CONTENT

Programme Coordinator : **Prof. Srivaramangai**
Head Dept. of I.T.
Dr. S.D. Sharma Bhavan,
Vidnagari, Mumbai-400 097.

Course Writer : **Mr. Nikhil Pawanikar**
University Dept. of I.T.
3rd Floor, Dre. S.D. Sharmar Bhavan,
Santacruz (E), Mumbai - 400 098

: **Mr. Jayesh Shinde**
University Dept. of I.T.
3rd Floor, Dre. S.D. Sharmar Bhavan,
Santacruz (E), Mumbai - 400 098

M.C.A. (Sem.I) Paper I PROGRAMMING WITH C

Reprint September, 2015

Published by : Incharge Director,
Institute of Distance and Open Education,
University of Mumbai,
Vidyanagari , Mumbai-400 098.

DTP Composed by : SHREE GRAPHIC CENTRE
28,MANGAL WADI, MUMBAI - 4

Printed By :

M.C.A. (Sem.I)

Paper I

PROGRAMMING WITH C

Syllabus

Programming with C

1. Introduction to Problem Solving
Flow charts, Tracing flow charts, Problem solving methods, Need for computer Languages, Sample Programs written in C

2. C Language preliminaries:
C character set, Identifiers and keywords, Deta types, Declarations, Expressions, statements and symbolic constants

3. Input-Output:
getchar, putchar, scanf, printf, gets, puts, functions.

4. Pre-processor commands:
#include, #define, #ifdef

5. Preparing and running a complete C program

6. Operators and expressions:
Arithmetic, unary, logical, bit-wise, assignment and conditional operators

7. Control statements
While, do-while, for statements, nested loops, if else, switch, break, Continue, and goto statements, comma operators

8. Storage types:
Automatic, external, register and static variables.

9. Functions
Defining and accessing, passing arguments, Function prototypes, Recursion, Library functions, Static functions

10. Arrays:
Defining and processing, Passing arrays to a function, Multi dimensional arrays.

11. Strings
Defining and operations on strings.

12. Pointers
Declarations, Passing pointers to a function, Operations on pointers, Pointer Arithmetic, Pointers and arrays, Arrays of pointers, function pointers.

13. Structures
Defining and processing, Passing to a function, Unions, typedef, array of structure, and pointer to structure

14. File structures:

Definitions, concept of record, file operations: Storing, creeting, retrieving, updating Sequential, relative, indexed end random access mode, Files with binary mode(Low level), performance of Sequential Files, Direct mapping techniques: Absolute, relative and indexed sequential files (ISAM) concept of Index, levels of index, overflow of handling.

15. File Handling:

File operation: creation, copy, delete, update, text file, binary file.

❖❖❖

1

INTRODUCTION TO PROBLEM SOLVING

Contents

1.1 Flow Charts

A flowchart is a type of diagram that represents an algorithm or process, showing the steps as boxes of various kinds, and their order by connecting these with arrows. This diagrammatic representation can give a step-by-step solution to a given problem. Process operations are represented in these boxes, and arrows connecting them represent flow of control. Data flows are not typically represented in a flowchart, in contrast with data flow diagrams; rather, they are implied by the sequencing of operations. Flowcharts are used in analyzing, designing, documenting or meaning a process or program in various fields.

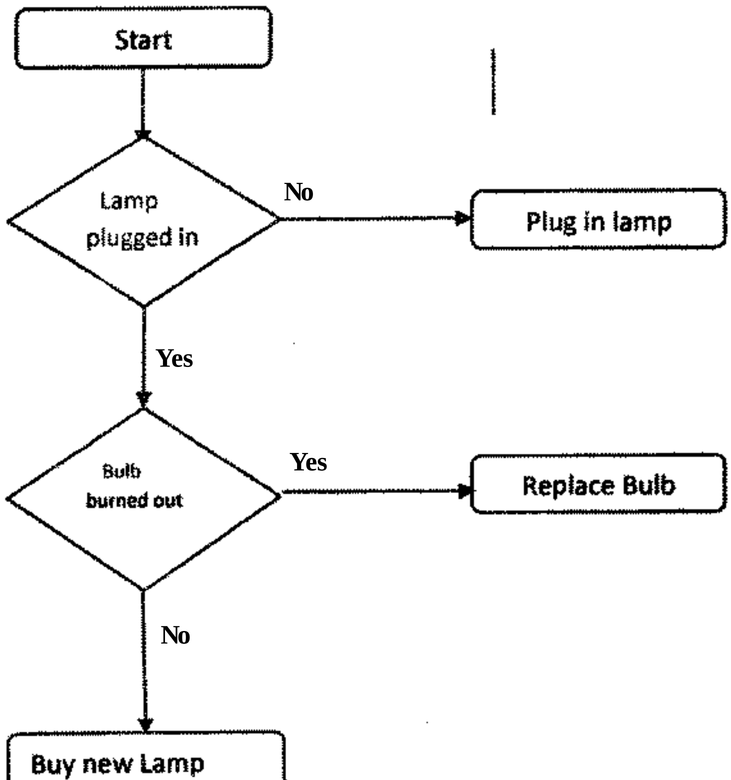


Figure: 1.1 A simple flowchart representing e process for dealing with a non-functioning lamp.

1.2 Flowchart Symbols

Symbols
A typical flowchart from older basic computer science textbooks may have the following kinds of symbols:
Start and end symbols
Represented as circles, ovals or rounded rectangles, usually containing the word "Start" or "End".

Arrows
It is known as "flow of control" in computer science. An arrow coming from one symbol and ending at another symbol represents that control passes to the symbol the arrow points to.

Generic processing steps
It's represented as rectangles. Examples: "Add 1 to X"; "replace identified part"; "save changes" or similar.

Subroutines
It's represented as rectangles with double-struck vertical edges; these are used to show complex processing steps which may be detailed in a separate flowchart.

Input/output
It's represented as a parallelogram. Examples: Get X from the user; display X.

Prepare conditional
It's represented as a hexagon. Its shows operations which have no effect other than preparing a value for a subsequent conditional or decision step.

Conditional or decision
Represented as a diamond (rhombus) showing where a decision is necessary, commonly a Yes/No question or True/False test. The conditional symbol is peculiar in that it has two arrows coming out of it, usually from the bottom point and right point, one corresponding to Yes or True, and one corresponding to No or False. (The arrows should always be labeled.) .

Junction symbol
Generally represented with a black blob, showing where multiple control flows converge in a single exit flow. A junction symbol will have more than one arrow coming into it, but only one going out.
In simple cases, one may simply have an arrow point to another arrow instead. These are useful to represent an iterative process (what in Computer Science is called a loop). A loop may, for example, consist of a connector where control first enters,

processing steps, a conditional with one arrow exiting the loop, and one going back to the connector.

Labeled connectors
Represented by an identifying label inside a circle. Labeled connectors are used in complex or multi-sheet diagrams to substitute for arrows. For eech label, the "outflow" connector must always be unique, but there may be any number of "inflow" connectors. In this case, a junction In control flow is implied.

Concurrency symbol
Represented by a double transverse line with eny number of entry and exit arrows. These symbols are used whenever two or more control flows must operate simultaneously. The exit flows are activated concurrently when all of the entry flows heve reeched the concurrency symbol. A concurrency symbol with a single entry flow is a *fork*; one with a single exit flow is a *join*.

It is importent to remember to keep these connections logical in order. All processes should flow from top to bottom and left to right.

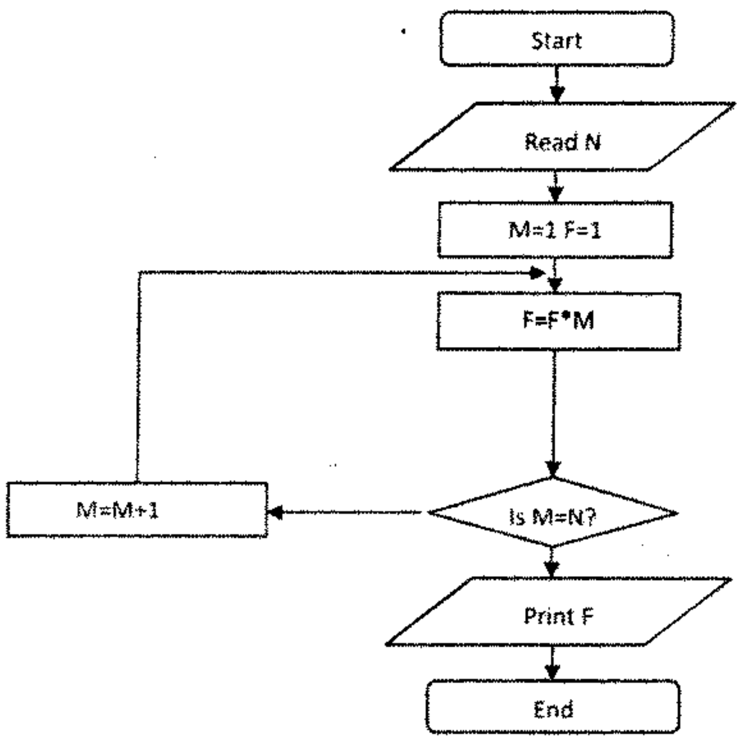


Figure: 1.2 The Flow Chart of Factoriel of given Number

1.3 Need for Computer Languages

The flowchart is essential for drawing a diagram/Symbols based on the certain algorithm; here given certain example of looping structure and Conditional statement.

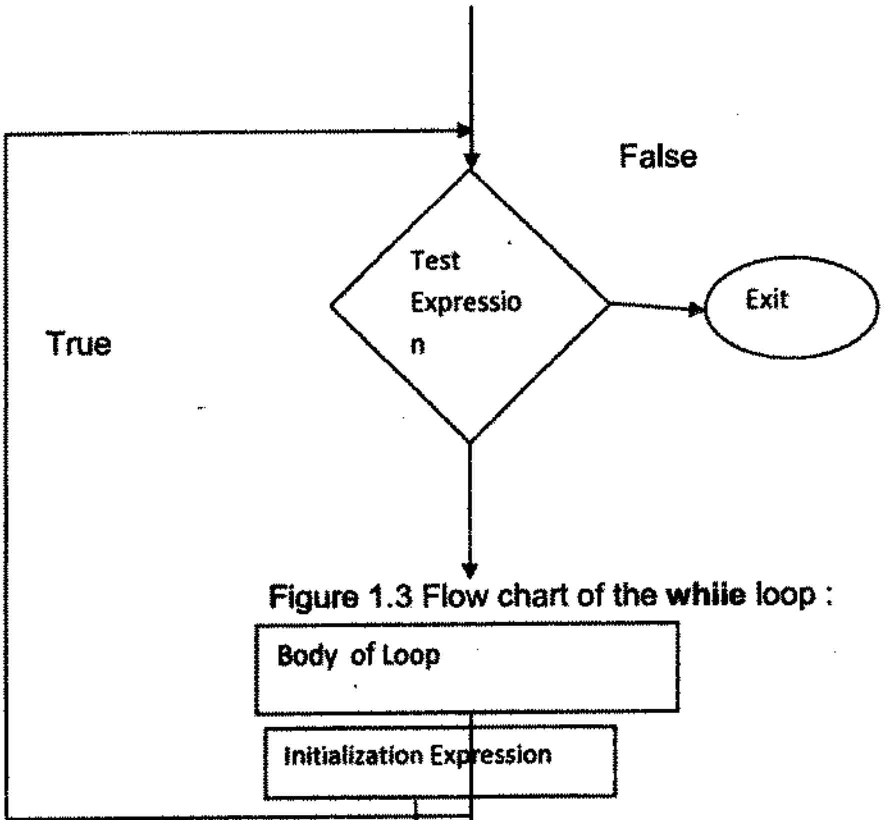


Figure 1.3 Flow chart of the **while** loop :

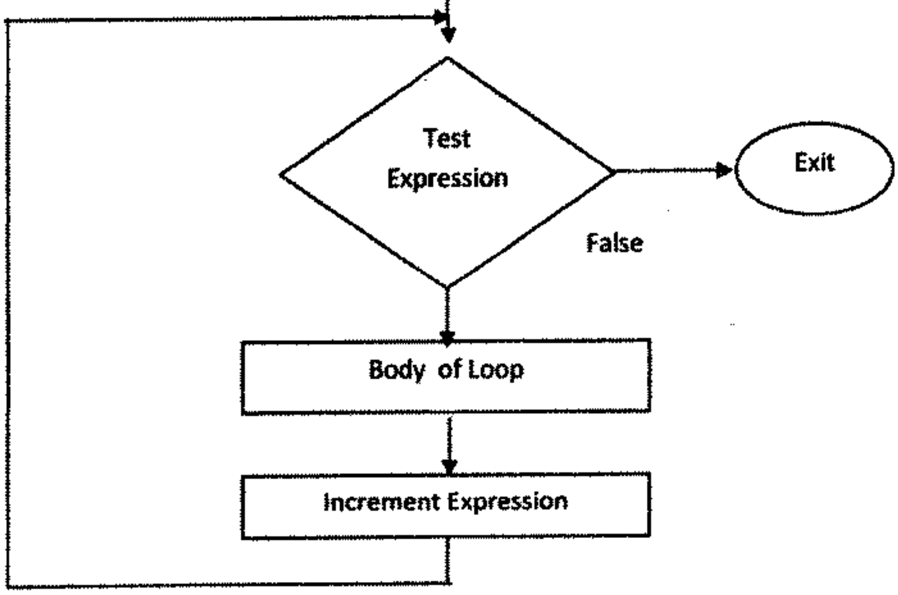


Figure 1.4 Flow chart of the **for** loop:

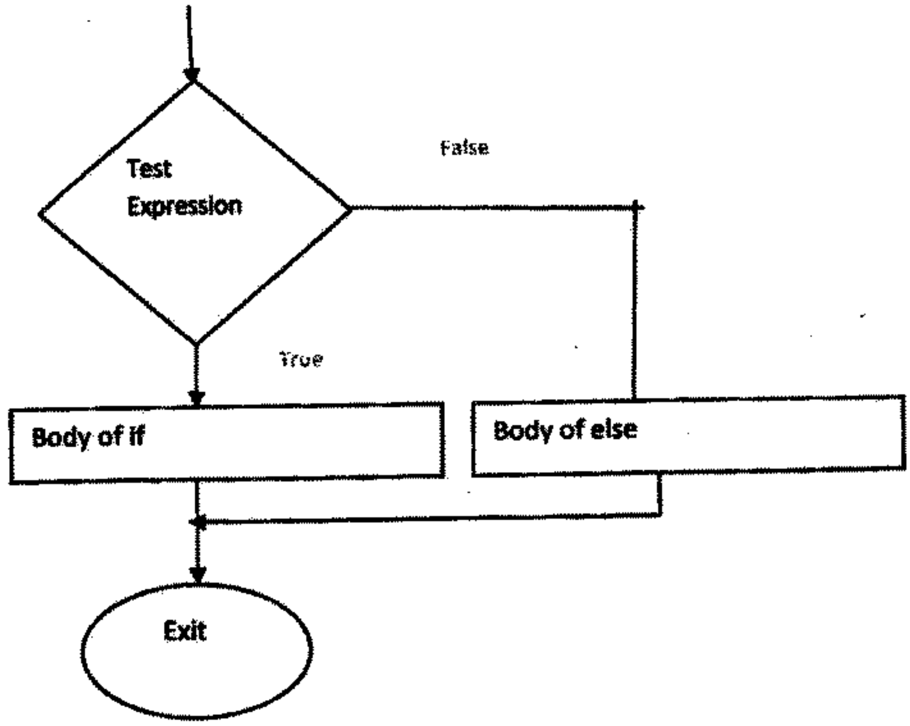
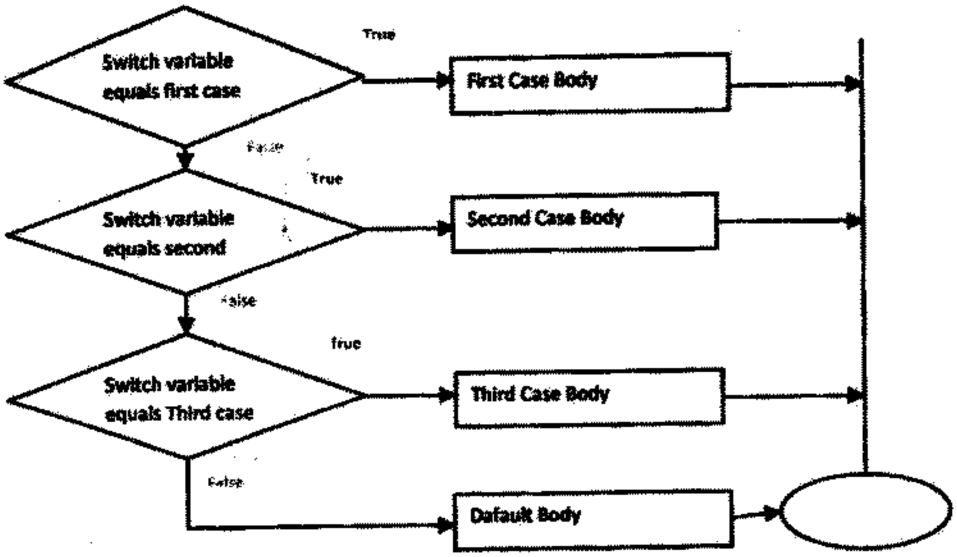


Figure 1.5 The flow chart of the **if...else** statement:



The flow chart of the switch statement:

Flowchart (Example):

Figure 1.6 Flowchart to find the sum of first 50 natural numbers.

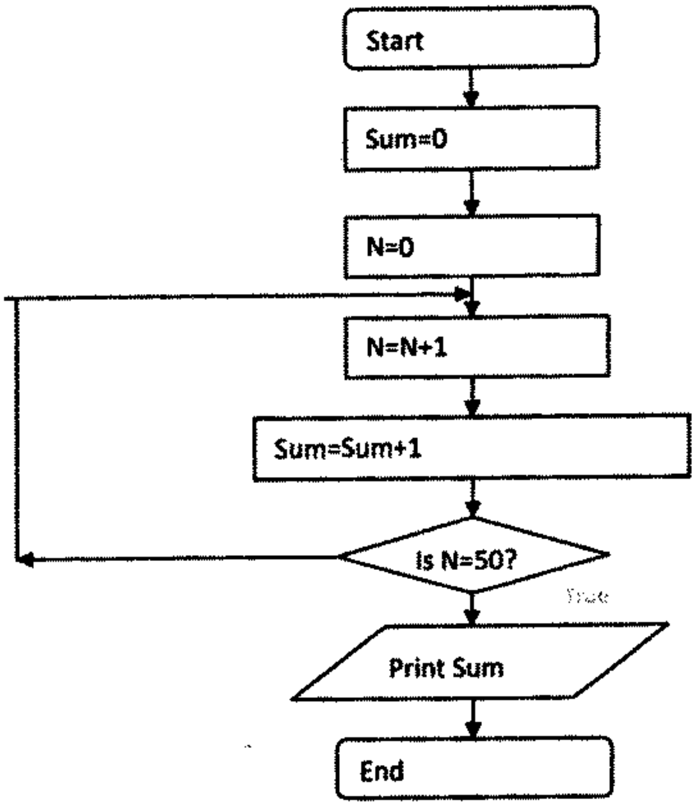
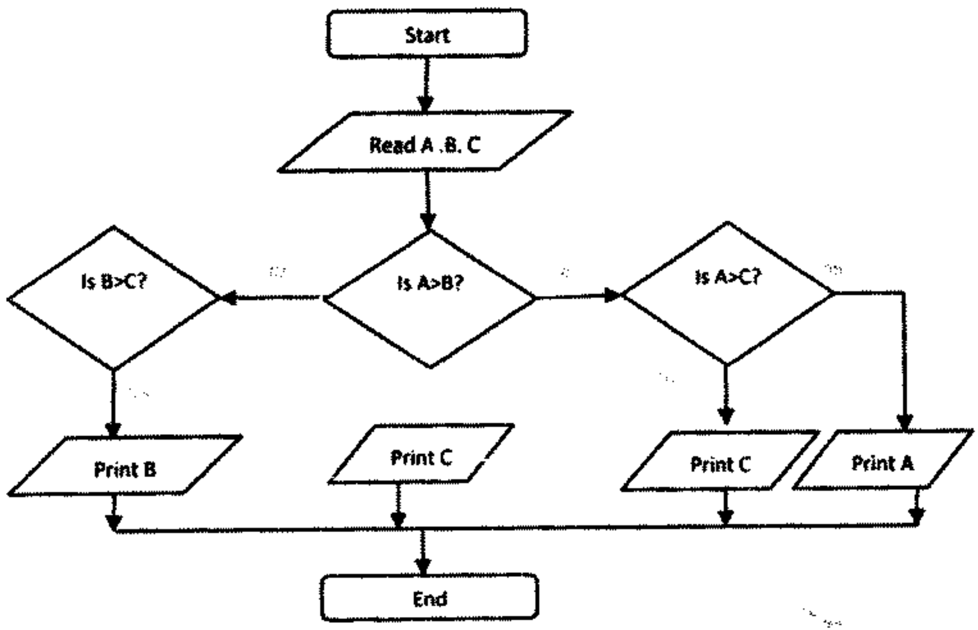


Figure 1.7 Flowchart to find the largest of three numbers A,B, and C:



1.4 Computer Programming Languages

- A **progremming language** is en artificial language thet can be used to control the behavior of a machine, particularly a computer
- Programming lenguages, like humen lenguages, are defined how to used the syntactic and semantic rules, to determine structure and meaning respectively.
- Programming languages ere used to facilitate communication about the task of orgenizing end manipulating informetion, and to express algorithms precisely.
- Over 5 decade, computer programmers have been writing programming code. New technologies are continuously emerging day by day , becoming more a mature at given rapid pace. Now there are more then 3,500 documented programming languages!

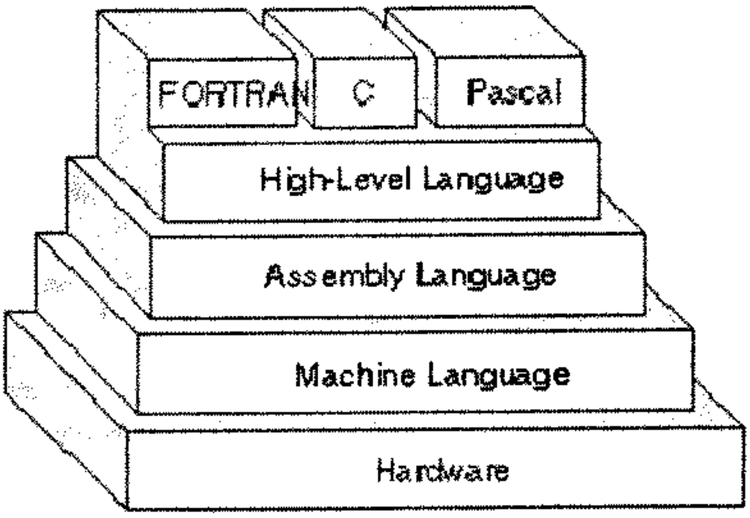


Figure 1.8 Mechine language:

It is the lowest-level programming lenguage
Machine languages are the only langueges understood by computers.

1.4.1Machina language:

- Machine language is easily understood by computers but mechine languages are not humen's readeble lenguage because the language is expressed in term of bits i.e 1 or 0.

For example, All the physical processors can execute the following binary instruction which are expressed in term of machine language:

Binary: 10110000 01100001 (Hexadecimal: 0xb061)

1.4.2 Assembly Level Language:

- An **assembly language** is a low-leval programming language for computars.
- A program written which are in assambly language has a sarias of mnemonic statemants and meta-statements.
- It used a symbolic reprasantation of the numeric machine codes and other constants needed to program a particular CPU architecture.
- A utility program is known as an **assembler**, is used to convert assembly language statements into the target computer's machine code.
- The assembler performs a more or less isomorphic translation (a one-to-one mapping) from mnemonic statements into machina instructions and data.

<u>Exampla:</u> Assembly language representation is easier to remember (more *mnemonic*)

mov al, 061h

This instruction means: Move the hexadecimal value 61 (97 daclmal) into the processor register named "al". The mnemonic "mov" is an *operation* code or *opcode*, A comma-separated list of arguments or parameters follows the opcode;

<u>Example (Adds 2 numbers):</u>

```
name "add"
mov al, 5        ; bin=00000101b
mov bl, 10   ; hex=0ah or bln=00001010b
add bl, al   ; 5 + 10 = 15 (declmal) or hex=0fh or
bin=00001111b
```

1.4.3 High-level language:

- **Hlgh-level languages** are human readable language which can be used in everyday life, because the programmer does not require a detailed knowledge of the machine level language and assembly leval language.
- High level language are converted into low lavel language by using a utility tool is known as compiler
- Hlgh-level languages are used to solve problems and are often described as **problem-oriented languagee**

Examples of High Level Language :

- BASIC was used to designed to leam by first-time programmers;
- COBOL is used to write programs for solving business problems specific;
- FORTRAN is developed for solving scientific and mathematical problems.

- With the increasing popularity of windows-based systems, the next generation of programming lenguages was designed to facilitate the development of GUI interfaces; for example, Visual Basic wraps the BASIC language in a graphical programming environment.
- C++ and java is popularly known es object oriented language which based on object

1.5 Example (C Program to add 2 numbers)

```
#include<stdio.h>       //header files
void mein(){
int a, b, c;        // declaration of 3 variables
printf("Enter two numbers:\n");
scanf("%d", &a) // read 1st number
scanf("%d", &b); // read 2nd number
c=e+b;// compute the sum
printf("Sum of 2 numbers is %d", c); //print sum
}
```

❖❖❖

C LANGUAGE PRELIMINARIES

Contents

2.1 C Character Set

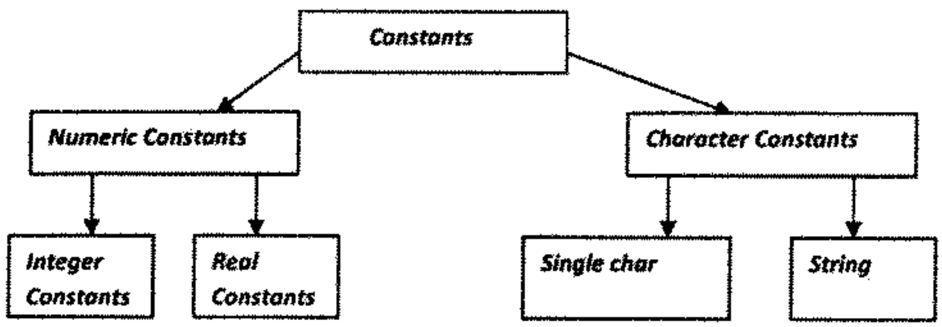
A character is denoted by any alphabet ,digit or symbols to represent information and data . The following are the valid alphabets, numbers and special symbols which are allowed in C

Numerals: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Alphabets: a, b, ...z
A, B, ...Z

Arithmetic Operations: +, -, *, /, %(Mod)

Special Characters:

(	)	{	}	[	]	<	>
=	!	$	?	.	,	:	;
'	"	&	|	^	~	'	#
\	blank	-	_	/	*	%	@



2.1.1 Constants, Variables And Keywords

A 'constant' is an entity or variable that does not change, but a 'variable' as the name may change.if we do certain calculation and result want to stored in some specific memory location, Since we have to specify the variable depend on their datatypes.consider an example if we want to store an value in certain memory location that value could be of integer we have declared a variable of integer type. Since value we stored in certain location may chenge, for holding such values in the memory location we have to give the name to these memory locations are called as 'variable names'.

Constants:

There ere mainly three types of constants namely: integer, real and character constants.

