  std_record;
{
        float   age:
        int   roll_no;
        struct  std_record *next;
};
struct  std_record std1, std2;
```

This statement creates space for two nodes (std1, std2) each containing three empty fields (as) std1

std1		std2	
	std1·age		std2·age
	std1·roll_no		std2·roll_no
	std1·next		std2·next

The next pointer of std1 can be made to point to std2 by the statement

```
std1·next = & std2;
```

Thus statement stores the address of std2 into the field std1·next and thus establishes a "link" between std1 and std2 nodes. We may continue this process to create a linked list of any number of values. For example

```
std2·next = & std3;
```

The end of the linked list can be made by assigning a null value to the last pointer of the last node. e.g.

```
std 37·next = 0;
```

This is necessary for processing the list. We must therefore indicate the end of the linked list. The value of the age member of std2 can be accessed using the next member of std1 as follows.

```
printf (''%f\n'', std1·next→age);
```

5.1. Inserting an Element/ Deleting an element : A record is created holding the new item and its next pointer is set to link it to the item which is to follow it in the list. The next pointer of the item which is to precede it must be modified to point to the new item. For instance if we want to insert a new student (say std3) between std1 and std2, we have to make following assignments

```
std1·next = & std3;  and
```

```
std3·next = & std2;
```

Similarly to delete an element from a linked list the next pointer of the item immediately preceding the one to be deleted is altered and made to point to the item following the deleted item. e.g. to delete std2 from inbetween of std1 and std3, we write

```
std1· next = &std3;
```

5.2. Advantages of Linked Lists : (i) A linked list is dynamic data structure so they can grow or sink in size during the execution of a program. A linked list can be made just as long as required.

(ii) Linked list does not waste memory space. It uses the memory that is just needed for the list at any point of time, because it is not necessary to specify the number of nodes to be used in the list

(iii) The more important advantage is that the linked lists provide flexibility in allowing the items to be rearranged efficiently or we can say it is easier to insert or delete items by rearranging the links. (as above)

The major limitation of linked lists is the access to any arbitrary item is little time consuming. Hence it would be better to use an array rather than a linked list, whenever we deal with a fixed length list. We also note that linked list will use more storage than an array with the same number of items. This is because each item has an additional link field.

6. Passing Arguments to a Function

Argument to a function are a means of passing data to the function. Many programming languages pass arguments by reference, which means they pass a pointer to the argument. As a result called function can actually change the value of the argument. But in C arguments are passed **by value**, which means that a copy of the argument is passed to the function. The function can change the value of this copy, but can not change the value of the argument in the calling routine. The argument that is passed is often called an **actual argument**, while the received copy is called a **formal argument** or **formal parameter**. Because of c passes argument by value, a function can assign values to the formal arguments without affecting the actual arguments. For example.

```
# include <stdio.h>
main ( )
{
        extern  void  f( );
        int   a = 10;
        f(a);  /* pass a copy of ″a″ to ″f( )″*/
        printf (″%d\n″, a);
```

```
        exit (0);
}
void  f(num)
int   num;
{
        num = 3; /* Assign 3 to argument copy */
}
```

Here the printf ( ) function prints 10, not3, because the formal argument, num in f( ), is just a copy of the actual argument a. c matches actual arguments in the call to the corresponding formal arguments in the function definition, regardless of the names used. That is the first actual argument is matched to the first formal argument, the second actual argument to the second formal argument and so on. For correct results, the types of the corresponding actual and formal arguments should be the same. Here only a copy of the values of actual argument is passed in to the called function, so, what occurs inside the function will have no effect on the variable used in the actual argument list. This is an example of passing argument but no return values.

If we want to change the values of an object with the help of function, we must pass a pointer to the object and then make an assignment through the dereferenced pointer. For example, to swap the values of two integer variables, we write

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

To call this function, we need to pass two addresses as

```
main ( )
{
        int   a = 10, b = 20;
        swap (&a, &b);
```

```
        printf (''a = %d\+b = %d\n'', a, b);
}
```

The output of this program will be

a = 20, b = 10.

The pass by value method explains the purpose of the address of operator in scanf ( ) calls. e.g. when we write

```
scanf (''%d'', & num);
```

the two arguments tells the function what type of the data to read and where to store it (at the address of num).

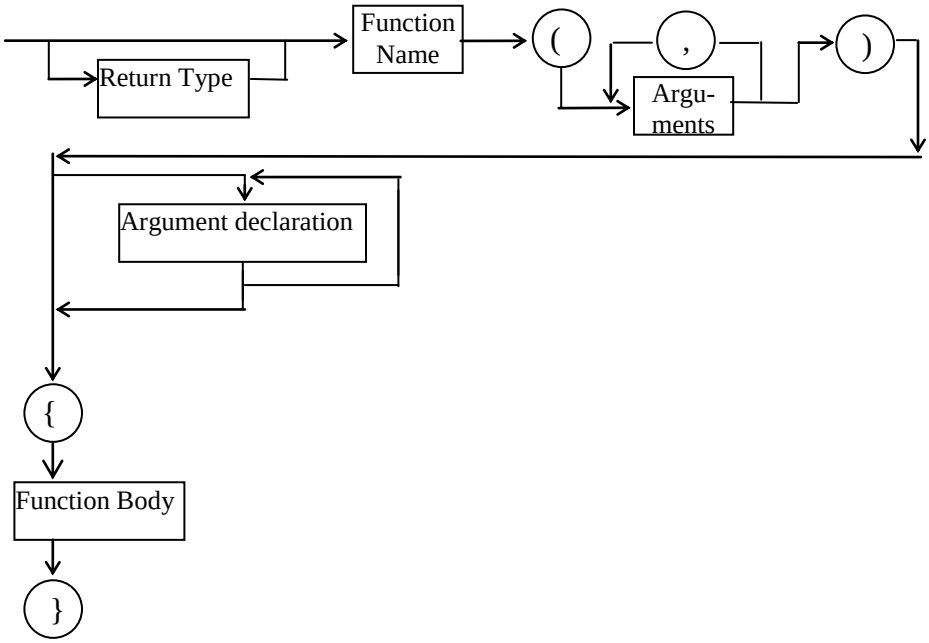
6.1. Declarations And Calls : A function can appear in a program in three forms :

(i) Definition
(ii) Function Allusion
(iii) Function Call.

6.2. Function Definition Syntax : In a function, we can specify any number of arguments, including zero. The return type defaults to int, if we leave it blank. However, even if the return type is int, we should specify it explicitly to avoid confusion. If the function does not return an int, we must specify the true return type. If the function does not return any value we should specify a return type of void. If our compiler does not support void, we can avoid this by defining a preprocessor macro that changes void to int as

```
# define  void  int.
```

The General syntax is



6.3. Argument Declaration: Argument declarations obey the same rules as other variables declarations, with the following exceptions

(i) The only legal storage class is register.

(ii) Char's and short's are converted into int's and float's are converted into double's.

(iii) A formal argument declared as an array is converted to a pointer to an object of the array type.

(iv) A formal argument declared as a function is converted to a pointer to a function

(v) You may not include an initializer in an argument declaration

It is also legal to omit an argument declaration, in which case of argument type defaults to int. We can also declare the type of argument when we list the parameters. For example

```
int  function (a, b, c)
        int   a;
        char  *b;
        float  c;
{
        . .....
```

we could write

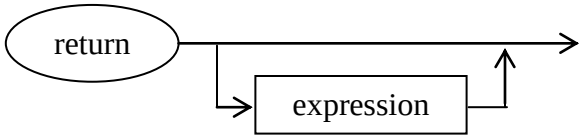
```
int   function (int a, char *b, float c)
        {
          .....
```

6.4. The Function Body : The body of a function is enclosed by a set of right and left braces. The only type of statement allowed outside a function body is a declaration. The body of a function can be empty, which can be useful in the design stages of a software product. One of the first tasks in designing a large program is to define a set of high-level operations that corresponds to functions. During, this stage, it can be useful to have a function that does nothing but return, in order to serve as a place holder for future functionality. There are called **stubs.** For example

```
void  operation1 ( )  { }
```

is a legal c function that does nothing but return when called. Later, we can fill in the function with some meaningful code.

6.5. Return Values : Function can return only a single value directly via the return statement. The return value can be any type except an array or function. The syntax for a return statement is



A function may contain any number of return statements. The first one, encountered in the normal flow of control, is executed and causes program control to be the returned to the calling routine. If there is no return statement, program control returns to the calling routine when the right brace of the function is reached. In this case the value returned is undefined.

The return value must be assignment compatible with the type of the function. This means that the compiler uses the same rules for allowable types on either side of an assignment operator to determine allowable return types. For example if func( ) is declared as a function returning an int, it is legal to return any arithmetic type, since they can be converted to an int. It would be illegal to return an aggregate type or a pointer, since these are incompatible types. e.g.

```
float  func( )
{
        float   f2;
        int   a;
        char  c;

        f2 = a;          /* ok quietly converts a to float */
        return  a;       /* ok quietly converts a to float */
        f2 = c;          /* ok quietly converts c to float */
        return  c;              /* ok quietly converts c to float
*/
}
```

The above example shows a function that returns a float and some legal return values. The behaviour is undefined if we return nothing e.g.

```
return;
```

we can safely use return without an expression only when the function type is void.

6.6. Function Allusions : A function allusion is a declaration of a function that is defined elsewhere, usually in a different source file. The main purpose of the function allusion is to tell the compiler what type of value the function returns. With the new ANSI prototyping feature, it is also possible to declare the number and types of arguments that the function takes. The syntax for a function allusion is

```
storage_class  data-type  function_name ( );
```

By default, all functions are assumed to return an int. Therefore we have to include function allusions for functions that do not return an int. However, it is good style to include function allusions for all functions that we call.

It we omit the storage class, it defaults to extern, signifying that the function definition may appear in the same source file or in another source module. The only other legal storage class is static, which indicates that the function is defined in the same source file. The data type in the function allusion should agree with the return type specified in the definition. If we omit the type, it defaults to int. Note that if we omit both the storage class and the data type, the expression is a function call if it appears within a block and it is an allusion, if it appears outside a block. For instance

```
funct1( ); /* Function Allusion - default type is int */
main ( )
{
        ……..
funct2( );    /* Function Call */
…….
```

The scope of a function allusion follows the same rules as other variables. Functions alluded to within a block have block scope and functions alluded to outside a block have file scope.

6.7. Function Calls : A function call, also called a function invocation, passes program control to the specified function. The syntax for a function call is

```
function name (arguments);
```

A function call is an expression and can appear anywhere an expression can appear. Functions always return a value that is substituted for the function call, unless they are declared as returning void. For example if f( ) returns 2, the statement

```
a = f( )/3;
```

is equivalent to

```
a = 2/3;
```

It is also possible to call a function without using the return value. Normally, we would ignore the return value only if the function returns void. However if we want to ignore a real return value, it is better cost it to void. For example

```
(void) f( );
```

is functionally equivalent to

```
f( );
```

But we frequently break this rule when we call printf ( )and scanf ( ), which both return values. The return value of scanf ( ) is very useful since it returns the number of objects that are actually assigned values.

6.8. Automatic Argument Conversions : In the absence of prototyping all scalar arguments smaller than int are converted to int, and all float arguments are converted to double. If the formal argument is declared as a char or short, the receiving function assumes that it is getting an int, so the receiving side converts the int to the smaller type. Similarly, if the formal argument is declared as a float, the receiving function assumes that it is getting a double, so it converts the received argument to float. This means that every time a char, short, or float is passed, at least one conversion takes place on the sending side where the argument is converted to int or double. Also the argument may be converted again on the receiving side if the formal argument is declared as a char short or float. e.g.

```
{
        char  a;
        short  b;
        float  c;
        funct (a, b, c); /* Here a and b are promoted to ints,
                        * and c, is promoted to double */
        ……..
    funct (x, y, z)
          char  x ;    /* Received argument is
           * converted from int to char */
          short y;   /* Received argument
          *is converted from int to short */
   float y;   /* Received argument is
          *converted from double to float */
     {
```

…….
…….

Note that these conversions are invisible. So as long as the types of the actual arguments match the types of the formal arguments, the argument will be passed correctly.

6.9. Prototyping : Function prototype requires that the function declaration must include not only the type of return value but also the type and number of arguments to expect. The general form function prototyping is

data_type function_name (type1, a_1, type 2 a_2,…, typeN a_N);

for example :-

float mul (float a, float b, int c);

This helps the compiler to perform automatic type conversions on the function parameters.

6.10. Pointers to Functions : A function, like a variable, has an address location in the memory. It is therefore, possible to declare a pointer to a function, which can then be used as an argument in other function. Pointers to functions are powerful tools because they provide an elegant way to call different functions based on the input data. A pointer to function declared as follows :

type (∗ fptr) ( );

This tells the compiler that fptr is a pointer to a function which returns type value. The parentheses around ∗fptr are necessary.

We can make a function pointer to point to a specific function by simply assigning the name of the function to the pointer. For example

double (∗p_1) ( ), mul ( );

p_1 = mul;

declare p_1 as a pointer to a function and mul as a function and then make p_1 to point to the function mul. To call the function mul, we may now use the pointer p_1 with the list of parameters that is,

(∗p_1) (x, y)

is equivalent to

mul (x, y)

To obtain the address of a function, we merely enter a function name without the argument list enclosed in parentheses. e.g.

p_1 = mul; /* assign address of mul to p_1 */

we cannot assign the value to a function pointer as

```
p1 = mul ( );          /* mul returns on double but p1 is pointer */

p1 = & mul ( );    /* can not take the address of a function result */

p1 = & mul;        /* & mul is a pointer to a pointer, but p1
                      * is a pointer to an double */
```

The other important point to remember about assigning values to function pointer is that return types must agree. If we declare a pointer to a function that returns an int, we must assign the address of a function that returns an int, not the address of a function that returns some other type. If the types do not agree, we should receive a compiler time error. For example

```
extern  int   if1( ), if2( ), (*pi) ( );

extern  float   ff1( ), ff2( ), (*pf) ( );

extern  char  cf1( ), cf2( ), (*pc) ( );

/* Three pointers to functions having different data types */

main ( )

{

        pi = if1;              /* LEGAL – Types match */

        pi = cf1;              /* 1LLIEGAL – Types mismatch */

        pf = if2;              /* 1LIEGAL – Types mismatch */

        pc = cf1;              /* LEGAL – Types match */

        if1 = if2;             /* 1LLEGAL – Assign to a Constant */
```

/* Illustration of Pointers to Functions */

```
# include <math.h>

# define   P1    3.1415926

main ( )

{

        double  y( ), Cos (), table ( );
```

```
        printf (''Table of y(x) = 2 * x * –x + 1 \n\n'');
        table (y, 0.0, 2.0, 0.5);
        printf (''\n Table of Cos(x) \n\n'');
        table (Cos, 0.0, P1, 0.5);
}
double  table (f, min, max, step)
double  (*f) ( ), min, max, step;
{
        double  a, value;
        for (a = min; a < = max; a + = step)
    {
            value = (*f (a);
            printf (''% 5.2f % 10.4f \n'', a, value);
        }
}
double  y(x)
double  x;
{
        return (2* x * x – x + 1)
}
```

6.11. Recursion : When a called function in turn calls another function a process of 'chaining' occurs. Recursion is a special case of this process, where a function calls itself. For example

```
main ( )
{
        printf (''  This is an example of recursion \n'');
        main ( );
}
```

Another useful example of recursion is evaluation of factorials. e.g.

