C Programming

Book · February 2011

Jayaram M.A
Siddaganga Institute of Technology
132 PUBLICATIONS 861 CITATIONS

SESSION 4.
C-Preliminaries

Before we venture in to direct laboratory session on C-programming and implementing, we shall equip ourselves with some very basic features of this language.

Communication between human beings normally comes through a language. We definitely need a language to communicate with computers which they understand. Thus, we can not think of **English** for communicating with computers but it is possible to use **English-like** language to work with computers. C is English like language because there is a glaring similarity in learning English and in learning C.

Before we develop a meaningful paragraph in English, we need alphabets; alphabets are to be grouped together to form words, words to be assembled in to sentences and finally when sentences are arranged expeditiously there emerges the Paragraph!. Similarly, before we develop a workable program in C, we need to meddle with alphabets (this includes digits and special symbols), alphabets grouped together can generate constants, variables and keywords, assembling these words will lead to formation of instructions (sentences) and finally group of instructions culminate in a C program.

C-Alphabets

It is better we use the term " Character set" in place of alphabets. Because, C-language has more than alphabets in its store. Table 1.1 presents different characters available in C.

Table 1.1 C- Character set

Alphabets	A,B,C…………….Z A,b,c ……………..z
Decimals(digits)	0,1,2…… 9
Special Characters	~ ` ! @ # $ % ^ & * ( ) _ - + = | \ { } [ ] : ; " ' < , > . ? /

Constants, Variables and Key words

As explained earlier, the alphabets, digits and special characters can be tagged to form constants, variables and keywords. We shall now see these word like items one by one.

Constants

As the name itself would imply, constants are quantities that do not change. For example in the equation $2x^2+5x+10=0$,the numerals 2,5, and 10 are constants. These constants can be divided in to two major categories. Namely,

i. Primary Constants.
ii. Secondary Constants.

Primary constants would encompass the following;
- Integer Constant
- Real Constant.
- String or Character Constant.

While secondary constants would include the following;
- Array
- Pointer
- Structure
- Union
- Enumerated Data etc.

Integer Constants

A whole number is an integer constant. Integer constants do not have a decimal point. Further, an **integer constant**;
- **Must** have at least one digit.
- **May** be positive or negative. In the absence of a sign it is taken as positive.
- **Can** not have blanks or commas in between the digits.

On a 16-bit computer (see points to ponder) the allowable range for integer constants is –32768 to +32767.

Examples: 420
- 111
-999
2222

Real Constants

Constants with decimal point (the decimal point is called period) are real constants. They are also called as **Floating point constants** or **Numeric Constants**. Further, A real constant;
- **Must** have at least one digit.
- **Must** have a decimal point.
- **May** be positive or negative and in the absence of sign taken as positive.
- **Must** not contain blanks or commas in between digits.
- **May** be represented in exponential form, if the value is too high or too low.

The range of values of real constants is –3.4 e 38 to 3.4e38.

Examples:
420.56
-111.11
-999.9999
+3.1 e-5
-0.1e+6

Character Constants and String Constants

Any character enclosed with in single quotes (')is called character constant. And any character or characters placed with in double quotes(") are known as String constants. A **character constant;**

- **May** be a single alphabet, single digit or single special character placed with in single quotes (both quotes are oriented to left).
- **Has** a maximum length of 1 character.

Examples:' N' ,'Z','?', '$'

A string constant;

- **Can** have one or more than one characters bounded by double quotes.
- **Can** have nothing in between quotes (it is called a Null string).
- **Is** also called as an array of character constants, whose last character is null character represented by \0. This character is placed at the end of the string by the compiler.

```
Examples:      " "
              " I/II Semester"
               "C"
              "10"
```

Variables

Variables are quantities that vary during program execution. Basically, variable is nothing other than a name given to the memory location where particular constant is stored. Thus integer variable name will refer integer constant, real variable name refer real constant and character variable refer to character constant. Probably this is the reason why very often we refer variables as **Identifiers**.

A variable;

- **Is** a naming word for location of a constant.
- **Can** comprise 1 to 8 alphabets, digits or underscores.
- **Can** comprise up to 40 characters (in some compilers). However such long names would be more a handy cap than handy.
- **Must** have its first character to be an alphabet.
- **Can** not have commas or blanks in between.
- **Can** not have any special character other than under score.

Examples: S_int, roots,mark,grade,rs

The compiler should be made known as to what type of constant the variable is referring to. C-language dictates that, the type of variable must be declared before being used (at the beginning of the program). Following examples will make it clear.

```
Examples:   int mark,sl-no;
             float int-rate,percent,root;
```

```
            char code,grade;
```

The semicolons are placed as a rule at the end of each declaration. This is a feature of C language.

It is worth giving meaningful names to variables. A variable with a fitting name will give a notion of what value it holds.

Instructions in C

As it was mentioned earlier, instructions in C are akin to sentences in English. Instructions are generated by proper use of constants, variables, characters and keywords (see points to ponder). Four types of instructions are possible in C. They are explained here with examples.

Type declaration

This instruction is to be made to give a clue to the compiler as to what type of variables are used in a program and to set aside a particular amount of memory. It is a must that, all the variables used in a program are to be declared at the beginning of the program. Some examples are listed in table 1.2.

Table 1.2 Type declaration

Type declaration	Explanation
int slno,I,j=1;	slno, I and j are integers, j is initialised to 1.
float root,gr-sal,pi=3.143;	root,gr-sal and pi are floats, pi is set to 3.143
char code;	code is a character variable.
double gamma;	Gamma is double precisioned

Input and output instructions

These instructions perform the task of supplying data to a program and taking out the results from it. This job will be rendered by functions scanf and printf respectively. The **scanf** and **printf** are built in functions .

Examples:

```
scanf("%d%d%d",&mark1,&mark2,&mark3);
printf("The percentage mark is=%f",per);
```

scanf is inputting three variables ie., mark1,mark2,mark3 through key board.
printf is outputting the variable **per** on the screen.
The input and output instructions will be explained at length in later sessions.

Arithmetic Instructions or Expressions

These instructions are written to perform arithmatic operations between constants and variables. The syntax(grammer) of these instructions would involve an equation like pattern with variable name on left hand side of = , variables and constants on the right hand side of = sign. Right hand side of = sign may be studded with arithmatic operators like +,-,*,/ and %.

Example:

```
int sum,a=7,b=80,prod,rem;
sum =a+b;
prod=a*b;
rem=b%a;
div=b/a;
```

Here, sum,a,b,prod,rem and div are integer variables
7 and 80 are integer constants
= is called assignment operator.
+ is addition operator, * is for multiplication,/ for division, - for subtraction and % is modulus operator which returns the reminder to variable rem after division.

Variables and constants are operands, they are manipulated (operated) by operators.

Further **arithmetic operation;**

- **Of** type a+b= c is not correct ,while c=a+b is correct.
- **Can** be performed on characters also

  Ex:
  ```
  char x,y;
  int z;
  x='a';
  y='b';
  z=x+y;
  ```
  However, operations are not done directly on characters but done on their respective ASCII values.

- **Can** not be done presuming that an operator is present.

  Ex : x =a.b.c(x+y)➔ normal arithmatic statement.
  x = a*b*c*(x+y);➔ C statement.
- **Can** not be done for exponentiation by using operators(this facility is avilable in other highlevel languages like FORTRAN).

  Ex: x = y**2 ; ➔ ** can not be used.
  x = y^3; ➔ ^ can not be used.
- **Between** an integer and integer would always produce an integer.
- **Between** a real and real always produce a result which is of real type.
- **Between** an integer and real always produce a result which is of real type.

Table 1.3 would clarify these types of operations.

Table 1.3

Operation	Result	Operation	Result
7/3	2	4/8	0
7.0/2	3.5	4.0/8	0.5
7/2.0	3.5	4/8.0	0.5
7.0/2.0	3.5	4.0/8.0	0.5

Control Instructions

These instructions are written to dictate the order in which various instructions (statements) in a program are to be executed by the computer. Four types of control instructions are available in C. They are;

I. Sequential control ➔ Execution takes place line after line.
II. Decision control ➔ Skips Certain part of the program.
III. Loop control ➔ To do repeated operations (repetition of statements).
IV. Case control ➔ Instructions are executed based on a parameter.

All the above instructions are to be used in the subsequent sessions.

Points to Rehearse

➔ Following are the invalid names of variables because;
What? ………. Question mark
420 ………. Numeral
si int ……….. Space in between
int ……….. Key word.

➔ Following are invalid statements because;
float= (3.142*180)/60 ………… float is a key word.
name= 'ram' ; …………… ram is bounded by single quote.
1m1 =(b*d**3)/12 …….. ** can not be used.

➔ If a.b.c(a+b) is algebraic expression , then its equivalent C expression is
a*b*c*(a+b)

➔ If **i** is an integer variable and if **r** is a real variable,

i=	Will produce	r=	Will produce
3/8	0	3/8	0.0
3/8.0	0	3./8	0.375
8/3	2	8/3	2.0

➔ If **i** is an integer and **r** is real variable (float) and if we write i=8.5 and r=85, then, what is being stored in **i** is 8 and in **r** is 85.000000.

Points to Ponder

The basic memory unit is **bit**. Following covertions could come in handy.

8 bits	**1 byte**
1024 bytes	**1 Kilo byte(Kb)**
1024 Kb	**1 Mega bite(Mb)**
1024 Mb	**1 Giga bite(Gb)**
1024 Gb	**1 Tera byte(Tb)**

Besides, Nibble is used to represent Four bits.

Microprocessor (CPU) is the heart of all PCs. It is this microprocessor which carries out variety of computations, numeric comparisons and data transfers in response to programs stored in memory. The data **flow** between CPU and memory cells happens through **databus.** Each wire in this bus carries a bit of data at a time (a bit is either zero or one in the form of an electric pulse). If the data bus has 16 wires then 16 bits or 2 bytes of data can flow in at a time.

T able 1.4 gives the datatypes, their numerical range and the digits of precision (in case of real numbers only) and the bytes of memory occupied in M S DOS environment. However, the range depends on memory capacity of Computer.

Table 1.4 Data types and ranges

Keyword	**Numerical range**		**Digits of Precision**	**Bytes of memory**
	Min	**Max**		
char	-128	127	Not applicable	1
int	-32768	32767	-----do---------	2
long int	-2,147,483,648	2147483647	------do--------	4
float	$3.4x10^{-38}$	$3.4x10^{38}$	7	4
double	$1.7x10^{-308}$	$1.7x10^{308}$	15	8
long double	$3.4x10^{-4932}$	$1.1x10^{4932}$	19	10

C Language was developed by Dennis Ritchies at AT & T's Bell laboratories of USA in 1972. It evolved as an improvement over **CPL**(**C**ombined **P**rogramming

Language),BCPL(**B**asic Combined **P**rogramming **L**anguage) and B(by Ken Thompson).

C is a middle level language. Because it stands between high level languages like FORTRAN, BASIC,PASCAL and low level languages like Assembly language and Machine language. C inherits both the capabilities of turning to high and low.

C can be used on M.S.DOS, UNIX and XENIX operating systems.

There are 32 key words available in C. Following table gives these key words enlisted in alphabetical order. In subsequent sessions will use them if need be. Care should be taken to see that the variable names in a program would not be any among the key words.