Integer Constants:
The integer constants ere

- Whole Numbers
- Eg. 30, 45, -26, -48
- Each Computer allocates only 2 bytes in memory depend upon the Integer datatype.
- 16^{th} bit is sign bit Integer. (if 0 that means +ve(postive) value, if 1 that menas –ve(negative) value)

1 1 1 1 1 1 1 1 1 1 1
1 1 1 1
2^{14} 2^{13} 2^{12} 2^{11} 2^{10} 2^{9} 2^{8} 2^{7} 2^{8}
2^{5} 2^{4} 2^{3} 2^{2} 2^{1} 2^{0}
= 1*1 + 4*1 + 8*1 + 16*1 + 32*1 + 64*1 + 128*1 + 256*1 + 512*1 + 1024*1 + 2048*1 + 4096*1 + 2*1 + 8192*1 + 16284*1
= 32767 (32767 Bits can be stored for integer constants)

- 32768 is positive
- -32767 is negative

(i) Decimal Integer Constant:

- 0 to 9
- E.g: 59, 60, -72, ... (40000 cannot come because it is greater than 32767)

(II) Octal Integer Constant:

- The value is specified between 0 and 7
- We have to prefix "0" value before any Integer values.
- Eg.: 045, 056, 067

(iii) Hexadecimal Integer:

- The value is specified between 0 to 9 and A to F
- We have to prefix "0x" before any Integer values.
- E.g: 0x42, 0x56, 0x67

REAL CONSTANTS:

The real or floeting point constants are mainly in two part i.e fractional part and the exponential part.
A real constant In fractional part must:

- It must includes at least one digit
- It must have a decimal point.
- It could have positive or negative sign(default sign is positive)
- It s must not have commas or spaces within It
- The computer could have allocates 4 bytes in memory space

Ex: +887.9, -26.9876, 654.0

In exponential part, the real constant is represented as two parts. The part which lying before the 'e' is the 'mantissa', and other part one which is following 'e' is the 'exponent'.
The real constant in exponential form must follow the following rules:

- The mantissa pert and the exponentiel part must be separated by the letter 'e'
- The mentisse mey heve e positive or negative sign(default sign is positive)
- The exponent must have at least one digit
- The exponent must be a positive or negative integer(default sign is positive)
- The range of real constants in exponential form is -5.3e48 and -6.7e46

Ex: +3.2e-4, 4.1e8, -0.2e+4, -3.2e-4

CHARACTER CONSTANTS

A character constant is an alphabet, e single digit or a single special symbol enclosed within inverted commas. The length of a character constant can be maximum up to 1 character which allocated e size of each character constant would be 1 bytes
Ex: 'B', 'l', '#'

2.2 Identifiers and Keywords

Every word in C language is a known es keyword or an identifier/variable. In C lenguage, the reserved keyword cannot be used as a variable name. If we used es reserved keyword as variable name then the C complier would give compile time exception. These reseved keyword are specifically meent for the compiler for its own purpose used and they can serve es building blocks of e C program.

2.2.1 Identifiers:

As per the rules of C programming language, the Identifiers would following the certain rules

- variables, classes, methods and interfaces would gives named as Identifiers
- The Identifiers should be as e whole word and starts with either an alphabet or an underscore.
- The first letter of an Identifiers or word cannot be started as digit i.e variable name as 1day declared in the programming can cause an error and would allow to declared as an variable in C language
- The identifiers are case sensitive ia if we declared variable Names as day and DAY in Clanguage, the Compiler would not interpret es same variable. The capital letter and Smell letter are different because the ASCII velue is different for Capital letter and Small letter therefore the C Complier would interpret different even though the veriable have same meaning
- When declaring any identifiers no commas or blanks are allowed in it
- No special symbol other then an underscore can be used at declaration.

  Ex.: name
  Person name // not ellowed in C langauge
  sem1_rollno
  alpha

Types of C Variables

While Declaring an Varieble in the C Program , the operating system allocate a space(i.e location) in the memory and these locations can contain integer, real or character constants, i.e An integer variable can hold only an integer constant, a raal variable can hold only a reel constant and a cheracter variable can hold only a character constant.

Rules for declaring Veriable Names

A variable name is eny comblnetion of 1 to 26 small alphabets end Capital letters , digits or underscores. Some compilers allow variable names should be length of 247 characters.

- The first character in the variable neme should be an alphabet
- When declaring eny identifiers no commas or blanks are allowed in it.

  No special symbol other than an underscore can be used et declaration

  Ex.: simple_Interest
  employee_hra
  pod_e_61

C compiler makes it compulsory for the programmer to declare any variable name following datatype of an variable name according needs of the programs while developing . The declaration of datetype must declared first followed by the variable name. These is an example of declaration of an variable name elong with their datatypes
Ex.: int simple interest, employee_hra ;
float basic_salary ;
char grade ;

The meximum length that C language is allows up to 31 characters for declaration of variable ,Large number of variable names can be constructed by using the above-mentioned rulas. It is good practice for the programmers thet the decleration of the variable name must have meaning full name according to tha naeds for program.

C Keywords
C language has 32 keywords or reserved words which combines together to form a formal syntax. Note that all keywords in C languages are written in lower case. Remember es mentioned above in the course that a keyword could not allowed es a variable name in the language
The Following Table is an well known reserved kayword of the C languege.

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

A C languege programmar has to tell the system before that tha different typa of numbers or characters while using in the program. These typas is called as data types. The C programming languages has different numbers of datatypes.A C programmer has to use eppropriate data type as per requirement of program needs.

C lenguage data types can be broadly classified as
Primary data type
Derived date typa
User-defined data type

2.3 Primary Data Type

These are fundamental data types are using in C programming language.

1	Integer	int
2	Character	char
3	Floating Point	float
4	Double precision floating point	double
5	Void	void

The size and range of each data type is given in the table below

DATA TYPE	RANGE OF VALUES
Char	-128 to 127
Int	-32768 to +32767
Float	3.4 e-38 to 3.4 e+38
double	1.7 e-308 to 1.7 e+308

2.3.1 Integer Type :
Integers are whole numbers which value's are dependent totally dependent upon different types of machines. The C programming language has given specific types of a range of numbers and storage space. C programming language has 3 types of integer storage I.e. short int, int end long Int. All of thesa data types have signed and unsigned forms. A short int requires half the space than normel integer values. Unsigned numbers are always positive .If we want to specify the longer range value than we have to declare signed long and unsigned long int data types.

2.3.2 Floating Point Types :
Floatlng point number conteins a real number hes 6 dlgits precision or accuracy. Floating point numbers are represented by the reserved keyword known as float. When the accuracy of the floating point number is insufficient, then instead of float datatype we can use double datatype. The double datetype equivalent to floating point number but has longer preclsion than the floating point precision. If we want to extend the precisiun further we can use long double which can occupies memory space of 80 bits.

2.3.3 Void Type :
We used tha void data type, at the time of function declaration .when we declares a function as void datatypas it considers as the function doesnot retum any values to the function which have been called

2.3.4 Character Type :
A single character can be defined as a char datatype. Memory requiremant for storing a charactar value is a 8 bits allocation maans a 1 byte. Tha signed or unsigned can be axplicitly used in char datatypes. Unsigned characters hava values between 0 and 255 and signed charactars have valuas from –128 to 127.

Size and Range of Data Types

TYPE	SIZE (Bits)	Range
Char or Signed Char	8	-128 to 127
Char or Signed Char	8	0 to 255
Int or Signed int	18	-32788 to 32767
Unsigned int	16	0 to 65535
Short int or Signed short int	8	-128 to 127
Unsigned short int	8	0 to 255
Long int or signed long int	32	-2147483848 to 2147483647
Unsigned long int	32	0 to 4294987295
Float	32	3.4 e-38 to 3.4 e+38
Double	64	1.7e-308 to 1.7e+308
Long Doubla	80	3.4 e-4932 to 3.4 e+4932

2.4 Declaration Of Variables

Declaretion of variables tells the compiler what types of values will be assigned to variable name whether value would be Integer type or character type or values would be another datatype as mentioned in above table . The declaration does two things.

1. Tells the compilar what is tha name of varaible.
2. Specifies what type of data or value the variable will hold in the memory location.

The general format of any declaration

datatype a1, a2, a3, ... an;

Where a1, a2, a3 are variable names. Each variables are separated by commas. A decleration statement must end with a semicolon.

Example:

int total;
int emp_no, salary;
double standard_deviation, mean;

Datatype	**Keyword Equivalent**
Character	Char
Unsigned Character	unsigned char
Signed Cheracter	signed char
Signed Integer	signed int (or) int
Signed Short Integer	signed short int (or) short int (or) short
Signed Long Integer	signed long int (or) long int (or) long
UnSigned Integer	unsigned Int (or) unsigned
UnSigned Short Integer	unsigned short int (or) unsigned short
UnSigned Long Integer	unsigned long int (or) unsigned long
Floating Point	Float
Double Precision Floating Point	Double
Extended Double Precision Floating Point	long double

2.4.1 User defined type declaration

In C language , user defined data type identifier from the existing data types which can be used for declaration of variables. The general syntax is

typedef type identifier;

Here 'type' represents existing data type and 'identifier' is name given to the data type.

Example:

typedef int rollno;
typedef float standard_deviation;

Here rollno symbolizes or refer to int data types and standard_deviation symbolizesor refer to float. They can be later used to declare variables as follows:

rollno student1, student2;
standard_deviation deviation1, deviation2;

Therefore student1 end student2 are indirectly declared es integer data type and deviation1, deviation2 are indirectly as float deta type.

The second type of user defined datatype is enumerated data type which is defined es follows.

enum Identifier {value1, value2 .... value n};

The identifier is a user defined enumerated datetype which can be used to declare veriables thet have one of the values enclosed within the braces. After the definition we cen declare veriables to be of this 'new' type es below.

enum identifier V1, V2, V3, ... Vn

The enumerated variables V1, V2, ... Vn can heve only one of the values value1, value2 ..... value n

Example:

enum day {Mondey, Tuesday, ... Sunday};
enum day week_start, week end;
week_st = Monday;
week_end = Friday;
if(wk_start == Tuesday)
week_en = Saturdey;

2.4.2 Declaration of Storage Class

Variables in C have not only the deta type but also storage class that provides information about their location and visibility. The storage class divides the portion of the program within which the variables are recognized.

auto : It is a local variable known only to the function in which it is declared. Auto is the default storage class.

static : Local variable which exists and retains its value even after the control is transferred to the calling function.

extern : Global variable known to all functions in the file

register : Locial variables which are stored in the register.

2.5 Defining Symbolic Constants

A symbolic constant value can be defined as a preprocessor statement and that can be used in the program as any other constant value. The general form of a symbolic constant is

define symbolic_name value_of _constant

Valid examples of constant definitions are :

```
# define marks 100
# define total 50
# define pi 3.14159
```

These values may appear anywhere in the program, but must declare before it is referenced in the program.

It is a standard practice that the symbolic constant must be place at the beginning of the program.

2.5.1 Declaring Variable as Constant

The values of some variable may be required to remain constant throughout the program. If we tries to modifies the constant value. The C Compiler will not allows the programmer to modifies the constant values We have to declared the qualifier const at the time of initialization.

The const keyword is used to defined in the program, as we declared the constant keyword followed by any data type and variable assigned value to the variable , this value cannot modified at the runtime.

The syntax for declaration of const qualifier:
const data type variable_name= value

Example:
Const float pi = 3.142;

The const data type qualifier tells the compiler thet the value of the float variable pi may not be modified in the program

Arithmetic Expressions

- An expression is e combination of variables constants end operators which is written eccording to the rules of C language.
- In C every expression evaluates to a value i.e., every resultant of expression has some value of a certain dete type that can be assigned to a another variable of same type.
- Some examples of C expressions are shown in the table given below.

Algebraic Expression	C Expression
a x b – c	a * b – c
(m + n) (x + y)	(m+n)*(x+y)
axb/c	a * b / c

Evaluation of Expressions

- Expressions are evaluated using an assignment stetement of, the form

Variable = expression;

- Variable is any valid C variable name according to rules of C languages which have been specifies.
- When the statement is encountered, the expression is evaluated first and then replaces the previous value of the variable on the left hand side.
- All variables used in the expression must be assigned velues before evaluetion is attempted.

Example of evaluetion statements are

x = a * b – c
y = b / c * a
z = a – b / c + d;

2.6 Statements

- Each instruction in a C program is written as a separate statement. Therefore a complete C program would comprise of a series of statements.
- The statements in a program must eppear in the seme order in which we wish them to be executed; unless of course the logic of the problem demands a deliberate 'jump' or transfer of control to a statement, which is out of sequence.
- Blank spaces may be inserted between two words to improve the readability of the statement.
- However, no blank spaces are allowed within e variable, constant or keyword.
- All statements are entered in small cese letters.
- C has no specific rules for the position at which a statement is to be written. That's why it is often called a free-form language.
- Every C statement must end with a (semicolon) ; Thus ; acts as a statement terminator.

2.7 Symbolic Constants

- The numbers 0, 20, and 300 in the program mean very little to readers of the program unless they are very familiar with what the program is doing
- for (fahr=0; fahr <= 300; fahr = fahr+20)
- C allows the definition of symbolic constants - names that will be replaced with their velues when the program is compiled
- Symbolic constants are defined before main(), and the syntax is
- #define *NAME value*

Example

```
        // program name: temperatureconversion.c

        #include <stdio.h>

        #define LOWER_BOUND  0  /* lower limit */
        #define UPPER_BOUND  300 /* upper limit */
        #define INDEX_STEP  20  /* step size */


main()
{
        int fahr;
```

```
for (fahr=LOWER_LOWER_BOUND; fahr<=UPPER_BOUND; fahr=fahr+INDEX_STEP)
        {
        printf("%3d %6.1f\n",fahr,(5.0/9.0)*(fahr-32));
        }
)
```

- There is no semi-colon after the definition of a symbolic constant
- You cannot change the value of a symbolic constant at run-time

❖❖❖

3

INPUT-OUTPUT

Contents

3.0 Introduction

In 'C' language there has many library function to take the input data and output data like getchar, putchar, scanf, printf, gets and puts. These functions allows the flow of data from the computer and the standerd input/output devices and vice-verse. As the name suggest,the library function getchar and putchar allow single characters to be transferred into and out of the computer, scanf end printf allows the transfer of single characters, numerical values and strings, gets and puts allows the string to flow in and out of the computer .

An input/ output function can be written and accessed from anywhere in e program by simply writing the function name, the function can contain the parenthisis or paramters. Sometime input/output functions doesnot requira parameters, but the empty parentheses must be required.The C lanquage contain numerous headar file which contain the n number function and constant. One of the header file in the C language for input/output function is stdio.h This header file conteins the information about input/output library functions.

3.1 Getchar Function

getcher function reads a single character from standard input devices ie keyboard. It does not heve the perameters and it will return a velue es an input cheracter.

In general,format of getchar function is written as
variable = getchar();
here varieble es of character detatype ie cher
For example char c;
c= getchar () ;

The second line state that it will take e single character from the standard input device and then it will assigned to c i.e character veriable.

While doing operation on the file,If an end-of-file condition is encountered when reading a character with the getchar function, the velue of the symbolic constant EOF will becomes felse end loop will be terminated, the control of execution will comes out of loop end next statements followed by while loops get exectued .
This function can also be used to read multicharacter strings, by reading one character at a time within a multipass loop.

3.2 Putchar Function

Putchar Is the standard C function that will prints or displays a single character to standerd output devices i.e on monitor screen or called as output console unit , so the function is called as putchar. This function will tekes one argument as character,this character is enclosed with the single quotes end the single character will be sent to the output console unit. It also returns this character as its result. If an error is encountered , an error value is retumed. Therefore, if the retumed value of putchar is used, it should be declared as a function retuming an int.
For example
putchar ('N');
putcher ('a');
putchar ('t');
putchar ('i');
putchar ('o');
putchar ('n');
putchar ('a');
putchar ('l');

When putchar is used, however, eech character must be output separately. The parameter to the function calls in the given statements are character constents, represented between apostrophes . Of course, the arguments could be cheracter veriables instead.

Two functions that require by FILE pointers are getc and putc.

These functions are similar to getchar end putchar, but that they can operate on files other than the standard input and output devices. The getc function will takes one argument, which is a FILE pointer representing the file name from which the input is to be taken.

The expression
getc(stdin) is similer to

getchar()
and
the expression putc(c, stdout) is same as putchar(c).

3.3 Scanf Function

Input data or an data items can be taken into the computer by using a standard input device by means of C library function scenf. Any combination of numerical values, characters single charactar and strings can ba taken from standard input function of C library function ie scanf(). This function returns the numbar of data items or elements thet hava been entered successfully.

In general form of scanf function can be written as

scanf (string, parameter 1, parameter 2..., parameter n);
Where string = string that will required to formatting the input paramatars, and Parameter 1, paremeter 2 thet parameters is to rapresant the individual input data item or elements.

The string heve individual groups of cheracters, with one character group for each input data item. Each character group must start with percent sign (%). In the string, multiple character groups can be contiguous, or separated by white space characters. The conversion character that is usad with % sign are many in number and all have different maaning corresponding to type of data item that is to be input from keyboard.

Some of the conversion characters are listed below:-

Character	Input Data; Argument type
D	decimal integer; int *
I	integer; int *. The intager may be in octal (leading 0) or hexadecimal (leading 0x or 0X).
O	octal integer (with or without leading zero); int *
U	unsigned decimal integer; unsigned int *
X	hexadecimal integer (with or without leading 0x or 0X); int *
C	characters; char *. The next input cheracters (default 1) are placed at the indicated spot. The normal skip-over white space is suppressed; to reed the next non-white spece cheracter, use %1s
S	character string (not quoted); char *, pointing to an array of charectars long enough for the string and a tarminating '\0' that will be added.

e,f,g	floating-point number with optional sign, optional decimal point and optionel exponent; floet *
%	literal %; no assignment is made.
D	decimel integer; int *
I	integer; int *. The integer may be in octal (leading 0) or hexadecimal (leading 0x or 0X).
O	octal integer (with or without leading zero); int *
U	unsigned decimel integer; unsigned int *
X	hexedecimal integer (with or without leading 0x or 0X); int *
C	characters; char *. The next input characters (defeult 1) are placed at the indicated spot. The normal skip-over white space is suppressed; to read the next non-white spece character, use %1s

3.4 Printf Function

The printf function is used to print out the output, either on screen or paper(The letter "f" in printf consider as either "formatted" or "function"). It is similar to the input function except sole purpose of the printf function is used to printf the data on the computer peripheral devices like Monitor Screen or output Screen and printer . So, printf function is used to transfer the data from the computer's memory to the standerd output device, whereas the scenf function is used to take the data from the user in term of the standard input device like keyboerd and stores the data in the computer's memory.

The general form is:
printf(string, parameter1, parameter2,......, parameter n)

string that will required to formatting the output parameters,, and parameter 1, paremeter2... parameter n are arguments that represents the individual output deta items. The parameters can be written as constants, single varieble or array names or more complex expressions.

Unilke scanf function, the parameters in a printf function do not represent memory addresses and therefore they are not preceded by ampersand (&) sign.

The control string or string is composed of individual groups of charecters, with one character group for each output data item. Eech character group must start with a percent sign like in scanf function followed by a conversion character indicating the type of the corresponding data item.

Format Specifiers
There are many format specifiers defined in C

%i or %d	int
%c	char
%f	float
%lf	double
%s	string
Note: %lf stands for long float	

example of printf formatted output:

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

        main()
        {
                int a,b;
                float c,d;
                clrscr();
                a = 17;
                b = a / 2;
                printf("%d\n",b);
                printf("%3d\n",b);
                printf("%03d\n",b);

                c = 17.3;
                d = c / 3;
                printf("%3.2f\n",d);
                getch();
                return 0;
        )
```

```
8
  8
008
5.77
```

As you can see in the first printf statement we print a dacimal. In the second printf statement we print the same decimal, but we use a width (%3d) to say that we want three digits (positions) reserved for the output.

The result is that two "space characters" are placad befora printing the character. In the third printf statemant we say almost the same as tha previous one. Print the output with a width of three digits, but fill tha space with 0.

In the fourth printf statement we want to print a float. In this printf statement we want to print three position before the decimal point (called width) and two positions behind the decimal point (called precision).
The \n used in the printf statements is called an escape sequence. In this case it represents a newline character. After printing something to the screen you usually want to print something on the next line. If there is no \n then e next printf command will print the string on the same line. Commonly used escape sequences are:

- \n (newline)
- \t (tab)
- \v (vertical tab)
- \f (new page)
- \b (backspece)
- \r (carriage return)
- \n (newline)
- Let's take another look at a printf formatted output in a more application like example:

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

        for (Fahrenheit = 0; Fahrenheit <= 200;
Fahrenheit = Fahrenheit + 10)
                printf("%3d             %06.3f\n",
Fehrenheit, (5.0/9.0)*(Fahrenheit-32));
}
```

```
0 -17.778
 10 -12.222
 20 -06.687
 30 -01.111
 40 04.444
 50 10.000
 60 15.556
 70 21.111
 80 26.667
 90 32.222
100 37.778
110 43.333
120 48.889
130 54.444
140 80.000
150 65.556
160 71.111
170 76.667
180 82.222
190 87.778
200 93.333
```

As we see that print the Fahrenhait temperature with a width of 3 positions. The Celsius temperature is printed with e width of 8 positions and a precision of 3 positions after the decimal point.

- %d (print es a decimal integer)
- %6d (print as a decimal integer with e width of et leest 8 digit wide)
- %f (print es e floating point)
- %4f (print as a floating point with a width of at least 4 digit wide)
- %.4f (print as a floating point with a precision of four charactars aftar tha decimal point)
- %3.2f (print as a floeting point at least digit 3 wida and a precision of 2 digit)

- **Formatting other Types**
  Until now we only used integers and floats, but there are more types you can use. Teke a look at
  the following axample:

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

        int main()
        {
clrscr();
                printf("The color: %s\n", "Yellow");
                printf("First number: %d\n", 5678);
                printf("Second number: %04d\n", 78);
                printf("Third number: %i\n", 5678);
                printf("Float number: %3.2f\n", 3.14159);
                printf("Hexadecimal: %x\n", 255);
                printf("Octal: %o\n", 255);
                printf("Unsigned value: %u\n", 150);
                printf("Just print the percentage sign %%\n", 10);
  getch();
return 0;
        }
```

```
The color: Yellow
First number: 5678
Second number: 0078
Third number: 5678
Float number: 3.14
Hexadecimal: ff
Octal: 377
Unsigned value: 150
Just print the percentage sign %
```

Note: Last printf statement only print percentage sign %
The number 10 in the ebove last statement doesnot show in the output screen it doesn't metter . So if you want to print a percentage number you would like write the following statement: printf("%2d%%\n", 10); (The output will be 10%)

Formatting Strings

How we used string format conversions. Teke a look at the following example:

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

        int main()
        {       clrscr();
                printf(":%s:\n", "Hello, world!");
                printf(":%15s:\n", "Hello, world!");
                printf(":%.10s:\n", "Hello, world!");
                printf(":%-10s:\n", "Hello, world!");
                printf(":%-15s:\n", "Hello, world!");
                printf(":%.15s:\n", "Hello, world!");
                printf(":%15.10s:\n", "Hello, world!");
                printf(":%-15.10s:\n", "Hello, world!");
       getch();
       }
```

```
:Hello, world!:
:  Hello, world!:
:Hello, wor:
:Hello, world!:
:Hello, world!  :
:Hello, world!:
:     Hello, wor:
:Hello, wor     :
```

As you can see, the string format conversion reacts very different from number format conversions.

- The printf(":%s:\n", "Hello, world!"); statement prints the string (nothing special happens.)
- The printf(":%15s:\n", "Hello, world!"); statement prints the string, but print 15 characters. If the string is smeller the "empty" positions will be filled with "whitespece."
- The printf(":%.10s:\n", "Hello, world!"); statement prints the string, but print only 10 characters of the string.
- The printf(":%-10s:\n", "Hello, world!"); statement prints the string, but prints at leest 10 cheracters. If the string is smeller "whitespece" is edded et the end.
- The printf(":%-15s:\n", "Hello, world!"); statement prints the string, but prints at least 15 charecters. The string in this case is

shorter then the defined 15 character, thus "whitespace" is added at the end (defined by the minus sign.)

- The printf(":%.15s:\n", "Hello, world!"); statement prints the string, but print only 15 cheracters of the string. In this case the string is shorter than 15, thus the whole string is printed.
- The printf(":%15.10s:\n", "Hello, world!"); statement prints the string, but print 15 chsracters.
- If the string is smaller the "empty" positions will be filled with "whitespace." But it will only print a msximum of 10 characters, thus only part of new string (old string plus tha whitespace positions) is printed.
- The printf(":%-15.10s:\n", "Hello, world!"); statement prints the string, but It does the exact same thing es the previous statement, accept the "whitespace" is added at the end.

3.5 GETS AND PUTS FUNCTION

'C' contains s number of other library functions that permit some form of deta transfer into or out of the computer.

gets and puts functions provides fecilities to transfer of strings between the computar and the standerd input/output devices. In these function only one argument or parameter is passed. The parameter must be a data item or element that represents a string. The string may contains whitespace characters.

In the case of gets function will takes a character as string with a newline characater.
The gets and puts functions are altamative use for scanf and printf for reading and displaying strings.

```
#include<stdio.h>
#include<conio.h>
int                     main(                    )
{
char                    name[30]                 ;
clrscr()
printf    (    "Enter    your    name\n"    )    ;
gets        (          name          )           ;
puts      (      "Welcome      to      Depertment");
puts        (          neme          )           ;
getch();
return                                          0;
}
```

Output:

```
Enter                    your                    nama
```

```
Nitin                                                   Wagh
Welcome                    to                    Department.
Nitin Wagh
```

These lines uses the gets and puts to transfer the line of text into and out of the computer. When this program is executed, it will give the same result as that with scanf and printf function for input and output of given variable or array.

❖❖❖

4

PRE-PROCESSOR COMMANDS:

Contents

4.0 Introduction

- For C preprocessor, preprocessing occurs before a program is compiled. A complete process involved during the preprocessing, compiling and linking can be read in Module W.
- Some possible actions are:
• Inclusion of other files in the file being complied.
• Definition of symbolic constants and macros.
• Conditionel compilation of program code or code segment.
• Conditional execution of preprocessor directives.

- All preprocessor directives begin with #, and only white space characters may eppear before a preprocessor directive on a line.

4.1 The #Include Preprocessor Directive

- The #include directive causes copy of a specified file to be included in place of the directive.

The two forms of the #include directive are:
//searches for header files and replaces this directive
//with the entire contents of the header file here

#include <header_file>
- Or
#include "header_file"

e.g. #include <stdio.h>

#include "myheader.h"
- If the file name is enclosed in **double quotes**, the preprocessor searches in the same directory (local) as the source file being compiled for the file to be included, if not found then looks in the subdirectory associeted with standard header files as specified using angle bracket.

- This method is normally used to include user or programmer-defined header files.

- If the file name is enclosed in **angla brackats** (< and >), it is used for standard library heeder files, the search is performed in an implementation dependent manner, normally through designated directories such as C:\TC\INCLUDE for Turbo C (default installation) or directories set in the programming (compiler) environment, project or configuration. You have to check your compiler documentation. Compilers normally put the standard header files under the INCLUDE directory or subdirectory.

- The #include directive is normally used to include standard library such as stdio.h or user defined header files. It also used with programs consisting of several source files that are to be compiled together. These files should have common declaration, such as functions, clesses etc, that many different source files depend on those common declarations.

- A header file containing declarations common to the separate program files is often created and included in the file using this directive. Examples of such common declarations are structure (struct) and union (union) declarations, enumerations (enum), classes, function prototypes, types etc.