```
factorial (n)
```

```
int   n;
{
        int   fact;
        if (n = = 1)
                return (1);
        else
                fact = n* factorial (n–1);
        return (fact);
}
```

Here if n = 3, the sequence of operations can be summarized as

fact = 3* factorial (2)

= 3 * 2 * factorial (1)

= 3 * 2 *1 = 6

Recursive functions can be effectively used to solve problems where the solution is expressed in terms of successively applying the same solution to subsets of the problem. When we write recursive functions, we must have an if statement somewhere to force the function to return without the recursive call being executed. Otherwise, the function will never return.

6.12. The main( ) Function : All C programs must contain a function called main ( ), which is always the first function executed in a C-program. When main( ) returns, the program is done. The compiler treats the main( ) function like any other function, except that at runtime the best environment is responsible for providing two arguments. That is main( ) can take two arguments called **argc** and **argv**. The variable argc is an argument counter that counts the number of arguments on the command line. The argv is an argument vector and represents an array of character pointers that point to the command line arguments. The size of this array will be equal to the value of argc.

The **Command line argument** is a parameter supplied to a program when the program is invoked. This parameter may represent a filename, the program should process. For example if we want to execute a program to a copy the contents of a file named Sham to another named Hooda, then we may use a command line as

C > PROGRAM Sham Hooda

PROGRAM is the file name where the executable code of the program is stored.

For instance for the above command line, argc is three and argv is an array of three pointers to strings as

argv[0] → PROGRAM

argv[1] → Sham

argv[2] → Hooda

In order to access the command line arguments, we must declare the main function and its parameters as follows

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

6.13. Complex Declarations : Declaration in C have a tendency to became complex, making difficult to determine exactly what is being declared. For instance, the following declaration, declares x to be a pointer to be a function returning a pointer to a 5-element array of pointers to ints,

```
int  *(* (*x) ( )) [5];
```

The main reason that complex declarations look so forbidding in C is that the pointer operator is a prefix operator, where as the array and function operators are postfix operators. As a result, the variables becomes sandwiched between operators. To compose and decipher complex declarations, we must proceed inside-out adding asterisks to the left of the variable name and parentheses and brackets to the right of the variable name. It is also remember (i) That the array operator ([]) and function operator ([]) have higher precedence than the pointer operator (*)

(ii) The array and function operators group from L→R, whereas the pointer groups from R→L.

For example, the declaration char *x[] ;

would be deciphered through the following steps.

(i) x[] is an array

(ii) ∗x[] is an array of pointers

(iii) char ∗x[] is an array of pointers to char's.

Note that, in the absence of parentheses to affect binding, we would add all of the function and array operators on the right side of the variable name first (since they have higher precedence) and then add the pointer operators on the left side (as above). Parentheses can be used to change the precedence order. For example int (∗ x[]) ( );

would be decomposed as follows

(i) x[] is an array

(ii) (∗x[]) is an array of pointers

(iii) (∗x[]) ( ) is an array of pointers to functions

(iv) int (∗x[]) ( ) is an array of pointers to functions returning ints.

This declaration had been written without parentheses as

int ∗x [] ( ); , translated as

an array of functions returning pointers to ints.

which is illegal, since array of functions are invalid.

To compose a declaration, we perform the same process. For example to declare a pointer to an array of pointers to functions that return pointers to arrays of structures with tag name S is written as

struct S(∗(∗(∗x) []) ( )) [] .

UNIT - V

1. C Preprocessor

One more unique feature of the C language is the preprocessor which provides several tools that are not available in other HLL's. These tools make the program easy to read, easy to modify, portable and more efficient.

We may think of the C preprocessor as a separate program that processes the source code before it passes through the compiler. It has its own simple, line-oriented grammar and syntax. It operates under the control of what is known as **preprocessor command line** or **directive.** All preprocessor directives begin with a pound sign (#), which must be the first nonspace (i.e. in column one) character on the line and they do not require a semicolon at the end. We have already introduced two preprocessor directives i.e. the # define command for naming a constant and the # include command for including additional source files.

A set of ANSI preprocessor directives and their functions is given below :

Directive	**Function**
# define	Defines a macro substitution
# elif	Analogous to else if construct
# else	Specifies alternatives when # if test fails
# end	if Specifies the end of # if
# error	Stops compilation when an error occurs

# if	Tests a compile-time condition
# if	def Tests for a macro definition
# ifndef	Tests whether a macro is not defined
# include	Specifies the lines to be included
# line	Changes the compiler's knowledge of the current line number of the source file
# pragma	Performs implementation specific tasks
# undef	Undefines a macro

The ANSI standard also includes two new preprocessor operators given below.

#	Stringinzing operator →	Converts a formal argument within a macro definition to a string.
##	Tokenpasting operator→	Combines two tokens within a macro definition to form a single token

On the basis of their utility, the above listed directives can be divided into three categories.

(i) **Macro substitution directives** that allow us macro processing.

(ii) **File inclusion directives** which enable inclusion of addition C source files.

(iii) **Compiler control directives** which allow conditional compilation i.e. which enable us to conditionally compile sections of C source contingent on the value of an arithmetic expression.

Let us discuss the preprocessor directives in detail.

1.1. Macro Substitution : A macro is a name that has an associated text string, called the **macro body.** Macro names that represent symbolic constants, by convention, should consist of upper case letters only. This makes it easy to distinguish macro names from variable names which are composed of lowercase characters. According to ANSI standard, macro names are unique upto at least 31 characters.

Macro substitution is a process where an identifier (macro name) in a program is replaced by a predefined string (macro body) composed of one or more tokens. (In a C program the smallest individual units are known as C tokens). The preprocessor does this task under the direction of # define statement. This statement, usually known as a macro definition or simply a macro, takes the following general form

```
# define  identifier  string          | No semicolon at the end
```

The simplest working of this statement has already been discussed in Unit-I. There are different forms of macro substitution.

The most common forms are

(i) Simple macro substitution.

(ii) Argumented macro substitution

(iii) Nested macro substitution

1.2. Simple Macro Substitution : Simple string replacement is commonly used to define constants. Examples of definition of constants are

```
# define    FALSE          0
# define    COUNT          1000
# define    NULL           0
# define    PAPERS         5
# define    CAPITAL        "DELHI"
# define  MAX_PAGE_WIDTH   50
# define    PI             3.1415
```

In such cases, the macro name will be replaced at all its occurrences, starting from the line of definition to the end of the program, by the macro body. However, a macro inside a string does not get replaced. For example, if we have

```
# define  MAX   20
```

and then the two lines

```
total = MAX * value;
printf (''MAX = % d\n'', MAX);
```

These two lines would be changed during processing as follows

```
total = 20 * value;
printf (''MAX = %d\n'', 20);
```

We note that the string ''MAX = %d/n'' remains unchanged.

A macro definition can include more than a simple constant value. It can include expressions as well. e.g.

```
# define    SIZE        sizeof (int)*5
# define    FOUR_PI     4.0 * 3.1415
# define    VOLUME      5.9 * 2.45 * 7
```

Whenever we use expressions for replacement, care should be taken to prevent an unexpected order of evaluation. For example, consider the evaluation of

```
ratio = A/B;
```

where A and B are macros defined as

```
# define    A    24–15
# define    B    28+35
```

The result of the preprocessor's substitution for A and B is

```
ratio = 24–15/28+35;
```

which is not the expected expression

```
(24–15)/(28+35)
```

Correct result can be obtained by using the parentheses around the strings as

```
# define    A    (24–15)
# define    B    (28+35)
```

Thus, we should use parentheses for all expressions used in macro definitions. Since the preprocessor performs a literal text substitution wherever the defined name occurs, we can use a macro to define almost anything. For example, we can use the definitions

```
# define    COMPARE    if (x > y)
# define    AND
# define    PRINT      printf(″Valid case.\n″);
```

to build the statement

```
COMPARE AND PRINT
```

The preprocessor will translate this line to

```
if(x > y) printf (″Valid case.\n″);
```

In order to avoid mistakes and confusions, such as using the token = in place of the token = = , we may use the following few definitions to make the program error-free and more readable

```
# define    START        main ( ) {
# define    EQUALS       = =
# define    NOT EQUAL    ! =
# define    AND          &&
# define    OR           ||
# define    END          }
# define    MOD          %
```

```
# define        BLANK LINE        printf (′′\n′′);
# define        INCREMENT         ++
# define        DECEREMENT        – –
```

and so on.

An example using these replacement is as follows

```
START
……
…….
if (sum EQUALS 500 AND mean EQUALS 50)\

     INCREMENT count;
……..
……..
END
```

1.3. Argumented Macro Substitution : Upto now, we have discussed a simple form of a macro, in which the macro serves as a name for a constant. There is another form of macros that is similar to a C function in that it takes arguments that can be used in the macro body. It has the form

define identifier (f_1, f_2,…, f_n) string

The identifiers f_1, f_2,…, f_n are the formal macro arguments that are analogous to the formal arguments in a function definition. It should be noted that there is no space between the macro identifier and the left parentheses.

The subsequent occurrence of a macro with arguments is known as a **macro call** (similar to a function call). When a macro is called, the preprocessor substitutes the string, replacing the formal parameters with the actual parameters. In such cases, the string behaves like a template (shape former). For example, let us consider

define CUBE(x) (x ∗ x∗ x)

Later in the program, if we have

volume = CUBE (side);

then the preprocessor would expand this statement as

volume = (side ∗ side ∗ side);

Further, if we have the statement

volume = CUBE (a + b);

then this would expand to

volume = (a+b ∗ a+b ∗ a+b);

Obviously, this does not produce the correct result. This is because the preprocessor performs a blind text substitution of the argument a+b in place of x. Such drawback can be removed by using

```
# define    CUBE(x)    ((x) * (x) *(x))
```

which gives the correct result as

```
volume = ((a+b) * (a+b) *(a+b));
```

Thus, we should use separate parentheses for each formal argument to ensure correct binding when the macro is expanded. Some commonly used definitions are

```
# define    ABS(x)          (((x) > 0)? (x) : (–(x)))
# define    MAX(a, b)       (((a) > (b)) ? (a) : (b))
# define    MIN (a, b)      (((a) < (b)) ? (a) : (b))
# define    STREQ (s1, s2)  (strcmp ((s1), (s2)) =  = 0)
# define    STRGT (s1, s2)  (strcmp ((s1), (s2)) > 0)
```

The argument supplied to a macro can be any series of characters. For example, the definition

```
# define    PRINT (variable, format)    printf (″variable
                                        = % format\n″, variable)
```

can be called by

```
PRINT(price × quantity, f);
```

The preprocessor will expand it as

```
printf (″price × quantity = % f\n″, price × quantity);
```

We note that the actual arguments are substituted for formal arguments in a macro call, although they are within a string. This definition can also be used for printing integers and character strings.

In general, macros execute more quickly than functions because there is no function overhead involved in copying arguments and maintaining stack frames. Therefore, when trying to speed up slow programs, we should be on the lookout for small, heavily used functions that can be implemented as macros. Converting functions to macros will have a noticeable impact on execution speed only if the function is called frequently.

From operational point of view, the macro CUBE may seem identical to the following function cube ( ).