Table 1.5 Key words in C

auto	**double**	**if**	**static**
break	**else**	**int**	**struct**
case	**enum**	**long**	**switch**
char	**extern**	**near**	**type def**
const	**float**	**registered**	**union**
continue	**far**	**return**	**unsigned**
default	**for**	**short**	**void**
do	**goto**	**signed**	**while**

C language sentences, labels, arrays etc.. are separated by special characters, which are called **delimiters**. These delimiters have meaning and significance. Their list is presented in table 1.5. Their utility can be understood in subsequent sessions.

Table 1.6 C- delimiters

Delimiter	**Symbol**	**Explanation**
Hash	#	Pre-processor directive to be written while including files.
Comma	,	Used in variable list for separation.
Semicolon	;	Used at the end of every C-statement.
Colon	:	Used at the end of labels when goto is used.
Parenthesis	( )	Used in C expressions.
Square brackets	[ ]	Used in Arrays.
Curly braces	{ }	Used to block C-instructions.

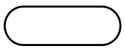
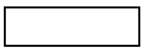
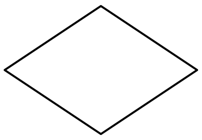
C has following library functions (Arithmetic functions). They are listed in table 1.7.
Table 1.7 Library functions(Arithmetic functions).

Function	Utility
Abs	Returns absolute value of an integer.
Cos	Calculates Cosine of an angle.
Cosh	Calculates hyperbolic cosine.
exp	Raises the exponential e to the xth power.
fabs	Finds the absolute value of real number.
fmod	Finds floating point remainder.
hypot	Calculates hypotenuse of right angled triangle.
log	Calculates natural logarithm.
log10	Calculates base 10 logarithm.
modf	Breaks down argument in to integer and fractional parts.
pow	Calculates a value raised to a power.
sin	Calculates sine of an angle.
sinh	Calculates hyperbolic sine.
sqrt	Finds square root of a number.
tan	Calculates tangent of an angle.
tanh	Calculates hyper bolic tangent.

Finally, we ponder a bit on **flow chart.** A flow chart is a pictorial representation of the method (logic) used to solve a particular program. It sketches the sequence of steps that must be taken (from beginning to end) to arrive at the solution of the problem and it turns out to be a valuable tool. Understanding of structured programming concepts becomes very easy if we just visually scan through the complete flow chart of a program very carefully. A flow chart basically involves two components.

- The physical processing of activities in a program such as reading data, writing data, looping, starting, ending etc..
- Mental activities involving decision.

Eventually, program flow charts must be translated in to a particular highlevel language. Figure 4.1 illustrates a flow chart. The language displayed is obviously not a programming language that can be processed by a computer. It is an example of **pseudo code** which is an informal design language somewhat similar to C. Pseudo code does not have to be extremely precise and grammatical considerations are of secondary importance. He various symbols used in developing a flow chart is explained in the table. A sample flow chart is sketched in figure 4.1.

Oval		: Start or stop of program.
Rectangle		: Process block.
Diamond		: Decision block

Parallelogram ▱ : Input /Output statements.

Arrow ⇨ : Flow of logic.

Circle ○ : Connector.

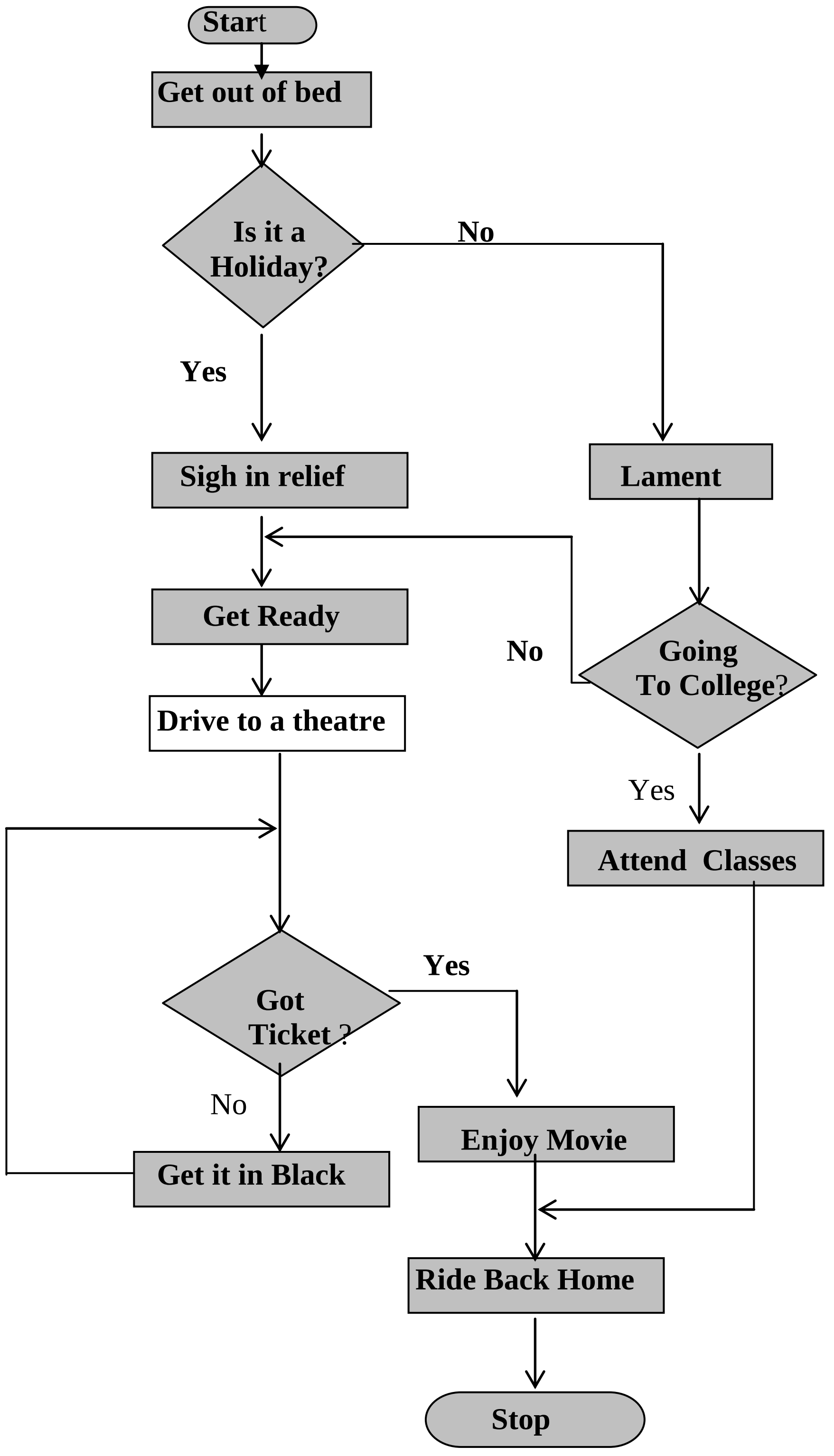


Figure 4.1 Flow Charting

SESSION 5

Few Simple C-Programs

Having gone through session 4,where we gained some knowledge about types of variables, constants and keywords, we shall now try to develop few simple programs in this session.

First C-Program

This program is written to calculate volume, surface area of a box.

```
/* Pre-processor directives */
#include<stdio.h>
/* main function starts now*/
main()
{ /* list of variables */
  float l,b,h,vol,s_area;
  /* variables assigned to constant values*/
  l=22.5;
  b=12.2;
  h=6.5;
  vol=l*b*h;
  s_area=2*(l*b+b*h+h*l);
  /* output of results*/
  printf("The Volume is:%f\n",vol);
  printf("The Surface area is:%f\n",s_area);
}
```

Output:

```
The Volume is: 1784.250000
The Surface area is: 1000.099976
```

Explanation:

- **T**he Very first line is comment statement. Comments are to be enclosed with in /* */ . Any number of comments can be used at any place in the program. A comment can be split in to different lines as in,

```
/*  This program calculates Volume and
      Surface area of a box with known
```
Side dimensions .*/

Comments are a must for programs because they figure out what a program does. With appropriate comments at confusing positions of the program, the program turns out to be **User Friendly**.

- The next line we see is a **pre-processor directive.** Pre-processor directive will process our program before it is passed to the compiler. This directive begin with a **hash** (#) symbol before main. We can certainly write C programs with out knowing pre-processor or its facilities. The # include directive causes one file to be included in other. The general command for file inclusion look like this,

  #include"file name"
  or #include<file name>
- It is common for the files which are to be included to have a **.h** extension. This extension stands for **header file**. The **stdio** stands for standard input output, this file when included facilitates input and output of the data through keyboard and screen respectively. While writing this directive , the # symbol must be in **column 1** and there should be **no space between the symbol and include.**

- After the pre-processor directive what we see is main(). Any C program is nothing but combination of functions and main() is one such function. It is from this function the operating system will start functioning. Each and every C program must have one and only one main() function. Empty parenthesis after main are a must.

- We find series of executable statements one below the other. An executable statement is an expression followed by a semicolon or a control construct(explained later). All these statements are bounded by curly braces. These braces herald the **beginning and ending** of the program. The first executable statement is declaration statement. In this statement variables d,b,h,vol,s_area are declared to be float type.

- In the next lines, the variables l,b and d are **assigned** with constant values 22.5, 12.2 and 6.5 respectively.

- Next two lines are arithmetic expressions, one to calculate the volume of the box while the other to find surface area of the box. We find **operands l, b** and **d** . and **operators** + and * in these expressions.

- Finally the answers (outputs) are drawn out by **printf** statements. **Printf** is a function which facilitates to display the output on the screen. In this program it displays the contents of the variables vol and s_area. The general syntax(form or grammar) of printf is:

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

- Format string may include brief explanation or a statement, a caption , a cautionary message along with the field specification of the variable to be displayed on the screen. Field specification could be;

  %f for printing real(float) value
  %d for printing integer value.
  % c for printing character value.
  %ld for long integer.

%lf for long float.
%u for unsigned integer.

Along with this format string we find a character **\n** this is, **newline** character . This character makes the cursor (also printer) to move to the beginning of the next line.

- **F**inally, we see a **closing brace** signifying the **closure** of the program.

Example 2.

In this program , the marks obtained by a student in five different subjects are taken through the keyboard, and aggregate mark and percentage mark are calculated.

```
#include<stdio.h>

main()

{ /* List of variables to be used in the program*/
int mark1,mark2,mark3,mark4,mark5,tot_mark;
float percent;
/* the marks are inputs*/
scanf("%d%d%d%d%d",&mark1,&mark2,&mark3,&mark4,&mark5);
tot_mark=(mark1+mark2+mark3+mark4+mark5);
percent=(tot_mark)/500. *100;
/* output statements*/
printf("Aggregate marks:%d\n",tot_mark);
printf("Percentage of marks:%f",percent);
}
```

Output :

```
38 46 67 89 62
Aggregate marks: 302
Aggregate marks:60.400002
```

Explanation:

- ➔ There are few additional ornaments for this program.
- ➔ Unlike the first program, here in this program, the values of variables are not assigned instead , the values are read through the key board.
- ➔ The data can be taken in through the key board by the use of **scanf** function. The rules governing **scanf** function is similar to those governing printf function. The general syntax is;

scanf ("control string" , list of variables);

Control string contains the conversion specifications(field specification) for variables. If more than one variable are to be read through the key board , corresponding number of conversion specifiers must be written and their number must match with number of variables. Further, the variables should accompany an **ampersand(&)** operator. This is called **address operator**.