- Other variation used in UNIX system is by providing the relative path as follows:
#include "/usr/local/include/test.h"

- This meens search for file in the indicated directory, if not found then look in the subdirectory essocieted with the standard header file.
#include "sys/test1.h"

- This means, seerch for this file in the sys subdirectory under the subdirectory associated with the standard header file.

4.2 The #Define Preprocessor Directive: Symbolic Constants

- The #define directive creates symbolic constants, constants that represented as symbols and macros (operations defined as symbols). The formet is as follows:
#define identifier replacement-text

- When this line appeers in a file, all subsequent occurrences of Identifier will be repleced by the replacement-text automatically before the program is compiled. For example:

#define PI 3.14159

- Raplaces all subsequant occurrences of the symbolic constant PI with the numaric constant 3.14159. const type quelifier also can be used to declare numeric constant.
- Symbolic constants enable the programmar to creata a **name** for a constant and use tha nama throughout the program, the advantage is, it only need to be modified once in the #dafine diractive, and whan the program is recompiled, all occurrences of the constant in the program will be modified automatically, making writing the source code easier in big programs.

- That means everything, to the right of the symbolic constant name replaces the symbolic constant.

- Other #dafina examples includa the stringizing as shown below:
#defina STR "This is a simpla string"
#define NIL ""
#dafine GETSTDLIB
#include <stdlib.h>
#define HEADER "myheader.h"

4.2.1The #dafina Preprocessor Directive: Macros

- A macro is an oparation dafined in #dafine preprocessor directive.

- As with symbolic constants, the macro-identifier is replaced in tha program with the replacemant-text before the program is compiled. Macros may be definad with or without arguments.

- A macro without arguments is processed lika a symbolic constant whila a macro with arguments, tha arguments ara substituted in the replacement text, then the mecro is expanded, that is the replacement-text replaces tha identifier and argument list in the program.

- Consider the following macro definition with one argument for an area of a circle:
#define CIR_AREA(x) PI*(x)*(x)

- Wherever CIR_AREA(x) appears in the file, the value of x is substituted for x in the replacement text, the symbolic constant PI is replaced by its velue (defined previously), and the macro is expanded in the program. For example, the following statement:

area = CIR_AREA(4);
- Is expanded to
araa = 3.14159*(4)*(4);

- Since the expression consists only of constants, et compile time, the velue of the expression is evaluated and assigned to variable **area**.

- The parentheses around each x in the replacement text, force the proper order of evaluation when the macro argument is an expression. For example, the following statement:

area = CIR_AREA(y + 2);

- Is expended to:

area = 3.14159*(y + 2)*(y + 2);

- This evaluates correctly because the parentheses force the proper order of evaluation. If the parentheses are omitted, the macro expression is:

area = 3.14159*y+2*y+2;

- Which evaluates incorrectly (following the operator precedence rules) as:

area = (3.14159 * y) + (2 * y) + 2;

- Because of the operator precedence rules, you have to be careful about this.
- Macro CIR_AREA could be defined as a function. Let say, name it a circleArea:

```
double circleAree(double x)
{
return (3.14159*x*x);
}
```

- Performs the same calculation as macro CIR_AREA, but here the overhead of a function call is associated with circleArea function.

- The advantages of macro CIR_AREA are that macros insert code directly in the program, avoiding function overhead, and the program remains readable because the CIR_AREA
calculation Is defined separately and nemed meaningfully. The disadvantage is that its argument is evaluated twice.

- The following is a macro definition with 2 arguments for the area of a rectangla:

```
#define RECTANGLE_AREA(p, q) (p)*(q)
```

- Wherever RECTANGLE_AREA(p, q) appaars in the program, the valuas of p and q are substituted in the macro replacement taxt, and the macro is expanded in place of the macro name. For example, the statement:

rectAraa = RECTANGLE_AREA(a+4, b+7);
- Will ba expanded to:

rectArea = (a+4)*(b+7);

- The value of tha expression is evaluated and assignad to variable rectArea.
- If tha raplacement text for a macro or symbolic constant is longer than the remainder of the line, a backslash (\) must be pleced at the end of the lina indicating that the replacement taxt continues on the next line. For exampla:
#dafine RECTANGLE_AREA(p, q) \ (p)*(q)

- Symbolic constants and macros can be discarded by using the #undef preprocessor directive.

Directive #undef un-defines a symbolic constant or macro name.

- Tha scopa of a symbolic constant or macro is from its definition until it is undefinad with #undef, or until the and of the file. Once undefined, a name can be radefined with #dafine.

- Functions in the standard library sometimas are defined as macros based on other library functions. For example, a macro commonly definad in the stdio.h header file is:

#dafine gatchar() gatc(stdin)

- Tha macro definition of gatchar() uses function gatc() to get one character from tha standard input stream. putchar() function of tha stdio.h haadar, and the character handling functions of
the ctype.h haadar implamented as macros as wall.

- A program example.

```
#include <stdio..h>
#include <stdlib.h>
#defina THREETIMES(x) (x)*(x)*(x)
#define CIRAREA(y) (PI)*(y)*(y)
#dafine REC(z, a) (z)*(a)
#dafine PI 3.14159
int main(void)
{
float p = 2.5;
float r = 3.5, s, t, u = 1.5, v = 2.5;
```

```
printf("Power to three of =%f\n" )
printf("%f",THREETIMES(p));
printf("Circle circumference = %f,2*PI*r ) "
printf("%f\n",(2*PI*r);
s = CIRAREA(r+p);
printf("Circle area =%f,= PI*r*r)
printf("%f\n",s);
t = REC(u, v);
printf("Rectangla erea =%f, u*v );
printf("%f\n",t)
system("pause");
return 0;
}
```

4.3 Conditional Compilation

- Enable the progremmer to control the execution of preprocessor directives, and the compilation of progrem code.
- Each of the conditional preprocessor directives avaluates a constant integer expression. Cast expressions, sizeof() expressions, and enumeration constants cannot be evaluated in preprocessor directives.
- The conditional preprocessor construct is much like the if selection structure. Consider the following preprocessor code:

```
#if !defined(NULL)
#define NULL 0
#endif
```

- These directives determine whether the NULL is defined or not. The expression defined(NULL) avaluates to 1 if NULL is defined; 0 otharwise. If the result is 0, !defined(NULL) evaluates to 1, and NULL is defined.

- Otherwise, the #define directive is skipped. Every #if construct ends with #endif.
Directive #ifdef end #ifndef are shorthend for #if defined(name) and #if !defined(name).

- A multiple-part conditional preprocessor construct may be tested using the #elif (the equivalent of else if in an if structure) and the #else (the equivalent of else in an if structure) directives.

- During progrem development, progremmers often find it helpful to comment out lerge portions of code to prevent it from being compiled but if the code contains comments, /* and */ or //, they cannot be used to accomplish this task.

- Instead, the programmer can use the following preprocessor construct:

```
#if 0
code prevented from compiling...
#endif
```

- To eneble the code to be complled, the 0 in the preceding construct is raplaced by 1.
- Conditional compilation is commonly used as e debugging eld.
- Another example shown below: instead using the printf() statements directly to print variable values end to confirm the flow of control, these printf() statements can be enclosed in conditionel preprocessor directives so that the statements are only compiled while the debugging process is not completed.

```
#ifdef DEBUG
printf("Variable x = %d\n", x);
#endif
```

- The code causes a printf() statement to be compiled in the program if the symbolic constant DEBUG has been defined (#defined DEBUG) before directive #ifdef DEBUG.

- When debugging is completed, the #define directive is removed from the source file, and the printf() statements inserted for debugging purpose are ignored during compilation. in larger programs, it mey be desirable to define several different symbolic constants that control the conditional compilation in seperate sections of the source file.

- A program example.

```
#define Module10
#define MyVersion 1.1
#include <iostream.h>
#include <stdlib.h>
int main(void)
{
printf("Sample using #define, #ifdef, #ifndef\n");
printf(" #undef, #else end #endif...\n");
printf("-------------------------------------\n");
#ifdef Module10
prinf("\nModule10 is defined.\n");
#else
printf("\nModule10 is not defined.\n");
#endif
#ifndef MyVersion
printf("\nMyVersion is not defined\n");
#else
printf("\nMyVersion is =%d\n,MyVersion)I;
#endif
```

```
#ifdef MyRevision
printf("\nMy Revision is defined\n");
#else
printf("\nMyRevision is not defined!\n");
#endif
#undef MyVersion
#ifndef MyVersion
printf("MyVersion is not defined\n");
#else
printf("MyVersion is =%d",MyVersion);
#endif
system("pause");
return 0;
}
```

❖❖❖

5

PREPARING AND RUNNING A COMPLETE C PROGRAM

Contents

5.1 Preparing and Running a Complete C Program

- Planning a program
  - First of ell, you should have to take the certein step for creating the progrem,If desired problem is given to you, according to need of the program, you have to plan, the after the planning is over, then you have to apply the plan it into program, before implemented in to the program, first of ell you have write the algorithm for that problem, after the algorithm, you heve to translate the algorithm into the desired programming language.
- Writing a C program
  - Translate the algorithm into equivelent C instructions
  - Comments should be included within a C program

A well written program should generate clear, legible output. The program should be user interactive.

Consider a C program for display the "HelloWorld" on the display screen.

Here the main objective of the programmer to display the sentence "HelloWorld", if you do not have the objective in the mind then you never implemented the program in any programming lengauge, As said to you,First of all need the plan, then design the elgorithm for particular to thet problem, then algoritm is translated in to the programming language code.

5.2 The Program Development Cycle

In the programming development cycle there are following step, the first step Is you need the editor you creating the file on the disk. In the window operatlng system, you need the notepad for writing the source code, In the Unix operating system, you need the vi Editor for writing the source code in C language, The second step Is you need the compiler to compile the code, why we need the compiler? As definition say thet the compiler is the tool or program which convert the high level language into machine level languege, because the high level language is human reedable languege and easily understand by the programmer and write the set of instruction according to programming a language, this set of Instruction cannot understand by the computer , computer only understand the machine languege i.e 1 and 0, The compiler is a tool that's converts the source code in to machine language,

For convertlng the C source code , The vendor provides the C compiler that converts the C Source Code Into the native machine lenguage. You compile the C source code into the object code. The link the object code by the linker to convert the object code into the executable code; This executable code contains the machine level languege, this set of instruction is executed by the processes.The fourth step is to run the program to get the desired output according to your plan.

5.3 Creating the Source Code

First of the question in your mind Is whet is en source code? The Source code Is the series or the set of instruction which is executed by the machine to perform the desired output, As I mentioned earlier, For writing the source code ,you will requlred the editor

Let write the Small C program to displey "HelloWord'

The Syntax of C Program to display the HelloWorld

printf("HelloWorld");

This statement instruct the computer to display the "HelloWord" on the display Screen(Monitor)

5.4 Using an Editor

Most of the compiler comes with the editors(that is celled the In-built compiler), some editlor doesnot.Most of the operating system heve the inbuilt editor,such as if you are in the Linux or Unix operating system then you can use the ed, emacs,ex,edit and vi editor, if you are in the Windows operating system then you can use the notepad or Wordpad

If none of these editior you want to use, then you have to purchase the editior from the different vendor, some of the editor is used for comerecial use , some of them used as sharewere.depend upon the need of your project you can purchase or free download from the internet.

As used are using the notepad on the windows operating system,

The Step for open a editor , Open a notepad->Write the Source code in C Langauage->after completing the source code->then save the Source Code by used the extension .c

If I name the program as HelloWorld.c

5.5 Compiling the Source Code

As C languege is known es the high level language.the computer cannot understand the high level lenguege, A computer need a digital or binary instruction in what we called as machine languaga. Before you run the C program,first of all tyou have to translated the C progrem or Source code which is human readable, in to a mechine code that's the computer understand, For conversion form the high level language into machine level language, you need the tools for conversion is known as Compiler. The compiler takes a source code es en input and produce a file which is known as exectable file (.exe) which is corresponds to the source code.The machine instruction which is created by the compiler is known as object code . The file containing the object code is known es object file.

Each compiler has its own set of command to create a object code. To compile, you use the commend to run the compiler following with the filename. The following exemple show the use of the command to run the compiler to compiler the source file called as Helloworld using various DOS/Windows Compiler.

Compiler	*Command*
Microsoft C	cl HelloWorld.c
Borlend's Turbo C	tcc HelloWorld.c
Borland C	bcc HelloWorld.c
Zortec C	ztc HelloWorld.c

To Compile HelloWorld.c under the Unix Operating system, the following Command is used.

cc HelloWorld.c

If you do not know which exact complier is used for compile the source code ,contact the consultant compiler manual

If you're using the GUI based development environment for creating a source file end compiling the source file, it very easy to used because the Integrated Development enivornment(IDE) contain the compile menu,by clicking the compile menu the source code file and for running the program you can used the run command from the IDE environment

Consider the Exemple famous IDE TOOL Known es TURBO C, which contain the rich set of library file, include file and exe file .you can creete e new file from the file menu, use can save the source file on the herd disk by using seve menu or by using the shorthand key F2, For compiling the source code the shorthand key is ATL+F9 and running the executable file is CTRL +F9

5.6 Linking to Create an Executable File

One more step is required before running your program. C language consist is a function library that contains *object code* (precompiled code) for predefined functions. A *predefined function* contains C code that has already been written and is supplied in e ready-to-use form with your compiler peckage.

We have used the predefined function in C in the previous example like printf() . These library functions frequently need in the program, such as for displeying information console and reeding data from files. If your are uing this predefined function in your program, the object file is created when you compile the source code and combine object code from the function library to create the final executable program. (*Executable* is the file which contains the set of instruction that process required to execute the instruction.) This process is called *linking*, and it's performed by a program called (you guessed it) a *linker*.

Figure5.1 shows creation of source code to object code to executable program.

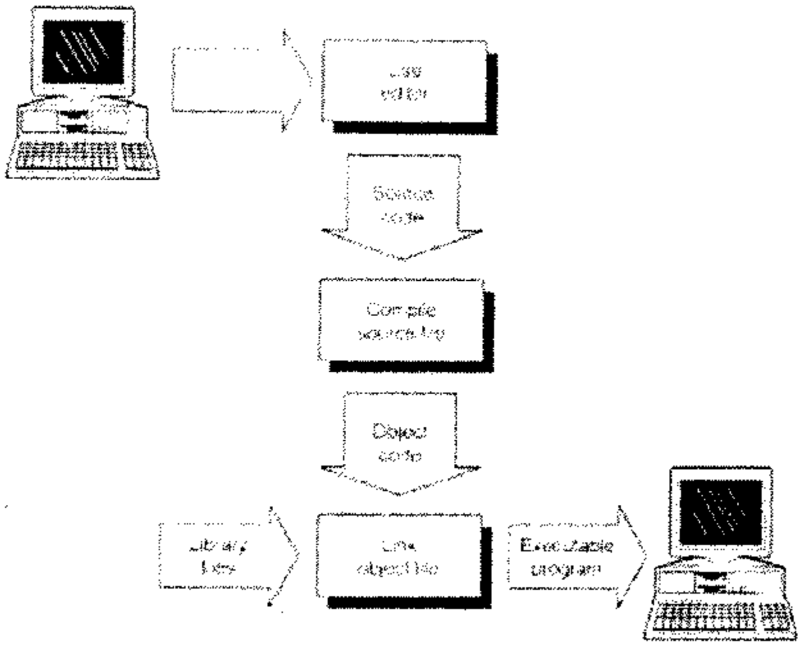


Figure 5.1. The C source code that you write is converted to object code by the compiler and then to an executable file by the linker.

5.7 Completing the Development Cycle

Once your progrem is compiled and linked to create an executable file, you can run it by entering its name at the system prompt or just like you would run any other program. If you run the program and receive results different from what you thought you would, you need to go back to the first step. You must identify what caused the problem and correct it in the source code. When you make a change to the source code, you need to recompile and relink the program to create a corrected version of the executable file. You keep following this cycle until you get the program to execute exactly as you intended.

One finel note on compiling and linking: Although compiling end linking are mentioned es two separate steps, many compilers, such as the DOS compilers mentioned earlier, do both as one step. Regardless of the method by which compiling end linking ere accomplished, understand that these two processes, even when done with one command, are two separate actions.

5.8 The C Development Cycle

Step 1	Use an editor to write your source code. By tradition, C source code files have the extension .C (for example Helloworld.C, demo.c and so on).
Step 2	Compile the program using a compiler. If the compiler doesn't find any errors in the program, it produces an object file. The compiler produces object files with an .OBJ extension and the same name as the source code file (for example, Helloworld.C compiles to Helloworld.OBJ). If the compiler finds errors, it reports them. You must retum to step 1 to make corrections in your source code.
Step 3	Link the program using a linker. If no errors occur, the linker produces an executable program loceted in a disk file with an .EXE extension and the same name as the object file (for exemple, Helloworld.OBJ is linked to create Helloworld.EXE).
Step 4	Execute the program. You should test to determine whether it functions properly. If not, start again with step 1 and make modifications and additions to your source code.

5.9 Your First C Program

You're probably eager to try your first program in C. To help you become familiar with your compiler, here's a quick program for you to work through. You might not understand everything at this point, but you should get a feel for the process of writing, compiling, and running a reel C program.

This demonstration uses e program named **HELLOWORLD**.C, which does nothing more than display the words Hello, World! on-screen. This program, a traditional introduction to C programming, is a good one for you to leam. The source code for **HELLOWORLD**.C is in Listing5.1. When you type in this listing, you won't include the line numbers or colons.

Listing 5.1. HELLOWORLD.C.

```
1: #include <stdio.h>
2:
3: main()
4: {
5:    printf("Hello, World!\n");
6:    retum 0;
7: }
```

Be sure that you have installed your compiler as specified in the installation instructions provided with the software. Whether you are working with UNIX, DOS, or any other operating system, make sure you understand how to use the compiler and editor of your choice. Once your compiler and editor are ready, follow these steps to enter, compile, and execute HELLO.C.

5.10 Entering and Compiling HELLOWORLD.C

To enter and compile the **HELLOWORLD**.C program, follow these steps:

1. Meke ective the directory your C programs ere in and start your editor. As mentioned previously, any text editor can be used, but most C compilers (such as Borland's Turbo C++ and Microsoft's Visual C/C++) come with an Integrated development environment (IDE) that lets you enter, compile, and link your programs in one convenient setting. Check the manuals to see whether your compiler has an IDE available.

2. Use the keyboard to type the **HELLOWORLD**.C source coda exactly as shown in Listing 5.1. Press Enter at the end of each line.

3. Save the source code. You should name the file **HELLOWORLD**.C.

4. Verify that **HELLOWORLD**.C is on disk by listing the files in the directory or folder. You should see **HELLOWORLD**.C within this listing.

5. Compile and link **HELLOWORLD**.C. Execute the eppropriete command specified by your compiler's manuals. You should get a message stating that there were no errors or warnings.

6. Check the compiler messages. If you receive no errors or wemlngs, everything should be okay.

If you made an error typing the program, the compiler will catch it and display an error message. For example, if you misspelled the word printf as pmtf, you would see a message similar to the following:

Error: undefined symbols:_pmtf in **HELLOWORLD**.c (**HELLOWORLD**.OBJ)

7. Go back to step 2 if this or any other error message is dlspleyed. Open the **HELLOWORLD**.C file in your editor. Compere your file's contents carefully with Listing 5.1, make any necessary corrections, and continue with step 3.

8. Your first C program should now be compiled and ready to run. If you display a directory listing of ell files named **HELLOWORLD** (with eny extension), you should see the following:

HELLOWORLD.C, the source code file you creeted with your editor

HELLOWORLD.OBJ or **HELLOWORLD**.O, which conteins the object code for HELLO.C

HELLOWORLD.EXE, the executable program created when you compiled and linked **HELLOWORLD**.C

9. To *execute*, or run, **HELLOWORLD.EXE**, simply enter hello. The messege Hello, World! is displayed on-screen.

❖❖❖

6

OPERATORS AND EXPRESSIONS

Contents

6.1 Arithmetic, Unary, Logical, Bit-Wise, assignment and Conditional Operators

You will leam about C operator i.e. Arithmetic operators, Relationel Operators, Logical Operators, Assignment Operators, Increments and Decrament Operators, Conditional Operators, Bitwise Operators end Special Operators.

Definition of Operators: Operator is symbol which helps the user to command to do certein mathematical or logical operation on data and verieble. C has the large amount of operators which has been classified as.

1. Arithmetic operators
2. Relational Operators
3. Logical Operators
4. Assignment Operetors
5. Increments and Decrement Operators
6. Conditional Operators
7. Bitwise Operators
8. Special Operators

6.1 Arithmetic Operators

All basic operators can be carried out in C language. All the operetors have almost the same meanlng as in other languages. C has two unery mathematical operators and five binery mathematical operators.

Arithmetic Operators

Operator	Meaning
+	Addition or Unary Plus
–	Subtraction or Unary Minus
*	Multiplication
/	Division

```
#include <stdio.h>//include header file stdio.h
void main() //tell the compiler the start of the program
{
int numb1, num2, sum, sub, mul, div, mod; //declaration of variables
scanf ("%d %d", &num1, &num2); //inputs the operands

sum = num1+num2; //addition of numbers and storing in sum.
printf("\n Thu sum is = %d", sum); //display the output

sub = num1-num2; //subtraction of numbers and storing in sub.
printf("\n Thu difference is = %d", sub); //display the output

mul = num1*num2; //multiplication of numbers and storing in mul.
printf("\n Thu product is = %d", mul); //display the output


div = num1/num2; //division of numbers and storing in div.
printf("\n Thu division is = %d", div); //display the output

mod = num1%num2; //modulus of numbers and storing in mod.
printf("\n Thu modulus is = %d", mod); //display the output
}
```

Operation	Operator	Comment	Value of Sum before	Value of sum after
Multiply	*	sum = sum * 2;	4	8
Divide	/	sum = sum / 2;	4	2
Addition	+	sum = sum + 2;	4	6

Subtraction	-	sum = sum - 2;	4	2
Increment	++	++sum;	4	5
Decrement	--	--sum;	4	3
Modulus	%	sum = sum % 3;	4	1

Operator	**Symbol**	**Action**	**Examples**
Increment	++	Increments the operand by one	++x, x++
Decrement	--	Decrements the operand by one	--x, x--

The increment and decrement operators can be used only with variables, not with constants. The operation performed is to add one to or subtract one from the operand. In other words, the statements
++a;
--a;
are the equivalent of these statements:
a = a + 1;
a = a - 1;

C has two unary operators for incrementing and decrementing the operand. The increment operator ++ adds 1 to its operand; the decrement operator --subtracts 1to its operand. Both increment ++ and decrement-- can be used either es prefix operators (before the operand: ++a) or postfix operators (after the operand: n++)
A prefix(++a) operator first add 1 to operand and then the result is assigned to the variable on the left
A postfix (a++)operator first essign the value to the variable on the left end then increments the operand

```
For an example :
#include<stdio.h>
#include<conio.h>
Void main()
{
int m=10; int n=20;
printf("m=%d\n",m);
printf("n=%d\n",n);
printf("++m=%d\n",++m);
printf("n++=%d\n",n++);
printf("m=%d\n",m);
printf("n=%d\n",n);
}
m=10
```

```
Output:

n=20
++m=11
n++=20
m=11
n=21
```

6.1.1 Integer Arithmetic

The arithmetic operation is performed on two whole numbers or integers that operation is celled as integer erithmetic. This operation always gives an integer as the result. Let x = 7and y = 6 be 2 integer numbers. Then the integer operation leads to the following results.

x + y = 13
x – y = 1
x * y = 48
x % y = 1
x / y = 1

In integer division the fractional part is truncated.

6.1.2 Floating point arithmetic

When an arithmetic operation is preformed on two reel numbers or fraction numbers such an operation is called floating point arithmetic. The floating point results can be truncated according to the properties requirement. The remainder operator is not applicable for floating point arithmetic operands.

Let x = 14.0 and y = 4.0 then

x + y = 18.0
x – y = 10.0
x * y = 56.0
x / y = 3.50

6.1.3 Mixed mode arithmetic

When one of the operand is real and other is an integer and if the arithmetic operation is carried out on these 2 operands then it is called as mixed mode arithmetic. If any one operand is of real type then the result will alweys be real thus 15/10.0 = 1.5

6.1.4Precedence of operators

Level	Operator	Description	Grouping
1	::	Scope	Left-to-right
2	() [] . -> ++ -- dynamic_cast static_cast reinterpret_cast const_cast typeid	Postfix	Left-to-right
3	++ -- ~ ! sizeof new delete	unary (prefix)	Right-to-left
	* &	indirection and reference (pointers	
	+ -	unary sign operator	
4	(type)	type casting	Right-to-left
5	.* ->*	pointer-to-member	Left-to-right
6	* / %	Multiplicative	Left-to-right
7	+ -	Additive	Left-to-right
8	<< >>	Shift	Left-to-right
9	< > <= >=	Relational	Left-to-right
10	== !=	Equality	Left-to-right
11	&	bitwise AND	Left-to-right
12	^	bitwise XOR	Left-to-right
13	|	bitwise OR	Left-to-right
14	&&	logical AND	Left-to-right
15	||	logical OR	Left-to-right
16	?:	Conditional	Right-to-left
17	= *= /= %= += -= >>= <<= &= ^= |=	Assignment	Right-to-left
18	,	Comma	Left-to-right

6.2 Relational Operators

There are six relation operators in C. They are

These six operators are used to form logical expressions, which represent conditions that are either true or false. The resulting expressions will be of type integer, since ***true*** is represented by the integer value 1 and ***false*** is represented by the value ***0.***

Here there are some examples:

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

```
(7 == 5)   // evaluates to false.
(5 > 4)    // evaluates to true.
(3 != 2)   // evaluates to true.
(6 >= 6)   // evaluates to true.
(5 < 5)    // evaluates to false.
```

of course, instead of using only numeric constants, we can use eny velid expression, including veriables.
Suppose that a=2, b=3 end c=6,

```
(e == 5)     // evaluates to false since a is not equal to 5.
(e*b >= c)   // evaluates to true since (2*3 >= 6) is trua.
(b+4 > a*c)  // evaluates to false since (3+4 > 2*6) is false.
((b=2) == a) // evaluates to true.
```

Be careful! The operator = (one equel sign) is not the same as the operator == (two equel signs), the first one is en assignment operator (assigns the velue at its right to the variable at its left) and the other one (==) is the equality operator that compares whether both expressions in the two sides of it are equel to each other. Thus, in the last expression ((b=2) == a), we first assigned the value 2 to b end then we compared it to e, that elso stores the velue 2, so the result of tha operation is true.

C's logical operators.

C has the following logical operators, they compare or evaluate logical and relational expressions.

Operator	Meaning
&&	Logical AND
||	Logical OR
!	Logical NOT

6.3 Logical and (&&)

The logical operators && and || are used when evaluating two expressions to obtain a single relational result. The operator && corresponds with Boolean logical operation AND. This operetion results true if both its two operands are true, and false otherwise. The following panel shows the result of operetor && avalueting the exprassion a && b:

&& OPERATOR

e	b	a && b
true	trua	true
true	felse	false
felse	true	felse
false	false	false

This operator is used to evaluate 2 conditions or expressions with relational operators simultaneously. If both the expressions to the left end to the right of the logical operator is true then the whole compound expression is true.

Example

a > b && x = = 10

The expression to the left is a > b end that on the right is x == 10 the whole expression is true only if both expressions are true i.e., if e is greeter than b and x is equal to 10.

6.4 Logical Or (||)

The operator || corresponds with Boolean logical operation OR. This operetion results true if either one of its two operands is true, thus being false only when both operands ere false themselves. Here ere the possible results of a || b:

|| OPERATOR

a	b	a || b
true	true	true
true	felse	true
false	true	true

false	false	false

The logical OR is used to combine 2 expressions or the condition evaluates to true if any one of the 2 expressions is true.