```
int   cube (x)
int   x;
```

```
{
    return x * x * x;
}
```

In the function version cube ( ), we must pass an integer value and the function must return an int. In the macro version CUBE, we can use any type of value for x (int, float etc) i.e. there is no type checking for macros, and the result may be of that type. We have observed that it is extremely difficult to write a function that works for all data types. Thus, the lack of type checking for macro and macro arguments can be a powerful feature if used with care.

1.4. Macros vs. Functions : We observe that macros and functions are similar in that they both enable a set of operations to be represented by a single name. Sometimes it is difficult to decide whether to implement an operation as a macro or as a function. The following points sum up the advantages and disadvantages of macros compared to functions.

1.5. Advantages : (i) Macros are faster than functions as they avoid the function call overhead.
(ii) The number of macro arguments is checked to match the definition i.e. for macros, the preprocessor checks to make sure that the number of arguments in the definition is the same as the number of arguments in the macro call.
(iv) No type restriction is applied to arguments so that one macro may serve for several data types.

1.6. Disadvantages : (i) Macro arguments are re-evaluated at each mention in the macro body, which can lead to unexpected results if an argument contains side effects.

(ii) Function bodies are compiled once so that multiple calls to the same function can share the same code without repeating it each time. On the other hand, macros are expanded each time they appear in a program. Due to this, a program with many large macros may be longer than a program that uses functions in place of the macros.
(iii) It is more difficult to debug programs that contain macros because the source code goes through an additional layer of translation, making the object code even further removed from the source code.

1.7. Nested Macro Substitution : We can use one macro in the definition of another macro i.e. macro definitions may be nested. e.g.

```
# define    N           10
# define    M           N+5
# define    SQUARE (x)  ((x) * (x))
# define    CUBE (x)    (SQUARE(x) *(x))
# define    FIFTH       (SQUARE(x) * CUBE(x))
```

The preprocessor expands each # define macro until no more macros appear in the text. Thus, the last definition, in the above cases, is first expanded as

((SQUARE (x)) * (SQUARE(x) * (x))

and then as

(((x) * (x)) * (((x) *(x)) * (x)))

which finally results to x^5

Macros can also be used as parameters of other macros. e.g., given the definitions of M and N, we can define the macro to give the maximum of these two, as

define MAX (M, N) (((M) > (N) ? (M) : (N))

We make the conclusion that macros can be nested in much the same fashion as functions and further, macro calls are nested similar to the function calls.

1.8. Removing a Macro Definition : A defined macro retains its meaning until the end of the source file or until it is undefined (removed) with a # undef directive, using the statement

undef identifier

This is useful when we want to restrict the definition only to a particular part of the program or we want to redefine the macro by removing the previous definition.

1.9. Built in Macros : The ANSI standard defines five macro names that are built into the preprocessor. They are termed as Built-in Macros (or Predefined Names). Each of these macro names begins and ends with two underscore characters. We can not redefine or undefine these macros. These macros may not be supported by older compilers. They are as follows:

Macro	**Expansion**
LINE	Expands to the source file line number on which it is invoked
FILE	Expands to the name of the file in which it is invoked
TIME	Expands to the time of program compilation i.e. string of the form ″h:mm:ss″.
DATE	Expands to the date of program compilation i.e. string of the form ″mm–dd–yyyy″
STDC	Expands to the constant 1 if the compiler conforms to the ANSI standard.

1.10. Special Preprocessor Operators : The macro replacement process is influenced by two special operators. These two preprocessor operators are the symbols # and ##, known respectively as stringizing (string producer)

operator and token pasting operator. The operator # adds the quotes round the macro argument.

For example, the following macro defines an easier method instead of the printf statement to print single strings

```
#define P(str) printf (# str)
```

After this, the statement

```
P(Hello);
```

will translate to

```
printf(″Hello″);
```

Thus the operator has converted the formal argument within a macro into a string. On the other hand, the definition

```
#define P(str) printf(″str″)
```

would not work, since the identifier str inside the quotes is not substituted. Further, we note that the ANSI standard also stipulates that adjacent strings will be concatenated.

The operator ## concatenates two arguments passed to it. For example, let us consider the macro definition

```
# define cat(a, b) a ## b
```

After this, the statement

```
cat(Hello, India)
```

will result to Hello-India

Thus the operator combines two tokens within a macro definition to form a single token.

Again, consider the macro definition

```
# define print(i) printf(″a″ #i ″=%f ″, a ## i)
```

This macro will convert the statement

```
print(5);
```

into the statement

```
printf (″a5 = %f ″, a5)
```

1.11. File Inclusion (Include Facility) : The C runtime library contains a number of header files that must be selectively included in the program in order to invoke the associated functions. This is achieved by # include directive which has already been explained in until-I.

Further, nesting of included files is allowed i.e. an included file can include other files. However, a file cannot include itself. Also, if an included file is not found, an error is reported and compilation is terminated.

2. Compiler Control Directives (Conditional Compilation)

Parts of a program may be compiled conditionally. The C preprocessor offers a feature known as conditional compilation which can be used to switch on or off a particular line or group of lines in a program. The preprocessor commands for such action are as follows.

(a) # ifdef identifier

By using this directive, alongwith the #endif directive, we can conditionally leave out parts of the code. For example, let us consider the following segment of a program

```
if(a = = b)
{
 # ifdef    DEBUG
        printf(″a and b are equal.\n″);
 # endif
        value = true;
}
```

At the beginning of the file, we can place a directive such as

```
# define      DEGBUG
```

that will cause the printf statement to execute. The printf can be prevented from executing by just removing this directive. It is good practice to place such statements in the code. When the code is ready and bug-free, remove the DEBUG directive on top and recompile.

(b) # if constant expression

This is an alternative way of conditionally executing a code. For example, the ifdef statement given above can be replaced by

```
# if        defined        DEBUG
```

A more general form of test condition, # if directive, has the following form

```
# if constant-expression
{
        statement 1;
        statement 2;
```

```
…….
…….
}
# endif
```

The constant expression may be any logical expression such as

```
VALUE < = 5
(COUNT = = 1  || COUNT = = 2)
MACHINE = = 'IBM'
```

If the result of the constant-expression is non zero (true), then all the statements between the # if and # endif are included for processing, otherwise they are skipped. The names VALUE, COUNT etc may be defined as macros.

(c) # ifndef identifier

This is opposite of the # ifdef directive. While the # ifdef directive includes the code if the identifier is defined before, # ifndef includes it is the identifier has not been defined before.

(d) # else

This works with any of the # if, # ifdef, #ifndef directives. It is analogous to the else statement associated with the if statement. Note that the #if and # else are processed by the preprocessor but not converted into machine code whereas the if and else statements are processed by the compiler and are translated into actual machine code. For example, let us consider

```
# ifdef    IBM–PC
        group–A lines
# else
        group–B lines
# endif
```

Here, group–A lines are included if IBM–PC is defined, otherwise, group–B lines are included.

(e) # elif constant-expression

This statement is analogous to the else if construct. Using this, a witch-case construct can be constructed for preprocessing purposes.

(f) # endif

This ends a block of statements and serves to signal the end of one of the preceding directives.

In addition to the above discussed directives, we have two more ANSI directives as follows.

2.1. # pragma Directive : It is an implementation oriented directive that allows us to specify various instructions to be given to the compiler. It takes the form

```
# pragma    name
```

where name is the name of the pragma we want. For example, a compiler might support the name no_size_effects which informs the compiler that it need not worry about side effects for a certain block of statements. We write

```
# pragma        no_side_effects
```

Under Microsoft C,

```
# pragma        loop_opt (on)
```

causes loop optimization to be performed.
In the Borland C compiler,

```
# pragma        argsused
```

will suppress the warning 'Parameter not used' for the function that follows.

2.2. # error Directive : This directive enables us to report errors during the preprocessing stage of compilation. It is used to produce diagnostic messages during debugging. It takes the form

```
# error       error message
```

when the # error directive is encountered, it displays the error message and terminates processing. For example, if a program is written following the ANSI standards, then anyone compiling the program by using K & R compiler will get cryptic errors. To avoid this, we place the following at the beginning of the program

```
# ifndef        _STDC_
# error         This program requires ANSI C
# endif
```

Every ANSI C compiler will compile the source code after defining the identifier _STDC_.
Typically, the # error directive is used to check for illegal conditional compilation values. For example, suppose we use

```
# if      DATASIZE < 10
# error       DATASIZE too small.
# endif
```

If we attempt to compile a file with DATASIZE = 8, we will receive the error message

DATASIZE too small.

3. Line Control

The ANSI standard defines a preprocessor directive called # line which allows us to change the compiler's knowledge of the current line number of the source file and the name of the source file. The syntax for #line is as follows

line line number ″file name″

where "file name" is optional. If the quoted file name is absent, then the existing fine name does not change. The line number that we enter, represents the line number of the next line in the source file. Most compilers use this number when they report an error and source–level debuggers make use of line numbers. The following example illustrates the behaviour of #line.

```
/* Example of # line preprocessor directive */
main ( )
{
    printf (″Current line : %d\n Filename : % s\n\n″,
                        _LINE_, _FILE_);
# line 100
    printf (″Current line :%d\n Filename : % s\n\n″,
                        _LINE_, _FILE_);
# line 200 ″new_name″
    printf (″Current line : %d\n Filename : % s\n\n″,
                        _LINE_, _FILE_);
    exit (0);
}
```

Assuming that the source file for this program is named as line_example.c, its execution gives

```
Current line : 6
Filename : line_example.c
Current line : 101
Filename : line_example.c
```

Current line : 201

Filename : new_name

The preprocessor evaluates _LINE_ before deleting comments. However, if an #include directive appears before the occurrence of _LINE_, the preprocessor inserts the include file before computing the value of _LINE_.

The # line feature is particularly useful for programs that produce C source text.

4. Input and Output

I /O facilities are not part of the C language itself. Operating systems vary greatly in the way they allow to access data in files and devices. This variation makes it very difficult to design I/O capabilities that are portable from one implementation of a programming language to another. The C language performs I/0 through a large set of runtime routines. In the ANSI library, all I/O functions are buffered, although we have the capability to change the buffer size. In addition, the ANSI I/0 functions make a distinction between accessing files in binary mode and accessing them in text mode. In UNIX environments, this distinction is disputed because the UNIX environments, this distinction is disputed because the UNIX operating system treats binary and text files the same. In some other operating systems, the distinction is extremely important. Further, the UNIX library performs unbuffered I/O.

The standard library contains nearly forty functions that perform I/O operations (The working of each function is described in detail in Appendix A, page 431 to 460 of Peter A. Darnell's Book). Here, we describe some general informations about these functions. We use the ANSI standard as the basis of our discussion.

4.1. Streams : C makes no distinction between devices such as terminal or tape drive and logical files located on a disk. In all cases, I/O is performed through streams. A stream is a source or destination of data that may be associated with files or devices. It consists of an ordered series of bytes. We can think of it as a one-dimensional array of characters. Reading and writing to a file or device involves reading data from the stream or writing data onto the stream. We must associate a stream with a file or device to perform I/O operations. We do this by declaring a pointer to a structure type called FILE. The FILE structure, which is defined in the stdio.h header file, contains several fields to hold such information as the file's name, its access mode, and a pointer to the next character in the stream. These fields are assigned values when we open the stream and access it, but they are implementation dependent, so they vary from one system to another.

The FILE structures provide the operating system with book keeping information, but our only means of access to the stream is the pointer to the

FILE structure, called a **file pointer.** The file pointer, which we must declare in our program, holds the stream identifier returned by the fopen ( ) function. We use the file pointer to read from, write to, or close the stream. A program may have more than one stream open simultaneously, however each implementation imposes a limit on the number of concurrent streams.

One of the fields in each FILE structure is a **file position indicator** which points to the byte where the next character will be read from or written to. As we read from and write to the file, the operating system adjusts the file position indicator to point to the next byte. However, we cannot directly access the file position indicator, we can fetch and change its value through library functions, thus enabling us to access a stream in non serial order.

We should not get confused with the file pointer and the file position indicator. The file pointer identifies an open stream connected to a file or device whereas the file position indicator refers to a specific byte position within a stream.

4.2. Standard Streams : When a program begins execution, the three streams stdin, stdout and stderr are automatically already open. Usually, these streams point to our terminal but many operating systems permit us to redirect them. For example, we may want error messages written to a file instead of the terminal. The I/O functions such as printf ( ) and scanf ( ), which we have already introduced, use these default streams. printf ( ) writes to stdout and scanf ( ) reads from stdin. We can use these functions to perform I/O to files by making stdin and stdout point to files with freopen ( ) function. However, an easier method is to use the equivalent functions, printf ( ) and fscanf ( ), which enable us to specify a particular stream.

4.3. Text and Binary Streams : Data can be accessed in one of the two formats, text and binary. Thus as a result, ANSI library supports text streams and binary streams. On some systems, such as UNIX, these are identical. A text stream consists of a series of lines, where each line is terminated by a new line character. An environment may need to convert a text stream to or from some other representation, such as '\n' to carriage return and linefeed. We should be very careful when performing textual I/O, since programs that work on one system may not work exactly the same way on another.

In binary stream, the compiler performs no interpretation of bytes. It simply reads and writes bits exactly as they appear. Binary streams are used primarily for nontextual data, where there is no line structure and it is important to preserve the exact contents of the file. If we are more interested in preserving the line structure of a file, we should use a text stream. Further, the three standard streams are all opened in text mode.

4.4. Buffering : The secondary storage devices, such as disk drives and tape drives, are extremely slow. For most programs which involve I/O, the time taken to access these devices is much larger than the time CPU takes to perform operations. Therefore, it is needed to reduce the number of physical

read and write operations as much as possible. Buffering is the simplest way to do so.

A buffer is an area in the main memory where data is temporarily stored before being sent to its ultimate destination. Buffering provides more efficient data transfer because it enables the operating system to minimize accesses to I/O devices. All operating systems use buffer to read from and write to I/O devices. The operating system accesses I/O devices only in fixed-size chunks, called **blocks.** Typically, a block is 512 or 1024 bytes, although it is defined by the operating system. This means that even if we want to read only one character from a file, the operating system reads the entire block on which the character is located. For a single read operation, this is not very efficient, but suppose we want to read 1000 characters from a file. If the I/O were unbuffered, the system would perform 1000 disk seek and read operations. On the other hand, with buffered I/O, the system reads an entire block into memory and then fetches each character from memory when necessary. This saves 999 I/O operations.

The C runtime library contains an additional layer of buffering which comes in two forms as **line buffering** and **block buffering.**

In line buffering, the system stores characters until newline character is encountered, or until the buffer is filled, and then sends the entire line to the operating system to be processed. This happens when we read data from the terminal. The data is saved in a buffer until we enter a newline character. At that point, the entire line is sent to the program.
In block buffering, the system stores characters until a block is filled and then passes the entire block to the operating system. By default, all I/O streams that point to a file are block buffered. Streams that point to our terminal, such as stdin and stdout, are either line buffered or unbuffered, depending on the implementation.

The C library standard I/O package includes a **buffer manager** that keeps buffers in memory as long as possible. So if we access the same portion of a stream more than once, there is good chance that the system can avoid accessing the I/O device multiple times. It should be noted however, that this can create problems if the file is being shared by more than one process.

In both line buffering and block buffering, we can explicitly direct the system to flush the buffer at any time, with the fflush ( ) function, sending whatever data is in the buffer to its destination.

Although line buffering and block buffering are more efficient than processing each character individually, they are unsatisfactory if we want each character to be processed as soon as it is input or output. For example, we may want to process characters as they are typed rather than waiting for a newline to be entered. C allows us to tune the buffering mechanism by changing the default size of the buffer. In most systems, we can set the size to zero to turn buffering off entirely. This results to unbuffered I/O which we shall discuss later on.

4.5. The <stdio.h> Header File : To use any of the I/O functions, we must include the stdio.h header file. The I/O functions, types and macros defined in stdio.h represent nearly one third of the C library. This file contains

(i) Prototype declarations for all the I/O functions.

(ii) Declaration of the FILE structure

(iii) Several useful macro constants, including stdin, stdout and stderr.

Another important macro is EOF, which is the value returned by many functions when the system reaches the end-of-file marker. Historically, stdio.h is also where NULL, the name for a null pointer, is defined. Presently, the ANSI standard has transferred the definition of NULL to a new header file called stddef.h. Therefore, to use NULL, we must include stddef.h or we should define NULL ourself as