Thus the meaning of the statement
scanf("%d%d%d%d%d",&mark1,&mark2,&mark3,&mark4,&mark5);
Is , read the integer type values one after the other and store them in the addresses named mark1,mark2,mark3,mark4 and mark5 respectively.

➔ The next two statements are arithmetic statements written to calculate total mark and percentage.
➔ Two separate statements are written to display the total mark and percentage.

Example 3.

In this program , we shall input two numbers in to the memory location named a and b and let us exchange their contents(values) .

```
#include<stdio.h>
main()
{ float a,b,temp;
/* prompt for entering two values*/
printf("Enter two numbers first to a , then to b\n");
/*values are to be read through key board*/
scanf("%f%f",&a,&b);
/*exchange(swap) is made in next lines*/
temp=a; /* contents of a goes to temp, a is freed/
a=b;    /* contents of b goes to a, b is freed/
b=temp; /* contents of temp goes to b*/

/* printing the results*/
printf(" Now a contains :%0.2f\n",a);
printf(" While b contains: %0.2f\n",b);
}
```

Output:

```
Enter two numbers first to a, then to b
23.45 45.67
Now a contains:45.67
While b contains:23.45
```

Explanation

➔ This program is no different in its style, but for a printf statement before scanf statement. This statement is called **prompt** statement. An **interactive program** should contain **prompt** statement before each and every **scanf** statement to alert the user regarding the type of entry to be made. This makes the program **User Friendly**.

➔ This program is of some interest because the variables **a** and **b** are exchanged in their values (swapping). The situation here is similar to exchanging contents of two cups , one of them containing Coffee and Tea in other. To avoid mixing, we need an empty cup and this empty cup helps in temporarily storing either Coffee or Tea. In the same token, we need temporary variable **temp** to hold the value contained in a. By doing this the value in **a** is not lost and **a** becomes **free** to hold the value contained in **b**. Then, **b** takes the value stored in temp(that is value of **a**).

Example 4.

This program reads a four digit number and sums up its digits.

```
#include<stdio.h>
main()
{ int sum,digit,num,dupli;
/*prompt to enter a four digit number*/
printf("Enter a four digit number:");
scanf("%d",&num);
/* this number is duplicated for further use*/
dupli=num;
sum=0;   /* a variable for accumulating the sum of
digits*/
digit=num%10; /* remainder will be returned to digit*/
sum=sum+digit;
num=num/10;
digit=num%10;
sum=sum+digit;
num=num/10;
digit=num%10;
sum=sum+digit;
digit=num/10;
sum=sum+digit;
printf("/nThe sum of digits of the number %dis\n",dupli);
printf("%d",sum);
}
```

Output:

```
Enter a four digit number: 2537
The sum of digits of the number2537 is
17
```

Explanation:

➔ All the variables used in the program are declared to be of integer type.
➔ We see a prompt which alerts the user to enter a four digit number.
➔ The number is taken in and it is duplicated into another variable **dupli.** This is because the variable **num** is operated repeatedly to detach individual digits from it for their addition. Thus duplication of this number would avoid missing the original number.
➔ A variable called **sum** is initialised to zero for accumulation of summed up value. Therefore **sum** is **accumulator**.

➔ We find series of arithmetic statements thereafter, these lines are explained with a specific example. Explanation is to be followed along with program statements line by line.

If the num is 2537,
digit = num%10 returns the remainder 7 to digit.
sum becomes 7 (0+7).
num=num/10 gives 253 (because of integer division, fraction gets truncated)
digit=num%10 gives 3
sum becomes 10 (7+3).
num=num/10 gives 25.
digit=num%10 returns 5 to digit.
sum boosts to 15 (10+5).
digit=num%10 yields 2.
Finally sum will accrue to 17 (15+2)

➔The program will conjure up in printing the sum of the digits.

We will realise in later sessions that while adding the digits of a four digit number, we have made a roundabout program. Thus if it were to be a ten digit number the program would have gone verbose(too many statements) and inefficient. Above all, it would have taken hell out of us!

Points to rehearse

➔ It is essential that , each new C-instruction has to be written on a separate line.
➔ Usually all C statements are entered in small case letters. However, keywords, built in functions must be in small case letters.
➔ The program written below is wrong because;

```
 main()
{
   printf("Enter a number:");
   scanf("%d",&num);
   printf("The number entered is\n",num)
 }
```

- The header file stdio.h is not included(However it is optional for some compilers).
- Variable num is not declared.
- In the printf statement there is no conversion specifier for num and there is no semicolon at the end.

➔ In the interest of users of a program ,it is always better to use prompts before each and every scanf statement.

Points to Ponder

Almost every program in C will have **printf** and **scanf** functions. A printf function have a message ,suggestion, heading, explanation or a caution. In some input output statements these functions will have conversion specifications also. Following table presents various ways of writing printf statements with their outputs.

If the value of the integer variable **x** is **421**, then;

Statement	**How it prints**	**Remarks**
printf("x value:%d\n",x);	x value:421	Normal Way.
printf("x value:%8d\n",x);	x value: 421	Right justified in a field of 8
printf("x value:%-8d\n",x);	x value:421	Left justified in a field of 8

If the value of the real variable y is 420.420 then;

Statement	**How it prints**	**Remarks**
Printf("y value:%f\n",y);	y value:420.420000	Prints with six decimal digits
Printf("y value:%e\n",y);	y value:4.204200E+2	Prints with e format
Printf("y value:%0.2f\n",y);	y value:420.42	Prints with two decimal digits.

Some characters used with printf do not appear themselves on screen but they perform special functions other than producing text. The function may be back spacing, returning to beginning of the line, moving over to new line, tabbing or beeping . Listed down are some commonly used characters(they are called **escape sequences**).

Escape sequence	**What they do**
\n	Takes cursor to beginning of next line.
\t	Makes cursor to jump to tab space.
\b	Cursor backspaces by one column.
\r	Carriage return.
\f	Form feed .
\007 or \a	Makes beep sound

Most of the rules governing **scanf** function are similar to that of **printf** function. However, a **scanf** function should contain only conversion specifications and not messages. There should be one to one matching between conversion specifiers and the variables. All the variables must be preceded by **&** (address of) operator except strings. Any blank space,tabs , newline characters included in the quotes are generally ignored. The table below brings out some features of scanf function.

Scanf function	**What it does**
scanf("%d%d",&x,&y);	Takes two integer variables x and y.
scanf("%dand%d",&x,&y);	Takes two integers separated by word **and**.
scanf("%d%c%d",&x,&c,&y);	Takes two integers separated by a character.
scanf("%d%*c%d",&x,&y);	Takes two integers separated by a character.

When an asterisk(*)follows the % in a scanf conversion specification, it only means that the scanf should skip the value in the input, without reading it into any variable.

Try yourself

1. The distance between two cities is to be input(in km). Develop a program which prints this distance in meters and centimeters.
2. Read a four digit number and write a program to reverse it.

3. Read temperature values in Fahrenheit and convert it into centigrade.

SESSION 6.

Using Decision Control Structures

As human beings, we need to decide on many occasions. Like, while selecting type of dress from variety of dresses, selecting a particular branch of engineering among host of branches so on and so forth. C –language too has some decision making **constructs** which are helpful in arriving at a decision in programming environment. These constructs are easy to use and when used, may cause a one-time jump to a different part of the program, or may lead to appropriate processing to be done on the data.

In session 5, the programs were sequential in nature, that is, all the steps were executed one after the other, in the same order in which they appear in the program. To make this to happen, programmer need not have to do any thing at all. By default(usually) , the instructions in a program are processed by the compiler sequentially. But , in the event of a condition ,the execution can not happen sequentially instead execution jumps between lines depending on truth or falsity of the condition.

Four different control structures are supported by C, they are explained here.

i. **Unconditional control.**
ii. **Bi-directional control**
iii. **Loop control**
iv. **Multi-directional control.**

Let us deal with first two controls in this session. We shall see the next two in subsequent sessions. But before harping, it is required that we should glean through **relational** and **logical** operators.

Relational operators

These operators are essentially symbols used to test relationship between two variables or between a variable and a constant. There are six relational operators in C. Here is how they look and what they mean.

Relational operator	It's Meaning
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
<=	Less than or equal to
>=	Greater than or equal to

Logical operators

Again, these are symbols, used to combine expressions containing relational operators. There are three logical operators in C.

Logical operator	Symbol	What it does
AND	&&	Combines the conditions(both must be true).
OR	||	Negates one of the conditions (any one to be true).
NOT	!	Reverses the truth value of the expression

Unconditional Control

In certain situations, the flow of control may be transferred to another part of the program with out testing for any condition. In such circumstances, **goto** is used. **goto** should be followed by a label name. However, use of this type of control is ill advised as it would make the program confusing and **unstructured**. The example program would reveal how **goto** works.

Example 1:
```
#include<stdio.h>
main()
{
   int some_no;
printf("Enter the number 420\n");
scanf("%d",&some_no);
if(some_no==420)
goto thank;
printf("It is not the number I expected\n");
goto end;
thank:
printf("You are real 420\n");
end:
printf("\n");
}
```

output
```
Enter the number 420
421
It is not the number I expected
```

Explanation

- Program starts with a prompt asking the user to input the number 420.
- The number keyed in (taken in) is stored in the variable some_no.
- A verification is made to ascertain whether the number is 420, if the number is 420, the control will move to the label **thank**. If the number is not 420, a message is printed and then the control will be passed to the label **end**. After the label **end** we find a printf statement which prints nothing.
- In this example the line **goto end:** is typically unconditional.

Bi-directional Controls

As mentioned earlier, a decision control instruction can be implemented in C using the **if** statement and the **if-else** statement.

if Statement

The general **if** statement looks like this;

```
  if (the  condition is true)
  Execute this statement;
```

It may be possible that in a program, we may need more than one statement to be executed if the condition following **if** is satisfied**.** Multiple statements are to be enclosed in a pair of **curly braces** to make them executable under the ambit of **if**. The **if** format under such circumstance is;

```
 if (This condition is true)
 {  Execute statement 1;
    Execute statement 2;
    Execute statement 3;
     -      -     -     -
     -      -     -     -
  }
```

if- else statement

if statements will do their job of executing a statement or statements when the condition following **if** is true. It will not bother if the accompanying condition becomes false. To facilitate the execution of one group of statements if the condition is true and to execute another group of statements if the condition is false, C provides **if-else** construct.

The general form is;

```
If(this condition is true)
 Execute this statement;
else
 Execute this statement;
```

If more number of statements are to executed under each clause(**if** clause and **else** clause), the statements should be bounded by pair of braces.

Nested if –else

It is not at all wrong if we write an entire **if-else** construct within the body of **if** clause or with in the body of **else** clause. Then it becomes nesting of **ifs.**
The syntax is;

```
if(condition)
 {  do this;
    and this;
  }
 else
     { if(condition)
         do this;
       else
            {  do this;
              do this;
             }

      }
```

We shall go through examples in this session to understand various formats of **if** constructs.