Example

e < m || a < n

The expression evaluates to true if any one of them is true or if both of them era true. It evaluetes to true if a is less than either m or n and when a is less than both m and n.

or example:

```
( (5 == 5) && (3 > 6) ) // evaluates to false ( true && false ).
( (5 == 5) || (3 > 6) ) // evaluates to true ( true || false ).
```

6.5 Logical Not (!)

The Operator ! is the C operator to perform the Boolean operation NOT, it has only one operand, located at its right, and the only thing that it does is to inverse the value of it, producing false if its operand is true and true if its operand is false. Basically, it returns the opposite Boolean velue of evaluating its operand. For example:

```
!(5 == 5)   // evaluates to false because the expression et its right
(5 == 5) is true.
!(6 <= 4)   // evaluates to true because (6 <= 4) would be false.
!true       // evaluates to false
!false      // evaluates to true.
```

6.5.1 Compound Assignment Operators

The compound assignment operators consist of e binary operetor and the simple assignment operator. They perform the operation of the binery operator on both operands and store the result of that operation into the left operand, which must be a modifiable velue.

Assignment Operators

Operator	Meaning
=	Store the value of the second operand in the object specified by the first operand (simple assignment).
*=	Multiply the value of the first operand by the value of the second operand; store the result in the object specified by the first operand.
/=	Divide the value of the first operand by the value of the second operand; store the result in the object specified by the first operand.
%=	Take modulus of the first operand specified by the value of the second operand; store the result in the object specified by the first operand.
+=	Add the value of the second operand to the value of the first operand; store the result in the object specified by the first operand.
–=	Subtract the value of the second operand from the value of the first operand; store the result in the object specified by the first operand.
<<=	Shift the value of the first operand left the number of bits specified by the value of the second operand; store the result in the object specified by the first operand.
>>=	Shift the value of the first operand right the number of bits specified by the value of the second operand; store the result in the object specified by the first operand.
&=	Obtain the bitwise AND of the first and second operands; store the result in the object specified by the first operand.
^=	Obtain the bitwise exclusive OR of the first and second operands; store the result in the object specified by the first operand.
|=	Obtain the bitwise inclusive OR of the first and second operands; store the result in the object specified by the first operand.

you want to increase the value of x by 5, or, in other words, add 5 to x and assign the result to x. You could write

```
x = x + 5;
```

Using a compound assignment operator, which you can think of as a shorthand method of assignment, you would write

```
x += 5;
```

In more general notation, the compound assignment operators have the following syntax (where op represents a binary operator):

exp1 op= *exp2*

This is equivalent to writing

exp1 = *exp1* op *exp2*;

Whan we want to modify the value of a variable by performing an operation on tha value currently stored in that variable we can use compound assignment operators:

expression	is equivalent to
value += increase;	value = value + increase;
a -= 5;	a = a - 5;
a /= b;	a = a / b;
price *= units + 1;	price = price * (units + 1);

and the same for all other operators. For example:

// compound assignment operators
 	
#include <sdtio.h>
int main ()
{
int a, b=3;
a = b;
a+=2; // equivalent to a=a+2
printf("a=%d"\n,a);
return 0;
}
	5

6.6 The Conditional Operator

C has one last operator which we haven't seen yet. It's called the conditional or ``ternary'' or ?: operator, and in action it looks something like this:
The syntax of the conditional operator is
e1 ? *e2* : *e3*
and what happens is that *e1* is evaluated, and if it's true then *e2* is evaluated and becomes the result of the expression, otherwise *e3* is evaluated and becomes the result of the expression. In other words, the conditional expression is sort of an if/else statement buried inside of an expression.

```
#include<stdio.h>
int main(){
int a,b;
printf("Enter a Number:");
scanf("%d",&a);
printf("\n Enter 2nd Number:");
scanf("%d",&b);
a>b ? printf("A is big") : printf("B is Big");
return 0;
}
```

A classic if/else would use this syntax:

```
int max;
if (a > b)
{
max = a;
}
else
{
max = b;
}
```

Using the ternary or conditional operator ? : we could shorten this to:

```
int max;
max = (a > b) ? a : b;
```

It means that first (a > b) is evaluated; if it is true, the value of expression would be the value given before : (the value between ? and :), otherwise, it would be the value after :

6.7 Bitwise Operators

One of C's powerful features is a set of bit manipulation operators. These permit the programmer to access and manipulate individual bits within a piece of data.

Operator	Meaning
~	One's Complement Operator
>>	Right Shift Operator
<<	Left Shift Operator
&	Bitwise AND Operator
|	Bitwise OR Operator
^	Bitwise XOR Operator

```
//A program to demonstrate the working of bitwise operators.
#include<stdio.h>
int main()
{
int n = 149;
int res;
res = n & 0017;
printf("The resultant of Bit wise AND operator is : %d \n", res);
res = n | 0017;
printf("The resultant of Bit wise OR operator is : %d \n", res);
res = n && 0017; // this is logical AND . Truth or false will be output
printf("The resultant of Logical AND operator is : %d \n", res );
res = n || 0017; // this is logical OR . Truth or false will be output
printf("The resultant of Logical OR operator is : %d \n", res);
res = n ^ 0017;
```

```
printf("The resultant of Exclusive operator is : %d \n", res);
res = n <<2;
printf("The resultant of shift left ( by 2 bits) operator is : %d \n", res);
res = n >>2;
printf("The rasultant of shift right ( by 2 bits) oparator is : %d \n",
res);
res = ~n;
printf("The resultant of NOT operator is : %d \n", res);
retum 0;
}
```

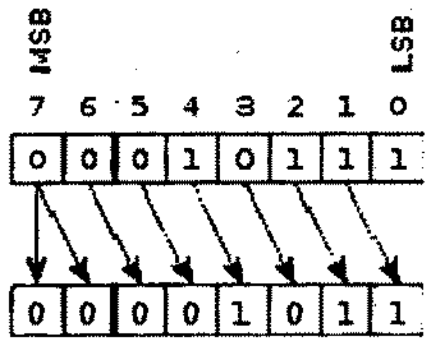
The resultant of Bit wise AND operator is : 5
Tha resultant of Bit wisa OR operator is : 159
The resultant of Loglcal AND operator is : 1
Tha resultant of Logical OR operator is : 1
The resultant of Exclusiva oparator is : 154
The resultant of shift left (by 2 bits) operator is : 576
Tha resultant of shift right(by 2 bits) oparator is : 36
The resultant of NOT operator is : -150

6.7.1 ~ Tilde operator . one's complement Operator. This is a unary operator, used to find one's
complement of a givan number .

n = 1 0 0 1 0 1 0 1 =149(decimal)
~n = 0 1 1 0 1 0 1 0 = bit wise complement

6.7.2 Rlght Shift Operator

The right shift operator Is represented by >>. It needs two oparands. It shifts each bit in its left operand to the right. The numbar of places tha bits are shifted depends on the number following tha operator (i.e. its right operand).
Thus, **ch >> 4** would shift all bits in **ch** four places to the right. Similarly, **ch >> 6** would shlft all bits **6** places to the right.



Right arithmetic shift

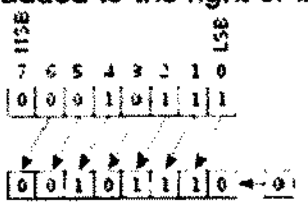
Note that as the bits are shifted to the right, blanks are created on the left. These blanks must be filled somehow. Thay are always filled with zeros.

6.7.3 Right Shift. Shifting right by one position , bits of a binary number is equal to division of the given number with 2.

n = 1 0 0 1 0 0 0 0 =144(decimal)
n>> 1 0 1 0 0 1 0 0 0 = 72(decimal)
n>>2 0 0 1 0 0 1 0 0 = 36(decimal)

6.7.4 Left Shift Operator

This is similar to the right shift operator, the only difference being that the bits are shifted to the left, and for each bit shifted, a 0 is added to the right of the number



Left arithmetic shift

6.7.5 << Left Shift. Shifting left by one position , bits of a binary number is equal to multiplying the number with 2.

n = 1 0 0 1 0 0 0 0 =144(decimal)
n<< 1 1 0 0 1 0 0 0 0 0 = 288(decimal)
n<< 2 1 0 0 1 0 0 0 0 0 0 = 576

6.7.6 Bitwise AND Operator

This operator is represented as **&**. Remember it is different than **&&**, the logical AND operator. The **&** operator operates on two operands. While operating upon these two operands they are compared on a bit-by-bit basis. Hence both the operands must be of the same type (either **char** or **int**). The second operand is often called an AND mask.

The **&** operator operates on a pair of bits to yield a resultant bit.

First bit	Second bit	Resultant bit
0	0	0
0	1	0
1	0	0
1	1	1

The example given below shows more clearly what happens while ANDing one operand with another.

This operand when ANDed bitwise

1	0	1	0	1	0	1	0

With this operand yields

1	1	0	0	0	0	1	1

this result

1	0	0	0	0	0	1	0

6.7.7 & Bit wise AND. Used for masking operation. For axample if you want to mask first four bits of a number 'n' , then we will mask n with e number whose last four bits are 1s. i.e. 0001111. In Octal representation it is 017(Remamber an octel number starts with 0 and e hexa number starts with 0x)

n= 1 0 0 1 0 1 0 1 =149(decimal)
& 0 0 0 0 1 1 1 1 = 017(octal)
result n = 0 0 0 0 0 1 0 1

Note that last four bits ere 0101 and are uneffected l.e. they are just reproduce in the result , whereas , tha left four bits are all 0s. i.e. they are *masked.*

Bitwtse OR Operator

Another importent bitwise operator is the OR operator which is represented es **|**. The rules that govern the value of the resulting bit obtained after ORing of two bits is shown in tha truth table below.

First bit	Second bit	Resultant bit
0	0	0
0	1	1
1	0	1
1	1	1

Using the Truth table confirm the result obtained on ORing the two operands as shown below.

11010000 Original bit pattern
00000111 OR mask

11010111 Resulting bit pattern

6.7.6 Bitwise XOR Operator

The XOR operetor is represented as ^ and is elso called an Exclusive OR Oparator. The OR operator retums 1, when eny one of the two bits or both the bits are 1, whereas XOR retums 1 only if one of the two bits is 1.

The truth table for the XOR operator is given below

First bit	Second bit	Resultant bit
0	0	1
1	1	0

XOR operator is used to toggle a bit ON or OFF. A number XORed with another number twice gives the original number

6.7.9 ^ Bit wise Exclusive OR. Exclusive OR also known as odd function , produces output

1 , when both bits are not same (odd) and produces a 0 when both bits are same.

n = 1 0 0 1 0 1 0 1 =149(decimal)

^ = 0 0 0 0 0 1 0 1 = 005(octal)

result n = 1 0 0 1 0 0 0 0 =149(decimal)

❖❖❖

7

CONTROL STATEMENTS

Contents

While, do-while, for statements, nested loops, if else, switch, break, Continue, and goto statements, comma operators

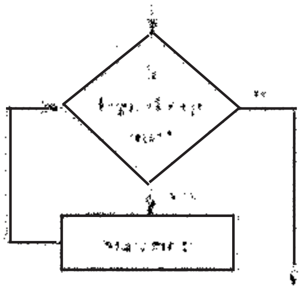
7.1 The While Statement

In most programming lenguages, e **while loop** is a control flow statement that allows code to be executed repeatedly based on e given Boolean condition i.e True or Flase. The while loop can be consider as a repeation of if statement.
The *while* statement consists of a block of code or series of Instruction or staement end a condition. The condition is firstly evaluated, and if the condition is true, the code within the block of code is executed. This execution of statement repeated until unless the condition becomes felse. Because *while* loops check the condition before the block is executed, the control structure is often also known as a **pre-test loop**.

1. The expression *condition* is evaluated.
2. If *condition* evaluates to false (that is, zero), the execution of while statement terminates, and control of execution is passes to the first statement following *stetement*
3 .If *condition* evaluates to true (that is, nonzero), the execution of while statement follows, the series of instruction or statement is executed unless the condition becomes false.
4. Execution returns to step 1

Figure. *The operation of a while stetement.*

WHILE inup-test at beginning



The general format for a while loop is

```
while (condition) {
      simple or compound statement ;//body of the loop
   }
#include<stdio.h>
#include<conio.h>
main(){

   int i = 0;
   Clrscr();
   while (i<5){
      printf(" the value of i is %d\n", i);
      i = i + 1;
   }
getch();
}
```

output:

```
the value of i is 0
     the value of i is 1
     the value of i is 2
     the value of i is 3
```

Program for Calculating the Factorial of a number

```
unsigned int counter = 5;
unsigned long factorial = 1;
 void main()
{

while (counter > 0)
{
 factorial *= counter--;    /* Multiply and decrement */
}

printf("%lu", factorial);
```

}
Output:125

The do...while Loop

The "do while loop" is almost the same as the while loop. The "do while loop" has the following form:

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

Notice thet the condition is tested at the end of the block instead of the beginning, so the block of statement will be executed at least once. If the condition is true, The cursor jump back to the beginning of the block and execute the block of statement again. A do..while loop is almost same es e while loop except that the loop body is permitted to execute at leest once.

A while loop says "Loop while the condition is true, and execute this block of code",

A do... while loop says "Execute this block of code, and then continue to loop while the condition is true".

Figure *The operation of a do...while loop.*

The statements associated with a do...while loop are always executed at least once. This is because the test condition is evaluated at the end, instead of the beginning, of the loop. In opposite, for loops end while loops evaluate the test condition at the beginning of the loop, so the associated statements are not executed at all ,if the test condition is initially false.

Example

```
#include<stdio.h>

    int main()
    (
            int counter, howmuch;
            scanf("%d", &howmuch);
            counter = 0;
            do
            {
            counter++;
            printf("%d\n", counter);
            )
            while ( counter < howmuch);
            return 0;
    }
```

In this Program , the block of statement is executed at least once that is counter variable and printf() stetement is exected at one time here the counter variable is initialized to zero et declaration of the variable, when the instruction is flows inside do while loops than the counter variable is increment by one and followed by the printf() statement is executed i.e it will prints the value of the counter variable is 1 to once, then the while condition is checked whether the condition is false or true , if the condition is false, the loop will terminated, only the value at beginning will print at once , if the condition is true , the control jump back at beginning of loop , the block of statement of an do..while untill unless the codition becomes false.

Be eware that you must put e tralling semi-colon after the while in the above example. A naïve programmer elways forget to put a trailing semicolon after the while statement it consider that do while loop must be terminated with a semicolon (the other loops should not be terminated with a semicolon, adding to the confusion). Notice that this loop will execute once, beceuse it automatically executes before checking the condition.

7.2 The For Statement

```
for (initielization_expression;loop_condition;increment_expression){
  // statements
}
```

For loops are divided in to three parts that is seperated by semi-colons in control block of the for loop.

The first part is initialization_expression is executed before execution of the loop starts. This is pert is used to initielize a counter for the number of loop iterations. You can initialize a counter for the loop in this part.

The second part is used to check the conditlon, depend upon the condition execution of the loop is continues until unless the *loop_condition* is false. This expression is checked at the beginnlng of each loop iteration.

The *increment_expression*, is usually used to increment the loop counter. This is executed at the end of each loop iteration.

```
#include <stdio.h>
  void main(){
   // using for loop statement
   int max = 5;
   int i = 0;
  for(i = 0; i < max;i++){

      printf("%d\n",i);
     }
}
```

And the output is

```
1
2
3
4
5
```

This programm indicates that the max variable is initialized to 5 Integer value, he for loop contains the initialized expression i.e. 1 = 0, the loop_condition i.e. i<max is checked, the first iteration checked 0<5, the condition is true the block of statements is executed, then the loop counter variable is increment by one i.e. i++, the i variable becomes 1 (i=1), then the condition is checked (i<1), the condition is true, the block of statement is executed, the counter variable is incremented by 2(i=i+1) the for loop is iterated until unless the condition becomes false(5<5), the condition is checked, the condition is false and loops is terminated, the control is come out of loops, the statement after the for loop is executed.

7.3 Nested Loops

Nested for statements

The loop within the loop is called a nested loops. These types of loops are used to create matrix. Any loop can contain a number of loop statements in iteseIf. If we are using loop within loop that is called nested loop. In this the outler loop is used for counting number of rows and the internal loop is used for counting number of columns.

```
SYNTAX:-
for (initializing ; test condition ; increment / decrement)
{
statement;
for (initializing ; test condition ; increment / decrement)
{
body of inner loop;
}
statement;
}
PROGRAM
#include
void main ( )
{
int i, j;
clrscr ( );
for (j=1; j<=4; j++){
for (i=1; i<=5; i++){
printf ("*")
}
```

```
printf ("\n");
}
getch ( );
}
```

OUTPUT OF THIS PROGRAM IS

```
****
****
****
****
```

7.3.1 Nested Loops

The term *nested loop* refers to a loop that is contained within another loop. You have seen examples of some nested statements. C places no limitations on the nesting of loops, except that eech inner loop must be enclosed completely in the outer loop; you can't have overlapping loops. Thus, the following is not allowed:

```
for ( count = 1; count < 100; count++)
{
do
{
/* the do...while loop */
} /* end of for loop */
}while (x != 0);
```

If the do...while loop is placed entirely in the for loop, there is no problem:

```
for (count = 1; count < 100; count++)
{
do
{
/* the do...while loop */
}while (x != 0);
} /* end of for loop */
```

When you use nested loops, remember that changes made in the inner loop might affect the outer loop as well. Note, however, that the inner loop might be independent from eny variables in the outer loop; in this example, they are not. In the previous exemple, if the inner do...while loop modifies the value of count, the number of times the outer for loop executes is affected.

7.4 The If Statement

This allows us to control the flow of the program, lets the statements it make decisions on what code to execute, it is important for the programmer point of view. The if statement ellows you to control if a program enters a section of code or not based on whether a given condition is true or false. One of the important aspect of the *functions that is the if statement ellows the program* to select an action based upon the user's input. For example, by using an if statement to check e user antered password, your program can decide whether a user is allowed access to the program.

Tha form of an if statement is as follows:

```
if (expression)
stetement;
```

If *expression would be combination of reletional operator or the logical or arthemtic* evaluates to true, *statement* is executed. If *expression* avaluetes to false, *statement* is not executed.

A block of statement can be used anywhere a single statement can be used. Therefore, you could write an
if statement es follows:

```
if (axpression)
{
statement1;
statement2;
/* edditional code goes here */
stetementn;
}
```

LIST.C: Demonstrates if statements.

```
/* Demonstrates the use of if statements */

#include <stdio.h>

main()
{

int x, y;
/* Input the two values to be tested */
printf("\nInput an integer value for x: ");
scenf("%d", &x);
printf("\nInput an integer value for y: ");
scanf("%d", &y);

/* Test values end print result */

if (x == y)
printf("x is equal to y\n");
```

```
if (x > y)
printf("x is greater than y\n");

if (x < y)
printf("x is smaller then y\n");

return 0;
}
```

```
Input en integer value for x: 100
Input an integer value for y: 10

x is greater then y
Input an integer value for x: 10
Input an integer value for y: 100
x is smaller than y
Input an integer value for x: 10
Input an integer value for y: 10
x is equel to y
```

7.5 The Else Clause

if..else syntax is as follows:

```
if ( condition ) {
expr_set1;
}
else {
expr_set2;
}
```

If given condition is evaluated as TRUE then expr_set1 will get executed.

If given condition is evaluated as FALSE (not TRUE), expr_set2 will get executed

if..else example

The progrem for find out large number of two from given input:

```
include<stdio.h>
int main(){
      int x,y;
      printf("Enter value for x :");
      scanf("%d",&x);
      printf("Enter value for y :");
      scanf("%d",&y);
      if ( x > y ){
            printf("X is large number - %d\n",x);
      }
      else{
            printf("Y is large number - %d\n",y);
      )
```

```
    return 0;
}
```

Output:
Enter value for x : 20
Enter value for y: 10

X is large number- 20

7.6 The If Statement

Form 1

if(*expression*)
statement1;
next_statement;
if statement as an simplest form. If *expression* Is true, *stetement1* is executed. If *expression* is not true, *statement1* is not executed.
Form 2
if(*expression*)
statement1;
else
statement2;
next_statement;

This is the most common form of the if statement. If *expression* is true, *statement1* is executed; otherwise, *statement2* is executed.
Form 3

if(*expression1*)
statement1;
else if(*expression2*)
statament2;
else
statement3;
next_statement;

This is a called as nested if. If the first expression is evaluated and *expression1* is true, *statement1* is executed before the program continues with the *next_statement*. If the first expression is not true(i.e), the second expression, *expression2*, is checked. If the first expression is not true, end the second is true, *statement2* is executed. If both expressions are false, *statement3* is executed. Only one of the three stetements is executed.

Following example of nested if statement.

```
void main
{
 int result;
Printf("Enter the percentage of students\n");
```

```
Scanf("%d",result);
if (result >= 75)
      printf("Passed: Grade A\n");
else if (result >= 60)
      printf("Passed: Grade B\n");
else if (result >= 45)
      printf("Passed: Grade C\n");
else
        printf("Failed\n");
}
```

Out put
Enter the percentage of students
80
Passed: Grade A

7.7 The Switch Statement

Switch case statements are applicable for long if statements that compare a variable to several "integral" values ("integral" values are simply values that can be expressed as an integer, such as the value of a char). The basic format for using switch case is outlined below.
The value of the variable given into switch is compared to the value following each of the cases, and when one value matches the value of the variable, the computer continues executing the program from that point.

```
switch ( <variable> ) {
case this-value:
  Code to execute if <variable> == this-value
  break;
case that-value:
  Code to execute if <variable> == that-value
  break;
...
Default
```

In this statement, *expression* is any expression that evaluates to an integer value: type long, int, or char.
The switch statement evaluates *expression* and compares the value against the templates following each case label, and then one of the following happens:
If a match is found between *expression* and one of the templates, execution is transferred to the statement that follows the case label.
If no match is found, execution is transferred to the statement following the optional default label.

If no match is found and there is default label is executed, execution passes to the first statement following the switch statement's closing brace.

```
main(){
int a;
printf("\n Enter a value:");
scanf("%d",&a);
switch(a)
{
case 1:
printf(" \nThis is case 1");
break;
case 2:
printf(" \nThis is case 2");
break;
case 3:
printf(" \nThis is case 3");
break;
default:
printf("\nDefault case");
}
}
```

Output:

Enter a value:2
This is case 2

Example 1

```
switch( letter )
{
case `A':
case `a':
printf( "You entered A" );
break;
case `B':
case `b':
printf( "You entered B");
break;
...
...
default:
printf( "I don't have a case for %c", letter );
}
```

7.8 The Break Statement

The **break** statement terminates the execution of the nearest enclosing **do**, **for**, **switch**, or **while** statement in which it appears. Control passes to the statement that follows the terminated statement.

Syntax

jump-statement:
break;

The **break** statement is frequently used to terminate the processing of a particular case within a **switch** statement. Lack of an enclosing iterative or **switch** statement generates an error.
Within nested statements, the **break** statement terminates only the **do**, **for**, **switch**, or **while** statement that immediately encloses it. You can use a **return** or **goto** statement to transfer control elsewhere out of the nested structure.
With loops, break can be used to force an early exit from the loop, or to implement a loop with a test to exit in the middle of the loop body. A break within a loop should always be protected within an if statement which provides the test to control the exit condition.

```
#include<stdio.h>

    int main()
    {
            int i;

            i = 0;
            while ( i < 20 )
            {
            i++;
            if ( i == 10)
                    break;
            }
            return 0;
    }
```

In the example above, the while loop will run, as long i is smaller then twenty. In the while loop there is an if statement that states that if i equals ten the while loop must stop (break).

7.9 The Continue Statement

With "continue;" it is possible to skip the rest of the commands in the current loop and start from the top again. (the loop variable must still be incremented). Take a look at the example below:

```
#include<stdio.h>

    int main()
    {
            int I;

            i = 0;
            while ( i < 20 )
```

```
        {
        i++;
        continue;
        printf("Nothing to see\n");
        }
        return 0;
}
```

In the example above, the printf function is never called because of the "continue;".

7.10 The Goto Statement

The goto statement is a jump statement which jumps from one point to another point within a function. The goto statement is marked by label statement. Label statement can be used anywhere in the function above or below the goto statement. You can see in the given example, we want to display the numbers from 0 to 9. For this, we have defined the label statement loop above the goto statement. The given program declares a variable n initialized to 0. The n++ increments the value of n till the loop reaches 10. Then on declering the goto statement, it will jumps to the label statement and prints the value of n.

```
#include <stdio.h>
#include <conio.h>
int main() {
  int n = 0;
  loop: ;

  printf("\n%d", n);
  n++;
  if (n<10) {
    goto loop;
  }
  getch();
  return 0;
}
```

7.11 The Comma Operator

The comma operator (,) works almost like tbe semicolon ; that seperates one C statement from another. You can separate almost any kind of C statment from another with a comma operator. The comma-separated expressions are evaluated from left to right and the value of the whole comma-separated sequence is the value of the rightmost expression in the sequence. Consider the following code example.

```
#include <stdio.h>

/* To shorten example, not using argp */
int main (int argc, char *argv[], char *envp[])
{
 int a, b, c, d;

 a = (b = 2, c = 3, d = 4);
 printf ("a=%d\nb=%d\nc=%d\nd=%d\n",
           a, b, c, d);
 return 0;
}
```

The value of (b = 2, c = 3, d = 4) is 4 because the value of its rightmost sub-expression, d = 4, is 4. The value of e is thus also 4. When run, this example prints out the following text:

```
a=4
b=2
c=3
```

❖❖❖

8

STORAGE TYPES

Contents

8.0 Objectives

After reading this chapter you will be able to answer the following:

1. What is meant by storage classes in C?
2. What are the different types of storage classes end how to use them?

8.1 Introduction

We have already seen how variables are declared. Every variable has a name and **type** associated with it. But that's not entiraly true. A variable also has a **STORAGE CLASS**. We haven't used storage class so far while declaring variablas and still managed to get along is because if a veriables storage class is not specified while declaration a defeult storage class is assumed by the compiler. In the sections to come we will see whet does a storage class mean, its types and try to understand them.

8.2 Storage Classes In C

- . A Storage Class is a specifier thet tells the compller how to store a variable.
- Every variable declared in C has e physical location inside the computer where its value is stored. There are two possible locations where the value could be stored – The memory or the CPU registers.
- It is the Storage Class thet tells where the variables value has to be stored.
- The Storage Class indicates the following things about a veriable:
    1. The locetion where the variable would be stored.

2. The **default initial value** of the variable if it is not specified.
3. The **Scope** of the variable
4. The **lifetime** of the veriable.

- The Storage Class Specifiers supported by C are:
  - Automatic
  - External
  - Registar
  - Static
- The general form of a variable declaration that usas Storaga Class specifier is shown here:

```
storaga_specifier type var_name;
```

8.2.1 Automatic

- Local variables are automatic by default.
- A variable declared automatic has the following Properties:

No	Property	Description
1	Keyword	auto
2	Storage Location	Memory
3	Default Initial Value	Garbage Value
4	Scope	Local to the block in which the variable is defined
5	Life	Terminates when the control exits the block in which the variable is defined

Consider the following Example:

```
#include<stdio.h>
#include<conio.h>
void main( )
{
 auto int i = 10,j ;
  {
   auto int i = 11 ;
    {
       auto int i = 12 ;
       printf ( "\n%d %d ", i,j ) ;
    }
   printf ( "\n%d %d ", i,j ) ;
  }
 printf ( "\n%d %d", i,j ) ;
 getch();
}
```

Output

```
12  862
11  862
10  862
```

- In the above program the variables i and j heve been declared es automatic by using the keyword **auto**.
- The variable i is initialized to 10 for the first time while the variable j is not initialized at all.
- The output shows a value 862 when the value of variable j Is printed on the screen which is a garbage value and is unpredictable.
- The variable i has been declared and initialized three times. The compiler treats each i differently as they are declared in different blocks.
- Inside the innermost block the value of variable i is found to be 3 es the control exits this block the value of varieble i is found to be 2 and as it exits this block the value of variable i becomes 1.
- From the above statement it becomes cleer that the scope of I is local to the block in which it is defined. The moment the control comes out of the block in which the veriable is defined, the variable and its value is lost.