```
# ifndef        NULL
        # define    NULL (void *)      0
# endif
```

4.6. Error Handling : It is possible that an error may occur during I/O operations on a file. Typical error situations include the following

(i) Trying to read beyond the end-of-file mark.

(ii) Device overflow.

(iii) Trying to use a file that has not been opened.

(iv) Trying to perform an operation on a file, when the file is opened for another type of operation,

(v) Opening a file with an invalid filename.

(vi) Attempting to write to a write-protected file.

Every I/O function returns a special value if an error occurs, however, the error value varies from one function to another. Some functions return zero for an error, others return a non zero value, and some return EOF (end-of-file).

There are also two members of the FILE structure that record whether an error or end-of-file has occurred for each open stream. End-of-file conditions are represented differently on different systems. A stream's end-of-file and error flags can be checked via the feof ( ) and ferror ( ) functions respectively. In some situations, an I/O function returns the same value for an end-of-file condition as it does for an error condition. In such cases, we need to check one of the flags to see which event actually occurred. The following function checks the error and the end-of-file flags for a specified stream and returns one of the four values based on the results. The clearerr ( ) function sets both flags equal to zero. We must explicitly reset the flags with clearerr ( ) as they are not automatically reset when we read them, nor are they automatically reset to zero by the next I/O call. They are initialized to zero when the stream

is opened, but the only way to reset them to zero is with clearerr ( ). The function is as follows :

```
# include <stdio.h>
# define        EOF_FLAG    1
# define         ERR_FLAG    2
char   stream_stat (fp)
        FILE   * fp;
{
        char  stat = 0;
        if (ferror (f p))
            stat 1 = ERR_FLAG ;
        if (feof (f p))
            stat 1 = EOF_FLAG;
        clearerr ( );
        return  stat;
}
```

Here, if neither flag is set, stat will equal zero. If error is set, but not eof, stat equals 1. If eof is set, but not error, stat equal 2. If both flags are not set, stat equals 3.

4.7. Remark : In addition to the end-of-file and error flags, there is a global variable called errno that is used by a few of the I/O functions to record errors. errno is an integer variable declared in the errno.h header file. The errno variable is primarily used for math functions.

5. File Management in C

We know that a file is a place on the disk where a group of related data is stored. Like most other languages, C supports a number of functions that have the ability to perform basic file operations, which include the following

(i) opening a file

(ii) reading data from a file

(iii) writing data to a file

(iv) closing a file

There are two different ways to perform file operations in C. The first one is called low-level I/O and uses UNIX system calls. The second method is

known as the high-level I/O and uses functions in C's standard I/O library. Here, we shall discuss some of the important file handling functions that are available in the C library.

5.1. Opening and Closing a File : We know that the data structure of a file is defined as FILE in the library of standard I/O function definitions. Therefore, all files should be declared as type FILE before they are used. FILE is a defined data type. Before we can read from or write to a file, we must open it with the fopen ( ) function. The function fopen ( ) takes two arguments, where the first is the file name and the second is the access mode. The access mode tells what we want to do with the file. For example, we may write data to the file or read the already existing data. The general format for opening a file is as follows:

```
FILE * fp;

fp = fopen (″filename″, ″mode″);
```

The first statement declares the variable fp as a pointer to the data type FILE, where FILE is a structure that is defined in the I/O library. The second statement opens the file named filename and assigns an identifier to the FILE type pointer fp. This pointer which contains all the information about the file is subsequently used as a communication link between the system and the program. Thus, fopen ( ) returns a file pointer that we can use to access the file later in the program. The second statement also specifies the purpose of opening this file. The mode does this job. Both the filename and mode are specified as strings i.e. they should be enclosed in double quotes.

There are two sets of access modes, one for text streams and one for binary streams. The text stream modes are listed below:

Mode Name	**Purpose**
″r″	Open an existing text file for reading only. The file position indicator is initially set to the beginning of the file i.e. reading occurs at the beginning of the file.
″w″	Create a new text file for writing only. If the file already exists, it will be truncated to zero length. Writing occurs at the beginning of the file.
″a″	Open an existing text file in append mode. We can write only at the end-of-file position. Even if we explicitly move the file position indicator, writing still occurs at the end-of-file.
″r+″	Same as ″r″ except both for reading and writing
″w+″	Same as ″w″ except both for reading and writing.
″a+″	Open an existing text file or create a new one in append

mode. We can read data anywhere in the file but we can write data only at the end-of-file marker.

The binary modes are exactly the same as the text modes, except that they have b appended to the mode name. For example, to open a binary file with read access, we would use ″rb″.

The following table summarizes the file and stream properties of the fopen ( ) modes.

	″r″	″w″	″a″	″r+″	″w+″	″a+″
File must exist before open	√			√		
Old file truncated to zero length		√			√	
Stream can be read	√			√	√	√
Stream can be written		√	√	√	√	√
Stream can be written only at end			√			√

As an illustration of fopen ( ), let us consider the following function which opens a text file called test with read access.

```
# include <stddef.h>
# include < stdio.h>
FILE  * open_text ( );   /* Returns a pointer to FILE */
{                              /* struct */
      FILE  * fp;
      fp = fopen (″test″, ″r″);
      if (fp = = NULL)
            fprintf (stderr, ″Error opening file test\n″);
      return  fp;
}
```

The fopen ( ) function returns a null pointer (NULL) if an error occurs. If successful, fopen ( ) returns a non zero file pointer. The fprintf ( ) function is

just like printf ( ), except that it takes an extra argument indicating which stream the output should be sent to. In this case, we send the message to the standard I/O stream stderr. By default, this stream usually points to our terminal.

In the above example, the open_test ( ) function is written somewhat more detailed than usual. Typically, the error test is combined with the file pointer assignment, as follows.

```
if (( fp = fopen (′′test′′, ′′r′′)) = = NULL)
    fprintf (stderr, ′′Error opening file test\n′′);
```

Note that in the above statements, the parentheses around

```
fp = fopen (′′test′′, ′′r′′)
```

are necessary because = = has higher precedence than =. Without the parentheses, fp gets assigned zero or one, depending on whether the result of fopen ( ) is null pointer or a valid pointer. This is a common programming mistake and should be taken care of.

The open_test ( ) function is a little too specific to be useful since it can only open one file, called test, and only with read-only access. A more useful function, given below, can open any file with any mode.

```
# include <stddef.h>
# include <stdio.h>
FILE * open_file (file_name, access_mode)
    char * file_name, * access_mode;
{
    FILE * fp;
    if ((fp = fopen (file_name, accesss_mode)) = = NULL)
        fprintf (stderr, ′′Error opening file %s with\
          access mode % s\n′′, file_name , access_mode);

    return fp;
}
```

The above open_file ( ) function is essentially the same as fopen ( ), except that it prints an error message if the file cannot be opened.

To open test from main ( ), we can write

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

```
main ( )
{
    extern  FILE  * open_file ( );
    if ((open_file (″test″, ″r″)) = = NULL)
        exit (1);
  ……….
   ……….
}
```

We note that the header files are included in both routines. We can include them in any number of different source files without causing conflicts.

5.2. Closing a File : A file must be closed as soon as all operations on it have been completed. To close a file, we use the fclose ( ) function which takes the form

```
fclose (file_pointer);
```

i.e.

```
fclose (fp);
```

Closing a file frees up the FILE structure that fp points to so that the operating system can use the structure for a different file. This ensures that all outstanding buffers associated with the stream are flushed out. It also prevents any accidental misuse of the file. Most operating systems have a limit on the number of streams that can be open simultaneously, so it is good habit to close files after finishing the job. In any event, all open streams are automatically closed when the program terminates normally. Most operating systems will close open files even when a program aborts abnormally, but we should not depend on this behaviour.

5.3. Reading and Writing Data : After opening a file, we use the file pointer to perform read and write operations. We can perform I/O operations on three different sizes of objects and thus as a result we have the following three cases

(i) One character at a time

(ii) One line at a time

(iii) One block at a time

Each of these cases have some special features. In the following discussion, we use three ways to write a simple function that copies the contents of one file to another

One rule that applies to all levels of I/O is that we cannot read from a stream and then write to it without an intervening call to fseek ( ), rewind ( ), or fflush ( ). The same rule holds for switching from write mode to read mode. These three functions are the only I/O functions that flush the buffers.

5.4. One Character at a Time : There are four functions that read and write one character to a stream. These are as follows:

getc ( ) : A macro that reads one character from a stream

fgetc ( ) : Same as getc ( ), but implemented as a function.

putc ( ) : A macro that writes one character to a stream

fputc ( ) : Same as putc ( ), but implemented as a function.

We note that getc ( ) and putc ( ) are usually implemented as macros whereas fgetc ( ) and fputc ( ) are guaranteed to be functions. Due to being implemented as macros, getc ( ) and putc ( ) usually run much faster, but they are susceptible to side effect problems. For example, the following is a dangerous call that may not work as expected.

```
putc (′x′,  fp [i + +]);
```

If an argument contains side effect operators, we should use fgetc ( ) or fputc ( ), which are implemented as functions. Further, it should be noted that getc ( ) and putc ( ) are the only library calls for which this side effect problem applies. For the rest of the library, the ANSI standard states that if a function is implemented as a macro, its arguments may appear only once in the macro body. This restriction removes side effect problems. The following example uses getc ( ) and putc ( ) to copy one file to another.

```
# include <stddef.h>
# include <stdio.h>
# define        SUCCESS     1
# define        FAIL        0
int  copyfile (infile, outfile)
     char  * infile,  * outfile;
{
     FILE  * fp1,  * fp2 ;
      if (( fp1 = fopen (infile, ″rb″)) = = NULL)
           return  FAIL;
      if (( fp2 = fopen (outfile, ″wb″)) = = NULL
      {
           fclose (fp1);
           return  FAIL
      }
```

```
        while ( ! feof (fp1))
                putc (getc (fp1), fp2);
        fclose (fp1);
        fclose (fp2);
        return  SUCCESS;
    }
```

In the above example, we open both files in binary mode because we are reading each individual character and not concerned with the file's line structure. This function copyfile ( ) will work for all files, regardless of the type of data stored in the file. The getc ( ) function gets the next character from the specified stream and then moves the file position indicator one position Successive calls to getc ( ) read each character in a stream. When the end-of-file is encountered, the feof ( ) function returns a non zero value. Note that we cannot use the return value of getc ( ) to test for an end-of-file because the file is opened in binary mode. For example, if we write

```
int   c;
while ((c = getc (fp1)) ! = EOF)
```

the loop will exit when ever the character read has the same value as EOF. This may or may not be a true end-of-file condition. The feof ( ) function, on the other hand, is unambiguous.

5.5. One Line at a Time : We can write the above function in such a way that it reads and writes lines instead of characters. There are two line-oriented I/O functions as fgets ( ) and fputs ( ). The fgets ( ) takes three arguments and has the following form

```
char   * fgets (char  *s, int n,  FILE   * stream);
```

The three arguments have the following meanings :

s is a pointer to the first element of an array to which characters are written ,

n is an integer representing the maximum number of characters to read,

stream is the stream from which to read i.e. the file poitner.

fgets ( ) reads characters until it reaches a newline, an end-of-file, or the maximum number of characters specified. fgets ( ) automatically inserts a null character after the last character written to the array. This is why, in the following copyfile ( ) function, we specify the maximum to be one less than the array size. fgets ( ) returns NULL when it reaches the end-of-file. Otherwise, it returns the first argument. The fputs ( ) function takes two arguments and writes the array identified by the first argument to the stream identified by the second argument.

The following function illustrates how we may implement the copyfile ( ) using the line-oriented functions. We open the files in text mode because we want to access the data line by line.