Example 2: Let us write a program to find the biggest among three numbers.

```
#include<stdio.h>
main()
{
 float a,b,c;
 printf("Please enter three real numbers:");
 scanf("%f%f%f",&a,&b,&c);
 printf("\n\n Biggest among %f,%f and%0.2f is:",a,b,c);
 /* conditions are tested now*/
 if(a>b&&a>c)
 printf("%f\n",a);
 if(b>a&&b>c)
 printf("%f\n",b);
 if(c>a&&c>b)
 printf("%f\n",c);
 getch();
 }
```

Output :

```
Please enter three real numbers:22.0 34.0 89.0

Biggest among 22.000000,34.000000 and 89.000000 is:
```

```
89.000000
```

Explanation

- Three numbers of float type are keyed in, on the suggestion from the prompt.
- Each number is compared with the other two by using independent **if** statements.
- Whichever is greater than the other two will be printed as the biggest number.
- Finally, in earlier programs, after correct execution we used to see the output(answers) on the screen by pressing hot keys alt and F_5 , we can as well see the answers without using the hotkey. For this, we need to use getch() function, this function takes any character from key board, that is unless we press a key the answers will not vanish from the screen.

Example 3:Let us write a program to check whether a given number is odd or even.

Logic: If a number is even, it is divisible by 2, and there will be no remainder left, if the number is odd, it is not divisible by 2 and it will have a remainder.

```
#include<stdio.h>
main()
{ int num;
printf("Enter a number to check whether it is odd or even\n");
scanf("%d",&num);
if(num%2==0)
printf("The number %dis even\n",num);
else
printf("The number %d is odd\n",num);
getch();
}
```

Output:

```
Enter a number to check whether it is odd or even
23
```

The number is odd

Explanation:

- This program reads a number of integer type on suggestion from prompt.
- Modulus operator is used to find the remainder. When the number (num) is divided by 2, if the remainder is zero, then it will be treated as even number otherwise it is treated as odd.
- On checking appropriate message is displayed by printf statements.
- This program uses if-else structure.

Example 4.

We shall write a program that reads the length of three sides of a Triangle, checks whether the triangle can be formed out of these sides and then types the type of triangle.

```
#include<stdio.h>

main()
{ /* declaring the sides to be of float type */
float a,b,c;
printf("Enter the three sides of the triangle==>\n");
scanf("%f%f%f",&a,&b,&c);
/* checking whether it forms a triangle */
if(a>=(b+c)||b>=(a+c)||c>=(a+b))
printf("\nTriangle can not be formed with these
sides\n");
else if(a==b&&b==c)
printf("\n The triangle is equilateral\n");
else if(a==b||b==c||c==a)
printf("The triangle is isosceles\n");
else  if(a*a+b*b==c*c||b*b+c*c==a*a||c*c+a*a==b*b)
printf("The triangle is right-angled");
else
printf("The triangle is scalene\n");
getch();
}
```

Output:
First run;

```
Enter the three sides of the triangle==>
3
3
5
The triangle isosceles
```

Second run;

```
Enter the three sides of the triangle==>
12
4
4
Triangle can not be formed with these sides
```

Explanation

- Three sides of (float type)the triangle are taken in ,on suggestion by the prompt.
- The formation of the triangle is verified based on the condition of the length of the third side. That is the third side must be lesser than the sum of the other two side lengths.
- Based on the equality or inequality of side lengths the triangle is categorised into equilateral, isosceles, right angled or scalene. This is achieved through **nested if-else** construct.

Example 5:In this example we shall write a program to find roots of a quadratic equation. For non zero coefficients , the program will print all types of roots.

Logic: The nature of the roots of a quadratic equation is dependent on the value of the discriminant. That is if the discrimenent is positive we get real and distinct roots, if it is negative the roots will be complex and if discriminent is zero the roots will be equal.

Discriminent is given by disc = $b^2 - 4ac$

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

main()
{ clrscr();
  float a,b,c,disc,root1,root2,root,rp,imp;
  printf("Enter the coefficients of quadratic
equation\n");
  scanf("%f%f%f",&a,&b,&c);
/* Checking for non zero roots */
  if(a==0||b==0||c==0)
  printf("Equation not possible\n");
  else
 { disc=b*b-4*a*c;
   /* If discriminent is positive print real roots*/
     if(disc>0)
     { root1=(-b+ sqrt(disc))/(2*a);
       root2=(-b-sqrt(disc))/(2*a);
       printf("Roots are real and distinct\n");
       printf("root 1 is= %f\n",root1);
       printf("root 2 is=%f\n",root2); }
   else if(disc<0)
     {  rp=-b/(2*a);
     imp =sqrt(fabs(disc));
     printf("First complex root is %f + i %f\n",rp,imp);
     printf("Second complex root is %f - i
%f\n",rp,imp);}
      else
     { printf("The roots are equal\n");
       root = -b/(2*a);
       printf("The roots are=%f and %f\n",root,root);}
     }
      getch();
     }
```

Output:

```
First run:
Enter the coefficients of quadratic equation
1
-4
4
The roots are equal
The roots are 2.000000 and 2.000000
```

Second run:

```
Enter the coefficients of quadratic equation
1
-5
6
The roots are real and distinct
Root 1 is =3.000000
Root 2 is =2.000000
```

Third run:

```
Enter the roots of quadratic equation
5
8
9
First complex root is =-.800000+i 10.770329
Second complex root is=.800000-i 10.770329
```

Explanation

- ❖ The include file list has **math.h** , this file facilitates use of **sqrt** function in the program.
- ❖ Three float type variables a b and c are read for coefficients of quadratic equation.
- ❖ The coefficients are tested for their non zero value.
- ❖ If the discreminent is positive then the real roots are printed.
- ❖ If the discreminent is negative then the complex roots are printed. To print the complex roots the absolute value of **disc** is placed under the square root. It may be recalled that, **fabs** returns the absolute value of a float type variable.

Example 6. This program identifies type of the character entered through key board.

Hint: All the characters on the key board have their respective ASCII value. When a key is hit , the character taken to memory is converted to its integer value and depending on the value, the character type is identified. The table gives the range of ASCII values for characters

.

A to Z	65-90
a to z	97-122
0-9	48-57
Special Characters	0-47,58-64,91-96,123-127

```
#include<stdio.h>
main()
{
  char c;
  int con;
  /* prompt for entering a character*/

  printf("Hit any character from key board\n");
  scanf("%c",&c);
  /* character is stored in integer type variable
     for converting it to its ASCII value */
  con=c;
  if(con>=65&&con<=90)
  printf("You have typed a capital letter\n");
  else if(con>=97&&con<=122)
  printf("You have typed a small letter\n");
  else if(con>=48&&con<=57)
  printf("You have typed a digit between 0 and 9\n");
  else
  printf("You have typed a special character\n");
  printf("Thank you!");
  getch();
  }
```

Out put:

```
Hit any Character from key Board
A
You have typed a capital letter
```

Explanation

- On the suggestion from the prompt, any key is pressed.
- The character keyed in is scanned by **scanf** function.
- The character is converted to respective integer value(by statement con=c).
- Then the value is tested in the known ranges for identification.

Points to Rehearse

- The conditions in a program may be tackled by **if** and **if- else** constructs.
- Group of statements after the **if** and not including the **else** is called **if block** and the statements related to **else** is **else block** . The content of these blocks(statements) must be delimited by curly braces.
- It is always better that we place **else** exactly below **if**. This type of indenting will always enable clear understanding.
- Care need to be exercised to match the **ifs** with corresponding **elses**. Corresponding braces should also be matched carefully.

- Relational operators can be used in variety of ways in a program, however the clarity of condition is more important. The examples will make this point clear.

Ex 1. if(n<=5)
May be written as,
if(n>5)
Or
if (!(n<=5))

Ex 2. if (n!=0)
May be succinctly written as
if(n)

The relational operators must be carefully written, otherwise the answers could be disastrous. Look at this example;

```
#include<stdio.h>
main()
{ int num,rem;
printf("Please enter a number\n");
scanf("%d",&num);
rem=num%2;
if(rem=0)
printf("You entered an even number\n");
else
printf("You entered an odd number\n");
}
```

The output for this program is;
First run:

```
Please enter a number
200
You entered an odd number
```

Second run:

```
Please enter a number
13
You entered an even number
```

This is ridiculous! . This happened because, the assignment operator = is used along with **if** instead of relational operator **==** . As a result, the condition got reduced to **if(rem**). Irrespective of what **rem** has in its store , the truth is always a non-zero value of **rem** and falsity is always zero.

➢ The programs enlisted are wrong because;

Example 1

```
#include<stdio.h>
main()
   {
      int a= 10;
      if(a<=2) then
      printf("%d\n",a);
   }
   ➔ word then can not be used.
```

Example 2

```
#include<stdio.h>
main()
{
   int x,y;
   if(x=y);
   printf(" Both the variables are equal\n");
}

➔ scanf is not used to read the variables and assignment
operator = is used instead of relational operator ==.
```

Points to Ponder

As we have understood few elementary aspects of C-language with regard to its vocabulary(not fully!),decision making constructs etc. , it is the time that we look in to aspects of errors in a program. Errors are known as **Bugs** , because errors can creep or crawl in to every program unnoticed. Driving out these **Bugs** from a program is **Debugging.**

There can be three distinct types of errors in a program they are **syntactic** and **logical and data errors..** If we violate the grammatical rules of the language, we commit **syntactic** errors. As examples; writing = in place of ==, writing => in place of >=, mismatching the parenthesis, missing semicolons etc.. When syntactic errors are committed the compiler flags messages specifying the nature of error and the line on which they have occurred.

It is logical errors which are more dangerous!, because compilers do not complain about these errors and as a consequence some unwieldy results are brought out. It is very difficult to trace out the cause. In case of unreliable output owing to logical error(s) , it is better that the outputs are verified for small sections of the program in a step by step manner .

While the instructions are executed, the compiler will always exercise the priority or precedence in which the operations are to be performed. This order of execution is called **hierarchy** of operations. Mathematical operations are performed before the relational operators and these in turn take precedence over the logical operators. Table 3.1 lists the operators in descending order of precedence.

Table 3.1 Hierarchy of operations

Operators	**Explanation**
( )	Parenthesis
!	Logical NOT
* / %	Multiplication, division and modulus.
+ -	Addition and Subtraction.
< > <= >=	Relational operators.
== !=	Relational operators.
&&	Logical AND
||	Logical OR
=	Assignment

Try Yourself

1. Write a program that reads an integer and states whether the integer is evenly divisible by 7.
2. Input a number and determine whether it is with in a given range. The user may also input the low and high value of the range.
3. Declare the result of a student based on the percentage of marks he/she has obtained. Use the following ranges;

Percentage	Result
Less than 35	Fail
35-50	Third Class
51-59	Second class
60-69	First class
70-100	Distinction

Session 7.

Using Loop Control Structures

In session 5, we worked with programs that went line by line sequentially, in session 6, we used decision control structures. But in both the sessions the statements in the programs executed only once. Situation may arise in problems, to repeat a certain set of statements until a condition is reached. Loop control structures in C can cause section of a our program to be repeated a certain number of times. The repetition continues while a condition is true. When the condition becomes false, the loop ends and control passes to the statements following the loop. There are three kinds of loop structures in C, viz.,

- The **for** loop
- The **while** loop
- The **do-while** loop

The for loop

Of the three loops, **for** loop is the easiest to understand and implement. The **for** loop has its control elements in a single line. The general form of **for** loop is;

```
 for (initialisation; test expression; increment expression)
         {  Statement 1;
            Statement 2;
            Statement 3;
            …………….
          }
```

We shall go through an example to understand **for** loop in a greater detail. Let us write a program that can produce numbers from 1 to 10 and also their squares.

Example 1. To print numbers from 1 to 10 and their squares.