8.2.2 Externel

- Externel variables have e global scope.
- They ere declared outside ell functions and are available whenever needed.
- A variable declered external has the following Properties:

No	Property	Deecription
1	Keyword	extern
2	Storage Location	Memory
3	Default Initial Value	Zero
4	Scope	Global (It is avellable throughout the program)
5	Life	Terminates only when the program terminates

- The keyword extern is used to indicate to the compiler thet the object (variable) is declared elsewhere in the progrem.
- Consider the following example:

```
include<stdio.h>
#include<conio.h>
void main( )
{
 extern int x,y;                // x is declared not defined
 printf("x=%d & y=%d",x,y);
 getch();
}
int x=10;               // x is global, external
```

Output:

```
x = 10  & y = 0
```

In the above example, variables x & y are declared external hence they are global. Variable x is initialized outside the main()
The output shows that velue of y is zero, since y is not initialized its default value is printed which is zero since its declared extern.
The variable x is shown to have the velue 10 because it is extern and tells the complier thet its value is declared elsewhere.

8.2.3 Register

- The keyword register tells the compiler to store the variable in the processors register rather than the regular memory.
- The central processing unit (CPU) of a computer contains a few data storage locations called **registers**. It is the register where the actual operations on data are performed. To manipulated data, the CPU transfers the data from memory to the registers performs the operations on it and transfers it back to the memory.
- The benefit of storing a veriable in a register is that it is fester to manipulete the data it contains.
- A variable declared with keyword register has the following Properties:

No	Property	Description
1	Keyword	register
2	Storage Location	Register inside CPU
3	Default Initial Value	Garbage Value
4	Scope	Local to the block in which the variable is defined
5	Life	Terminates when the control exits the block in which the variable is defined

If a variable is used multiple times in a program it is better to declare its storage class as register. Ex. Loop Counters

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

void main( )
{
register int i ;

for ( i = 1 ; i <= 5 ; i++ )
        printf ( "\n%d", i ) ;
getch();
}
```

- In the above program, although variable i is declared to be stored in register it is not sure if that would happen. This is because the number of registers available with any CPU are very limited and they may already be busy in doing some task.

- In such a situation the compiler would treat the variable to be of euto storage class.
-

8.2.4 Static

- Variables declared as **static** ere permenent veriables within their own function or file and maintain their values between calls.
- A variable declared with keyword static has the following Properties:

No	Property	Description
1	Keyword	stetic
2	Storage Location	Memory
3	Defeult Initiel Value	zero
4	Scope	Local to the block in which the variable is defined
5	Life	Terminates when the control exits the program

Consider the Following example:

```
#include<stdio.h>
#include<conio.h>
void add();
void main( )
{
 edd( ) ;
 edd( ) ;
 add( ) ;
 getch();
}
void add( )
{
stetic int x = 11 ;
printf ( "%d\n", x ) ;
x = x + 1 ;
}
```

Output:

```
11
12
13
```

- In the above program, variable x is declared in the function definition of add() and is static.
- A static variable is initialized only once and exists till the program terminates.
- Since x is static, after being initialized to 11 it does not loose its value hence the second call to add() prints the value 12 and third call to add() prints the value 13.

Comparison between Storage Classes in C:

		Automatic	External	Register	Static
No	**Property**	**Description**			
1	Keyword	auto	extern	register	static
2	Storage Location	Memory	Memory	Register inside CPU	Memory
3	Default Initial Value	Garbage Value	Zero	Garbage Value	zero
4	Scope	Local to the block in which the variable is defined	Global (It is available throughout the program)	Local to the block in which the variable is defined	Local to the block in which the variable is defined
5	Lifa	Terminates when the control exits the block in which the variable is defined	Terminates only when the program terminates	Terminates when the control exits the block in which the variable is defined	Terminates when the control exits the program

8.3 References & Further Reading

1. Let us C – Yashwant Kanetkar
2. C The Complete Reference – Herbert Schildt
3. The C Programming Language - Brian W. Kernighan and Dennis M. Ritchie.
4. Teach yourself C in 21 days - Peter Aitken and Bradley L. Jones

8.4 Review Questions

1. What is meant by Storage classes in C?
2. What is special about storing variables in CPU registers? How is it done?
3. What is the difference between static and auto ?

❖❖❖

9

FUNCTIONS

Contents

9.1 Objectives

After reading this chapter you will be able to answer the following:

1. What is a function and why use it?
2. How do we define and use e function?
3. Whet is a function prototype?
4. How to pass arguments to a function?
5. Types of functions : Built-in(library functions) & User defined
6. Static Functions

9.2 Introduction

- Usually programs are much larger than the programs that we heve seen so far.
- To make large programs manageable and less complicated, they are broken down into subprograms. These subprograms are called functions.
- The basic principle of Functions is ***DIVIDE AND CONQUER***. Using functions we can divide a larger tesk into smaller subtasks that are manegeable.

9.3 What Are Functions?

We can understand functions by answering the following questions:

1. **What is a function?**
2. **Why use a function?**
3. **How does a function work?**

1. What is a function?

1. A function is a self contained block of statements.
   A function is self contained in the sense it may hava its own variebles and constants

2. It is designed to do a well dafined task.
   Since a function is a part of a larger program (i.e. a subprogram) it has a particular job to perform.

3. It has e unique name.
   A function can be used (invoked) by the name given to it.

4. It may have retum type
   A function invoked by a celling program may or may not return a velue to it. In case it retums a value tha functions ratum type is the same as tha variables data type.

5. A program that has a function should have the following three things in it:
   a. Function Declaration or Prototype
   b. Function Call
   c. Function dafinition, which are discussed in a latar part of this chaptar.

2. Why use a function? (Adventagas of using e function)

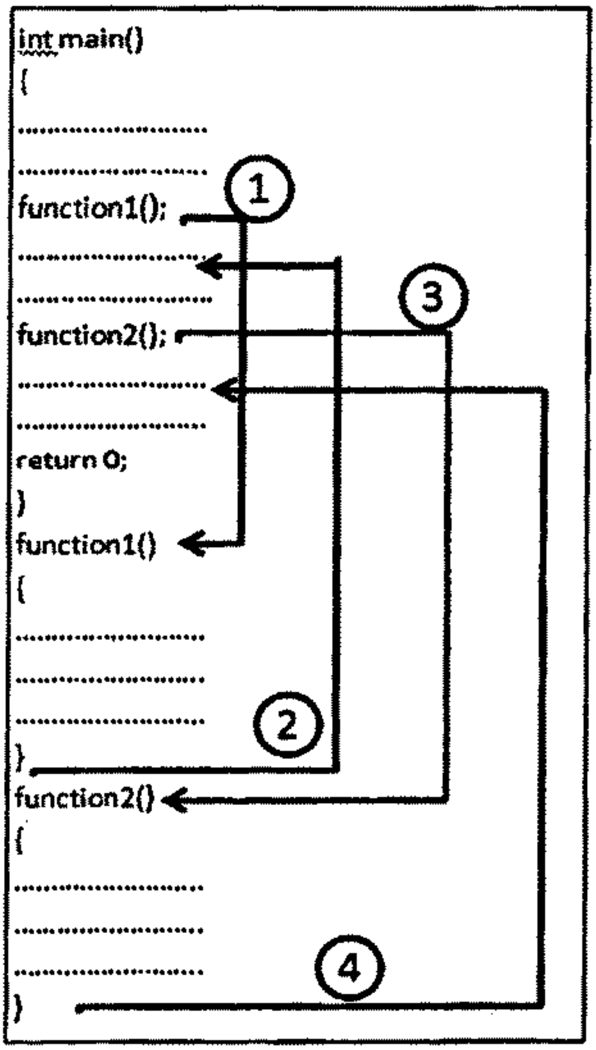
1. Functions are a structured way to programming. Larger progrems get divided Into smaller manageable units.

2. If a specific block of statements has to be axecuted multiple times (for example. taking contact datails from 100 users), it can be written as a function and that function can be repeetedly exacuted. This implies that redundency in writing the sama piece of code multiple times is removed.

3. Dividing e large program into smaller subprograms using functions help to easily code them and reduces the dabugging effort.

3. How does e function work?

Consider the following:



The program to the left contains three functions. First one is the main() function, second is function1() and third is function2().
The execution of any program begins with the execution of main function. Unless there is e decision or looping construct the execution of the program proceeds in a serial manner.
In the diagram to tha left, the program exacution begins with main(), all the statements get executed.
A function gets called when the function name is followed by a semicolon.

A function is defined when function name is followed by a pair of braces in which one or **more statements** may **be present.**

When the system encounters a call to function1() the program control jumps outside the main() function to execute the block of statements named function1() shown by arrow number1.

Once the last statement in function1() is executed the program control is again transferred to main() and the immediete statement aftar mein is axecuted, shown by errow number2.

When the system encounters e call to function2() the program control jumps outside the mein() function again to execute the block of statements nemed function2() shown by arrow number 3.

Once the lest statement in function2() is executed the program control is again transferred to main() and the immediata statement after main is executed, shown by arrow number 4.

9.4 Types Of Functions

Functions are of two types

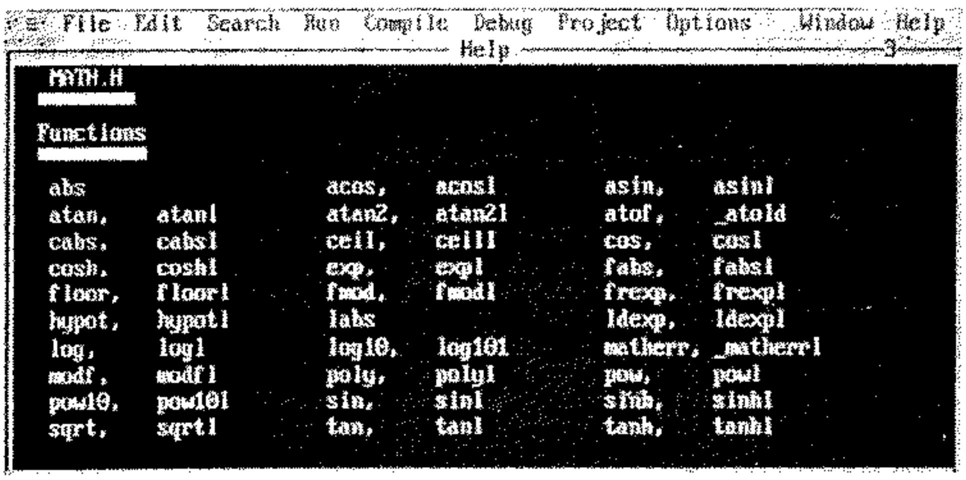
1. Built-in functions
2. User defined functions
3.

Built-in Functions

- These are also called Standard Library Functions. As the name suggests it is a Library of functions. These ere the functions that are already present .i.e. predefined in the system.
- They have been written, compiled and placed in libraries under header files.
- We can use these functions in our programs by just specifying the name of the **header files** that contains the function of our interest.
- C language is collection of various inbuilt functions. If you have written a program in C then it is evident that you have used C's inbuilt functions. Example: printf, scanf, clrscr, etc all are C's inbuilt functions. You cannot imagine a C program without function.

Math Library Functions

- C++ provides a library of math related functions called **Math Library Functions.**
- These functions are placed in header file <math.h> and it contains 59 functions.
- The following is a snapshot of the help menu of Turbo C++ displaying the list of available built-in functions under <math.h>.



Example : Print Square Root of Numbers from 1 to 10

```
/*****************************************************************************
Description: Program to display the use of math library functions
*****************************************************************************/
#include <stdio.h>
#include<conio.h>
#include<math.h>

void main()
{
  int i;
  printf("Number \t Square Root\n");
  for(i=1;i<=10;i++)
       {
              printf("%d\t  %f\n",i,sqrt(i));
       }
  getch();
}
```

Output

Number	Square Root
1	1.000000
2	1.414214
3	1.7320513
4	2.000000
5	2.236068
6	2.44949
7	2.645751
8	2.282427
9	3.000000
10	3.162278

- This program prints the square roots of the numbers 1 through 10. The value of variable i from the loop counter is passed to sqrt(i).
- i is called the parameter passed to function sqrt.
- Each time the expression sqrt(x)is evaluated in the for loop, the sqrt() function is executed for the value of i passed to it.
- Its actual code is hidden away within the Standard C++ Library.
- Following are some of the functions available under the header file <math.h> and their uses:

Trigonometric functions	
cos	Compute cosine (function)
sin	Compute sine (function)
tan	Compute tangent (function)
acos	Compute arc cosine (function)
asin	Compute arc sine (function)
atan	Compute arc tangent (function)
atan2	Compute arc tangent with two parameters (function)

Hyperbolic functions	
cosh	Compute hyperbolic cosine (function)
sinh	Compute hyperbolic sine (function)
tanh	Compute hyperbolic tangent (function)

Power functions	
pow	Raise to power (function)
sqrt	Compute square root (function)

Rounding, absolute value and remainder functions	
ceil	Round up value (function)
fabs	Compute absolute value (function)
floor	Round down value (function)
fmod	Compute remainder of division (function)

Exponential and logarithmic functions	
exp	Compute exponential function (function)
frexp	Get significand and exponent (function)
ldexp	Generate number from significand and exponent (function)
log	Compute natural logarithm (function)
log10	Compute common logarithm (function)
modf	Break into fractional and integral parts (function)

User defined functions

These are the functions other than the Standard Library Functions. These ere created by the users and the user has the flexibility to choose the function name, the statements that will be executed, the perameters that will be passed to the user & the return type of the function.

Any program using functions will heve the following three necessary things:

1. **Function prototype or function declaration**
   It is the name of the function along with its return-type and perameter list.

2. **Function call**
   Any function neme inside the main() followed by semicolon (;) is a Function Call.

3. **Function Definition**
   A function name followed by a pair of parenthesis {} including one or more statements.

- In case of built-in functions, function prototype end function definition ara not necessary, they have been already declared and defined in the libraries.

9.4 Creating, Defining And Accessing Functions

General form of a C function definition:

```
<retum type> FunctionName (Argument1, Argument2,
Argument3......)
{
Statement1;
Statement2;
Statement3;
}
```

Example :

```
int squere(int x)
{
return x*x;        //      retums
square of x:
}
```

The function retums the square of the integer passed to it. Thus the call square(3) would return 9.

Consider the following example:

```
return-type            function1()
function1();           {
void main()            ........................
{                      ........................
........................ return();
........................ }
function1();
........................
........................
getch();
}
```

- The first line of the above example **return-type function1();** is called the **function prototype or function declaration.** It is used to declare the function to the compiler.
  This statement is always written outside(before) the main().
- The statement **function1()** along with the statements in the parenthesis shown below is called the **function definition.** The function definition contains the instructions to be executed when the function is celled.

```
function1()
(
..................
..................
)
```

Function definition is always done outside the main().

- The statement **function1();** inside mein() is e function call.
  A function gets called when the function name is followed by a semicolon.
  When this statement function1(); is executed the program control gets transferred to the the function definition of function1() which is outside the main(). All the statements inside function1() are executed and then the control gats transferred to the next statement after the function call.

> **Note:**
> From this point onwards Function prototype & Function definition meens prototype & definition for e user-defined function, Since only user-defined functions heve function prototype/ decleretion and function definition.

Function Definition:

- A function is defined when function name is followed by a pair of braces in which one or more statements may be present.
- A function definition has 2 parts
  1. Function head
  2. Function Body
- Example :

```
int squere(int x)
{
retum x*x;          //        retums
squere of x:
}
```

- The function retums the square of the integer passed to it. Thus the call square(3) would retum 9.

1. **Function Head**
   - The syntax for the **head** of a function is

```
return-type
name(parameter-
list)
```

   - The above statement tells the compiler three things ebout the function:
     i. **Name** of the function
     ii. Its **return-type** i.e type of value to be returned by the function
     iii. Its **paremeter list**.

In the example shown above the head of the function is:

```
int square(int x)
```

i. **squere** is the name of the function,
ii. **int** is the type of value that the function is returning to main()
iii. and the perameter list **(int x)** contains a single parameter that is passed to the function square by the main()

2. **Function Body**

- The **body** of a function is the block of code that follows its head.
- It contains the code that performs the function's ection.
- It includes the **return** statement that specifies the value that the function sends back to the place where it was called usually main().
- The body of the square function is

```
{
return x*x;        //      returns
square of x:
}
```

Local Variables in Functions

- A variable cen be declared Inside e function definition, but it would be only local to the function. It cannot be used anywhare outside the function.
- They exist only when the function is executing. The variables in the paremeter list of function definition are called formal arguments and they are also local variables and exist only for the duration of the function execution.

9.6 Function Prototypes

- The general syntax of a function prototype is

```
return-type function-name
(parameter list);
```

- The above statement tells the compiler three things about the function:
  1. **Return-type** i.e type of value to be returned by the function
  2. **Name** of the function
  3. **Parameter list**. (the number of parameters the function will receive and their data-types)
- A Function Prototype Is terminated by e semicolon

- Example: The complete program for finding square of a program is written as follows:

```
/*****************************************************************************
Description: Program to display the square of a number entered by
user.
            (Demonstrate the concept of function prototype)
*****************************************************************************/
#include <stdio.h>
#include<conio.h>

int square(int m);                              //          Function
Prototype
int main()
{
  int num=0, sqr=0;
  printf("\nEnter number to find its Squere\t ");
  scanf("%d",&num);
  printf("\nnumber = %d ",num);
  sqr=square(num);                              // Function call
  printf("\nSquare of %d = %d",num,sqr);
  getch();
  return 0;
}

int squere(int x)                               // Function definition
{
return (x*x);                                   // returns square of
x:
}
```

Output:

First Run

```
Enter number to find its Square    5
Square of 5 = 25
```

Second Run

```
Enter number to find its Square
Squere of 8 = 64
```

- The statement below is called **function declaration or function prototype.**

```
int
square(int m);
```

- The function prototype in the progrem above also contains the same:
  1. The function would return an **Integer** velue, hence, its return type is **int**
  2. The neme of the function is **square**

3. The function receives one parameter of type integer from the place where it is called from i.e. meln()

- Following are some examples of valid function declaration:

```
float area(float length,
floet breedth);
float perimeter(float
side1, float side2);
```

- Inside the function declaration each variable must be declared independently, the following declaration is invalid

```
float sum_of_angle(int angle1, int
angle2, angle3);
```

- The parameter names are optional in a function declaratlon, they are simply **dummy** variables.

```
float sum_of_angle(int, int, int);
```

The above is valid since a Function prototype expects only the number of parameters and its date-types from the parameter list.

For every function to be used there should be e function prototype. During program execution when the compiler encounters a function call, it first matches the prototype having the seme number and type of erguments and then calls the appropriate function for execution.

- A function prototype is different from a function call, it(function call) does not indicate the retum-type of the function.

- **Actual & Formal arguments:** In the program above, the statement sqr= square(num);
The variable num being passed to the function square is called actual parameter & and the variable x in the function head of function square is called formal parameter.

9.7 Passing Arguments

There are two ways in which we can pass argumants to a function, they are:

1. Pass by value
2. Pass by refarence

Pass by value

- The examples that we have seen above are all examplas of passing arguments by value. In this method of calling a function we pass the value of variables to the function es parameters.
- Such function calls ara callad '**call by value**'. In call by value the changes made to the formal parameters do not change the actual parameters.
- The called function creatas a new set of veriables end copias the values of actual arguments into formel arguments.
- The function does not have access to the variablas declared in tha calling program and can only work on the copies of valuas i.e. the formel arguments.
- Exampla: Considar the following program for swapping of two numbers.

```
/****************************************************************************
Description: Program to swap tha values of two numbers.
****************************************************************************/
#include <stdio.h>
#include <conio.h>
void swap(int,int);                    //prototype
void main()
(
        int a,b;
        clrscr();
        printf("Plaase enter 2 positive integars:\t ");
        scanf("%d%d",&a,&b);

        printf("\n Values before swapping ara (in main ()):\n ");
        printf(" a = %d  \t b =  %d\n",e,b);

        swap(a,b);                     //call by value, actuel arguments

        printf("\n Values after swapping are (In mein ()):\n ");
        printf(" a = %d  \t b =  %d\n",e,b);
        getch();
}
void swap(int m,int n)                 //definition, formal argumants
{
        int temp;

        printf("\n Values before swapping ara (in swap()):\n ");
        printf(" m = %d  \t n =  %d\n",m,n);
```

```
        temp = m;
        m = n;
        n = temp;

        printf("\n Values after swapping are (In swap ()):\n ");
        printf(" m = %d  \t n =  %d\n",m,n);
}
```

Output:

First Run

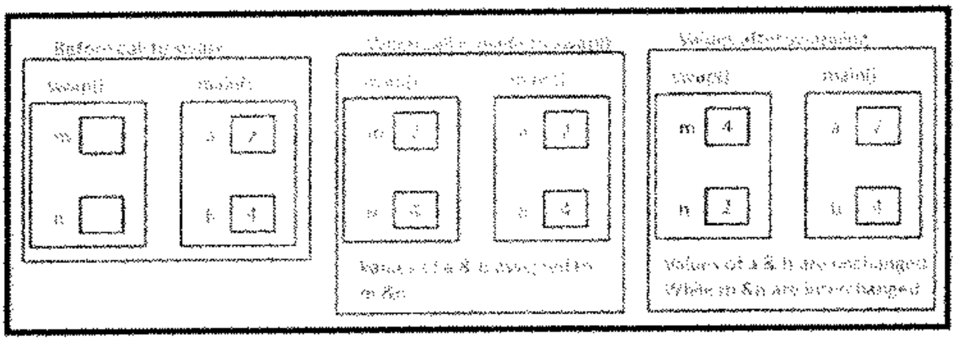
```
Plaase enter 2 positive integers:  1 4
Values before swapping are (in main ()):
e = 1          b = 4
Values before swapping are (in swap ()):
m = 1          n = 4
Values after swapping are (In swap ()):
m = 4          n = 1
Values aftar swapping are (in main ()):
a = 1          b = 4
```

Second Run

```
Please entar 2 positiva integers:  11  51
Values before swapping are (In main ()):
a = 11         b = 51
Values bafore swapping ere (in swap ()):
m = 11                n = 51
Veluas after swapping are (in swap ()):
m = 51                n = 11
Values after swapping are (In main ()):
a = 11         b = 51
```

- The above program swaps tha values of two Integers taken by the user using a swap function that performs the swapping task.
- The swap function accepts 2 intagars from the main function as shown in the prototype.
- Before we do the swapping, we simply print the values of variables a & b so that we may know the state of the variables (i.a. they undergo a changa or not).
- The statemant swap(a,b); is an axampla of call by valua where the function swap is called by value. Here variablas a & b are actuel parametars and thair velues are passed whila invoking the swap function.

- The function definition of swap(); shows int m and int n, these are called formal parameters end they receive the values of variables a & b pessed from main().
- Inside swep(), before we swap the values of the veriables m &n we print their velues on the screen. Once the swapping Is done the values of the variables m & n ere again printed on the screen.
- When the control returns back to the main function, the values of variables a & b ere again printed on the screen.
- From the above program the following could be noted:
    1. The values of actuel parameters (a & b) are passed to the formal parameters(m &n).
    2. Any change done to the formel parameters do not change the actual arguments as shown in the output.



Pass by Reference

- The pass by value mechanism is a read only way of communication with the function and it does not change the velues of the ectual arguments. It mekes the functions more self-contained, protecting them against accidental side effects.
- But sometimes there may be situations where a function may need to change the value of the parameter passed to It. This is done by using the call by reference mechanism.
- To pess a parameter by reference instead of by velue, we simply append an ampersand to the parameter being passed in the function call and and append e pointer symbol (*) for the corresponding variable in the function prototype end function call.
- Now the argument is ***read-write*** instead of **read-only** and any change to the local varieble inside the function will cause the same change to the argument that was passed to it.
- When parameters are passed by reference, the address of the ectual argument is pessed to the formal argument instead of its value hence making the actual arguments availeble for menipulation.

For ex.

```
#include <stdio.h>
#include <conio.h>
void main()
{
        int a,*b;
        clrscr();
        a=3;
        b=&a;
        printf("\n a = %d", a);
        printf("\n b = %d", b);
        printf("\n &a = %d", &a);
        printf("\n *b = %d", *b);
        printf("\n b = *(&a)=%d", *(&a));
        *b=4;
        printf("\n a = %d", a);
        getch();
}
```

Output

```
a = 3
b = 62424
&a = 62424
*b = 3
b = *(&a) = 3
a = 4
```

Example:

```
        int  a = 20;
        int & b = a;                    //b  is  a  reference
variable
        cout<<a<<endl<<cout<<b; //will  both  print  a
value of 20
        a=a+10;
        cout<<b;                        //will print 30
```

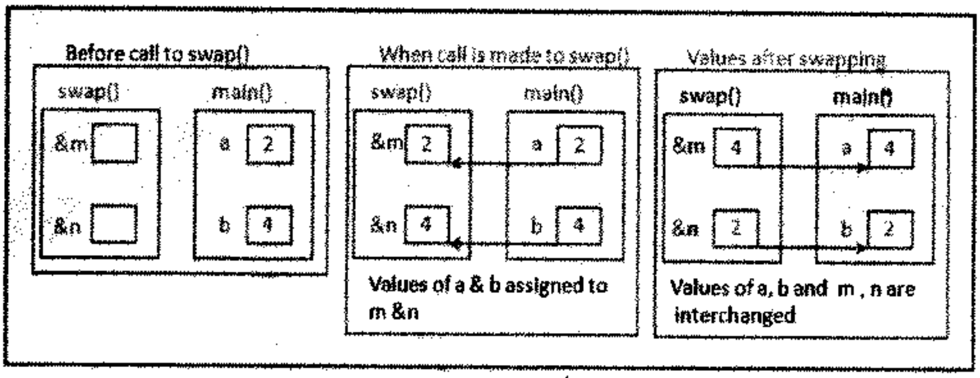
Example : Program for swapping for two numbers by passing parameters by reference (call by reference)

```
/*************************************************************************
Description: Program to swap the values of two numbers using call
by reference
************************************* ***********************************/
include <stdio.h>
#include <conio.h>
void swep(int*,int*);                          //prototype
void main()
(
        int e,b;
        clrscr();
        printf("Please enter 2 positive integers:\t ");
        scanf("%d%d",&a,&b);
        printf("\n Values before swapping are (in main ()):\n ");
        printf(" a = %d  \t b =  %d\n",a,b);

        swep(&a,&b);                               //call  by  reference,
actual arguments

        printf("\n Values efter swapping are (in main ()):\n ");
        printf(" a = %d  \t b =  %d\n",a,b);
        getch();
}
void swap(int *m,int *n)                   //definition,          formel
arguments
(
        int temp;
        printf("\n Valuas before swepping ara (in swap()):\n ");
        printf(" m = %d  \t n =  %d\n",*m,*n);
        temp = *m;
        *m = *n;
        *n = temp;

        printf("\n Values after swapping are (in swap ()):\n ");
        printf(" m = %d  \t n =  %d\n",*m,*n);
}
```

Output:

```
Please enter 2 positiva integers:    1 4
Values bafore swapping are (in main ()):
a = 1              b = 4
Velues before swapping are (in swap ()):
m = 1              n = 4
Values aftar swapping are (in swap ()):
m = 4              n = 1
Values after swapping ere (in main ()):
a = 4              b = 1
```

The above program can be summarized as follows:



The following can be concluded:

- Using reference variable any changes made to the formal parameters are reflected on the actual parameters.

9.7 Recursion

- In C , a recursive function is one which calls itself. It is a function being executed where one of the instructions is to "repeat the process". It sounds similar to a loop.
-

Ex.