```
# include <stddef.h>
# include <stdio.h>
# define        SUCCESS     1
# define        FAIL        0
# define        LINESIZE    100
int   copyfile (infile, outfile)
        char  * inflile,  * outfile;
{
        FILE   * fp1,  * fp2;
        char line (LINESIZE];
        if (( fp1 = fopen (infile, ″r″)) = = NULL)
                return  FAIL ;
        if ((fp2 = fopen (outfile, ″w″)) = = NULL)
        {
                fclose (fp1);
                return  FAIL;
        }
        while (fgets (line, LINESIZE–1, fp1) ! = NULL)
                fputs (line, fp2);
        fclose (fp1);
        fclose (fp2);
        return  SUCCESS;
  }
```

It should be noted that most compilers implement fgets ( ) and fputs ( ) using fgetc ( ) and fputc ( ), so they are not as efficient as the macros getc ( ) and putc ( ).

5.6. One Block at a Time : Now, we discuss the case when data is accessed in blocks. We can think of a block as an array. When we read or write a block, we need to specify the number of elements in the block and the size of each element. The two block I/O functions are fread ( ) and fwrite ( ). fread ( ) takes four arguments and has the form

```
size_t  fread (void  * ptr, size_t size,  size_t  nelem,  FILE  * stream);
```

where size_t is an integral type defined in stdio.h. The four arguments have the following representation ptr is a pointer to an array in which to store the data. size is the size of each element in the array. nelem is the number of elements to read. stream is the file pointer.

The fread ( ) function returns the number of elements actually read. This should be the same as the third argument unless an error occurs or an end-of-file condition is encountered.

The fwrite ( ) function is the mirror image of fread ( ). It takes the same arguments, but instead of reading elements from the stream to the array, it writes elements from the array to the stream. Now, we write the copyfile ( ) function using block I/O functions. Note that we test for an end-of-file condition by comparing the actual number of elements read (the value returned from fread ( )) with the number specified in the argument list. If they are different, it means that either end-of-file or an error condition occurred. We use the ferror ( ) function to find out which of the two possible events happened. For the final fwrite ( ) function, we use the value of num_read as the number of elements to write, since it is less than BLOCKSIZE. Further note that we have written the function in such a way that it can be modified easily. If we want to change the size of each element in the array, we need only change the typedef statement at the top of the function. If we want to change the number of elements read, we need only redefine BLOCKSIZE. The function is as follows.

```
# include <stddef.h>
# include <stdio.h>
# define        SUCCESS     1
# define        FAIL        0
# define        BLOCKSIZE   512
typedef  char  DATA ;
int  copyfile (infile, outfile)
     char  * infile,  * outfile ;
{
     FILE  * fp1,  * fp2;
      DATA block (BLOCKSIZE];
      int  num_read ;
      if (( fp1 = fopen (infile, ″rb″)) = = NULL)
      {
           printf (″Error opening file %s for input.\n″, infile);
```

```
            return  FAIL;
        }
        if (( fp2 = fopen (outfile, ″wb″)) = = NULL)
        {
            printf (″Error opening file %s for output.\n″, outfile);
            fclose (fp1);
            return  FAIL;
        }
        while (( num_read = fread (block, sizeof (DATA),
                        BLOCKSIZE, fp1)) = = BLOCKSIZE)
            fwrite (block, sizeof (DATA), num_read, fp2);
        fwrite (block, sizeof (DATA), num_read, fp2);
        fclose (fp1);
        fclose (fp2);
        if (ferror (fp1))
        {
            printf (″Error reading file %s\n″, infile);
            return  FAIL;
        }
        return  SUCCESS;
    }
```

Similar to the cases of fgets ( ) and fputs ( ), the block I/O functions are usually implemented using fgetc( ) and fputc ( ) functions, so they too are not as efficient as the macros getc ( ) and putc ( ). Further, the block sizes (512 or 1024 bytes) used for fread ( ) and fwrite ( ) functions, do not effect the number of device I/O operations performed.

6. Selecting an I/O Method

When selecting an I/O method, we take care of simplicity, efficiency and portability. From efficiency point of view, the macros getc ( ) and putc ( ) are usually fastest. However, most operating systems have very fast block I/O operations that can be even faster than getc ( ) and putc ( ). Though efficiency is important but sometimes the choice of an I/O method is based on simplicity. For example, fgets ( ) and fputs ( ) are relatively slow functions, but they are simple in use. Thus, when execution speed is not important, the version using these functions is the best.

The last consideration in choosing an I/O method is portability. In terms of deciding between character, line, or block I/O, portability does not play a role.

Portability is a major concern in choosing between text mode and binary mode. If the file contains textual data, such as source code files and documents, we should open it in text mode and access it line by line. On the other hand, if the data is numeric and does not have a clear line structure, it is best to open it in binary mode and access it either character by character or block by block.

6.1. Unbuffered I/O : We can turn off buffering. To do so, we can use either the setbuf ( ) function or the setvbuf ( ) function. The setbuf ( ) function takes two arguments, the first is a file pointer and the second is a pointer to a character array which is to serve as the new buffer. If the array pointer is a null pointer, buffering is turned off, as by the statement

```
setbuf (stdin,  NULL);
```

where the **stdin** stream is line buffered, requiring the user to enter a newline character before the input is sent to the program. The setbuf ( ) function does not return a value.

The setvbuf ( ) function is similar to setbuf ( ), but it is a bit more complicated. It takes two additional arguments that enable us to specify the type of buffering (line, block, or no buffering) and the size of the array to be used as the buffer. The buffer type should be one of the following three symbols (defined in stdio.h)

_IOFBF	block buffering
_ IOLBF	line buffering
_ IONBF	no buffering

To turn off buffering, we should write

```
stat = setvbuf (stdin,  NULL,  _IONBF,  O);
```

The setvbuf ( ) function returns a non zero value if it is successful. If due to some reason, it cannot honour the request, it returns zero.

6.2. Random Access : So for we have discussed file functions that are useful for reading and writing data sequentially. In some situations, we are interested in accessing only a particular part of a file and not in reading the other parts. This can be achieved with the help of the random access functions fseek ( ), ftell ( ) and rewind ( ).

The fseek ( ) function moves the file position indicator to a specific location in a stream. It takes the following form

```
int  fseek (FILE  * stream,  offset,  position);
```

where ‘stream’, is a pointer to the file concerned, ‘offset’ is a number or variable of type long and ‘position’ is an integer number. The ‘offset’ specifies the number of positions (bytes) to be moved from the location specified by ‘position’.

There are three choices for the 'position' arguments, all of which are designated by names defined in stdio.h as follows :

SEEK_SET the beginning of the file

SEEK_CUR the current position of the file position indicator.

SEEK_END the end-of-file position.

For example,

```
stat = fseek (fp, 5, SEEK_SET);
```

moves the file position indicator to character 5 of the stream. This will be the next character read or written. Since streams, like arrays, start at zero position, so character 5 is actually the 6^{th} character in the stream. The value returned by fseek ( ) is zero if the request is legal. If the request is illegal, it returns a non-zero value.

The ftell ( ) function takes just one argument, which is a file pointer, and returns the current position of the file position indicator. ftell ( ) is used primarily to return a specified file position after performing one or more I/O operations. This function is useful in saving the current position of a file, which can be used later in the program. It takes the following form

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

where n is a number of type long that corresponds to the current position i.e. n would give the relative offset (in bytes) of the current position. This means that n bytes have already been read or written.
rewind ( ) takes a file pointer and resets the position of the start of the file. For example, the statement

```
rewind (fp);

n = ftell (fp);
```

would assign 0 to n since the file position has been set the start of the file by rewind ( ) function. This function helps us in reading or writing a file more than once, without having to close and open the file.

7. The Standard Library for Input/Output :

The standard C library contains nearly forty functions that perform I/O operations. Some of those are described below :

fclose ( )	Closes a stream
fflush ( )	Flushes a buffer by writing out everything currently in the buffer. The stream remains open
fget ( )	Same as getc ( ), but implemented as a function
fgets ( )	Reads a string from a specified input stream
fopen ( )	Open and create a file and associate it with stream
fprintf ( )	Same as printf ( ), except that output is a specified line
fputc ( )	Writes a character to a stream

fputs ( )	Writing a string to a stream
fread ( )	Reads a block of binary data from a stream
freopen ( )	Close a stream and then reopens it for a new file
fsconf ( )	Same as scanf ( ), except data is read from a specified line
fseek ( )	Random access
ftell ( )	Returns the position of a file position indicator
fwrite ( )	Writes a block of data from a buffer to a stream
getc ( )	Reads a character from a stream
getchar ( )	Reads the next character from the standard input stream
gets ( )	Reads characters from stdin until a new line or end-of-file is encountered
Impfile ( )	Creates a temporary binary file
putc ( )	Writes a character to a specified stream
putchar ( )	Output a single character to the standard output stream
puts ( )	Outputs a string of characters to stdout
remove ( )	Deletes a file
scanf ( )	Reads one or more values from stdin, as user want
sename ( )	Renames a file
tempnam ( )	Generates a string that can be used as the name of a temporary file.
ungetc ( )	Pushes a character onto a stream. The next call of getc ( ) returns this character

There are , also, three error handling functions named clearerr ( ), feof ( ) and ferror ( ). We have already discussed these functions.

UNIT - V

1. C Preprocessor

One more unique feature of the C language is the preprocessor which provides several tools that are not available in other HLL's. These tools make the program easy to read, easy to modify, portable and more efficient.

We may think of the C preprocessor as a separate program that processes the source code before it passes through the compiler. It has its own simple, line-oriented grammar and syntax. It operates under the control of what is known as **preprocessor command line** or **directive.** All preprocessor directives begin with a pound sign (#), which must be the first nonspace (i.e. in column one) character on the line and they do not require a semicolon at the end. We have already introduced two preprocessor directives i.e. the # define command for naming a constant and the # include command for including additional source files.

A set of ANSI preprocessor directives and their functions is given below :

Directive	**Function**
# define	Defines a macro substitution
# elif	Analogous to else if construct
# else	Specifies alternatives when # if test fails
# end	if Specifies the end of # if
# error	Stops compilation when an error occurs
# if	Tests a compile-time condition
# if	def Tests for a macro definition
# ifndef	Tests whether a macro is not defined
# include	Specifies the lines to be included
# line	Changes the compiler's knowledge of the current line number of the source file
# pragma	Performs implementation specific tasks
# undef	Undefines a macro

The ANSI standard also includes two new preprocessor operators given below.

#	Stringinzing operator	→ Converts a formal argument within a macro definition to a string.
##	Tokenpasting operator	→ Combines two tokens within a macro definition to form a single token

On the basis of their utility, the above listed directives can be divided into three categories.

(iv) **Macro substitution directives** that allow us macro processing.

(v) **File inclusion directives** which enable inclusion of addition C source files.

(vi) **Compiler control directives** which allow conditional compilation i.e. which enable us to conditionally compile sections of C source contingent on the value of an arithmetic expression.

Let us discuss the preprocessor directives in detail.

1.1. Macro Substitution : A macro is a name that has an associated text string, called the **macro body.** Macro names that represent symbolic constants, by convention, should consist of upper case letters only. This makes it easy to distinguish macro names from variable names which are composed of lowercase characters. According to ANSI standard, macro names are unique upto at least 31 characters.

Macro substitution is a process where an identifier (macro name) in a program is replaced by a predefined string (macro body) composed of one or more tokens. (In a C program the smallest individual units are known as C tokens). The preprocessor does this task under the direction of # define statement. This statement, usually known as a macro definition or simply a macro, takes the following general form

```
# define  identifier  string          | No semicolon at the end
```

The simplest working of this statement has already been discussed in Unit-I. There are different forms of macro substitution.
The most common forms are

(v) Simple macro substitution.

(vi) Argumented macro substitution

(vii) Nested macro substitution

1.2. Simple Macro Substitution : Simple string replacement is commonly used to define constants. Examples of definition of constants are

```
# define      FALSE          0
# define      COUNT          1000
# define      NULL           0
# define      PAPERS         5
# define      CAPITAL        "DELHI"
# define  MAX_PAGE_WIDTH     50
```

```
# define        PI              3.1415
```

In such cases, the macro name will be replaced at all its occurrences, starting from the line of definition to the end of the program, by the macro body. However, a macro inside a string does not get replaced. For example, if we have

```
# define    MAX     20
```

and then the two lines

```
total = MAX * value;
printf (''MAX = % d\n'', MAX);
```

These two lines would be changed during processing as follows

```
total = 20 * value;
printf (''MAX = %d\n'', 20);
```

We note that the string ''MAX = %d/n'' remains unchanged.
A macro definition can include more than a simple constant value. It can include expressions as well. e.g.