```
#include<stdio.h>

main()
{ int j ;
   printf("Number                 Square \n");
   for(j=1; j<=10;j++)
   printf(" %4d                      %4d \n", j,j*j);
   getch();
}
```

Output:

```
 Number                 Square
   1                      1
   2                      4
   3                      9
   4                      16
```

```
5                          25
6                          36
7                          49
8                          64
9                          81
10                         100
```

Let us examine the way in which the program worked.

- When the **for** statement is executed for the first time, the value of **j** is set to an initial value of 1.
- Then the condition is testified, that is **j<=**10 is tested, since **j** is 1, the condition is satisfied and the body of the **for** loop is executed.
- When the compiler reaches the closing **brace**, it sends back the control to for statement again where the value of **j** gets incremented by 1.
- The procedure repeats till the value of **j** is equal to 10

Initialisation Expression

This in a **for** loop is executed only once, when the loop starts. It gives the loop variable an initial value. In the example above, it sets j to 1.

Test Expression

The test expression usually involves a relational operator. It is evaluated each time through the loop, just before the body of the loop is executed. If the test expression is true, the loop is executed one more time. If it is false, the loop ends and control passes to the statements following the loop.

Increment expression

This expression changes the value of the loop variable, often by incrementing it. It is always executed at the end of the loop, after the loop body has been executed. In the above example, the increment operator **++** adds 1 to j each time through the loop**.**

How many iterations ?

In the above example it is exactly 10 times. The first time j is 1. This is ensured in the initialization expression. The last time through the loop, **j** is 10. This is determined by the test expression **j**<=10 . When j becomes 11 the loop ends up and loop ceases to work.

Few other points about `for` loop

- More than one statement in a loop body should always be enclosed by braces, just as we did for if blocks.

- Incrementation may be done with in the body of the **for** loop as shown below,

```
for(i=0;i<=10;)
{ printf("%d\n",i);
i++;
}
```

- Initialization may also be done outside the loop as shown below;

```
i=1;
for(;i<=10;i++)
printf("%d\n",i);
```

- Testing for condition and incrementation can be simultaneously done as shown;

```
for(i=0;i++<=10;)
printf("%d\n",i);
```

- For loops may be used one inside the other. Then it becomes **nested** for loops.

```
for(i=1;i<=2; i++)
for(j=1;j<=2;j++)
{  prod = j*i;
printf("i=%d,j=%d,product=%d\n",i,j,prod);
}
```

when we execute the above routine, what we get is;

```
i=1 j=1 product=1
i=1 j=2 product=2
i=2 j=1 product=2
i=2 j=2 product=4
```

- A for loop can have more than one initialization and incrementation. However only one expression is allowed in test expression. As in,

```
for(i=0,j=n;i<=n;i++,j--)
```

Notice that, each time loop is executed, j gets decremented by 1.

To make our understanding still more clear we shall go through examples later in this session.

The `while` loop

We observed in **for** loop that , it does a task repeatedly a fixed number of times. In real life programming we may come across a situation when it is not possible to fix up number of iterations before hand. In such cases **while** loop may be used. The syntax of **while** loop is;

```
while(Expression)
  {  statement 1;
     statement 2;
     statement 3;
     ……………
  }
```

The expression is evaluated first. The block of statements is executed repeatedly until the value of expression is false. We shall look into an example to make ourselves clear.

Example 2. This program finds the sum of the digits of a four digit number.

```
#include<stdio.h>

main()
{ int sum,num,dup_num;
  printf("Enter a four digit number\n");
  scanf("%d",&num);
  dup_num=num;
  sum=0;
  while(num!=0)
  { sum=sum+num%10;
    num=num/10;
    }
 printf("sum of digits of num %d is=%d\n",dup_num,sum);
    getch();
}
```

output:

```
Enter a four digit number
4567
Sum of the digits of the number 4567 is=22
```

Explanation

- A four-digit number is read through key board on prompting.
- The number is duplicated in **dup_num** so that original number is not lost.
- An accumulator **sum** is set to 0.
- A counter **i** for iteration is initialized to 1.
- While loop is invoked to execute the statements repeatedly written inside the loop body till **i** is less than or equal to 4.

We shall go along the loop with a four digit number 4567 as input to make an understanding of the loop.

In the very first cycle i= 1, sum = 0 + 4567%10 =7
num becomes num/10 ie 4657 becomes 465.
i gets incremented to 2,

In the second cycle i=2, sum= 7+465%10 = 12
num becomes 465/10 = 46
i gets inc**rem**ented to 3.

In the third cycle i=3, sum= 12 + 46%10 = 18
num becomes 46/10 = 4
i gets incremented to 4.

In the fourth cycle i=4, sum=18 + 4%10 = 22
num becomes 4/10 = 0

i gets incremented to 5

Fifth cycle can not happen because **i** has gone for 5.

The `do-while` loop

In a **while** loop, the test expression is evaluated at the beginning of the loop. If the test expression is false when loop is entered, the loop body won't be executed at all. But sometimes, we want to guarantee that the loop body is executed at least once, no matter what the initial state of test expression is. In such situations we have to use **do-while** loop. This loop structure places test expression at the end. The syntax is;

```
do
    {
       Statement 1;
       Statement 2;
       ...............
       ...............
     } while(condition);
```

Notice the semicolon at the terminal statement end . While making interactive programs the do-while loops are of great help. Example 3 will make the use of **do-while** clear.

Example 2: This program reads a positive number only.

```
#include<stdio.h>
main()
{ int i;
   do
 {
   printf("Enter a positive Number\n");
   scanf("%d",&i);
   } while(i<=0);

   }
```

Output

```
Enter a positive number
-5
Enter a positive number
5
```

The *break* and *`continue`* in loop

break

The **break** statement causes an immediate exit from the loop structure instantly. When the key word **break** is found inside any C loop, control automatically passes to first statement after the loop. The word **break** conjointly exists with an **if**. We will see the effect of **break** when we work with examples.

continue

This statement causes the next iteration of the loop structure. The keyword **continue** allows the compiler to bypass the statements inside the loop which have not yet been executed.

After having acquainted ourselves with basic features of these three loop structures, we shall now work with various problems involving them.

Example 3.
This program will add the series 1+(1+2)+(1+2+3)+……….+(1+2+3+…n)

```
#include<stdio.h>

main()
{ int nt,sum,i,bigsum;
  printf("Enter number of terms:");
  scanf("%d",&nt);
/* A counter is initialised to accumulate sum*/
sum=0;
/* This accumulates sum of sums */
bigsum=0;
for(i=1;i<=nt;i++)
  {sum=sum+i;
   bigsum=bigsum+sum;}
   printf("\n Sum of given series is=%d\n",bigsum);
  getch();
  }
```

output

```
Enter number of terms
3
Sum of given series is = 10
```

Explanation

- The number of terms in the series is the input.
- Two accumulators are initialized one for adding the terms in the individual term of the series(sum) and the other for adding these sums(bigsum).
- Then a for loop is invoked. In this for loop sum generates individual terms of the series while bigsum adds up these individual terms.

Ex : If nt=3,
    Then in the first cycle, sum = 0+1=1 , bigsum =0+1=1

In the second cycle sum=1+ 2 =3 , bigsum=1+3=4
In the third cycle sum=3+3 = 6 , bigsum=4+6=10

Example 4 : This program reads a four digit number and reverses it.

```
#include<stdio.h>
main()
{
 long int num,rev=0,rem,dupli;
 printf("Enter a number\n");
 scanf("%ld",&num);
  dupli=num;
  while(num)
  {
   rem=num%10;
   rev=rev*10+rem;
   num/=10;
   }
 printf("The reversal of number %ld is %ld\n",dupli,rev);
   getch();
   }
```

Output:

```
Enter a number
1234
The reversal of number 1234 is 4321
```

Explanation:

- A number of long integer type is read and duplicated into dupli.
- We take a number say 1234 to understand further steps. In the while loop, for the first cycle, rem = 4 , rev=4, num=123. In the second cycle, rem=3,rev=43,num=12. In the third cycle,rem=2,rev=432,num=1. In the fourth cycle,rem=1,rev=4321 and num=0. Thus at the end of the fourth cycle the control comes out of the loop.

Example 5. In this example we shall program for the calculation of factorial of a number.
(Factorial of a number say 5 is 1x2x3x4x5)

```
#include<stdio.h>

main()
{
 int num,i;
 long int fact;
 printf("Enter a number for factorial");
 scanf("%d",&num);
 /* Initialising for accumulation*/
 fact=1;
 for(i=1;i<=num;i++)
 fact=fact*i;
 printf("\n Factorial of %d!=%ld",num,fact);
 getch();
 }
```

output:

Enter a number for factorial

```
5
Factorial of 5=120
```

Explanation

- A number is read for finding its factorial.
- An accumulator **fact** is initialised to 1(since it is a multiplier it can not be taken as zero to begin with).
- When for loop starts its function with the number say 5

  In first cycle i=1 , fact = 1*1= 1
  In the second cycle i=2, fact=1x2=2
  In the third cycle i=3, fact =2x3=6
  In the fourth cycle i=4, fact =6x4=24
  In the fifth cycle i=5, fact =24x5 = 120
- Notice that the fact is declared as a long integer in order to enable large values to be printed.

Example 6. Let us list and count the numbers within 50 that are divisible by 7 and not by 5.

```
#include<stdio.h>

main()
{ int i,count;
count=0;
/* The for loop is repeated 50 times*/
printf("Numbers divisible by 7 and not by 5 \n”);
for(i=1;i<=50;i++)
if(i%7==0&&i%5!=0)
{printf("%d\n",i);
 count++; }
 printf("There are %d such numbers\n",count);
getch();
}
```

output

```
Numbers divisible by 7 and not by 5
7
14
21
28
```

```
42
49
There are 6 such numbers
```

Explanation

- The number when divided by 7 should not have remainder and when the same number is divided by 5 it should not be evenly divisible (remainder is present). This logic is applied and all the numbers with in 50 are testified for this.

Example 7 . This program checks whether a given number is prime or not.

Logic: Prime number is one which is divisible only by 1 and by itself. Numbers 1,2 and 3 are prime numbers. To test the prime property of a given number, the number need be divided by all the positive integers from 2 to half of the number itself(Also it can be divided from 2 to square root of the number itself). If at any one stage the remainder becomes zero , then the number is not a prime. If all the numbers between 2 and half the number does not divide the number equally, than the number is prime.