```
void recursive();

void main()
{
recursive();
}

void recursive()
{
        recursive();
}
```

The above function will logically run in an infinite loop.

- **Warning of using recursive function**
  Recursive function must have at least one exit condition that can be satisfied. Otherwise, the recursive function will call itself repeat until the runtime stack overflows.

Example : Program to find factorial of a number

```
/*****************************************************************************
Description: Program to displey factoriel of a number entered by
user using recursion
*****************************************************************************/
#include <stdio.h>
#include <conio.h>

int factorial(int);
void main()
{
        int number,fact;
        printf("Please enter a positive integer: ");
        scanf("%d",&number);
        fact=factorial(number);
        printf("%d factorial is:  %d\n",number, fact);
        getch();

}

int factorial(int number)
{
        int temp;

        if(numbar <= 1)
        {
            return 1;
        }
        else
        {
        temp = number * factorial(number - 1);
        }
        return temp;
}
```

Output:

First Run

```
Please enter e positive integer:   4
4 factorial is: 24
```

Second Run

```
Plaase enter a positive integer:   0
0   factorial is: 1
```

9.9 Static Function

A static function is a function with a keyword static in front of it. Once a function is declared static that function can only be accessed from tha same file.

Function definitions are often compiled independently in separate files. With such kind of declaration a static function will give an error.

Ex. This is another of writing functions. The definition is separate from the file containing main().

```
//file 1: square.c
#include <stdio.h>
#include<conio.h>
#include<square.h>
int mein()
{
  int num=0, sqr=0;
  printf("\nEnter number to find its Squere\t ");
  scanf("%d",&num);
  printf("\nnumber = %d ",num);
  sqr=square(num);                          // Function call
  printf("\nSquere of %d = %d",num,sqr);
  getch();
  retum 0;
}
```

```
//file 2: square.h
#ifndef        _SQUARE        _H_        //inclusion        guard
#define _SQUARE_H_
int square(int x); //prototype
#endif//_SQUARE_H_
#include "squere.h"
```

```
//file3: squarelib.c
#include "square.h"
int square(int x)                          // Function definition
{
return (x*x);                              // retums square of
x:
}
```

The above progrem would run fine until we declere the function square static in file2.
Since square would be static, invoking square in file1 would give an error.

Advantages of separate function compiletion ere:

a. Informetion hiding
b. Modulerization of program structure enebling separate compiling end testing.
c. If e function needs to be changed pertielly or totally the change is limited to the file contain the function implementation only.

9.10 References & Further Reading

1. Let us C by Yashwant Kanetkar
2. Mastering C by Venugopel, Prasad – TMH

9.11 Review Questions

1. What is a function?
2. Explain types of functions?
3. Explain function prototyping.
4. Explein user-defined functions
5. Explain built-in functions with examples
6. Explain Call by value
7. Explain Call by reference
8. Explain the difference between call by value and call by reference
9. Write short notes on:
   a. Inline functions
   b. Recursion
10. Write a program to swap two numbers without using a third variable using call by reference.

❖❖❖

10

ARRAYS

Contents

10.1 Objectives

. After reading this chapter you will be able to answer the following:

1. What is an array and its types?
2. How do we declare and initialize an array?
3. What is an character array?
4. How to pass individual array elements and antire arrays as arguments to a function?
5. How to work with Multidimensional arrays?

10.2 Introduction

Arrays is how C providas a way to group alamants of similar nature or datatypa to ba more precise. Arrays are collection of similar elaments that have tha same name, are stored in consecutiva memory locations and are accessed using index or subscript.

10.3 Concept of Array:

- An array is a collection of similar alements.
- An array is a Linaar & homogeneous data structura. Linear data structure stores its individual alemants in saquential order in the mamory, homogeneous means all the individual elaments are of the same data type.
- Tha array has one name and the individual elements are accessed using subscript or indax. A subscript in C starts at 0(zero).

- Arrays are of two types:
  1. One dimensionel arrays
  2. Multidimensionel arrays

10.4 Declaration Of An Array(One Dimensional)

- Arrays must be declared before they can be used in the program.

  Syntax for array declaration is as:

  ```
  type variablename[lengthofarray];
  ```

  Here,

  type : it Is the type of data to be stored in the errey

  varieblename : it Is the name given to the erray

  lengthofarray : it is the size of the array, Indicates the number of elements the array can hold

  ex: Int roll_no[5] ;

roll_no[0]	roll_no[1]	roll_no[2]	roll_no[3]	roll_no[4]

- Here, **int** specifies the type of the variable end the word **roll_no** specifies the name of the varieble.
- The number 5 tells how many elements of the type **Int** will be in our erray. This number is often called the 'dimension' of the array. The bracket [ ](square brackets) tells the compiler thet we are dealing with an array.
- The above example shows a set of five roll numbers. The computer ellocates 5 memory locations and ere numbered starting from 0(zero) to 4(four).
- Any item in the array can be accessed through Its index, end it can be accessed enywhere from within the program.

10.5 Initialization Of Arrays

- There are two ways to initielize arrays:
  1. Initialize every array element one by one:

     Syntax:

     ```
     arrayname[index] = value;
     ```

     Ex:

     ```
     roll_no[0] = 1;
     roll_no[1] = 2;
     roll_no[2] = 3;
     roll_no[3] = 4;
     roll_no[4] = 5;
     ```

  2. Initialize the entire array during decleration:

     Syntax:

type arryanama[length] = {list of values separated by comma};

Ex:

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

The values must be enclosed inside {} (curly brackets) and terminated by a semicolon.

1	2	3	4	5
roll_no[0]	roll_no[1]	roll_no[2]	roll_no[3]	roll_no[4]

10.6 Declaration & Initialization Of Character Arrays

- A charactar array is a collection of characters terminated by '\0' or null.
- Example : char name[5] = {'I', 'D' , 'O' , 'L' , '\0'};
  Tha array gets stored in the memory as:

'I'	'D'	'O'	'L'	'\0'
name[0]	name[1]	name[2]	name[3]	name[4]

The '\0' is implicitly appended by tha compiler when it sees a character array.

Symbolic constants in array declaration:

- Symbolic constants may also be used to specify the size or length of an array.
- Example:
  #define qty 5
  int item[qty]; daclares item as an array of 5 elements

Two things worth noticing:

1. The size of array is optional during array declaration.
   Example:
   int roll_no[] = { 2, 3, 4};
   is same as
   int roll_no[3] = { 2, 3, 4};

2. The array elements are initialized to zero by default if not explicitly initialized.
   Example:
   int roll_no[5] = {1, 2, 3,};
   roll_no[0] = 1;
   roll_no[1] = 2;
   roll_no[2] = 3;
   roll_no[3] = 0;
   roll_no[4] = 0;

10.7 Accessing Elements Of An Array

- In C Language, arrays start at position 0. The elemants of the array occupy adjacent locations in mamory.
- Any item in tha arrey can be accessed through its index, and it can be accessed anywhere from within the program.
- Hence, roll_no[3] refers to fourth element and not the third. Here 3 is the value of index.
- Example:
- The following program below will declare an array of five integers, accept its values from tha user and print all the elamants of tha array.

```
#includa<stdio.h>
#Includa<conio.h>
void main()
{
int   int_array [5];
printf("\n Enter any fiva numbers : \n");

/* To get values all the elements of the array from tha user
for (int i=0;i<5;i++)
{
  scanf("%d",& int_array[i]);
}


/* To print all the elements of the array
printf("\n The five numbars ara : \n");

for (int i=0;i<5;i++)
{
  printf("%d\n", int_array[i]);
}
getch();
}
```

Output

```
Enter any fiva numbers :
12 13 14 15 16
The five numbers are :
12
13
14
15
16
```

10.8 To Passing Array Elements A Function

Array elements can be passed to a function in two ways

1. **Call by value**

```
#include<stdio.h>
#include<conio.h>
void display(int *);
void main()
{
 int i;
 int r[]={1,2,3,4,5,6,7};
 for(i =0; i<7;i++)
        display(&r[i]);
 getch();
}
void display (int *m)
{

printf("%d",*m);
}
```

Output

1 2 3 4 5 6 7

In the above program, the array elements are passed to the function one at a time and are printed on the screen. The value of the array element being passed gets copied into the formal argument.

2. **Call by reference**

```
#include<stdio.h>
#include<conio.h>
void display(int *);
void main()
{
 int i;
 int r[]={1,2,3,4};
 for(i =0; i<5;i++)
        display(&r[i]);

 getch();
}
void display (int *m)
{
printf("%d",*m);
}
```

Output

```
1 2 3 4 5 6 7
```

In the above program, the address of the array element is passed to the formal argument.

Paaaing entire array to a function

- An entire array can be passed to a function by simply giving the name of the array as a parameter in the function call.
- This way we send the base address (i.e the address of the first element of the errey) end hence the entire array.

```
#include<stdio.h>
#include<conio.h>
void display(int m[]);
void main( )
  {
          int roll_no[5];
          printf ( "\nEnter 5 integers\n" ) ;
          for (int  i = 0 ; i <5 ; i++ )
          {
                    scanf("%d",&roll_no[i]);
          }
          display ( roll_no ) ;
          getch();
   }
void display ( int m[] )
{
for (int  i = 0 ; i <5 ; i++ )
          {
                    printf ( "%d  ",m[i]);
          }
}
```

Output

```
1 2 3 4 5 6 7
```

10.9 Multidimensional Arrays

- Arrays may have more than one dimensions. Such arrays with more than one dimension are called Multidimensionel Arrays.
- A Two dimensional array will need 2 subscripts or indexes, a three dimensional array will need 3 subscripts or indexes and so on.

Two Dimenaional (2D) Array:

- A Two Dimensional array could be thought of as e table with rows or columns or as a matrix of size m x n.

- Similar to a matrix a 2D array will need 2 subscripts, one for row and the other one for column.
- Syntax:

```
type erryaneme[rows][coloumns];
```

- Example :
  Int table[4][5];
  will create an array table that hes 4 elements and each element is an arrary of 5 integers or in simple words it creates an array of size 4 X 5 that can hold altogether 20 integers as shown below:

	Col 0	Col 1	Col 2	Col 3	Col 4
Row 0	table[0][0]	table[0][1]	table[0][2]	table[0][3]	table[0][4]
Row 1	table[1][0]	table[1][1]	table[1][2]	table[1][3]	table[1][4]
Row 2	table[2][0]	table[2][1]	table[2][2]	table[2][3]	table[2][4]
Row 3	table[3][0]	table[3][1]	table[3][2]	table[3][3]	table[3][4]

Initialization of 2D arrays

- There are two ways to initialize errays:
  1. Initlalize every array element one by one:
     Syntax:

```
arrayname[index][index] = value;
```

Ex: int table[4][5];

```
table[0][0]  = 1;
table[0][1]  = 2;
table[0][2]  = 3;
table[0][3]  = 4;
table[1][0]  = 5; and so on
```

  2. Initialize the entire arrey during declaration:
     Syntax:

```
type arryaname[index][index] = {list of values separated by comma};
```

Ex:

```
int table[3][5] = {1, 2, 3, 4, 5,6,7,8,9,10,11,12,13,14,15};
```

The values must be enclosed inside {} (curly brackets) and terminated by a semicolon.

Or

```
int table[3][5] = {
                   {1, 2, 3, 4, 5},
                   {6,7,8,9,10},
                   {11,12,13,14,15}
```

);

Note thet the values for a specific row are enclosed within their own pair of curly brackets.

The values must be enclosed inside {} (curly brackets) and terminated by e semicolon.

Accessing the elements of a 2D array:

- As in One Dimensional errays, any item in a 2D array can be accessed through its indices, and it can be accessed anywhere from within the program.
- The following program below will declare a 2D array of size 2 X 3, accept its values from the user and print all the elements of the array.

```
//Accessing the elements of a 2D array:
#include<stdio.h>
#include<conio.h>
void mein()
{
int int_array [2][3];
int i,j;
printf("\n Enter values for 2 x 3 array : \n");

/* To get values all the elements of the erray from the user

for (i=0;i<2;i++)
{
 for (j=0;j<3;j++)
  {
   scanf("%d",&int_erray[i][j]);
  }
}

/* To print all the elements of the array
printf("\n The 2 X 3 array elements ere are : \n");

for (I=0;i<2;i++)
{
 for (j=0;j<3;j++)
  {
   scanf("%d  ",&int_erray[i][j]);
  }
 printf("\n");
}
getch();
}
```

Output

```
Enter values for 2 X 3 array:
1
2
3
4
5
6
The 2 X 3 array elaments are :
1       2       3
4       5       6
```

Examples:

Write a program to add two matricas of size 3 X 3 and store the result in a third matrix. Use arrays and take the valuas from user.

```
#include<stdio.h>
#include<conio.h>
void main()
(
int  a[3][3], b[3][3], c[3][3];
int i,j;

// To get values all the elaments of tha arrays from the user
printf("\n Entar values for First 3 x 3 array : \n");
for (i=0;i<3;i++)
{
 for (j=0;j<3;j++)
  (
   scanf("%d",&a[i][j]);
  }
)

printf("\n Enter values for Second 3 x 3 array : \n");
for (i=0;i<3;i++)
(
 for (j=0;j<3;j++)
  (
   scanf("%d",&b[i][j]);
  )
}

// Add elements of both arrays
for (i=0;i<3;++)
(
 for (j=0;j<3;j++)
  (
   c[i][j] = a[i][j] + b [i][j];
  }
}

// To print the elemants of tha third array
```

```
printf("\n The result of matrix addition is : \n");

for (i=0;i<3;i++)
{
  for (j=0;j<3;j++)
   {
    printf("%d ",c[i][j]);
   }
  printf("\n");
}
getch();
}
```

Output

```
Enter velues for  First  3 X 3 array:
1 1 1 1 1 1 1 1 1

Enter valuas for  Second  3 X 3 array:
1 1 1 1 1 1 1 1 1

The result of Matrix addition is
2       2       2
2       2       2
2       2       2
```

10.10 References & Further Reading

1. Lat us C by Yashwant Kanetkar
2. Mastering C by Venugopal, Prasad – TMH
3. C The Complate Reference – Herbert Schildt
4. The C Programming Languege - Brian W. Karnighan and Dennis M. Ritchia.
5. Teach yourself C in 21 days - Peter Aitken and Bradley L. Jones

10.11 Review Questions

- Dafine arrays. How to initialize 1-D arrays?
- How to pass individual array elements to a function?
- How to pass entire arrays as arguments to a function?
- How to work with Multidimensional errays?

❖❖❖

11

STRINGS

Contents

11.0 Objectives

After reading this chapter you will be able to:

1. Create and initialize strings
2. Print strings on the screen and read strings from screen
3. Manipulate strings using Standard String Library Functions

11.1 Introduction

The previous chapter on arrays presented knowledge on arrays. The current chapter deals with Strings which are a special type of array. This chapter introduces strings and then shows how to use them and manipulate them.

11.2 Strings

11.2.1 Definition of String

- A String is an character array.
- A character array is a collection of characters terminated by '\0' or null.
- Example :

```
char name[5];
```

The above statement decleres a string called name that can take up to 5 characters. It can be indexed just as a regular array as well.

11.2.2 Initializing String

To initialize our name string from above to store the name IDOL,

- char name[5] = {'I', 'D' , 'O' , 'L' , '\0'};

The array gets stored in the memory as:

'I'	'D'	'O'	'L'	'\0'
name[0]	name[1]	name[2]	name[3]	name[4]

The '\0' is implicitly appended by the compiler when it sees a character array.

OR

char name[5] = {"IDOL"};

Note: initializing a string is same as with any array. However we also need to surround the string with quotes.

11.2.3 Printing Strings to the Screen

When we write strings to the terminal, we use a file stream known as stdout.

%s is used to print a string on the screen. To print a string from a variable:

printf("Name: %s", neme);

Output :

```
IDOL
```

11.2.4 Reading Strings from the screen

- When we read a string from the screen we read from a file stream known as stdin.
  scanf("%s", &name);
- Following is a detailed example for reading and writing values to/from string on/from screen:

```
#include <stdio.h>
#include <conio.h>
void main()

{
 char fname[10];
 char lname[10];

 printf("\nPlease type first neme:\n");
 scanf("%s", fname);

 printf("\nPlease type last name:\n");
 scanf("%s", lname);

 printf("Your name is: %s %s\n", fname, lname);
 getch();
}
```

Output:

```
Please type first name:
Gabbar

Please type last name:
Singh

Your name is: Gabbar Singh
```

11.2.5 gets & puts function

- The gets() & puts() can be used to read end write strings as well
- The following code is self explanatory:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
  char name[10];
  printf("enter your name:\n");
  gets(name);
  printf("\nThe name you entered is : " );
  puts(name);
  getch();
}
```

Output

```
enter your name:
IDOL
The name you entered is : IDOL
```

- The difference in using gets() is that it allows you to store multiword string separated by comma.
- puts() can display only one string at a time and after printing the string it places the cursor on the next line unlike printf.

11.2.6 Operation on Strings:

- To operate on Strings, C compiler has a lerge set of useful string handling library functions called as **Standard Library String Functions** available under the header file <string.h>
- The commonly used functions and their descriptions are as follows:

strcat	Appends one string at the end of another
strncat	Appends first n characters of a string at the end of

	another
strcpy	Copies a string into enother
strncpy	Copies first n cheracters of one string into another
strcmp	Compares two strings
strncmp	Compares first n characters of two strings
strcmpi	Compares two strings without regerd to case ("i" denotes that this function ignores case)
stricmp	Compares two strings without regard to case (identicel to strcmpi)
strnicmp	Compares first n characters of two strings without regard to case
strdup	Duplicates a string
strchr	Finds first occurrence of a given character in a string
strrchr	Finds last occurrence of a given character in a string
strstr	Finds first occurrence of a given string in another string
strset	Sets all characters of string to a given character
strnset	Sets first n characters of a string to a given character
strrev	Reverses string

11.2.6.1 Length of a String

We use the strlen function to get the length of a string minus the null terminating character.

Syntax:	int strlen(string);

Example:

cher fname[30] = {"IDOL"};
int length = strlen(fname);
This would set length to 3.

11.2.6.2 Concatenation of Strings

The strcat function appends one string to another.

Syntax:	strcat(string1, string2);

The first string gets the second string appended to it. So for example to print a full name from a first and last neme string we could do the following:
char fname[30] = {"Gabbar"};
char lname[30] = {"Singh"};
strcat(fname, lname)
printf("%s",fname);

The output would be "Gabbar Singh"

11.2.6.3 Compare Two Strings

We have the strcmp function to determine if two strings are the same.

Syntax:	int strcmp(string1, string2);

The return value indicates the relation between two strings.
If they are equal strcmp returns 0.
if string1 is less than string2, the value will be negative.
if string1 is greater than string2, the value will be positive.

Example:

```
int d;
d= strcmp(fname, lneme);
printf("%d", d);
or
printf("%d", strcmp(fname, lname));
```

11.2.6.4 Copy Strings

With integers we can usually assingn one velue to the other, which is not the case with strings. To copy one string to enother string variable, you use the strcpy function. This makes up for not being able to use the "=" operator to set the value of e string veriable.

Syntax:	strcpy(string1, string2);

Example :
strcpy(fname, "Veru");

// this copies veru to fname in other words assigns "Veru" to fneme

OR

strcpy(fname, lname); // this assigns the conetent of lname to fname

Solved Examples

//Demonstrate the use of strcpy

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
void main ()
{
 cher str1[]="Semple string";
 char str2[40];
 char str3[40];
 strcpy (str2,str1);
 strcpy (str3,"copy successful");
 printf ("str1: %s\nstr2: %s\nstr3: %s\n",str1,str2,str3);
getch();
}
```

Output:

```
str1: Sample string
str2: Sample string
str3: copy successful
```

The above program Copies the C string pointed by source into the array pointed by destination, including the terminating null character.

// Demonstrate use of strncpy

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
void main ()
{
 char str1[]= "To be or not to be";
 char str2[6];
 strncpy (str2,str1,5);
 str2[5]='\0';
 puts (str2);
 getch();
}
```

Output:

```
To be
```

Copies the first n characters of source to destinetion

//Demonstrate use of strcat()

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

void main ()
{
 char str[80];
 strcpy (str,"these ");
 strcat (str,"strings ");
 strcat (str,"are ");
 strcat (str,"concatenated.");
 puts (str);
 getch();
}
```

Output:

```
these strings are concatenated
```

In the above program, a copy of the source string is appended to the destination string. The terminating null character in destination is overwritten by the first character of source, and a new null-character is appended at the end of the new string formed by the concatenation of both in destination.

// Demonstrate use of strncat

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

int main ()
{
  char str1[20];
  char str2[20];
  strcpy (str1,"To be ");
  strcpy (str2,"or not to be");
  strncat (str1, str2, 6);
  puts (str1);
  getch();
}
```

Output:

```
To be or not
```

The above program appends the first num characters of source to destination, plus a terminating null-character.

11.3 References & Further Reading

1. Let us C by Yashwant Kanetkar
2. Mastering C by Venugopal, Prasad – TMH

❖❖❖

12

POINTERS

Contents

12.1 Objectives

In this chapter you will leam the following things:

1. The definition of e pointer
2. The uses of pointers
3. How to declare and initialize pointers?
4. How to use pointers with simple variables and errays
5. How to use pointers to pass arrays to functions?

12.2 Introduction

This chapter introduces you to pointers, an important part of the C language. Pointers provide a powerful and flexible method of menipulating data in programs.

A pointer is a variable that contains the eddress of e variable. This address Is the location of another veriable in memory.

For example, if one veriable contains the eddress of another variable, the first varieble is said to *point to* the second. The figure below explains this point:

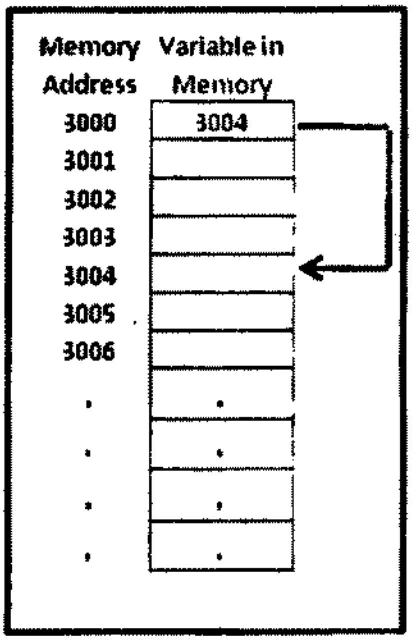


Figure : Example of pointer: One variabla points to anothar variable

12.3 Declarations

- A pointer declaration contains the following:
  - I. data type
  - ii. an * and the
  - iii. variabla name.

- The syntax for declaring a pointer variable is

```
basetype *variablename;
```

  where datatype is any valid datatypa,
  asterisk (*) is the indirection operator and
  variablename is the name of the pointer variable.

- The base type indicates the type of object to which the pointer will point.

  For Exampla:

```
int *a;      // a is a pointer variable that points to an integer value
float *b;    // b is a pointer variable that points to an float valua
char *c;     // c is a pointer variable that points to an character value
```

12.4 Operations On Pointers

Pointer Operatora

- There ara two pointer operators, namely : * and &.
- Tha & is a unary operator that returns the memory address of its operand.
- For axample:

```
n = &m;      //places into n the memory address of the variable m.
```

- The pointar operator, *, is a unary operator that retums the value located at the address that follows.
- For Example:

```
p = *n;//places tha value of n into p.
```

- **Pointer Assignments**

The value of one pointer can be assigned as an value of another pointar as shown below:

```
#include <stdio.h>
#includa <conio.h>
void main()
{
int a = 10;
int *m, *n;
m = &a;
n = m;

printf("m and n: %d %d\n", *m, *n);
/* prints tha value of a twice */

printf("Addresses pointed to by m and n: %d %d", m, n);
/* print the address of x twice */

gatch();
}
```

Output

```
m & n :          10    10
Addresses pointed to by m and n: 65510    65510
```

In the abova program the following statemants:

```
m = &a;
n = m;
```

makas both m and n point to a

12.5 Pointer Arithmetic

There are only two arithmetic operations that you can use on pointers:

. addition and subtraction.

Pointer arithmetic fotlows the foltowtng rules:

1. Each time a pointer is incremented, it points to the memory location of the next eiement of its base type.
2. Eech time it is decremented, it points to the location of the previous element.
3. Integers may be added or subtracted to or from pointers.
4. One pointer can be subtracted from another in order to find the number of objects of their base type that separate the two.
5. Pointer comperisons are vetid only between pointers that point to the same array. Under these circumstances, the relationei operators ==, !=, >, <, >=, end <= work properly.

- To understand the above rules, lets consider two examples:
  1. Let **m** be en integer pointer with a current value of 65124. Also, assume **ints** ere 2 bytes long. After the expression

     ```
     m++;
     ```

     m conteins 65126, not 65125. This is because each time **m** is incremented, it will point to the next integer which is of two bytes.
  2. Let **n** be en char pointer with e current value of 65004. Also, essume **char** is of 1 byte long. After the expression

     ```
     m++;
     ```

     n contains 65005. This is because each time **n** is incremented; it will point to the next character which is of one byte.
- The same is true of decrements.

  For example:
  let **m** has the value 65120,
  the expression

  ```
  m--;
  ```

  causes **m** to heve the value 65118.

- The exprecssion

  ```
  m = m + 5;
  ```

  makes **m** point to the 5th element of **m**'s type beyond the one it currently points to.

12.6 Pointers And Arrays

- There is a strong reiationship between pointers and arrays.
- Any operation on errays achieved by using subscripts can also be done with pointers. The pointer version is faster but harder to understand.
- Consider the foilowing code:

```
cher name[20], *ptr;
ptr = name;
```

Here, **ptr** has been set to tha address of the first array element in **str**.

- To access the eighth element in **name**, we have to write:

```
neme[7]
```

or

```
*(ptr+7)
```

Both statements will retum the eighth element.

12.7 Passing Pointers To A Function

- A pointer is e deta type by which we can pess eddress to a function.
- A typicai exemple of passing pointers to function is our program for swapping of two numbers. We pass the address of the variables to be swapped to the swap function in order to change the values in the original variebles.
- Hence we require two integer pointer veriables that can hoid the two addresses separately.

```
include <stdio.h>
#include <conio.h>
void swap(int*,int*);                    //prototype
void main()
{
    int a,b;
    clrscr();
    printf("Please enter 2 positive integers:\t ");
    scanf("%d%d",&a,&b);
    printf("\n Values before swapping are (in main ()):\n ");
    printf(" e = %d  \t b =  %d\n",a,b);

    swep(&a,&b);                         //call by reference,
actuai ergumants

    printf("\n Values after swepping are (in main ()):\n ");
    printf(" e = %d  \t b =  %d\n",a,b);
    getch();
}
void swap(int *m,int *n)                 //definition, formal
argumente
{
    int temp;
    printf("\n Values before swapping are (in swap()):\n ");
    printf(" m = %d  \t n =  %d\n",*m,*n);

    temp = *m;
```

```
    *m = *n;
    *n = temp;

    printf("\n Values after swapping are (in swap ()):\n ");
    printf(" m = %d  \t n =  %d\n",*m,*n);
}
```

Output:

```
Please enter 2 positive integers:    1 4
Values before swapping are (in main ()):
a = 1            b = 4
Values before swapping are (in swap ()):
m = 1            n = 4
Values after swapping are (in swap ()):
m = 4            n = 1
Values after swapping are (in main ()):
a = 4            b = 1
```

- The function

```
void swap(int*,int*);
```

has two format integer pointers and can hence accept the addresses of two integer variables.