```
# define        SIZE            sizeof (int)*5
# define        FOUR_PI     4.0 * 3.1415
# define        VOLUME      5.9 * 2.45 * 7
```

Whenever we use expressions for replacement, care should be taken to prevent an unexpected order of evaluation. For example, consider the evaluation of

```
ratio = A/B;
```

where A and B are macros defined as

```
# define        A       24–15
# define        B       28+35
```

The result of the preprocessor's substitution for A and B is

```
ratio = 24–15/28+35;
```

which is not the expected expression

```
(24–15)/(28+35)
```

Correct result can be obtained by using the parentheses around the strings as

```
# define        A       (24–15)
# define        B       (28+35)
```

Thus, we should use parentheses for all expressions used in macro definitions. Since the preprocessor performs a literal text substitution wherever the

defined name occurs, we can use a macro to define almost anything. For example, we can use the definitions

```
# define    COMPARE    if (x > y)
# define    AND
# define    PRINT      printf(″Valid case.\n″);
```

to build the statement

```
COMPARE AND PRINT
```

The preprocessor will translate this line to

```
if(x > y) printf (″Valid case.\n″);
```

In order to avoid mistakes and confusions, such as using the token = in place of the token = = , we may use the following few definitions to make the program error-free and more readable

```
# define    START          main ( ) {
# define    EQUALS         = =
# define    NOT EQUAL      ! =
# define    AND            &&
# define    OR             ||
# define    END            }
# define    MOD            %
# define    BLANK LINE     printf (″\n″);
# define    INCREMENT      ++
# define    DECEREMENT     – –
```

and so on.

An example using these replacement is as follows

```
START
……
…….
if (sum EQUALS 500 AND mean EQUALS 50)\
    INCREMENT count;
……..
……..
END
```

1.3. Argumented Macro Substitution : Upto now, we have discussed a simple form of a macro, in which the macro serves as a name for a constant. There is another form of macros that is similar to a C function in that it takes arguments that can be used in the macro body. It has the form

define identifier (f_1, f_2,…, f_n) string

The identifiers f_1, f_2,…, f_n are the formal macro arguments that are analogous to the formal arguments in a function definition. It should be noted that there is no space between the macro identifier and the left parentheses.

The subsequent occurrence of a macro with arguments is known as a **macro call** (similar to a function call). When a macro is called, the preprocessor substitutes the string, replacing the formal parameters with the actual parameters. In such cases, the string behaves like a template (shape former). For example, let us consider

define CUBE(x) (x ∗ x∗ x)

Later in the program, if we have

volume = CUBE (side);

then the preprocessor would expand this statement as

volume = (side ∗ side ∗ side);

Further, if we have the statement

volume = CUBE (a + b);

then this would expand to

volume = (a+b ∗ a+b ∗ a+b);

Obviously, this does not produce the correct result. This is because the preprocessor performs a blind text substitution of the argument a+b in place of x. Such drawback can be removed by using

define CUBE(x) ((x) ∗ (x) ∗(x))

which gives the correct result as

volume = ((a+b) ∗ (a+b) ∗(a+b));

Thus, we should use separate parentheses for each formal argument to ensure correct binding when the macro is expanded. Some commonly used definitions are

define ABS(x) (((x) > 0)? (x) : (–(x)))

define MAX(a, b) (((a) > (b)) ? (a) : (b))

define MIN (a, b) (((a) < (b)) ? (a) : (b))

define STREQ (s_1, s_2) (strcmp ((s_1), (s_2)) = = 0)

define STRGT (s_1, s_2) (strcmp ((s_1), (s_2)) > 0)

The argument supplied to a macro can be any series of characters. For example, the definition

```
# define        PRINT (variable, format)    printf (″variable
                                                = % format\n″, variable)
```

can be called by

```
PRINT(price × quantity, f);
```

The preprocessor will expand it as

```
printf (″price × quantity = % f\n″, price × quantity);
```

We note that the actual arguments are substituted for formal arguments in a macro call, although they are within a string. This definition can also be used for printing integers and character strings.

In general, macros execute more quickly than functions because there is no function overhead involved in copying arguments and maintaining stack frames. Therefore, when trying to speed up slow programs, we should be on the lookout for small, heavily used functions that can be implemented as macros. Converting functions to macros will have a noticeable impact on execution speed only if the function is called frequently.

From operational point of view, the macro CUBE may seem identical to the following function cube ( ).

```
int   cube (x)
int   x;
{
      return x * x * x;
}
```

In the function version cube ( ), we must pass an integer value and the function must return an int. In the macro version CUBE, we can use any type of value for x (int, float etc) i.e. there is no type checking for macros, and the result may be of that type. We have observed that it is extremely difficult to write a function that works for all data types. Thus, the lack of type checking for macro and macro arguments can be a powerful feature if used with care.

1.4. Macros vs. Functions : We observe that macros and functions are similar in that they both enable a set of operations to be represented by a single name. Sometimes it is difficult to decide whether to implement an operation as a macro or as a function. The following points sum up the advantages and disadvantages of macros compared to functions.

1.5. Advantages : (i) Macros are faster than functions as they avoid the function call overhead.
(ii) The number of macro arguments is checked to match the definition i.e. for macros, the preprocessor checks to make sure that the number of arguments in the definition is the same as the number of arguments in the macro call.
(viii) No type restriction is applied to arguments so that one macro may serve for several data types.

1.6. Disadvantages : (i) Macro arguments are re-evaluated at each mention in the macro body, which can lead to unexpected results if an argument contains side effects.

(ii) Function bodies are compiled once so that multiple calls to the same function can share the same code without repeating it each time. On the other hand, macros are expanded each time they appear in a program. Due to this, a program with many large macros may be longer than a program that uses functions in place of the macros.
(iii) It is more difficult to debug programs that contain macros because the source code goes through an additional layer of translation, making the object code even further removed from the source code.

1.7. Nested Macro Substitution : We can use one macro in the definition of another macro i.e. macro definitions may be nested. e.g.

```
# define      N            10
# define      M            N+5
# define      SQUARE (x)   ((x) * (x))
# define      CUBE (x)     (SQUARE(x) *(x))
# define      FIFTH        (SQUARE(x) * CUBE(x))
```

The preprocessor expands each # define macro until no more macros appear in the text. Thus, the last definition, in the above cases, is first expanded as

((SQUARE (x)) * (SQUARE(x) * (x))

and then as

(((x) * (x)) * (((x) *(x)) * (x)))

which finally results to x^5

Macros can also be used as parameters of other macros. e.g., given the definitions of M and N, we can define the macro to give the maximum of these two, as

```
# define      MAX (M, N)      (((M) > (N) ? (M) : (N))
```

We make the conclusion that macros can be nested in much the same fashion as functions and further, macro calls are nested similar to the function calls.

1.8. Removing a Macro Definition : A defined macro retains its meaning until the end of the source file or until it is undefined (removed) with a # undef directive, using the statement

```
# undef    identifier
```

This is useful when we want to restrict the definition only to a particular part of the program or we want to redefine the macro by removing the previous definition.

1.9. Built in Macros : The ANSI standard defines five macro names that are built into the preprocessor. They are termed as Built-in Macros (or Predefined Names). Each of these macro names begins and ends with two underscore characters. We can not redefine or undefine these macros. These macros may not be supported by older compilers. They are as follows:

Macro	**Expansion**
LINE	Expands to the source file line number on which it is invoked
FILE	Expands to the name of the file in which it is invoked
TIME	Expands to the time of program compilation i.e. string of the form ″h:mm:ss″.
DATE	Expands to the date of program compilation i.e. string of the form ″mm–dd–yyyy″
STDC	Expands to the constant 1 if the compiler conforms to the ANSI standard.

1.10. Special Preprocessor Operators : The macro replacement process is influenced by two special operators. These two preprocessor operators are the symbols # and ##, known respectively as stringizing (string producer) operator and token pasting operator. The operator # adds the quotes round the macro argument.

For example, the following macro defines an easier method instead of the printf statement to print single strings

```
#define    P(str)    printf (# str)
```

After this, the statement

```
P(Hello);
```

will translate to

```
printf(″Hello″);
```

Thus the operator has converted the formal argument within a macro into a string. On the other hand, the definition

```
#define    P(str)    printf(″str″)
```

would not work, since the identifier str inside the quotes is not substituted. Further, we note that the ANSI standard also stipulates that adjacent strings will be concatenated.

The operator ## concatenates two arguments passed to it. For example, let us consider the macro definition

```
# define    cat(a, b)    a ## b
```

After this, the statement

```
cat(Hello, India)
```

will result to Hello-India

Thus the operator combines two tokens within a macro definition to form a single token.

Again, consider the macro definition

```
# define    print(i)    printf(″a″ #i ″=%f ″, a ## i)
```

This macro will convert the statement

```
print(5);
```

into the statement

```
printf (″a5 = %f ″, a5)
```

1.11. File Inclusion (Include Facility) : The C runtime library contains a number of header files that must be selectively included in the program in order to invoke the associated functions. This is achieved by # include directive which has already been explained in until-I.

Further, nesting of included files is allowed i.e. an included file can include other files. However, a file cannot include itself. Also, if an included file is not found, an error is reported and compilation is terminated.

2. Compiler Control Directives (Conditional Compilation)

Parts of a program may be compiled conditionally. The C preprocessor offers a feature known as conditional compilation which can be used to switch on or off a particular line or group of lines in a program. The preprocessor commands for such action are as follows.

(a) # ifdef identifier

By using this directive, alongwith the #endif directive, we can conditionally leave out parts of the code. For example, let us consider the following segment of a program

```
if(a = = b)
```

```
{
 # ifdef    DEBUG
        printf(″a and b are equal.\n″);
 # endif
        value = true;
}
```

At the beginning of the file, we can place a directive such as

```
# define      DEGBUG
```

that will cause the printf statement to execute. The printf can be prevented from executing by just removing this directive. It is good practice to place such statements in the code. When the code is ready and bug-free, remove the DEBUG directive on top and recompile.

(b) # if constant expression

This is an alternative way of conditionally executing a code. For example, the ifdef statement given above can be replaced by

```
# if        defined        DEBUG
```

A more general form of test condition, # if directive, has the following form

```
# if constant-expression

{
        statement 1;
        statement 2;
        …….
        …….
}
# endif
```

The constant expression may be any logical expression such as

```
VALUE < = 5
(COUNT = = 1  || COUNT = = 2)
MACHINE = = ′IBM′
```

If the result of the constant-expression is non zero (true), then all the statements between the # if and # endif are included for processing, otherwise

they are skipped. The names VALUE, COUNT etc may be defined as macros.

(c) # ifndef identifier

This is opposite of the # ifdef directive. While the # ifdef directive includes the code if the identifier is defined before, # ifndef includes it is the identifier has not been defined before.

(d) # else

This works with any of the # if, # ifdef, #ifndef directives. It is analogous to the else statement associated with the if statement. Note that the #if and # else are processed by the preprocessor but not converted into machine code whereas the if and else statements are processed by the compiler and are translated into actual machine code. For example, let us consider

```
# ifdef    IBM–PC
        group–A lines
# else
        group–B lines
# endif
```

Here, group–A lines are included if IBM–PC is defined, otherwise, group–B lines are included.

(e) # elif constant-expression

This statement is analogous to the else if construct. Using this, a witch-case construct can be constructed for preprocessing purposes.

(f) # endif

This ends a block of statements and serves to signal the end of one of the preceding directives.

In addition to the above discussed directives, we have two more ANSI directives as follows.

2.1. # pragma Directive : It is an implementation oriented directive that allows us to specify various instructions to be given to the compiler. It takes the form

```
# pragma    name
```

where name is the name of the pragma we want. For example, a compiler might support the name no_size_effects which informs the compiler that it need not worry about side effects for a certain block of statements. We write

```
# pragma    no_side_effects
```

Under Microsoft C,

```
# pragma    loop_opt (on)
```

causes loop optimization to be performed.
In the Borland C compiler,

```
# pragma    argsused
```

will suppress the warning 'Parameter not used' for the function that follows.

2.2. # error Directive : This directive enables us to report errors during the preprocessing stage of compilation. It is used to produce diagnostic messages during debugging. It takes the form

```
# error    error message
```

when the # error directive is encountered, it displays the error message and terminates processing. For example, if a program is written following the ANSI standards, then anyone compiling the program by using K & R compiler will get cryptic errors. To avoid this, we place the following at the beginning of the program

```
# ifndef    _STDC_
# error     This program requires ANSI C
# endif
```

Every ANSI C compiler will compile the source code after defining the identifier _STDC_.
Typically, the # error directive is used to check for illegal conditional compilation values. For example, suppose we use

```
# if    DATASIZE < 10
# error    DATASIZE too small.
# endif
```

If we attempt to compile a file with DATASIZE = 8, we will receive the error message

```
DATASIZE too small.
```

3. Line Control

The ANSI standard defines a preprocessor directive called # line which allows us to change the compiler's knowledge of the current line number of the source file and the name of the source file. The syntax for #line is as follows

```
# line   line number    ″file name″
```

where "file name" is optional. If the quoted file name is absent, then the existing fine name does not change. The line number that we enter, represents the line number of the next line in the source file. Most compilers use this number when they report an error and source–level debuggers make use of line numbers. The following example illustrates the behaviour of #line.