```
#include<stdio.h>

main()
{
  int num,i,k;
  printf("Enter a num to check for prime property\n");
  scanf("%d",&num);
  k=0;
  for(i=2;i<=num/2;i++)
  if(num%i==0)k=1;
  if(k==0)
  printf("The number is  prime\n");
  else
  printf("The number is not prime\n");
  getch();
  }
```

output

```
Enter a number to check for prime property
11
The number is prime
```

Explanation

- A number is read for checking its prime property.
- A check counter is initialized to 0(k=0).
- The for loop is initiated for dividing the number repeatedly by 2 to num/2.
- If in any of the iterations the number gets divided equally, the value of k becomes 1.

- After the completion of the looping, if the value of k is still 0 then, it can be taken to mean that the number is not divisible by any numbers from 2 to half of its value, on the other hand if k holds 1 , it is meant that the number has got divided equally by a divisor between the 2 to num/2.

Alternatively, the number can be checked to be prime or not a prime by the code written below.

```
#include<stdio.h>

main()
{ int num,i,rem;
printf("Enter a number for testing its primeness\n");
  scanf("%d",&num);
  i=2;
  while(i<=num/2)
  { if(num%i==0)
   { printf("Not a Prime number\n");
    break;
   }
     i++;
  }
     if(i>num/2)
     printf("The number is prime\n");
  getch();
}
```

output

First run:

```
Enter a number for testing its primeness
27
Not a prime number
```

Second run:

```
Enter a number for testing its primeness
23
The number is prime
```

Explanation

- In this program while loop is used. While loop condition is same as that of for loop. If the remainder of num and i is zero , When the loop is operative, the loop breaks out with a message Not a prime number.
- If the remainder is not zero, the loop will complete its full iterations i.e. from 2 to num/2. We find a conditional check soon the while loop is over, the value of **i** should be more than num/2 had this loop completed its iterations fully. If it is true, the number is a prime. We could not have written as Number is not a prime with out any conditional check because if the number were to be really not a prime , we would have got both messages.

Example 8 This program generates the prime numbers with in a given number. Also it makes a count of them.

```
#include<stdio.h>

main()
{ int i,rem,num,j,k=1,l=2,no_prime;
printf("Enter a number\n");
scanf("%d",&num);
printf("The prime numbers with in %d are\n",num);
printf("%d\n%d\n",k,l);
no_prime=2;

for(i=3;i<=num;i++)
 {j=2;
 do
 { rem=i%j;
  if(rem==0)
  break;
  j++;
  } while(j<=i/2);
  if(j>i/2)
 { printf("%d\n",i);
   no_prime++;}
  }
  printf("The number of prime numbers:%d\n",no_prime);
  getch();
  }
```

Output

```
Enter a number
15
The prime numbers with in 15 are
1
2
3
5
7
11
13
The number of prime numbers:7
```

Explanation

- The first two numbers i.e. 1 and 2 are taken to be prime and the counter for number of primes is initialised to 2.

- Outer **for** loop sends numbers from 3 to 15 one after the other to the inner while loop where the number gets continuously divided by 2 to half of its value. If number is proved to be prime it will be printed. Simultaneously, the counter gets increased by one.

Alternatively,

```
#include<stdio.h>
main()
{
  int i,ch,j,np=2,num,np1,np2;
  printf("Enter a number for generating primes\n");
  scanf("%d",&num);
  np1=1;
  np2=2;
  printf("The list of primes\n");
  printf("%d\n%d\n",np1,np2);
  for(i=3;i<=num;i++)
   { ch=1;
     for(j=2;j<=i/2;j++)
     if(i%j==0)ch=0;
     if(ch==1)
       { printf("%d\n",i);
      np++;
       }
   }
   printf("Number of prime numbers within %d
is%d\n",num,np);
   getch();
   }
```

- In the above alternative method , the outer for loop sends the numbers starting from 3 upto the limit specified. In every looping the check counter **ch** will be set to 1. In the inner for loop each and every number sent by the outer for loop is divided from numbers 3 upto half of itself. While doing division, if at any time the remainder becomes zero, the check counter **ch** gains zero signalling non-primeness of the number. Outside the inner for loop, an if statement is placed which takes care of printing prime numbers only.

Example 9: A famous conjecture holds that, all positive integers converge to ONE when treated in the following fashion:

1. **If the number is odd, it is multiplied by 3 and then 1 is added.**
2. **If the number is even, it is divided by 2**
3. **Apply the above two operations to the intermediate results until the number reaches to 1.**

```
#include<stdio.h>
main()
{
 int num,i,dupli;
 printf("Enter a number\n");
 scanf("%d",&num);
 dupli=num;
 i=0;
 do
 {
  if(num%2!=0)
  num=num*3+1;
  else
  num=num/2;
  i++;
  }while(num!=1);
  printf("It has taken %d steps for the number %d to
become one\n",i,dupli);
  getch();
  }
```

output:

```
Enter a number
7
It has taken 16 steps for the number 7 to become 1.
```

Explanation:

After having read the number, the number gets processed till it becomes one. The **do while** loop is tailored for this condition only. When the number becomes 1 the control comes out of the loop. The number of steps of processing is nothing but the number of iterations.

Example 10: This programe accepts a number n, then finds the sum of the integers from 1 to 2, then from 1 to 3, then from 1 to 4 and so forth until it displays sum of integers 1 to n.
For exa: If input is 5, theb output will be 1 ,3 , 6 , 10 , 15

```
#include<stdio.h>
main()
{
  int i,j,sum,n;
  printf("Enter an integer\n");
  scanf("%d",&n);
  for(i=1;i<=n;i++)
   { sum=0;
     for(j=1;j<=i;j++)
     sum+ = j;
     printf("\t%d",sum);
   }
   getch();
 }
```

Output:

```
Enter an integer
5
1   3   6   10   15
```

Explanation:

- The moment the integer value is read, the outer for loop sets the limits from 1 through n such that n terms are generated. This can be seen from the output values, i.e., when n=5, five terms are generated.
- The inner for loop iterates from 1 to i, that is if i=2, then what gets added to sum is 0+1+2(3).

Alternatively,

```
main()
{
  int i , sum=o,n;
 printf(“Enter a number \n”);
 scanf(“%d”,&n);
 for(i=1; i<=n; i++)
  {
     sum+ = i ;
     printf(“ %d\t”,sum);
   }
getch();
}
```

Explanation:

- The above alternative program is just a straight forward way of getting the series. Every time the for loop is processed, sum accumulates the value and gets printed.

Example 11: This program generates Fibonacci series upto n terms.
{Fibonacci series: 1 1 2 3 5 8 13 21 34 55..................... }

```
#include<stdio.h>
main()
{
 int fib1=1,fib2=1,fib,nt,i;
 printf("Enter number of terms in fibonacci series\n");
 scanf("%d",&nt);
 printf("Fibonacci Series\n");
 printf("%d\n%d\n",fib1,fib2);
 for(i=0;i<=nt;i++)
 { fib=fib1+fib2;
   fib1=fib2;
   fib2=fib;
   printf("%d\n",fib);
   }
   getch();
   }
```

output

```
Enter number of terms in Fibonacci series
10
Fibonacci Series
```

```
1
1
2
3
5
8
13
21
34
55
89
144
233
```

Explanation:

- **Here a** for **loop is used to generate prime number. The for loop is delimited up to the number of terms sought. In every iterative cycle, the new term for the series comes out as a sum of previous two terms. An exchange mechanism will facilitate to obtain these previous terms.**

Example 12 : This program evaluates the value of π using the series, for a given accuracy level.

$$\pi = \text{sqrt}(6/1^2 + 6/2^2 + 6/3^2 + \ldots\ldots\ldots\ldots\ldots\ldots)$$

```
#include<stdio.h>
#include<math.h>
main()
{
 float i,pip,acc,term,sum=0,pin,diff;
 printf("Enter the accuracy(0.001 to 0.000001)\n");
 scanf("%f",&acc);
 pip=0;
 i=1;
 do
 {
  term=6./(i*i);
  sum+=term;
  pin=sqrt(sum);
  if((pin-pip)<=acc)break;
  else
  pip=pin;
  i++;
  }while(diff>acc);
  printf("Value of ã from series=%f\n",pip);
  getch();
  }
```

Output

```
Enter the accuracy(0.1 to 0.00001)
```

```
0.000001
The value of phi is 3.141274
```

Explanation

- The series gives accurate value of π depending on number of terms we consider. Naturally, more number of terms we consider more accurate the value would be. But to put a limit for the number of terms, an accuracy level is suggested. The accuracy value is the numerical difference between the value of π in the present (**pin**-phi present) cycle and value of π in the previous cycle(**pip**- phi previous).

Example 13: Let us find the value of sine of an angle using sine series. The program will read five angles and obtains the sine of that angle for a given accuracy, for given number of terms. It also calculates the value from library function.

```
#include<stdio.h>
#include<math.h>
main()
{
 float x,sum,term,acc;
 int i,opt,nt,j;
 clrscr();

 printf("To calculate sine of an angle by series\n");
 printf(" 1. By specifying number of terms\n");
 printf(" 2. By Designating accuracy\n");
 printf(" 3. By Built-in function\n");
 printf("Enter your option\n");
 scanf("%d",&opt);
 if(opt==1)
 {
  printf("Enter number of terms\n");
  scanf("%d",&nt);
  for(i=1;i<=5;i++)
   { printf("Enter angle no %d: ",i);
     scanf("%f",&x);
     x=(x*22.)/(7*180);
     sum=x;
     term=x;
     for(j=1;j<=nt;j++)
     { term=-term*x*x/(2*j*(2*j+1));
       sum=sum+term;
       }
       printf("Sine of the angle is=%f\n",sum);
       } /* close of outer for*/
     }/*close of if*/
    else if(opt==2)
    {  printf("Enter the accuracy level(0.01 to
0.000001)\n");
       scanf("%f",&acc);
        for(i=1;i<=5;i++)
     { printf("Enter angle no %d: ",i);
       scanf("%f",&x);
       x=(x*22.)/(7.*180.);
       sum=x;
       term=x;
       j=1;
       do {
       term=-term*x*x/(2*j*(2*j+1));
       sum=sum+term;
       j++;
        }while(fabs(term)>acc);
        printf("Sine of the angle is=%f\n",sum);
        } /* close of outer for*/
     }/*close of elseif*/
      else if(opt==3)
      {  for(i=1;i<=5;i++)
      { printf("Enter angle no %d: ",i);
        scanf("%f",&x);
        x=(x*22.)/(7*180);
        printf("Sine of the angle =%f\n",sin(x));
       }
      } /* end of else if*/
      else
       printf("Absurd input\n");

       getch();
       }
```

Output

```
To calculate sine of an angle by series
1. By specifying number of terms
2. By Designating accuracy
3. By Built-in function
Enter your option
2
Enter the accuracy (0.1 to 0.000001)
0.000001
Enter the angle no: 1
30
Sine of the angle is=0.500183
Enter the angle no: 2
60
Sine of the angle is=0.866236
Enter the angle no: 3 45
Sine of the angle is=0.707330
Enter the angle no: 4
```