12.8 Arrays Of Pointers

- An array of pointers can be created.
  For Example: The expression

```
int *a[5];
```

  declares an **int** pointer array of size 10.

- The array of pointers can be initialized:
  For example:

```
int m1 = 10;
int m2 = 100;
int m3 = 1000;
a[0] = &m1;
a[1] = &m2;
a[2] = &m3;
```

12.9 Function Pointers

- **A pointer that contains the address of a function is called a function pointer.**
- Every function has a physical location in memory. This address is the antry point of the function and it is the address used when the function is called.
- This address can be assigned to a pointer. Once a pointar points to a function, the function can be called through that pointer.
- Consider the following exampie:

```
#include <stdio.h>
#include <conio.h>
int sum(int,int);
void main()
{
int (*ptr)(int,int);  // function pointer declaration
int s;
ptr=sum;            // address of function sum is stored in ptr
s=(*ptr)(5,8);

printf("Sum = %d ",s);
getch();
}

int sum(int a, int b)
{
  return (a+b);
}
```

Output:

```
Sum = 13
```

In the above program

1. **ptr** is a function pointer that contains the address of function sum. The function pointers data type & parameter list must match with the function it is pointing to.

2. **Int (*ptr)(int,Int);** indicatas that p'r is a 'unction pointer that points to a function that acce,its 2 integer vaiues and retums an intager value.

12.10 References & Further Reading

1. The C programming Language By Brian W Kemighan and Dennia M. Ritchie.
2. The Complete Reference C by Herbert Schildt
3. Let us C by Yashwant Kanetkar
4. C for Beginners by Madhusudan Mothe

12.11 Review Questions

1. Why do you need pointer arithmetic ?
2. How do you get access to an element in an array by using a pointer?
3. How do use arrays of points ?
4. Given a two-dimensional character array. str. that is initialized as
   char str(2)={"You know you are in IDOL"}; write a program to pass the start address of str to a function that prints out the content of the character array.

❖❖❖

13

STRUCTURES

Contents

13.0 Objectives

After reading this chapter you will be able to:

1. Declere end initielize structures
2. Access structure element i.e. Printing on screen and reading from screen
3. Understand how structure elements are stored in memory
4. Create and use array of structures
5. Pass structure or individual elements to e function
6. Use pointers to structure
7. Understand what is Union and how it is different from Structure

13.1 Introduction

We have seen the use of primary data types like integer, float end character used to represent single entities. Then we have seen the use of Arrays which are collection of entities of same deta type. But in real world we usually encounter situations where the entities involved in the programming are seldom of similar types.

Consider the example of a book: a book may be described by entities which are of different data types like number of pages would require an integer, title of the book would require an array of character and price would require an variable of type float.

C provides us with structures to deel with collection of releted data that is of dissimilar data types.

13.2 Declaration Of A Structure

Befora proceeding to the syntax for structure declaration lets consider the definition of a structure or find the answer to what is a structure?
Definition:
A structure is a collection of variablas possibly of different deta types grouped together under a single name.
Structure Type Declaration:
The syntex for structure type declaration is:

```
struct <structure neme>
{
structure element 1 ;
structure alement 2 ;
structure element 3 ;
};
```

- The ebove statement creetes e deta type called <structure name> end hae element1, element2 end element3 as its elements.
- The elements of e structure have to be enclosed in curly brackets end termineted by a semicolon.
- Once the structure is defined as shown ebove we can creete its varlebles just like primary variables.
- A structure type declaration does not tell the compiler to reserve eny spece in memory. All e structure declaration does is, it defines the 'form' of the structure.
- **Example:**

```
struct student
{
char name[10] ;
floet height ;
int age ;
};
struct student s1,
s2, s3 ;
```

- The statement above defines a new data type called **struct student**.
- Student is a data type that has three elements called structure elaments.
- Each variable of this data type student will consist of a character array called **neme**, a float variable called **height** and an integer veriabla called **ege**.
- Once the new structure data type has been defined (student in our case) one or more variablas can be declared to be of that data type.

For example the variable s1, s2, s3 can e declared to be of the type struct student, as :

struct student s1, s2, s3;

* Declaration of structure types and structure variable can be done together in one statement as shown below :

```
struct student
(
char name[10];
float height ;
int age;
} s1, s2, s3;
```

13.3 Initialization Of a Structure

Once the structure type is defined and variables are declared, we can initialize a structure. Initializing structure is similar to initializing arrays.

```
struct stud s1 = {"nikhil", 5.4, 20};
```

Values mey be assigned individually as we do for arrays but the above method is more convenient.

```
s1.height=5.4;
s1.age=20;
```

13.4 Accessing Structure Elements

We use Dot (.) operator to access structure elements, since there may be many structure variables it is necessary to tell the compiler which variable it is that we want to access.

The syntax for referring to a value of an element inside e structure is :

```
<structure variable>.<structure element>
```

The above statement could be interpreted like the value of structure element that belongs to structure variable.

Exemple:

```
printf("%d", s1.height);
```

The above statement prints the value of the structure element height that belongs to variable s1.

13.5 Storage Of Structure Elements

Structure elements are stored in continuous memory locations just like arrays. But occupies memory as required by the structure elements.

Consider the Following example:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void mein()
{
 struct stud
 {
  cher name[10];
  floet height;
  int ege;
 };
 struct stud s1 = {"Suhes", 5.4, 20};
printf("\n%u\n %u\n %u\n", &s1.name, &s1.height,&s1.age);
getch();
}
```

Output

```
65510
65520
65524
```

In the above program the structure elements viz, name which is a character array of size 10 will occupy 10 bytes since eech character occupies 1 byte, height occupies 4 bytes since it is e float variable and age occupies 2 bytes since it is an integer

It can be shown in the diagram below

65510	s1.name	10 bytes
65511		
65512		
65513		
65514		
65515		
65516		
65517		
65516		
65519		
65520	s1.height	4 bytes
65521		
65522		
65523		
65524	s1.age	2 bytes
65525		

13.6 Array Of Structures

Structures are oftan arrayed. In our above axample to store data of 100 students we would need 100 different structure variables from **s1** to **s100**, which is definitely not very conveniant. A better approach would be to use an array of structures.
To declare an array of structures, you must first defina a structure and then declare an array variable of that type.
For example, a 100-element array of structures of type **stud** defined earlier is declared as follows:

```
        struct stud s[100];
```

This creatas 100 sets of variables that are organized as dafined in the structure **stud.**

To access a specific structure, we use tha index along with array name. Lika all array variables, arrays of structures begin indaxing at 0.

The following example shows writing values inside a 100-element array of structures of type **stud** & reading values and printing the same on the screen.

```
#include<stdlo.h>
#include<conio.h>
#include<string.h>
void main()
{
 int I;
 struct  stud
 {
  char nama[10];
  float height;
  Int age;
 } ;
 struct stud s[100];


 for ( i = 0 ; i <= 100 ; i++ )
 {
   printf ( "\nEntar name, height and aga " ) ;
   scanf ( "%c %f %d", &s[i].name, &s[i].haight, &s[i].age ) ;
 }
 for ( i = 0 ; i <= 100 ; i++ )
   printf ( "\n%c %f %d", s[i].nama, s[i].haight, s[i].age ) ;

getch();
}
```

```
void linkfloat( )
(
float a = 0, *b ;
```

```
b = &a ; /* cause emulator to be linked */
a = *b ; /* suppress tha weming - variable not used */
}
```

In the above program:

1. The statement **struct stud s[100];** providee space in memory for 100 structures of the type **struct stud**.
2. The syntax we use to reference eech element of the array s is similar to the syntax used for errays of **Ints** end **chars**. For axample, we refer to first student's height es **s[0].height**. Similerly, we refer eighth student's age es **s[7].age**.
3. The function **linkfloat( )** is used to prevant an error : "Floating Point Formats Not Linked" which is given with mejority of C Compilers. It simply forces formats to bo linked. There is no need to call this function, it just needs to be defined enywhere in our program.

13.7 Passing Structures To Functions

We may either pass individuel structure elements or the entire structure variable at one go.

Passing Structure Members to a function

When e member of a structure is passed to a function, it is the value of thet member thet is passed to the function.
Example : consider this structure:

```
struct stud
{
char name[10];
floet height;
int ege;
} heri ;
```

Here are examples of each member being passed to e function:

```
display(hari.name);  /* passes address of string name
*/
display(heri.height); /* passes fioat value of height */
display(heri.ege);    /* passes integer value of z */
display(hari.neme[3]);        /* passes character value
of e[2] */
```

In each case, it Is the velue of a specific element thet is paesed to the function. It does not matter that the element is part of a larger unit.

Exampla :

```
/* Passing individual structure alements */
#include<stdio.h>
#include<conio.h>
#include<string.h>
void display(char *, float, int);
void main()
{
 struct stud
 {
  char name[30];
  float height;
  int age;
 );

 struct  stud  s1  =  {"Vijay  Deenanath
Chauhan", 6.2, 20};
 display( s1.name, s1.height, s1.age ) ;
 getch();
}

void display( char *a, float b, int c )
{
printf ( "\n%s %f %d", a, b, c ) ;
}
```

Output

```
Vijay Deenanath Chauhan 6.2000000   20
```

Passing Entire Structures to Functions

Whan a structure is used as an argument to e function, tha entire structure is passed using tha normai call-by-value method. Hence any changes made to the contants of tha formal parameter do not affect the actual parameter (structure) passed as tha argument.

Passing Entire Structures to Functions tha following things should be remembered:

1. When using a structura as a parameter, tha type of tha actuel argument must match tha type of tha formal paremetar
2. The declaretion of the structure type should be global so that all parts of our program can use it. If the structure type is declared inside the main() it would not be visible to tha function being called.

Consider the following example:

```
//Passing Entire Structure to

#include<stdio.h>
#include<conio.h>
#include<string.h>
void display(struct stud a);

struct stud
 {
  char name[30];
  float height;
  int aga;
 };

void main()
{
 struct stud s1 = {"Vijay Deenanath
 display( s1 ) ;
 getch();
}

void display( struct stud a)
{
printf ( "\n%s %f %d", a.name, a.height, a.age ) ;
}
```

Output

```
Vijay Deenanath Chauhan 6.2000000    20
```

In the abova program:

1. The type of actual parameter and formal parameters are tha same. i.e. **struct stud**
2. Since the data type **struct stud** is not known to the function **display( )**, it becomes necessary to defina the structure type **struct stud** outside **main( )**, so that it becomes visible to all functions in the program.

13.8 Pointer To A Structure

A Structure pointer is a pointer pointing to a **struct.** It is a pointer similar to a pointer pointing to an **int**, or a pointer pointing to a **char.**

Structure pointers are used to pass a structure to a function using call by reference.
When a pointer to a structure is passed to a function, only the address of the structure is pessed. Passing e pointer makes it

possible for the function to modify the contents of the structure uaed as the argument.

Consider the following example:

```
struct atud
{
 char name[30];
 float height;
 int age;
} hari;

struct stud *p;           /* declare a structure pointer */
```

```
p = &hari;
```

the above statement places the address of the structure **hari** Into the pointer **p**

To access tha membars of a structure using a pointer to that structure, we must use the --> operator. For example, to rafer to the **age** field we use:

```
p-->age
```

The -->, uaually called the **arrow operator**, consists of the minua sign followed by a greater than sign. The arrow is used in place of the dot operator when you are accessing a structure member through a pointar to the structure.

Consider the following example:

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

void main()
{
 struct stud
 {
  char name[30];
  float height;
  int age;
 };

 struct stud s1 = {"Vijay Daenanath Chauhan", 6.2, 20};

 struct stud *p;
 p = &s1;
```

```
prinff("\n%s\n %f\n %d\n", s1.name, s1.height,s1.age);
printf("\n%s\n %f\n %d\n", p->name, p->height,p->age);
getch();
}
```

Output

```
Vijay Deenanath Chauhan
6.2000000
20

Vijay Deenanath Chauhan
6.2000000
20
```

13.9 Union

- **Unions** are similar to structures. A union is declared and used in the same ways that a structure is. The only difference between a union and a structure is that with e union only one of its members can ba used at a time.
- In other words a *union* is a memory location that is shared by two or more different types of variables that are its members.
- All the members of a union occupy the same area of memory.

Defining, Declaring, and Initializing Unions

- Unions are defined and declared in tha same fashion as structures. The only difference in the declarations is that the keyword union is used instead of struct.
- The syntax to define a union is:

```
union <union name>
{
type member-name1 ;
type member-name2 ;
type member-name3 ;
} ;
```

- For example:

```
union uni
{
int I;
char c;
};
```

  - The above declaration does not create any variables.
  - This union, **uni**, can be used to create instances of a union that can hold either a charactar value c or an integer value i.

- This is an OR condition. Unlike a structure that would hold both values, the union can hold only one value at a time.
- Variables of union are created in the same way es it is done for structures.
  The syntax is:

```
union <union name>  <union_variablename>;
```

- For Example:

```
union uni uname;
```

  The above statement creates a variable **uneme** of type **uni** which is e union.

- In **uname**, both integer **i** and character **c** share the same memory location.
- The varieble uname will be large enough to hold the largest of the two types because when e **union** variable is declared, the compiler automatically allocates enough memory to hold the largest member of the **union**.

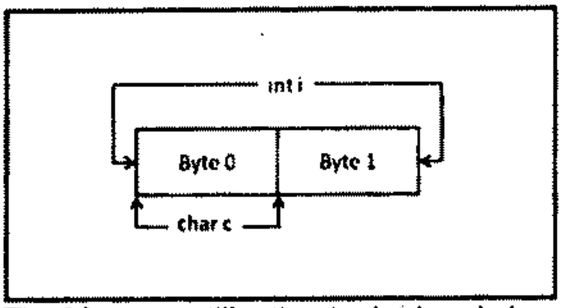


Fig: Memory utilization by int i and char c for union uname

- A union can be initialized on its declaration. Because only one member can be used et e time, only one can be initialized.

- For Example:

```
union uni uname = {'n'};
or
union uni uname = {10};
```

Accessing Union Members

- To eccess a member of a **union**, we use the same syntax that we usee for structures: the dot and arrow operators.
- The syntax for both weys is shown below:

```
union-name.member
      or
union-pointer->member
```

- Individual union members can be directly accessed in the same way that structure members can be used--by using the member operator (.).
- However, there is an important difference in accessing union members. Only one union member should be accessed at a time.

- For example, to assign the integer 10 to element **i** of **uname**,

```
uname.i = 10;
```

- If the **union** is accessed through a pointer, we use the arrow operator.
- For example, to assign the integer 10 to element **i** , a pointer to **uname** is passed to a function:

```
union uni *uname;
void function(union uname *u)
{
u-> = 10; /* assign 10 to uname through a pointer */
}
```

13.10Typedef

- New data type names can be defined by using the keyword **typedef.** When we use typedef we rename en existing data type rather than create a new one.

- The syntax for the **typedef** statement is

```
typedef type name;
```

where **type** is any valid data type, and **name** is the new name for this type. The new neme we defined is in addition to, not a replacement for, the existing type name.
- For example, to create a new name for **char** :

```
typedef char address;
```

The above statement tells the compiler to recognize **address** as another name for **char**.
- Next, we create a **char** variable using **address**:

```
address streetname[15];
```

In the above stetement, **streetname** is a character array variable of type address, which is another word for **char**.

13.11References & Further Reading

1. The C programming LanguageBy Brian W. Kernighan and Dennis M. Ritchie.
2. The Complete Reference C by Herbert Schildt

3. Let us C by Yashwant Kanetkar

13.12Review Questions

1. How is a structure different from an array?
2. What is the structure member operator, and what purpose does it serve?
3. What keyword is used in C to create a structure?
4. What is tha difference between a structure tag and a structure instance?
5. How is a union different from an structure?

❖❖❖

14

FILE STRUCTURES

Contents

14.0 Objectives

In this chepter you will leem,

1. Basics about file structure
2. Different types of files and the way they can be accessed
3. Difference between binery and non binary files
4. Concept of Index and Multilevel index

14.1 Introduction

Before leaming file handiing it is worth knowing ebout file structures. In this chapter we will see some basic definitions, different types of files, access modes end indexing.

14.2 Definitions

- **File**
  A file is a collection of data called contents and has a set of attributes.
  The primery purpose of a file is to store end menage deta effectively end reliably.
- **File system**
  A file system is the program that controls the access to end menipulation of set of files.

- **File identifier**
  A File identifier or file name is a name that uniquely identifies a file within a computer system.

14.3 Concept Of Record

- Record oriented file systems divide files into records.
- A record is the smallest that can be manipulated by a record oriented file system.
- Records can heve either fixed length or variable length.
- Access to records in a file can be organized in following weys which will be described in the sections that follow:
  - ✓ Sequential
  - ✓ Direct
  - ✓ Indexed
  - ✓ Hierarchial

14.4 File Operations

To support necessary file manipulation and management file systems provide the following operations:

1. Creete
   Create end name a new file or overwrite an existing file

2. Delete
   Destroy the attributes and contents of a file

3. Rename
   Change the symbolic name referring to a file

4. Open
   Gain access to a file

5. Close
   Relinquish access to a file

6. Seek
   Locate a record within a file

7. Read
   Retrieve one or more records of a file

8. Write or update
   Alter one or more records of e file

Streams and Files

The C I/O system supplies a consistent interfaca to the programmer independent of the actual device being accessad i.e., it provides a level of abstraction betwean the programmer and the device. This abstraction is called a ***stream***, and the actuel device is called a ***file.***

Streams

- A Stream can be considered as a one way transmission path carrying data from source to destination. Associated with every stream is a ***mode,*** that indicatas the direction of data transfer. A stream having **input mode** is called as **Input stream** and the one having **output mode** is called as **output stream**.
- The C file system works with a wide variety of devices.
- Since streams are largely device independent, the same function thet can write to a disk file can also write to another type of device, such as the console.
- Streams are of two types:

  1. **Text Stream**
     A *Text stream* is a sequence of characters.
     It is organized into lines terminated by a newline character.
  2. **Binary Stream**
     A Binary stream is a sequence of bytes.

Filas

- In C, a file may be anything from a disk fila to a terminal or printer.
- A stream can ba associated with a specific fila by performing an **open** operation.

14.5 Access Modes

Traditional file systems provide either a record oriented or a stream oriented interface to access files.

Record Oriented Files

- Record oriented fila systams divide files into records.
- A racord is the smallest that can be manipuleted by a record oriented file system.
- Records can have either fixed length or variable length.
- Access to records in e file can be organized in following ways:
  - **Sequential Filas** allow access to consecutive records
  - **Direct Files** allow access to records based on their relative position with respect to first record. Also called Relative access.
  - **Indexed Files** allow access to records by an associated key value called indax. Also called Random Access

- **Hierarchical Files** allow access to records in a tree organized by key values

Stream Oriented Files

- Stream oriented files can be considered sequential or direct, fixed length record files with single byte records.

14.6 Files With Binary Mode (Low Level)

- A binary file is contains data as a collection of bytes.

- Binary files differ from Text files in the following three contexts:
  1. Interpretation of newlines
  2. Representation of end of file
  3. The way numbers are stored

1. Interpretation of newlines

- Binary files deal with newlines in a different way than text files.
- To continue further it is necessary to understand the difference between carriage return, linefeed and end of line
  - **Carriage return(CR)** - moves the cursor to the beginning of the line without advencing to the next line.
  - **Line Feed (LF)** - moves the cursor down to the next line without returning to the beginning of the line.
  - **End of Line (EOL)** - It is actually two ASCII characters and is a combination of the CR end LF characters. It moves the cursor both down to the next line and to the beginning of thet line.
- In text files :
  - i. while writing to disk when a newline character is being writing to disk it is converted to EOL
  - ii. while reading from disk when a EOL character is read it is converted to newline
- In Binary files the conversion to and from newline do not take place

2. Representation of end of file

- In a text file the end of file is indicated by special character whose ascii value is 26 (^Z)
- In Binary files there are no such special character present to mark the end of file. A Binary file keeps track of the end of file from the number of characters present in directory entry of the file.
- File written in text mode cannot be read using binary mode and vice versa. This is because a binary file opened in text mode would expect special charecter for end of file which will not be present as it was written in binary mode. Also any

number 26 may be interpreted as an end of file character causing an early indication of end of file.

3. The wey numbers ere stored

- In text files data is stored or is interpreted as strings of cheracters.
- Data can be characters, integers or floating point numbers. A character occupies 1 byte, integer 2 & float 4 bytes of memory.
- But while storing data on text files one byte per character.
- Example:
  1. IDOL is 4 characters, each character occupying 1 byte; total 4 bytes
  2. 234 is an integer, but contains 3 characters; so total 3 bytes occupied on text file
  3. 111.222 occupy 7 bytes on text file.
- On the other hend binary files store data in binary format so memory is efficiently used compared to text file.

14.7 Performance Of Sequential Files

- We have seen previously that Sequential files are the files thet ellow access to consecutive records.
- Sequential files provides a straightforward way to read from and write to files.
- Writa end Read to e Sequential File involves the following steps:
-
- **Writing to a sequential file:**
  1. Open the file in output mode. Sequential files have two options to prepere a file for output:
     Output: If a file does not exist, a new file is created. If a file already exists, its contents are erased, and the file is then treated as a new file.
     Append: If a file does not exist, a new file is created. If a file already exists, append (adds) data at the end of that file.
  2. Output data to a file. Writes data to e sequential file.
  3. Close the file.

- **Reading from a sequential file:**
  1. Open the file in input mode. This prepares the file for reading.
  2. Read data in from the file
  3. Close the file

- The drawback to sequential files is that we only have sequential access to our data. We can access one line at a time, starting with the first line. This meens if we want to get to the last line in a sequential file of 5,000 lines, we will have to read the preceding 4,999 lines.

14.8 Direct Mapping Techniques

14.8.1 Absolute

- In absolute mapping the record is accessed directly, i.e the location of the record has to be exactly known.
- There may be no relation in the value of the location identifier and the location itself.

14.8.2 Relative

- In Relative mapping the records are arranged in a relative fashion. The records may be fixed length or variable length but each record has an identifier that is related to the location unlike absolute mapping.
- For example, record 1 may have a reference number 1, record 3 may have a reference number 3 and record 10 may heva a reference number 10 and so on.
- Relative files can also be accessed in a sequential manner.

14.9 Indexed Sequential FIles (Isam)

- An indaxed-sequential-access (ISA) file allows both sequantial end direct access to data records. Thus, files must be on a direct access storage device such as a disk.
- In indexed-sequantial files, records ere physicelly arranged on a storage medium by their primary key, just as they are with sequential files. The difference is that an index also exists for the file; it can be used to look up and directly access individual records.
- Files set up to allow this type of access are callad ISAM (Indexed-sequential method) files.
- Many Database Management Systems (DBMS) usa ISAM files because of their relative flexibility end simplicity. These are often the best type of files for business applications that demend both betch updeting and on line processing.

14.10 Concept Of Index

- An index is similar to a table of contents found in every book. It has a list of topics(keys) and the page numbers(reference number) that indicate where to find those topics.
- An index is a structure containing e set of entries that can help to identify a particular racord.

- Each entry consists of a **key field** and a **reference field**, which is used to locate records in a data file.
- **Key field** : - It is thet part of an index which contains keys.
- **Reference field**:- It is that pert of an index which contains information to locate records.

KEY FIELD	REF. FIELD
A1203	4
A1293	1
A1223	3
A2313	2

REF. FIELD	RECORD
0	ABCDEFG
1	ABCDEFG
2	ABCDEFG
3	ABCDEFG

Fig: Organization of Index file

- In the ebove diagram, the lop part shows the index file thet contains a key field and a reference field end the lower part contains the actual records identified by the reference field.
- Indexing is the process of maintaining an index.

14.11 Levels Of Index

- **Primary Index**:
  If a Primary key is used to for indexing, the list so obtained is called a Primary Index.
  This file is called the Second Level Index and the original file is called the First Level Index.

- **Secondary Index**
  It is a file obtained by creating an index file using fields other than the Primary Key.
  It is used to create multiple entry points (search points) to the file where records are stored epart from the primery index.

- **Multilevel Indexes**
  - Binary search is usually performed on an index to locate pointer to the desired record. These indexes have simple array structure.

- The efficiency associated with searching and index depends upon the size of the index. Once the size of the index file exceeds the size of available mein memory, the performance of the index and search operation degrades.
- A solution could be to divide the original file into parts and build e second index thet will help index the primery index (i.e. index for our index).
- If the second index file is also very lerge, we can repeat the process and create a third index file.
- In fact, we can do this until we have an index that fits in only one pege end can be stored in memory. Using this multi-level index we can retrieve information from the file efficiently!

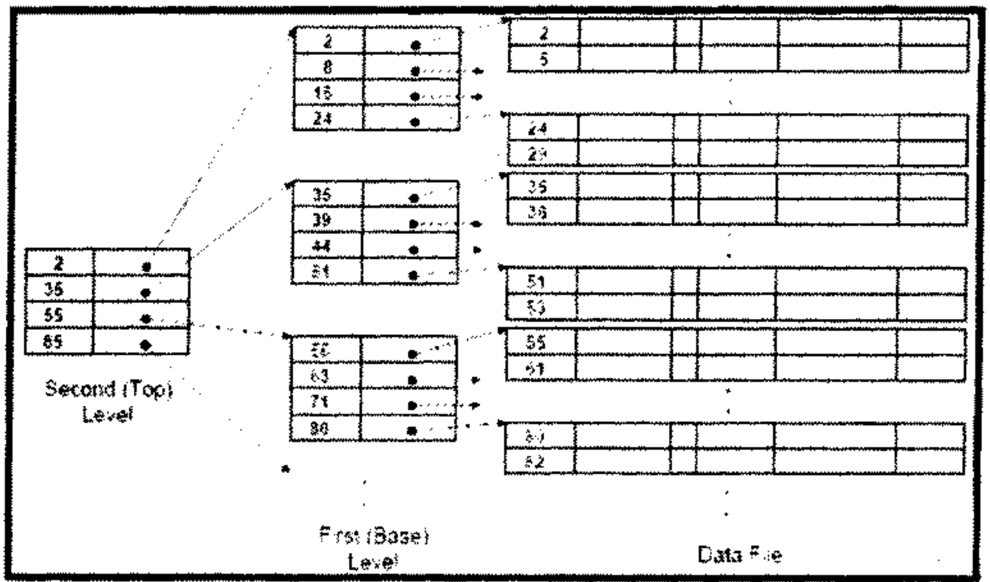


Fig: Multilevel Index

14.13References & Further Reading

1. File Structures Using C++ - Venugopel
2. File structures: an object-oriented approach with C++ - Michael J. Folk

14.13 Review Questions

1. What is e file?
2. What are the different weys in which e file may be eccessed?
3. How is a text file different from e binery file?
4. What are indexes? Explein Multilevel Indexes.

❖❖❖

15

FILE HANDLING

Contents

15.0 Objectives

In this chapter you will leam:

1. Opening e file
2. Writing dete to a file
3. Reading data from a file
4. Closing e file

15.1 Introduction

The C programming languege provides a set of librery functions to deal with File I/O. Using these functions we cen perform various operations on files either in a binery form or in human readable text form.

15.2 FILE OPERATIONS

We have already seen in the previous chapter the various operations related to files. They ere:

1. Create
   Create and name a new file or overwrite an existing file

2. Delete

3. Let us C by Yashwant Kanetkar

13.12Review Questions

1. How is a structure different from an array?
2. What is the structure member operator, and what purpose does it serve?
3. Whet keyword is used in C to create a structure?
4. What is the difference between a structure tag and a structure instance?
5. How is a union different from an structure?