```
/* Example of # line preprocessor directive */
main ( )
{
    printf (″Current line : %d\n Filename : % s\n\n″,
                        _LINE_,  _FILE_);
# line 100
    printf (″Current line :%d\n Filename : % s\n\n″,
                        _LINE_,  _FILE_);



# line 200 ″new_name″
    printf (″Current line : %d\n Filename : % s\n\n″,
                        _LINE_,  _FILE_);
    exit (0);
}
```

Assuming that the source file for this program is named as line_example.c, its execution gives

```
Current line : 6
Filename : line_example.c
Current line : 101
Filename : line_example.c
Current line : 201
Filename : new_name
```

The preprocessor evaluates _LINE_ before deleting comments. However, if an #include directive appears before the occurrence of _LINE_, the preprocessor inserts the include file before computing the value of _LINE_.

The # line feature is particularly useful for programs that produce C source text.

4. Input and Output

I /O facilities are not part of the C language itself. Operating systems vary greatly in the way they allow to access data in files and devices. This variation makes it very difficult to design I/O capabilities that are portable from one implementation of a programming language to another. The C language performs I/0 through a large set of runtime routines. In the ANSI library, all I/O functions are buffered, although we have the capability to change the buffer size. In addition, the ANSI I/0 functions make a distinction between accessing files in binary mode and accessing them in text mode. In UNIX environments, this distinction is disputed because the UNIX environments, this distinction is disputed because the UNIX operating system treats binary and text files the same. In some other operating systems, the distinction is extremely important. Further, the UNIX library performs unbuffered I/O.

The standard library contains nearly forty functions that perform I/O operations (The working of each function is described in detail in Appendix A, page 431 to 460 of Peter A. Darnell's Book). Here, we describe some general informations about these functions. We use the ANSI standard as the basis of our discussion.

4.1. Streams : C makes no distinction between devices such as terminal or tape drive and logical files located on a disk. In all cases, I/O is performed through streams. A stream is a source or destination of data that may be associated with files or devices. It consists of an ordered series of bytes. We can think of it as a one-dimensional array of characters. Reading and writing to a file or device involves reading data from the stream or writing data onto the stream. We must associate a stream with a file or device to perform I/O operations. We do this by declaring a pointer to a structure type called FILE. The FILE structure, which is defined in the stdio.h header file, contains several fields to hold such information as the file's name, its access mode, and a pointer to the next character in the stream. These fields are assigned values when we open the stream and access it, but they are implementation dependent, so they vary from one system to another.

The FILE structures provide the operating system with book keeping information, but our only means of access to the stream is the pointer to the FILE structure, called a **file pointer.** The file pointer, which we must declare in our program, holds the stream identifier returned by the fopen ( ) function. We use the file pointer to read from, write to, or close the stream. A program may have more than one stream open simultaneously, however each implementation imposes a limit on the number of concurrent streams.

One of the fields in each FILE structure is a **file position indicator** which points to the byte where the next character will be read from or written to. As we read from and write to the file, the operating system adjusts the file position indicator to point to the next byte. However, we cannot directly access the file position indicator, we can fetch and change its value through library functions, thus enabling us to access a stream in non serial order.

We should not get confused with the file pointer and the file position indicator. The file pointer identifies an open stream connected to a file or device whereas the file position indicator refers to a specific byte position within a stream.

4.2. Standard Streams : When a program begins execution, the three streams stdin, stdout and stderr are automatically already open. Usually, these streams point to our terminal but many operating systems permit us to redirect them. For example, we may want error messages written to a file instead of the terminal. The I/O functions such as printf ( ) and scanf ( ), which we have already introduced, use these default streams. printf ( ) writes to stdout and scanf ( ) reads from stdin. We can use these functions to perform I/O to files by making stdin and stdout point to files with freopen ( ) function. However, an easier method is to use the equivalent functions, printf ( ) and fscanf ( ), which enable us to specify a particular stream.

4.3. Text and Binary Streams : Data can be accessed in one of the two formats, text and binary. Thus as a result, ANSI library supports text streams and binary streams. On some systems, such as UNIX, these are identical. A text stream consists of a series of lines, where each line is terminated by a new line character. An environment may need to convert a text stream to or from some other representation, such as '\n' to carriage return and linefeed. We should be very careful when performing textual I/O, since programs that work on one system may not work exactly the same way on another.

In binary stream, the compiler performs no interpretation of bytes. It simply reads and writes bits exactly as they appear. Binary streams are used primarily for nontextual data, where there is no line structure and it is important to preserve the exact contents of the file. If we are more interested in preserving the line structure of a file, we should use a text stream. Further, the three standard streams are all opened in text mode.

4.4. Buffering : The secondary storage devices, such as disk drives and tape drives, are extremely slow. For most programs which involve I/O, the time taken to access these devices is much larger than the time CPU takes to perform operations. Therefore, it is needed to reduce the number of physical read and write operations as much as possible. Buffering is the simplest way to do so.

A buffer is an area in the main memory where data is temporarily stored before being sent to its ultimate destination. Buffering provides more efficient data transfer because it enables the operating system to minimize accesses to

I/O devices. All operating systems use buffer to read from and write to I/O devices. The operating system accesses I/O devices only in fixed-size chunks, called **blocks.** Typically, a block is 512 or 1024 bytes, although it is defined by the operating system. This means that even if we want to read only one character from a file, the operating system reads the entire block on which the character is located. For a single read operation, this is not very efficient, but suppose we want to read 1000 characters from a file. If the I/O were unbuffered, the system would perform 1000 disk seek and read operations. On the other hand, with buffered I/O, the system reads an entire block into memory and then fetches each character from memory when necessary. This saves 999 I/O operations.

The C runtime library contains an additional layer of buffering which comes in two forms as **line buffering** and **block buffering.**

In line buffering, the system stores characters until newline character is encountered, or until the buffer is filled, and then sends the entire line to the operating system to be processed. This happens when we read data from the terminal. The data is saved in a buffer until we enter a newline character. At that point, the entire line is sent to the program.
In block buffering, the system stores characters until a block is filled and then passes the entire block to the operating system. By default, all I/O streams that point to a file are block buffered. Streams that point to our terminal, such as stdin and stdout, are either line buffered or unbuffered, depending on the implementation.

The C library standard I/O package includes a **buffer manager** that keeps buffers in memory as long as possible. So if we access the same portion of a stream more than once, there is good chance that the system can avoid accessing the I/O device multiple times. It should be noted however, that this can create problems if the file is being shared by more than one process.

In both line buffering and block buffering, we can explicitly direct the system to flush the buffer at any time, with the fflush ( ) function, sending whatever data is in the buffer to its destination.

Although line buffering and block buffering are more efficient than processing each character individually, they are unsatisfactory if we want each character to be processed as soon as it is input or output. For example, we may want to process characters as they are typed rather than waiting for a newline to be entered. C allows us to tune the buffering mechanism by changing the default size of the buffer. In most systems, we can set the size to zero to turn buffering off entirely. This results to unbuffered I/O which we shall discuss later on.

4.5. The <stdio.h> Header File : To use any of the I/O functions, we must include the stdio.h header file. The I/O functions, types and macros defined in stdio.h represent nearly one third of the C library. This file contains

(iv) Prototype declarations for all the I/O functions.

(v) Declaration of the FILE structure

(vi) Several useful macro constants, including stdin, stdout and stderr.

Another important macro is EOF, which is the value returned by many functions when the system reaches the end-of-file marker. Historically, stdio.h is also where NULL, the name for a null pointer, is defined. Presently, the ANSI standard has transferred the definition of NULL to a new header file called stddef.h. Therefore, to use NULL, we must include stddef.h or we should define NULL ourself as

```
# ifndef        NULL
        # define    NULL (void *)       0
# endif
```

4.6. Error Handling : It is possible that an error may occur during I/O operations on a file. Typical error situations include the following

(vii) Trying to read beyond the end-of-file mark.

(viii) Device overflow.

(ix) Trying to use a file that has not been opened.

(x) Trying to perform an operation on a file, when the file is opened for another type of operation,

(xi) Opening a file with an invalid filename.

(xii) Attempting to write to a write-protected file.

Every I/O function returns a special value if an error occurs, however, the error value varies from one function to another. Some functions return zero for an error, others return a non zero value, and some return EOF (end-of-file).

There are also two members of the FILE structure that record whether an error or end-of-file has occurred for each open stream. End-of-file conditions are represented differently on different systems. A stream's end-of-file and error flags can be checked via the feof ( ) and ferror ( ) functions respectively. In some situations, an I/O function returns the same value for an end-of-file condition as it does for an error condition. In such cases, we need to check one of the flags to see which event actually occurred. The following function checks the error and the end-of-file flags for a specified stream and returns one of the four values based on the results. The clearerr ( ) function sets both flags equal to zero. We must explicitly reset the flags with clearerr ( ) as they are not automatically reset when we read them, nor are they automatically reset to zero by the next I/O call. They are initialized to zero when the stream is opened, but the only way to reset them to zero is with clearerr ( ). The function is as follows :

```
# include <stdio.h>
# define        EOF_FLAG    1
```

```
# define          ERR_FLAG    2
char   stream_stat (fp)
        FILE   * fp;
{
        char  stat = 0;
        if (ferror (f p))
            stat 1 = ERR_FLAG ;
        if (feof (f p))
            stat 1 = EOF_FLAG;
        clearerr ( );
        return  stat;
}
```

Here, if neither flag is set, stat will equal zero. If error is set, but not eof, stat equals 1. If eof is set, but not error, stat equal 2. If both flags are not set, stat equals 3.

4.7. Remark : In addition to the end-of-file and error flags, there is a global variable called errno that is used by a few of the I/O functions to record errors. errno is an integer variable declared in the errno.h header file. The errno variable is primarily used for math functions.

5. File Management in C

We know that a file is a place on the disk where a group of related data is stored. Like most other languages, C supports a number of functions that have the ability to perform basic file operations, which include the following

(v) opening a file

(vi) reading data from a file

(vii) writing data to a file

(viii) closing a file

There are two different ways to perform file operations in C. The first one is called low-level I/O and uses UNIX system calls. The second method is known as the high-level I/O and uses functions in C's standard I/O library. Here, we shall discuss some of the important file handling functions that are available in the C library.

5.1. Opening and Closing a File : We know that the data structure of a file is defined as FILE in the library of standard I/O function definitions. Therefore, all files should be declared as type FILE before they are used. FILE is a

defined data type. Before we can read from or write to a file, we must open it with the fopen ( ) function. The function fopen ( ) takes two arguments, where the first is the file name and the second is the access mode. The access mode tells what we want to do with the file. For example, we may write data to the file or read the already existing data. The general format for opening a file is as follows:

```
FILE * fp;

fp = fopen (″filename″, ″mode″);
```

The first statement declares the variable fp as a pointer to the data type FILE, where FILE is a structure that is defined in the I/O library. The second statement opens the file named filename and assigns an identifier to the FILE type pointer fp. This pointer which contains all the information about the file is subsequently used as a communication link between the system and the program. Thus, fopen ( ) returns a file pointer that we can use to access the file later in the program. The second statement also specifies the purpose of opening this file. The mode does this job. Both the filename and mode are specified as strings i.e. they should be enclosed in double quotes.

There are two sets of access modes, one for text streams and one for binary streams. The text stream modes are listed below:

Mode Name	**Purpose**
″r″	Open an existing text file for reading only. The file position indicator is initially set to the beginning of the file i.e. reading occurs at the beginning of the file.
″w″	Create a new text file for writing only. If the file already exists, it will be truncated to zero length. Writing occurs at the beginning of the file.
″a″	Open an existing text file in append mode. We can write only at the end-of-file position. Even if we explicitly move the file position indicator, writing still occurs at the end-of-file.
″r+″	Same as ″r″ except both for reading and writing
″w+″	Same as ″w″ except both for reading and writing.

″a+″ Open an existing text file or create a new one in append mode. We can read data anywhere in the file but we can write data only at the end-of-file marker.

The binary modes are exactly the same as the text modes, except that they have b appended to the mode name. For example, to open a binary file with read access, we would use ″rb″.

The following table summarizes the file and stream properties of the fopen ( ) modes.

	″r″	″w″	″a″	″r+″	″w+″	″a+″
File must exist before open	√			√		
Old file truncated to zero length		√			√	
Stream can be read	√			√	√	√
Stream can be written		√	√	√	√	√
Stream can be written only at end			√			√

As an illustration of fopen ( ), let us consider the following function which opens a text file called test with read access.

```
# include <stddef.h>
# include < stdio.h>
FILE  * open_text ( );   /* Returns a pointer to FILE */
{                              /* struct */
        FILE  * fp;
        fp = fopen (″test″, ″r″);
        if (fp = = NULL)
                fprintf (stderr, ″Error opening file test\n″);
        return  fp;
```

```
}
```

The fopen ( ) function returns a null pointer (NULL) if an error occurs. If successful, fopen ( ) returns a non zero file pointer. The fprintf ( ) function is just like printf ( ), except that it takes an extra argument indicating which stream the output should be sent to. In this case, we send the message to the standard I/O stream stderr. By default, this stream usually points to our terminal.

In the above example, the open_test ( ) function is written somewhat more detailed than usual. Typically, the error test is combined with the file pointer assignment, as follows.

```
if (( fp = fopen (′′test′′, ′′r′′)) = = NULL)
    fprintf (stderr, ′′Error opening file test\n′′);
```

Note that in the above statements, the parentheses around

```
fp = fopen (′′test′′, ′′r′′)
```

are necessary because = = has higher precedence than =. Without the parentheses, fp gets assigned zero or one, depending on whether the result of fopen ( ) is null pointer or a valid pointer. This is a common programming mistake and should be taken care of.

The open_test ( ) function is a little too specific to be useful since it can only open one file, called test, and only with read-only access. A more useful function, given below, can open any file with any mode.

```
# include <stddef.h>
# include <stdio.h>
FILE  * open_file (file_name, access_mode)
        char  * file_name,  * access_mode;
{
        FILE  * fp;
        if ((fp = fopen (file_name, accesss_mode)) = = NULL)
                fprintf (stderr, ′′Error opening file %s with\
                  access mode % s\n′′, file_name , access_mode);

        return  fp;
}
```

The above open_file ( ) function is essentially the same as fopen ( ), except that it prints an error message if the file cannot be opened.

To open test from main ( ), we can write

```
# include <stddef.h>
# include < stdio.h>
main ( )
{
      extern    FILE    * open_file ( );
      if ((open_file (′′test′′, ′′r′′)) = = NULL)
          exit (1);
   ……….
    ……….
}
```

We note that the header files are included in both routines. We can include them in any number of different source files without causing conflicts.

5.2. Closing a File : A file must be closed as soon as all operations on it have been completed. To close a file, we use the fclose ( ) function which takes the form

```
fclose (file_pointer);
```

i.e.