```
90
Sine of the angle is=1.000000
Enter the angle no: 5
0
Sine of the angle is=0.000000
```

Explanation

- This program works for three alternative ways of finding sine of an angle.
- If angle is to be evaluated by specifying number of terms, a nested **for** loop is being made use of. The outer **for** loop helps in reading 5 values of angles in degrees. While the inner **for** loop adds up the sine series to the accumulator **sum**.
- Outside the inner for loop, **sum** is initialised to **x**, the term also initialised to **x** and the denominator of the series i.e, is initialised to 1. In the first cycle of the loop the sum will add up to $x - x^3/3!$, which is being the sum of first two terms of the series. In the next cycle the term would become $x^5/5!$, the third term of the series. This gets added to sum in the second cycle only.
- If angle is to be calculated by fixing certain accuracy level, **do-while** loop is made use of. The loop will terminate when the given accuracy is satisfied.
- We can see in the program that if the option is to use the library function, then the method is just straightforward.

Example 14:Obtain cosine of an angle by inputting an angle in degrees to a given accuracy. Use cosine series.

$\cos(x) = 1-x^2/2! + x^4/4! - x^6/6! + \ldots\ldots\ldots\ldots\ldots..$

```
#include<stdio.h>
#include<math.h>
main()
{ float x,term,sum,prod,acc,i;
  int j;
  printf("Enter the accuracy\n");
  scanf("%f",&acc);
  for(j=1;j<=5;j++)
  { printf("Enter angle no:%d\n",j);
    prod=2;
    i=2;
    sum=1;
    printf("Enter the angle \n");
    scanf("%f",&x);
    x=(x*22.)/(7.*180);
  do
  {
   term=pow(x,i)/prod;
   sum= sum - term;
   prod=-prod*(i+1)*(i+2);
   i+=2;
   }while(fabs(term)>acc);
```

```
    printf("Value of the angle from series=%f\n",sum);
    getch();
    }
    getch();
    }
```

Output:

```
Enter the accuracy
0.000001
Enter angle no: 1
30
Value of the angle from the series = 0.865920
Enter angle no: 2
45
Value of the angle from the series = 0.706883
Enter angle no: 3
60
Value of the angle from the series = 0.499635
Enter angle no: 4
90
Value of the angle from the series = -0.000063
Enter angle no: 5
0
Value of the angle from the series = 1.000000
```

Explanation:

- This program is no different from that for sine series. The major differences are however, noticed in initialisations, generation of denominator. It is important to note that the alternate sign changes in the series is achieved by negating the variable prod.

Example 15: We shall program for finding GCD or HCF of two integers.

```
#include<stdio.h>

main()
{ int num1,num2,rem,gcd,t;
  printf("Enter two numbers for finding GCD\n");
  scanf("%d%d",&num1,&num2);
  do
    {  rem=num1%num2;
       if(rem ==0)
       gcd= num2;
       else
       num1=num2;
       num2=rem;
       } while(rem!=0);
       printf("The GCD of two numbers is %d\n",gcd);
       getch();
```

```
        }
```

Output

First run

```
Enter two numbers for finding GCD
8
4
The GCD of two numbers is 4
```

Second run

```
Enter two numbers for finding GCD
7
12
The GCD of two numbers is 1
```

Explanation

- Though we are knowing what **GCD** of two numbers is, we shall go through specific example to unearth as to how the program takes up the calculation,
- If two numbers are 7 and 12 , in the first iteration **rem** will get 7 to its store, **num1** i.e.7 is erased and **num1** becomes 12, and **num2** becomes 7.
- In the second cycle, **rem** will get (12%7) 5, **num1** will store 7 and **num2** will store 5.
- In the third cycle, **rem** will get 2(7%5), **num1** will store5 and **num2** will store 2.
- In the fourth cycle, **rem** will get 1(5%2), **num1** will store 2 qnd **num2** will store 1
- In the fifth cycle, **rem** will get 0(2%1) and GCD becomes equal to 1 and the loop breaks.

Example 16:This program examines all the numbers from 1 to 999, displaying all those for which the sum of cubes of the digits equals the number itself.

[For example : 563 : $5^3 + 6^3 + 3^3$ = 125 +216+27 =368 is not the original number.
371 : $3^3 + 7^3 + 1^3$ = 27 + 343 +1 = 371 is the original number.].

```
#include<stdio.h>
main()
{
   int i,digit,sum,k,same=0;
   for(i=1;i<=999;i++)
    {  sum=0;
        k=i;
        while(k!=0)
         {
             digit=k%10;
              sum+ = digit*digit*digit;
              k=k/10;
          } /* while closes */
      if(sum==i) printf("%d\n",i);
      same++;
    }/* for closes */
```

```
  printf(" There are only %d such numbers from 1 to 999
\n",same);
  getch();
}
```

Output:

```
1
153
370
371
407
There are only 5 such numbers from 1 to 999
```

Explanation:

- The outer for loop sends numbers 1 through 999 to inner for loop.
- The inner for loop just detaches the digits of the numbers obtained from outer for loop and to gets the sum.
- The value of the sum so obtained is checked for equality with the original number. If equality is established, it gets printed. The count of such numbers is done by the variable same.

Points to Rehearse

- Loop control structures are essentially used to repeat a particular set of statements a number of times.

- The special advantage of **for** loop is that, the parameters of looping i.e., initialisation, testing and incrementation can be incorporated in a single line. However, the disadvantage is that, the number of cycles is predetermined.

- The program segment written down would generate numbers from 100 to 1.

```
int i;
for(i=100;i<=1;i--)
printf("%d\n" ,i);
```

- The loop structure presented below is illegal because;

```
int k,i;
while(i<=10)
{ scanf("%d",&k);
i+++;
};
```

→i is not initialised.
→+++ is not allowed.
→; is placed after closing curly brace.

➢ Following loops will work infinitely because,

i.
```
int i=1;
while(i<=10);
{  printf("%d\n",i*i);
i++;
}
```
→There is a ; after the while.

ii.
```
int i= 1;
while(i<=10)
{
;
}
```
→the value of i is not getting incremented. Therefore control keeps rotating with in the loop.

Points to Ponder

When to use a particular loop? . There seems to be no clear-cut answer for this question. But, to say the least, **for** loop is appropriate when we know in advance the number of times the loop is to be executed. The **while** and **do** loops are used when we don't know in advance when the loop is to be closed. The **do-while** loop is to be used when we want to execute the loop body at least once.

In this session, we have used statements of the form;
sum=sum+term; → term gets added to sum.
fact=fact*i; → i gets multiplied to fact.
Arithmetic statements of this kind are usual components of C program. But C has number of ways to write such operations. These type of statements can be made short and crisp by using **updating assignment operators**, thus
```
   sum+=term;
   fact*=i;
```
Are perfectly valid. += means , add the expression on the right to the variable named on the left. The table 4.1 lists all such updating assignment operators.

Table 4.1 Updating assignment operators

Statement	Means this
x+=y;	x=x+y;
x-=y;	x=x-y;

x*=2;	x=x*2;
x/=2;	x=x/2;
x%=2;	x=x%2;
x*=y+z	x=x*(y+z);

When it becomes necessary to add 1 or subtract 1 from a variable, we have a concise way to do this job. If we wish to add 1 to a variable i, we can write **i++;** . Here , **++** is called **increment operator**. Similarly **--** is **a decrement operator**. We can also write them as **++i** and **--i** ; the position of this operator does not make much of a difference though in some cases their position is important. The example written below will explain the difference between post incrementing and pre incrementing operators.

```
main()
{ int i=10,j=15;
   printf("value of  ++i =%d\n",++i); /* here, i=11 */
   printf("value of  j++=%d\n",j++); /* here , j=15 */
}
```

It is syntactically correct incorrect to write expressions such as;

```
x--* =2;
x/*=2;
x++/=2;
```

Decisions can be made crisply by using **ternary operators ?** and **; .** This method of decision making reduces the big **if.. else** structure in to a single line. The syntax will look like,

Expression 1? Expression2: Expression3

The meaning of the above syntax is, if **Expression** 1 is **true**, then, the **Expression**2is to be executed otherwise **Expression** 3. We shall see few example segments to understand it better.

```
i.   int a,b,big;
     scanf("%d%d", &a,&b);
     big=a>b?a:b;
```

→ Here, big will store **a** if **a**>**b,** otherwise **b**.

```
ii   int j;
     scanf("%d", &i);
     j==1?printf("Male"): printf("Female");
```

→ Here, if j=1 What gets printed is Male otherwise Female.

```
ii.  int x,y,z,big;
```

```
        big=(x>y?(x>z?x:z): (y>z?y:z))
```

→ Here ternary operators are nested for finding biggest of three numbers.

Try Yourself

1.Write a program that asks the user to enter a radius value and computes volume of a sphere. The program should terminate when a negative or a zero is entered as a radius value.(Volume of a sphere = $4/3 \pi r^3$).
2.Write a program that generates fibonacci series up to a particular value n.
3.Read a binary number and count number of 1^s present in it.
4.Read 5 years and categorise them into leap and non-leap year types.
{ A leap year is one which is evenly divisible by 4 or 400 and not by 100 }.

Session 8.

Use of Case Control Structure

When we plan to write a program involving situation where, a choice between number of alternatives is to be encountered then, **case control structure** is of great help. In effect, case control structure is an extension of **if….else** statement. The if….else structure permitted us to do for a maximum of two branches, where the case control structure permits us to go for any number of branches.
The control statement which allows us to make a decision from the number of choices is called a switch, however, **switch-case-default** coined together make the complete case control structure. The syntax of the structure is:

```
switch (Integer expression or integer or character variable)
    {
            case 1:
                    statement;
                    ………..;
                    ………..;
            case 2:
                    statement;
                    ………..;
                    ………..;

            case 3:
                    statement;
                    ………..;
                    ………..;
            default:
                    ………..
                    ………..
    }
```

We shall go through the salient components of this structure one by one.

- After keyword **switch**, within the parenthesis, we may include an integer expression that will yield an integer value, it could be an integer variable which takes value 1, 2, 3, …….., or it can be character variable as well.
- The keyword **case**, is followed by an integer constant or a character constant, Each constant in each case should be different from all others.
- When we run a program containing switch, the integer expression or integer value with in the parenthesis is matched against the constant values that follow case statements. When there is a match, the statements attached to that **case** will be executed. If none of the case matches the statements attached to **default** are executed.

We shall work through few examples to get the grasp of this construct.

Example 1. We shall program to solve Quadratic equation for its roots (**you may recall that earlier we have done it using decision control structure).**

```
#include<stdio.h>
#include<math.h>
main()
{ float a,b,c,disc,root1,root2,root,rp,im;
  int code;
  printf("Enter coefficients of quadratic equation\n");
  scanf("%f%f%f",&a,&b,&c);
  if(a*b*c==0)
  {printf("Not-tenable\n");
   exit();}
 disc= b*b-4*a*c;
 if(disc>0) code=1;
 if(disc<0) code=2;
 switch(code)
 {
   case 1: printf(" The roots are real and distinct\n");
        root1= (-b + sqrt(disc))/(2*a);
        root2=(-b-sqrt(disc))/(2*a);
        printf("Root 1 is=%f\n",root1);
        printf("Root 2 is=%f\n",root2);
        break;

   case 2: printf("Complex roots\n");
        rp=-b/(2*a);
        im=sqrt(fabs(disc))/(2*a);
        printf("Root 1 is= %f +i%f\n",rp,im);
        printf("Root 2 is= %f-i%f\n",rp,im);
        break;
  default: printf("Roots are real and equal\n");
         root= -b/(2*a);
         printf(" The roots are %f and %f\n",root,root);
  }

  getch();
 }
```

Output:

First run

Please enter non-zero coefficients

```
1
-4
4
Roots are equal
The roots are 2.000000 and 2.000000
```

Second run

```
Please enter non-zero coefficients
1
-5
6
The roots are real and distinct
Root 1 is= 3.000000
Root 2 is =2.000000
```

Explanation

- The coefficients are checked for their non zero value. If one of the coefficients turn out to be zero, the program prints appropriate message and comes out. The return of the control is facilitated by the function **exit().**
- The integer variable **code** is used as a switch value. This **code** is given the value 1 and 2 for positive discremenent and negative discremenent respectively. Accordingly the **case** blocks are being constructed. The default block will take care if the discremenent is zero.

Example 2. Let us make menu which will have a facility to calculate factorial of a number, checks a number for prime property, finds its cube, raises it to a power.