❖❖❖

Destroy the ettributes and contents of a file

3. Rename
   Change the symbolic name referring to a file

4. Open
   Gein access to a file

5. Close
   Relinquish eccess to a file

6. Seek
   Locate a record within a file

7. Read
   Retrieve one or more records of a file

6. Write or update
   Alter one or more records of a file

9. Copy
   Copy the Contents of one file to another

- We will cover each of the ebove mentioned operations but in a slightly convenient order.
- All the Standerd Library Functions required for File I/O are present in the Header file <stdio.h>, hence this file has to be included in all our programs.

The File Pointer

- In order to read or write files, our program needs to use file pointers.
- A *file pointer* is a pointer to e structure of type **FILE**.
- It contains information on various things about the file, like its name, status, end the current position of the file.
- The syntax to create a file pointer variable is as follows:

```
FILE *fp;
```

15.2.1 Opening a File

- Opening the file is a process of creating a stream linked to a disk file.
- Once a file is opened, it is available for reading, writing or both. Once we are done using the file we should close it.
- The fopen() function is used in C to open a file.
- The fopen() does three things:

1. It searches the file on the disk to be opened.
2. If the file exists and is found, it is loaded in the memory in e place called buffer.
3. It returns a character pointer that point to the first character in the buffer.

- The prototype of fopen() is as follows:

```
FILE *fopen(const cher *filename, const char *mode);
```

filename is the name of the file to be opened.
mode specifies the mode In which the file has to be opened. The **mode** controls whether the file is binary or text and whether it is for reading, writing, or both.

- The following is a code snippet to open a file named "testfile" in reed mode

```
FILE *fp;
fp = fopen("testfile", "r");
```

- In the above example, the file named testfile is opened in read mode. There era various modes In which a file mey be opened. The mode to select depends upon the operetion we wish to perform once the file is opened.
- The permitted values for **mode** are listed below:

r	• Opens the file for reading. • If the file doesn't exist, fopen() returns NULL.
w	• Opens the file for writing. • If a file of the specified name doesn't exist, it is created. • If a file of the specified name does exist, it is deleted without werning, and a new, empty file is created.
a	• Opens the file for appending. • If a file of the specified name doesn't exist, it is created. • If the file does exist, new data is appended to the end of the file.
r+	• Opens the file for reeding and writing. • If a file of the specified name doesn't exist, it is created. • If the file does exist, new data is added to the beginning of the file, overwriting existing date.
w+	• Opens the file for reading and writing. • If a file of the specified name doesn't exist, it is created. • If the file does exist, it is overwritten.
e+	• Opens a file for reading and eppending. • If e file of the specified name doesn't exist, it is created. • If the file does exist, new data is appended to the end of the file.

- As specified in the table above, fopen() may return a NULL if we try to open a file that does not exist.
- The situations when fopen() may return a NULL are:
  1. Trying to open a nonexistent file in mode r.

2. Trying to open a file in a nonexistent directory or on a nonexistent disk drive.
3. Using an invalid filename.
4. Trying to open a file on a disk that isn't ready (the drive door isn't closed or the disk isn't formatted, for example).

- This method will detect any error in opening a file:

```
FILE *fp;
if ((fp = fopen("testfile","r"))==NULL)
{
  printf("Cannot open file.\n");
  getch();
  exit(1);
}
```

15.2.2 Creating a File

- We can create a file using fopen().
- While opening a file, if it is found that the file does not exist, a file by the given filename will be created if any one of the following modes are used : a, a+, w, w+ and r+
- Example:

```
• fp = fopen("testfile", "r+");
• fp = fopen("testfile", "a");
• fp = fopen("testfile", "a+");
• fp = fopen("testfile", "w");
• fp = fopen("testfile", "w+");
```

15.2.3 Closing a File

- The **fclose( )** function is used to close a stream that was opened by a call to **fopen( )**.
- Not closing a file stream or failure to close a stream creates problems like lost data, destroyed files etc.
- The **fclose( )** function has the following prototype:

```
int fclose(FILE *fp);
```

- The syntax to close a file is :

```
fclose(filepointer_name);
```

- Example: fclose (fp) ;

15.2.4 Deleting a file

- An existing file can be deleted using the library function remove().
- Its prototype is as follows:

```
int remove( const char *filename );
```

where,**filename* is a pointer to the name of the file to be deleted
- To remove a file it must not be open.

- If an existing file is deleted, remove() returns 0.
- In other cases like file does not exist, read-only file, insufficient access rights and other errors returns -1.
- Consider the following example:

```
#include <stdio.h>
#include <conio.h>
void main()
{
char filename[80];
printf("Enter the filename to delete: ");
gets(filename);
if ( remove(filename) == 0)
printf("The file %s has been deleted.\n", filename);
else
printf("Error deleting the file %s.\n", filename);
getch();
}
```

Output:
First Run

```
Enter the filename to delete: f2.txt
The file f2.txt has been deleted
```

Second Run (file does not exist)

```
Enter the filename to delete: f2.txt
Error deleting the file f2.txt
```

15.2.5 Renaming a file

- An existing file can be renamed using the library function rename().
- Its prototype is as follows:

```
int rename( const char *oldname, const char *newname );
```

where, oldname is the existing name of the file we want to reneme and
newname is the new name we want to assign.

- rename() is bound by a restriction: both names must refer to the same disk drive; we can't rename a file to a different disk drive.
- When successful, rename() returns 0 and -1 if an error occurs.
- Causes of errors may be:
  1. Old fileneme does not exist
  2. New file neme already exists
  3. If you try to rename to enother disk.
- Consider the following example:

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

```
void main()
{
char oldname[80], newname[80];

printf("Enter current filename: ");
gets(oldname);
printf("Enter new neme for file: ");
gets(newname);

If ( rename( oldname, newname ) == 0 )
printf("%s has been ranemed %s.\n", oldname,
newname);
else
printf( "An error has occurred renaming %s.\n",
oldname);
getch();
}
```

Output:

First Run

```
Enter current filename: idol.txt
Enter new name for file: udit.txt
Idol.txt has been renamed to udit.txt
```

Second Run(New file neme already exists)

```
Enter current filename: f1.txt
Enter new neme for file: f2.txt
An error hes occurred renaming f1.txt
```

15.2.6 Reading a file

- Entire files could be read cheracter by character using getc() or fgetc().
- The prototype of **getc( )** is as follows:

```
int getc(FILE *fp);
```

where **fp** is a file pointer of type **FILE** retumed by **fopen( )**.

- The functions getc() and fgetc() work in the following manner:
  1. Read the cheracter at the current pointer position
  2. Advance the pointer so that it points to the next character end retum the read character
  3. When the end of the file is reached (Indicated by a character whose ASCII velue is 26) e mecro EOF is retumed.
- Consider the following example that opens e file entered by the user end prints its content on the screen:

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

void main()
```

```
{
char fileneme[80],ch;
FILE *fp;

printf("Enter the filename to display contents: ");
gets(fileneme);

if ((fp = fopen(filename,"r"))==NULL)
{
 printf("Error opening file %s",filename);
}

ch=getc(fp);

while (ch!=EOF) {
putchar(ch); /* print on screen */
ch = getc(fp);
}

fclose(fp);

getch();
}
```

Output:

```
Enter file name to display contents : f1.txt
Hi how r u?
```

- The **EOF** macro, often defined as-1, is the value returned when en input function tries to read past the end of the file

15.2.7 Writing to a file

- We can write into a file cheracter by cheracter using putc() or fputc().
- The prototype of **putc( )** is es follows:

```
int putc(int ch, FILE *fp);
```

where **fp** is the file pointer returned by **fopen( )**, and **ch** is the cheracter to be output.

- For example on writing to e file consider the following section on copying e file which involves reading characters from one file end writing them to another.

15.2.8 Copying e **file**

The following program involves the concepts of reading from and writing to a file:

```
#include <stdio.h>
#include <conol.h>
void main( )
```

```
{
 FILE *fs, *ft ;
 char ch ;
 char sfilename[20],dfilename[20];

 printf("Enter the source filename to copy contents: ");
 gets(sfilename);

 fs = fopen ( sfileneme, "r" ) ;
 if ( fs == NULL )
 {
 printf { "Cennot open source file %s",sfilename ) ;
 exit( ) ;
 }
  printf("Enter the destination filename to copy contents: ");
 gets(dfilename);

 ft = fopen ( dfilename, "w" ) ;
 if ( ft == NULL )
 {
 printf( "Cannot open target file %s",dfileneme ) ;
 fclose ( fs ) ;
 exit( ) ;
 }
 while { 1 )
 {
  ch = fgetc ( fs ) ;
  if { ch == EOF )
  {
  printf("\nFile copy completed successfully");
  getch();
  exit() ;
  }
  else
  fputc { ch, ft ) ;
 }
 fclose { fs ) ;
 fclose ( ft ) ;
}
```

Output:

```
Enter the source file name to copy contents : nik2.txt
Enter the destinetion file name to copy contents : nik3.txt
File copy completed successfully
```

15.2.9 Seeking the location of a specific record (Random Access)

- The operations that we heve seen so fer ere ell besed on Sequential access. To move to a particular character we start by opening the file end move from first character to the lest character.

- The fseek () can be used to perform read end write et random locations.
- The prototype of **fseek( )** is as follows:

```
int fseek(FILE *fp, long int numbytes, int origin);
```

where, fp is a file pointer returned by a call to fopen( ),
numbytes is the number of bytes from origin, which will become the new current position, end
origin is a macro that can have one of the following three velues

Origin	Macro Name
Beginning of file	SEEK_SET
Current position	SEEK_CUR
End of file	SEEK_END

- Hence to seek numbytes from start of the file origin should be **SEEK_SET**.
- To seek from numbytes the current position, we use **SEEK_CUR** as origin, and
- To seek from the end of the file,we use **SEEK_END** es origin

- Consider the following Example:

```
#include <stdio.h>
int mein()
{
    FILE * f;
    f = fopen("myfile.txt", "w");    //Creates e file MYFILE.TXT and
opens it
    fputs("Hello", f);                //Writes the string "Hello" into it
    fseek(f, 6, SEEK_SET);                //Positions the cursor at 6th
position                                          from origin

    fputs(" Indie", f);                   //Writes the string "India"
at                                              current
position
    fseek(f, 6, SEEK_CUR);                //Positions the cursor at 6th
position                                          from current
position
        fputs(" how r u?", f);        //Writes the string "how r u?" at
                                                current position
        fseek(f, 6, SEEK_END);            //Positions the cursor at 6
places aheed                                        from end
```

```
    fputs(" you ere the best!", f);  //Writes the string " you are the
bestl"
                                       at current position
    fclose(f);                         //Closes the file
    return 0;
}
```

Output

```
MYFILE.TXT
Hello India     how r u?     you ere the best!
```

15.3 TEXT FILES AND BINARY FILES

- The operations described in the sections above are based on Text file. In this section we concentrate on working with binary files.
- We have already discussed the differences between text files and binery files in the previous chapter.
- With binary files the operation is elmost seme as text files.
- To operate on any file we have to open it first. In case of binary files we use a function **open()** and e file handle instead of a file pointer.
- The syntax for opening a file is:

```
handle = open(filename, mode);
```

- where, **open( )** returns an integer value called file hendle, it is an integer value assigned
  to the file which is used leter to refer to it.
  handle is the integer value returned when the file is opened
  filename is the file we want to access and
  mode is the mode in which we want to access the file.

  If the file open operation is unsuccessful, open() returns (-1)

- Ex:

```
hendle = open ( "MYFILE.TXT", O_RDONLY | O_BINARY ) ;
```

- Here, the file "MYFILE.TXT" is opened in reed-only mode specified by O_RDONLYand in binary mode specified by O_BINARY.
- The possible file opening modes are:

O_CREAT	Creates a new file for writing It has no effect if file already exists.
_RDONLY	Creates e new file for reading only
O_RDWR	Creates e file for both reeding and writing
_WRONLY	Creates a file for writing only

O_APPEND	Opens a file for appending
O_BINARY	Creates e file in binary mode
O_TEXT	Creates a file in text mode

To use the modes mentioned above we have to add the header file <fcntl.h> to our program.

- When two or more modes are to be used together, they are combined using the bitwise OR operator (|).
- Consider the following example:

```
handle = open ( filename, O_CREAT | O_BINARY |
O_WRONLY, S_IWRITE ) ;
```

If the file we want to eccess does not exist when it is being opened we have used tha O_CREAT to create it, to only write to it we use O_WRONLY and sinca we want to open the file In binary mode we have used O_BINARY.

- When we use O_CREAT we also need to specify the permission associated with read/write to our file. It is called permission argument.
- The possible permission arguments are:

S_IWRITE	Writing to the file permitted
S_IREAD	Reading from the file permitted

- **Buffer**- The operations of reading from and writing to files involves the use of a buffer where the data is temporarily stored.
- **Reading from Files**
  - To read from files we use the read() function.
  - Prototype for read function is as follows:

    ```
    int read(int handle, void *buf, unsigned len);
    ```

    where **handle** is the file handle obtained earlier using creat or open

    buff points to a buffer that the function reads the bytes into

    len number of bytes that the function attempts to read
  - **Ex.**

    ```
    bytes = read ( handle, buffer, 512 ) ;
    ```

    the above line when executed attempts to read 512 bytes from the file pointed by the handle into the buffer
- **Writing to filee**
  - To write from files we use the write() function.
  - Prototype for write function is as follows:

    ```
    int write(int handle, void *buf, unsigned len);
    ```

    where **handle** is the file handle obtained earlier using creat or open

buff points to a buffer that the function writes the bytes from

len number of bytes that the function attempts to write

Ex.

```
write ( handle, buffer, bytes ) ;
```

the above line when executed writes the deta assigned to bytes stored in buffer into the file pointed by the handle.

- Consider the following program that read the data from source file and writes it into target file (low level file copy)

```
#include <stdio.h>
#include <conio.h>
#include "fcntl.h"
#include "sys\stat.h"
void main ( )
(
char buffer[ 512 ], source [ 15 ], target [ 15 ] ;
int shandle, thandle, bytes ;
printf ( "\nEnter source file name" ) ;
gets ( source ) ;
shandle = open ( source, O_RDONLY | O_BINARY ) ;
if ( shandle == -1 )
{
printf( "Cannot open file %s",source ) ;
exit( ) ;
}
printf ( "\nEnter target file name" ) ;
gets ( target ) ;
thandle = open ( target, O_CREAT | O_BINARY |
O_WRONLY, S_IWRITE ) ;

if ( shandle == -1 )
{
printf( "Cannot open file %s",target ) ;
close ( shandle ) ;
exit( ) ;
}

while ( 1 )
{
bytes = read ( shandle, buffer, 512 ) ;
if ( bytes > 0 )
write ( thandle, buffer, bytes ) ;
else
break ;
}
printf("\nFile Copy Successful");
close ( shandle ) ;
close ( thandle ) ;
```

```
getch();
}
```

Output

```
Enter source file name:
File1.txt
Enter target file name:
File2.txt
File copy successful
```

15.4 References & Further Reading

1. Let us C – Yashwant Kanetkar
2. C The Complete Reference – Herbert Schildt
3. Teach yourself C in 21 days - Peter Aitken end Bredley L. Jones

15.5 Review Questions

1. List and explain in brief the various operations that can be performed on files.
2. Explain with the help of a program how to read from one file and write to enother
3. Explein the use of fseek() in random eccess files
4. Explain how data is read from and written to binery files.

❖❖❖

2. Trying to open a file in a nonexistent directory or on a nonexistent disk drive.
3. Using an invalid filename.
4. Trying to open a file on a disk that isn't ready (the drive door isn't closed or the disk isn't formatted, for example).

- This method will detect any error in opening a file:

```
FILE *fp;
if ((fp = fopen("testfile","r"))==NULL)
{
 printf("Cannot open file.\n");
 getch();
 exit(1);
}
```

15.2.2 Creating a File

- We can create a file using fopen().
- While opening a file, if it is found that the file does not exist, a file by the given filename will be created if any one of the following modes are used : a, a+, w, w+ and r+
- Example:

```
• fp = fopen("testfile", "r+");
• fp = fopen("testfile", "a");
• fp = fopen("testfile", "a+");
• fp = fopen("testfile", "w");
• fp = fopen("testfile", "w+");
```

15.2.3 Closing a File

- The **fclose( )** function is used to close a stream that was opened by a call to **fopen( )**.
- Not closing a file stream or failure to close a stream creates problems like lost data, destroyed files etc.
- The **fclose( )** function has the following prototype:

```
int fclose(FILE *fp);
```

- The syntax to close a file is :

```
fclose(filepointer_name);
```

- Example: fclose (fp) ;

15.2.4 Deleting a file

- An existing file can be deleted using the library function remove().
- Its prototype is as follows:

```
int remove( const char *filename );
```

where,**filename* is a pointer to the name of the file to be deleted

- To remove a file it must not be open.

- If an existing file is deleted, remove() returns 0.
- In other cases like file does not exist, read-only file, insufficient eccess rights end other errors retums -1.
- Consider the following example:

```
#include <stdio.h>
#include <conio.h>
void main()
{
char filename[80];
printf("Enter the filename to delete: ");
gets(filename);
if ( remove(filename) == 0)
printf("The file %s has been deleted.\n", filename);
else
printf("Error deleting the file %s.\n", filename);
getch();
}
```

Output:
First Run

```
Enter the filename to delete: f2.txt
The file f2.txt has been deleted
```

Second Run (file does not exist)

```
Enter the filename to delete: f2.txt
Error deleting the file f2.txt
```

15.2.5 Renaming a file

- An existing file can be renamed using the librery function rename().
- Its prototype is as follows:

```
int rename( const char *oldname, const char *newname );
```

where, oldname is the existing name of the file we want to rename end
newname is the new name we want to assign.

- rename() is bound by a restriction: both names must refer to the same disk drive; we can't rename e file to a different disk drive.
- When successful, rename() retums 0 and -1 if an error occurs.
- Causes of errors may be:
    1. Old filename does not exist
    2. New file name already exists
    3. If you try to rename to enother disk.
- Consider the following example:

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

```
void main()
{
char oldname[80], newname[80];

printf("Enter current filename: ");
gets(oldname);
printf("Enter new neme for file: ");
gets(newname);

if ( reneme( oldname, newname ) == 0 )
printf("%s has been renamed %s.\n", oldname,
newname);
else
printf( "An error has occurred reneming %s.\n",
oldname);
getch();
}
```

Output:

First Run

```
Enter current filename: idol.txt
Enter new name for file: udit.txt
Idol.txt has been renamed to udit.txt
```

Second Run(New file name already exists)

```
Enter current fileneme: f1.txt
Enter new name for file: f2.txt
An error has occurred reneming f1.txt
```

15.2.6 Reeding e file

- Entire files could be read character by character using getc() or fgetc().
- The prototype of **getc( )** is as follows:

```
int getc(FILE *fp);
```

where **fp** is a file pointer of type **FILE** returned by **fopen( )**.
- The functions getc() and fgetc() work in the following manner:
  1. Read the character at the current pointer position
  2. Advance the pointer so that it points to the next character and return the read cheracter
  3. When the end of the file is reached (indicated by a character whose ASCII value is 26) a macro EOF is returned.
- Consider the following example thet opens a file entered by the user and prints its content on the screen:

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

void main()
```

```
{
char filename[80],ch;
FILE *fp;

printf("Enter the filename to display contents: ");
gets(filename);

if ((fp = fopen(filename,"r"))==NULL)
{
 printf("Error opening file %s",filename);
}

ch=getc(fp);

while (ch!=EOF) {
putchar(ch); /* print on screen */
ch = getc(fp);
}

fclose(fp);

getch();
}
```

Output:

```
Enter file name to display contents : f1.txt
Hi how r u?
```

- The **EOF** macro, often defined as-1, is the value returned when an input function tries to read past the end of the file

15.2.7 Writing to a file

- We can write into a file character by character using putc() or fputc().
- The prototype of **putc( )** is as follows:

```
int putc(int ch, FILE *fp);
```

where **fp** is the file pointer returned by **fopen( )**, and **ch** is the character to be output.

- For example on writing to a file consider the following section on copying a file which involves reading characters from one file and writing them to another.

15.2.8 Copying a file

The following program involves the concepts of reading from and writing to a file:

```
#include <stdio.h>
#include <conoi.h>
void main( )
```

```
(
FILE *fs, *ft ;
char ch ;
char sfilename[20],dfilename[20];

printf("Enter the source filename to copy contents: ");
gets(sfilename);

fs = fopen ( sfilename, "r" ) ;
if ( fs == NULL )
{
printf ( "Cannot open source file %s",sfilename ) ;
exit( ) ;
)
 printf("Enter the destination filename to copy contents: ");
gets(dfilename);

ft = fopen ( dfilename, "w" ) ;
if ( ft == NULL )
{
printf( "Cannot open target file %s",dfilename ) ;
fclose ( fs ) ;
exit( ) ;
)
while ( 1 )
{
 ch = fgetc ( fs ) ;
 if ( ch == EOF )
 (
 printf("\nFile copy completed succassfully");
 getch();
 exit() ;
 )
 else
 fputc ( ch, ft ) ;
}
fclose ( fs ) ;
fclose ( ft ) ;
}
```

Output:

```
Enter the source file name to copy contents : nik2.txt
Enter the destination file name to copy contents : nik3.txt
File copy completed successfully
```

15.2.9 Seeking the location of a specific record (Random Access)

- The operations that we have seen so far are all based on Sequential access. To move to a particular character we start by opening the file and move from first character to the last character.

- The fseek () can be used to perform read and write at random locations.
- The prototype of **fseek( )** is as follows:

```
int fseek(FILE *fp, long int numbytes, int origin);
```

where, fp is a file pointer returned by a call to fopen( ),
numbytes is the number of bytes from origin, which will become the new current position, and
origin is a macro that can have one of the following three values

Origin	Macro Name
Beginning of file	SEEK_SET
Current position	SEEK_CUR
End of file	SEEK_END

- Hence to seek numbytes from start of the file origin should be **SEEK_SET**.
- To seek from numbytes the current position, we use **SEEK_CUR** as origin, and
- To seek from the end of the file,we use **SEEK_END** as origin

- Consider the following Example:

```
#include <stdio.h>
int mein()
{
    FILE * f;
    f = fopen("myfile.txt", "w");   //Creates a file MYFILE.TXT and
opens it
    fputs("Hello", f);              //Writes the string "Hello" into it
    fseek(f, 6, SEEK_SET);          //Positions the cursor at 6th
position                                           from origin

    fputs(" India", f);             //Writes the string "India"
at                                                 current
position
    fseek(f, 6, SEEK_CUR);          //Positions the cursor at 6th
position                                           from current
position
        fputs(" how r u?", f);      //Writes the string "how r u?" at
                                            current position
        fseek(f, 6, SEEK_END);      //Positions the cursor at 6
places ahead                                       from end
```

```
    fputs(" you are the best!", f); //Writes the string " you are the
best!"
                                        at current position
    fclose(f);                          //Closes the file
    return 0;
}
```

Output

```
MYFILE.TXT
Hello India     how r u?     you are the best!
```

15.3 TEXT FILES AND BINARY FILES

- The operations described in the sections above are based on Text file. In this section we concentrate on working with binary files.
- We have already discussed the differences between text files and binary files in the previous chapter.
- With binary files the operation is almost same as text files.
- To operate on any file we have to open it first. In case of binary files we use a function **open()** and a file handle instead of a file pointer.
- The syntax for opening a file is:

```
handle = open(filename, mode);
```

- where, **open( )** returns an integer value called file handle, it is an integer value assigned

  to the file which is used later to refer to it.

  handle is the integer value returned when the file is opened

  filename is the file we want to access and

  mode is the mode in which we want to access the file.

  If the file open operation is unsuccessful, open() returns (-1)

- Ex:

```
handle = open ( "MYFILE.TXT", O_RDONLY | O_BINARY ) ;
```

- Here, the file "MYFILE.TXT" is opened in read-only mode specified by O_ROONLYand in binary mode specified by O_BINARY.
- The possible file opening modes are:

O_CREAT	Creates a new file for writing It has no effect if file already exists.
_ROONLY	Creates a new file for reading only
O_ROWR	Creates a file for both reading and writing
_WRONLY	Creates a file for writing only

O_APPEND	Opens a file for appending
O_BINARY	Creetes a file in binary mode
O_TEXT	Creates a file in text mode

To use the modes mentioned above we have to add the header file <fcntl.h> to our program.

- When two or more modes are to be used together, they are combined using the bitwise OR operator (|).
- Consider the following example:

```
handle = open ( filename, O_CREAT | O_BINARY |
O_WRONLY, S_IWRITE ) ;
```

If the file we want to eccess does not exist when it is being opened we heve used the O_CREAT to create it, to only write to it we use O_WRONLY and since we want to open the file in binary mode we have used O_BINARY.

- When we use O_CREAT we also need to specify the permission associated with reed/write to our file. It is called permission argument.
- The possible permission arguments are:

S_IWRITE	Writing to the file permitted
S_IREAD	Reading from the file permitted

- **Buffer**- The operations of reading from and writing to files involves the use of a buffer where the data is temporarily stored.
- **Reading from Files**
  - o To read from files we use the read() function.
  - o Prototype for read function is as follows:

```
int read(int handle, void *buf, unsigned len);
```

where **handle** is the file handle obtained eerlier using creat or open

buff points to a buffer that the function reads the bytes into

len number of bytes that the function attempts to read

  - o **Ex.**

```
bytes = read ( handle, buffer, 512 ) ;
```

the above line when executed attempts to read 512 bytes from the file pointed by the hendle into the buffer

- **Writing to files**
  - o To write from files we use the write() function.
  - o Prototype for write function is as follows:

```
int write(int handle, void *buf, unsigned len);
```

where **handle** is the file handle obtained earlier using creat or open

buff points to a buffer that the function writes the bytes from

len number of bytes that the function attempts to write

Ex.

```
write ( handle, buffer, bytes ) ;
```

the above line when executed writes the data assigned to bytes stored in buffer into the file pointed by the handle.

- Consider the following program that read the data from source file and writes it into target file (low level file copy)

```
#include <stdio.h>
#include <conio.h>
#include "fcntl.h"
#include "sys\stat.h"
void main ( )
{
char buffer[ 512 ], source [ 15 ], target [ 15 ] ;
int shandle, thandle, bytes ;
printf ( "\nEnter source file name" ) ;
gets ( source ) ;
shandle = open ( source, O_RDONLY | O_BINARY ) ;
if ( shandle == -1 )
{
printf( "Cannot open file %s",source ) ;
exit( ) ;
}
printf ( "\nEnter target file name" ) ;
gets ( target ) ;
thandle = open ( target, O_CREAT | O_BINARY |
O_WRONLY, S_IWRITE ) ;

if ( shandle == -1 )
{
printf( "Cannot open file %s",target ) ;
close ( shandle ) ;
exit( ) ;
}

while ( 1 )
{
bytes = read ( shandle, buffer, 512 ) ;
if ( bytes > 0 )
write ( thandle, buffer, bytes ) ;
else
break ;
}
printf("\nFile Copy Successful");
close ( shandle ) ;
close ( thandle ) ;
```

```
getch();
}
```

Output

```
Enter source file name:
File1.txt
Enter target file name:
File2.txt
File copy successful
```

15.4 References & Further Reading

1. Let us C – Yeshwent Kanetkar
2. C The Complete Reference – Herbert Schildt
3. Teach yourself C in 21 deys - Peter Aitken and Bradley L. Jones

15.5 Review Questions

1. List and explain in brief the various operations that can be performed on files.
2. Explein with the help of a program how to read from one file and write to another
3. Explain the use of fseek() in random access files
4. Explain how data is read from end written to binary files.

❖❖❖