```
fclose (fp);
```

Closing a file frees up the FILE structure that fp points to so that the operating system can use the structure for a different file. This ensures that all outstanding buffers associated with the stream are flushed out. It also prevents any accidental misuse of the file. Most operating systems have a limit on the number of streams that can be open simultaneously, so it is good habit to close files after finishing the job. In any event, all open streams are automatically closed when the program terminates normally. Most operating systems will close open files even when a program aborts abnormally, but we should not depend on this behaviour.

5.3. Reading and Writing Data : After opening a file, we use the file pointer to perform read and write operations. We can perform I/O operations on three different sizes of objects and thus as a result we have the following three cases

(iv) One character at a time

(v) One line at a time

(vi) One block at a time

Each of these cases have some special features. In the following discussion, we use three ways to write a simple function that copies the contents of one file to another

One rule that applies to all levels of I/O is that we cannot read from a stream and then write to it without an intervening call to fseek ( ), rewind ( ), or fflush ( ). The same rule holds for switching from write mode to read mode. These three functions are the only I/O functions that flush the buffers.

5.4. One Character at a Time : There are four functions that read and write one character to a stream. These are as follows:

getc ( ) : A macro that reads one character from a stream

fgetc ( ) : Same as getc ( ), but implemented as a function.

putc ( ) : A macro that writes one character to a stream

fputc ( ) : Same as putc ( ), but implemented as a function.

We note that getc ( ) and putc ( ) are usually implemented as macros whereas fgetc ( ) and fputc ( ) are guaranteed to be functions. Due to being implemented as macros, getc ( ) and putc ( ) usually run much faster, but they are susceptible to side effect problems. For example, the following is a dangerous call that may not work as expected.

```
putc (′x′,  fp [i + +]);
```

If an argument contains side effect operators, we should use fgetc ( ) or fputc ( ), which are implemented as functions. Further, it should be noted that getc ( ) and putc ( ) are the only library calls for which this side effect problem applies. For the rest of the library, the ANSI standard states that if a function is implemented as a macro, its arguments may appear only once in the macro body. This restriction removes side effect problems. The following example uses getc ( ) and putc ( ) to copy one file to another.

```
# include <stddef.h>
# include <stdio.h>
# define        SUCCESS     1
# define        FAIL        0
int  copyfile (infile, outfile)
     char  * infile,  * outfile;
{
     FILE  * fp1,  * fp2 ;
      if (( fp1 = fopen (infile, ″rb″)) = = NULL)
           return  FAIL;
      if (( fp2 = fopen (outfile, ″wb″)) = = NULL
      {
```

```
            fclose (fp1);
            return  FAIL
        }
        while ( ! feof (fp1))
            putc (getc (fp1), fp2);
        fclose (fp1);
        fclose (fp2);
        return  SUCCESS;
    }
```

In the above example, we open both files in binary mode because we are reading each individual character and not concerned with the file's line structure. This function copyfile ( ) will work for all files, regardless of the type of data stored in the file. The getc ( ) function gets the next character from the specified stream and then moves the file position indicator one position Successive calls to getc ( ) read each character in a stream. When the end-of-file is encountered, the feof ( ) function returns a non zero value. Note that we cannot use the return value of getc ( ) to test for an end-of-file because the file is opened in binary mode. For example, if we write

```
int   c;
while ((c = getc (fp1)) ! = EOF)
```

the loop will exit when ever the character read has the same value as EOF. This may or may not be a true end-of-file condition. The feof ( ) function, on the other hand, is unambiguous.

5.5. One Line at a Time : We can write the above function in such a way that it reads and writes lines instead of characters. There are two line-oriented I/O functions as fgets ( ) and fputs ( ). The fgets ( ) takes three arguments and has the following form

```
char   * fgets (char  *s, int n,  FILE   * stream);
```

The three arguments have the following meanings :

s is a pointer to the first element of an array to which characters are written ,

n is an integer representing the maximum number of characters to read,

stream is the stream from which to read i.e. the file poitner.

fgets ( ) reads characters until it reaches a newline, an end-of-file, or the maximum number of characters specified. fgets ( ) automatically inserts a null character after the last character written to the array. This is why, in the following copyfile ( ) function, we specify the maximum to be one less than the array size. fgets ( ) returns NULL when it reaches the end-of-file.

Otherwise, it returns the first argument. The fputs ( ) function takes two arguments and writes the array identified by the first argument to the stream identified by the second argument.

The following function illustrates how we may implement the copyfile ( ) using the line-oriented functions. We open the files in text mode because we want to access the data line by line.

```
# include <stddef.h>
# include <stdio.h>
# define        SUCCESS     1
# define        FAIL        0
# define        LINESIZE    100
int   copyfile (infile, outfile)
        char  * inflile,  * outfile;
{
        FILE   * fp1,  * fp2;
        char line (LINESIZE];
        if (( fp1 = fopen (infile, ″r″)) = = NULL)
                return  FAIL ;
        if ((fp2 = fopen (outfile, ″w″)) = = NULL)
        {
                fclose (fp1);
                return  FAIL;
        }
        while (fgets (line, LINESIZE–1, fp1) ! = NULL)
                fputs (line, fp2);
        fclose (fp1);
        fclose (fp2);
        return  SUCCESS;
  }
```

It should be noted that most compilers implement fgets ( ) and fputs ( ) using fgetc ( ) and fputc ( ), so they are not as efficient as the macros getc ( ) and putc ( ).

5.6. One Block at a Time : Now, we discuss the case when data is accessed in blocks. We can think of a block as an array. When we read or write a block, we need to specify the number of elements in the block and the size of each element. The two block I/O functions are fread ( ) and fwrite ( ). fread ( ) takes four arguments and has the form

```
size_t  fread (void  * ptr, size_t size,  size_t  nelem,  FILE  * stream);
```

where size_t is an integral type defined in stdio.h. The four arguments have the following representation ptr is a pointer to an array in which to store the data. size is the size of each element in the array. nelem is the number of elements to read. stream is the file pointer.

The fread ( ) function returns the number of elements actually read. This should be the same as the third argument unless an error occurs or an end-of-file condition is encountered.

The fwrite ( ) function is the mirror image of fread ( ). It takes the same arguments, but instead of reading elements from the stream to the array, it writes elements from the array to the stream. Now, we write the copyfile ( ) function using block I/O functions. Note that we test for an end-of-file condition by comparing the actual number of elements read (the value returned from fread ( )) with the number specified in the argument list. If they are different, it means that either end-of-file or an error condition occurred. We use the ferror ( ) function to find out which of the two possible events happened. For the final fwrite ( ) function, we use the value of num_read as the number of elements to write, since it is less than BLOCKSIZE. Further note that we have written the function in such a way that it can be modified easily. If we want to change the size of each element in the array, we need only change the typedef statement at the top of the function. If we want to change the number of elements read, we need only redefine BLOCKSIZE. The function is as follows.

```
# include <stddef.h>
# include <stdio.h>
# define        SUCCESS     1
# define        FAIL        0
# define        BLOCKSIZE   512
typedef  char  DATA ;
int  copyfile (infile, outfile)
     char  * infile,  * outfile ;
{
     FILE  * fp1,  * fp2;
```

```
    DATA block (BLOCKSIZE];
    int   num_read ;
    if (( fp1 = fopen (infile, ″rb″)) = = NULL)
    {
        printf (″Error opening file %s for input.\n″, infile);
        return  FAIL;
    }
    if (( fp2 = fopen (outfile, ″wb″)) = = NULL)
    {
        printf (″Error opening file %s for output.\n″, outfile);
        fclose (fp1);
        return  FAIL;
    }
    while (( num_read = fread (block, sizeof (DATA),
                    BLOCKSIZE, fp1)) = = BLOCKSIZE)
        fwrite (block, sizeof (DATA), num_read, fp2);
    fwrite (block, sizeof (DATA), num_read, fp2);
    fclose (fp1);
    fclose (fp2);
    if (ferror (fp1))
    {
        printf (″Error reading file %s\n″, infile);
        return  FAIL;
    }
    return  SUCCESS;
}
```

Similar to the cases of fgets ( ) and fputs ( ), the block I/O functions are usually implemented using fgetc( ) and fputc ( ) functions, so they too are not as efficient as the macros getc ( ) and putc ( ). Further, the block sizes (512 or 1024 bytes) used for fread ( ) and fwrite ( ) functions, do not effect the number of device I/O operations performed.

6. Selecting an I/O Method

When selecting an I/O method, we take care of simplicity, efficiency and portability. From efficiency point of view, the macros getc ( ) and putc ( ) are usually fastest. However, most operating systems have very fast block I/O operations that can be even faster than getc ( ) and putc ( ). Though efficiency is important but sometimes the choice of an I/O method is based on simplicity. For example, fgets ( ) and fputs ( ) are relatively slow functions, but they are simple in use. Thus, when execution speed is not important, the version using these functions is the best.

The last consideration in choosing an I/O method is portability. In terms of deciding between character, line, or block I/O, portability does not play a role. Portability is a major concern in choosing between text mode and binary mode. If the file contains textual data, such as source code files and documents, we should open it in text mode and access it line by line. On the other hand, if the data is numeric and does not have a clear line structure, it is best to open it in binary mode and access it either character by character or block by block.

6.1. Unbuffered I/O : We can turn off buffering. To do so, we can use either the setbuf ( ) function or the setvbuf ( ) function. The setbuf ( ) function takes two arguments, the first is a file pointer and the second is a pointer to a character array which is to serve as the new buffer. If the array pointer is a null pointer, buffering is turned off, as by the statement

```
setbuf (stdin,  NULL);
```

where the **stdin** stream is line buffered, requiring the user to enter a newline character before the input is sent to the program. The setbuf ( ) function does not return a value.

The setvbuf ( ) function is similar to setbuf ( ), but it is a bit more complicated. It takes two additional arguments that enable us to specify the type of buffering (line, block, or no buffering) and the size of the array to be used as the buffer. The buffer type should be one of the following three symbols (defined in stdio.h)

```
_IOFBF      block buffering
_ IOLBF     line buffering
_ IONBF     no buffering
```

To turn off buffering, we should write

```
stat = setvbuf (stdin,  NULL,  _IONBF,  O);
```

The setvbuf ( ) function returns a non zero value if it is successful. If due to some reason, it cannot honour the request, it returns zero.

6.2. Random Access : So for we have discussed file functions that are useful for reading and writing data sequentially. In some situations, we are interested in accessing only a particular part of a file and not in reading the other parts. This can be achieved with the help of the random access functions fseek ( ), ftell ( ) and rewind ( ).

The fseek ( ) function moves the file position indicator to a specific location in a stream. It takes the following form

```
int fseek (FILE * stream, offset, position);
```

where 'stream', is a pointer to the file concerned, 'offset' is a number or variable of type long and 'position' is an integer number. The 'offset' specifies the number of positions (bytes) to be moved from the location specified by 'position'.
There are three choices for the 'position' arguments, all of which are designated by names defined in stdio.h as follows :

SEEK_SET	the beginning of the file
SEEK_CUR	the current position of the file position indicator.
SEEK_END	the end-of-file position.

For example,

```
stat = fseek (fp, 5, SEEK_SET);
```

moves the file position indicator to character 5 of the stream. This will be the next character read or written. Since streams, like arrays, start at zero position, so character 5 is actually the 6^{th} character in the stream. The value returned by fseek ( ) is zero if the request is legal. If the request is illegal, it returns a non-zero value.

The ftell ( ) function takes just one argument, which is a file pointer, and returns the current position of the file position indicator. ftell ( ) is used primarily to return a specified file position after performing one or more I/O operations. This function is useful in saving the current position of a file, which can be used later in the program. It takes the following form

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

where n is a number of type long that corresponds to the current position i.e. n would give the relative offset (in bytes) of the current position. This means that n bytes have already been read or written.
rewind ( ) takes a file pointer and resets the position of the start of the file. For example, the statement

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

would assign 0 to n since the file position has been set the start of the file by rewind ( ) function. This function helps us in reading or writing a file more than once, without having to close and open the file.

7. The Standard Library for Input/Output :

The standard C library contains nearly forty functions that perform I/O operations. Some of those are described below :

fclose ( )	Closes a stream
fflush ( )	Flushes a buffer by writing out everything currently in the buffer. The stream remains open
fget ( )	Same as getc ( ), but implemented as a function
fgets ( )	Reads a string from a specified input stream
fopen ( )	Open and create a file and associate it with stream
fprintf ( )	Same as printf ( ), except that output is a specified line
fputc ( )	Writes a character to a stream
fputs ( )	Writing a string to a stream
fread ( )	Reads a block of binary data from a stream
freopen ( )	Close a stream and then reopens it for a new file
fsconf ( )	Same as scanf ( ), except data is read from a specified line
fseek ( )	Random access
ftell ( )	Returns the position of a file position indicator
fwrite ( )	Writes a block of data from a buffer to a stream
getc ( )	Reads a character from a stream
getchar ( )	Reads the next character from the standard input stream
gets ( )	Reads characters from stdin until a new line or end-of-file is encountered
Impfile ( )	Creates a temporary binary file
putc ( )	Writes a character to a specified stream
putchar ( )	Output a single character to the standard output stream
puts ( )	Outputs a string of characters to stdout
remove ( )	Deletes a file
scanf ( )	Reads one or more values from stdin, as user want
sename ( )	Renames a file
tempnam ( )	Generates a string that can be used as the name of a temporary file.

ungetc ( )	Pushes a character onto a stream. The next call of getc ( ) returns this character

There are , also, three error handling functions named clearerr ( ), feof ( ) and ferror ( ). We have already discussed these functions.