```
#include<stdio.h>
#include<math.h>
main()
{
 int num,opt,i,p,num_to_pow,cube;
 long int fact;
 clrscr();
 printf("******** WORK WITH A NUMBER *******\n");
 printf("1. Raise to power\n");
 printf("2. Find the cube\n");
 printf("3. Find the factorial\n");
 printf("4. Check for prime\n");
 printf("5. Exit\n");
 printf("Enter the number\n");
 scanf("%d",&num);
 printf("Enter the option\n");
 scanf("%d",&opt);
 switch(opt)
 {
   case 1:
        printf("To what power ?\n");
        scanf("%d",&p);
        num_to_pow= pow(num,p);
        printf("Number raised %d is=%d\n",p,num_to_pow);
        break;
   case 2: cube=num*num*num;
```

```
        printf("Cube of number %d is= %d\n",num,cube);
        break;
   case 3: fact=1;
        for(i=1;i<=num;i++)
        fact*=i;
        printf("Factorial of the number is=%ld\n",fact);
        break;
   case 4: i=2;
        while(i<=num/2)
        { if(num%i==0)
          { printf("The number is not a prime\n");
           break;
               }
          i++;
        }
        if(i>num/2)
        printf("The number is a prime\n");
        break;
   case 5:printf("Thank you\n");
          break;
   default: printf("That is not an option\n");

   }
   getch();
   }
```

Output
First run:

```
******** WORK WITH A NUMBER ********
1.Raise to power
2.Find the cube
3.Find the factorial
4.Check for prime
5.Exit\n"
Enter the number
11
Enter the option
4
The number is not a prime
```

Second run:

```
******** WORK WITH A NUMBER ********
1. Raise to power
2. Find the cube
3. Find the factorial
4. Check for prime
5. Exit\n"
Enter the number
5
Enter the option
```

```
3
Factorial of the number is : 120
```

Explanation

- This program is not different from the earlier program. There is a use of function **clrscr**()(clear screen**)**this function will facilitate to clean the screen in every subsequent running of the program. This function is defined in **conio.h** header file.The option **opt** is taken as a switch variable. Depending on the number given by the user, the control will be transferred to the appropriate case, and the desired output will be flagged on the screen. Notice the use of power function in option 1.

Points to Rehearse

- Switch statement can be preferably used instead of series of **if..else** or **else if** constructs.

- Multiple statements to be executed in each **case** has to be enclosed in parenthesis.

- The closing curly brace of the **switch** block terminates with a **;.**

- The **case** in a **switch** do not take float value.

- The **switch** statement written down is erroneous because,

Example 3:
```
#include<stdio.h>
main()
{ float a
   scanf("%f",&a);
   switch(a)
  {
     case 0.0: printf("Not possible\n");
                    break;
     case 2.2: printf("Too bad");
                    break;
     default;   printf("Sorry\n");
  };
}
```

→ float variable is associated with switch.
→ case is tagged with float value.
→ semicolon is attached with closing brace of switch block.

Points to Ponder

Switch statement could be used with character values also. Example below will narrate this.

Example 4:
```
#include<stdio.h>
main()
{
  char c;
  printf("Enter M or m for male , F or f for female\n");
  scanf("%c",&c);
  switch(c)
  { case 'm':
    case 'M': printf("The tax reduction is 20%\n");
              break;
    case 'f':
    case 'F': printf("The tax reduction is 10%\n");
              break;
    default : printf("Invalid entry\n");
  }
  getch();
  }
```

output:
```
Enter M or m for male,F or f for female
m
```
The tax reduction is 20%

The case need not be associated with numbers arranged in ascending order. We can in fact put the cases in any order the situation demands. Look at this example.

Example 5:
```
#include<stdio.h>
main()
{ int speed;
printf("Enter the speed normally you drive at:\n");
scanf("%d",&speed);
switch(speed)
{
 case 40: printf("Fine, it is moderate\n");
          break;
 case 45: printf("Excellent maintain & save petrol\n");
          break;
 case 30: printf("Scope for improvement\n");
          break;
 case 70: printf("Dangerous!\n");
```

```
          break;
 case 20: printf(" Better buy a Bicycle!\n");
          break;
 default: printf("No comments!\n");
 }
 getch();
 }
```

output

```
Enter the speed normally you drive at :
20
```

Better buy a Bicycle!

The **Switch** construct can not be a complete replacement for **if..else** construct. Because in certain situations where in we have to take decisions on range of values, **if.else** is only the choise. Further **switch** can not have **case** like the one written below,

```
 case>=20<=35:
 case>=40:
```

Mixing up of integer constants and character constants in a single **switch** is also permitted. But, such switch statements lacks clarity.

Try Yourself

1. Write a program which will identify whether an alphabet typed is a oval or a consonant. (ovals → a,e,i,o,and u rest are consonants).
2. Make a menu for a grocery shop , which will facilitate selection of any item, and calculates the amount to be paid. (Use the format shown, assume suitable unit price for the items)

   *****MENU*****

   ABC GROCERY SHOP
   1. SOAP
   2. RICE
   3. WHEET
   4. DETERGENTS
   5. BISCUIT

Session 9. Using Arrays

It is well known that, we always tend to group similar objects into common units. When we buy books we put them in boxes, if it is apple, we put them in baskets. In computer language, we also need to put **data items** of **same type** together. This grouping can be done using **Arrays**. This grouping is done in Mathematics as well as in computer programming. An array may be defined as a group of elements that share a common name and of common data type.

Array Declaration

Like any other variable, array needs to be declared in the beginning of the program. This will enable the compiler to know the type of array and number of elements. The constituents of an array are called **elements**. An array containing 20 elements ,all of them being integers is declared as;

```
        int score[19];
```

Here, **int** specifies the **type** of array and the word **score** specifies the **name** of the array. The number 19 tells, that 20 score values will be stored in array[because count starts from 0]. The bracket**[ ]**tells the compiler that we are declaring an array.

Referring Elements of an Array

When we use an array in a program, the need of accessing individual elements would definitely arise. This is done with **subscript** . Thus if we write **score[2]** , we are referring to **third** element of the array and not the second element. This is because the numbering of elements(positioning) starts from **zero**. The number used for specifying a particular element in an array is called **subscript**.

Entering Data into an Array

The feature that makes arrays such powerful tools is that the subscript used to refer a particular element can be an integer expression. For example, if **j** is an integer variable which can be given series of values, score[j] becomes a general referral , as j keeps on changing, score[j] keeps on referring the element in the respective jth position. This technique is used in taking data into an array. The following example will make this point clear.

```
int arr[10],j
                 for(j=0; j<10; j++)
{ printf(" enter the elements of array arr\n");
   scanf("%d",&arr[j]);
}
```

The **for** loop causes the process of asking and receiving the elements of array **arr** from the user repeatedly 10 times. The first time through the loop **j** has the value **0**, so the scanf statement will cause the value typed in key board to be stored in the memory named **arr[0]**. Thus arr[0] is the first element. This process is repeated till **j** becomes **9** . When **arr[9]** gets filled up, the array **arr** will have gathered **10** values as elements.

The variable j in the loop is called **index variable**. Because, it is used to **index** the desired element of the array.

Array Manipulations

When we use an array in a program, we need to manipulate with it to arrive at desired result. The manipulation may be initialisation, duplication ,arithmetic operation, searching, sorting and so on. Type of manipulation depends on the requirement. We will deal with all types of manipulations of arrays in subsequent examples.

Array Initialisation

Some times , it is desirable to keep values in array locations before program execution. Initialisation is done at the stage of declaring an array. The following examples will demonstrate as to how arrays can be given predefined values.

```
   int score[6]= { 22,45,67,88,90,96,97};
   int score[ ]=  {33,44,66,88,91};
   float rate_int [ ] ={12.5,13.25,11.50,17.00 };
```

Following points are important,

- If array elements are not supplied with specific values , they simply hold some **garbage** values.
- The array size[**dimension]** need not be written separately, it is optional as in the second and third example above.

Example 1. The following program sets all elements of array a to zeros, b to variable x, c to numbers from 1 to 5 and creates a copy of array d in s .

```
#include<stdio.h>

main()
{
 int i, a[10],b[10],c[10],d[10],s[10],x;
 /* Read a variable x */
 printf("Enter value for x:\n");
 scanf("%d",&x);
 for(i=0;i<5;i++)
 { printf("Enter d[%d]:",i); /*prompt to enter elements*/
   scanf("%d",&d[i]);
   a[i]=0;
   b[i]=x;
   c[i]=i;
   s[i]=d[i];
 }
 for(i=0;i<5;i++)
 printf("%3d%3d%3d%3d%3d\n",a[i],b[i],c[i],d[i],s[i]);
 getch();
 }
```

output

```
Enter value for x
12
Enter d[0]: 10
Enter d[1]: 11
Enter d[2]: 12
Enter d[3]: 13
                                    Enter d[4]: 14
   0 12  0 10 10
   0 12  1 11 11
   0 12  2 12 13
   0 12  3 13 13
   0 12  4 14 14
```

Explanation

- Five arrays **a,b,c,d** and s are declared to be of integer type. Variable **x** is read.
- In the for loop the elements of array **d** is read one by one and then array **a** is initialised with 5 **zeros**, **b** with variable **x**, **c** gets filled up from numbers 0 to 4, **s** gets what **d** has in its store.

Accumulation of array Elements

This is a type of manipulation process where in all the elements of an array is accumulated in to a single(**discrete**) variable. Examples 2 and 3 will explain this.

Example 2: This program accumulates the product of all the elements of an array in to a single variable.

```
#include<stdio.h>
main()
{
   int a[]={ 2,3,4,5,7};
   int p=1,i;
   for(i=0;i<5;i++)
   p*=a[i];
   printf("The product of all the elements array=%d\n",p);
   getch();
   }
```

Output:

```
The product of all the elements array=840
```

Explanation

- **Array a is** initialised **with 5 elements.**
- Accumulator **p is initialised to 1.**
- **In the** for **loop all the elements gets multiplied and stored in** p(**p*=a[i] is p=p*a[i]).**

Example 3: In this example,two arrays a and b are summed in such a way that,
sum= a[0]+b[0]+a[1]+b[1]+…

```
#include<stdio.h>

main()
{
 int sum=0,i,a[5],b[5];
 printf("Enter the elements of array a then b\n");
 for(i=0;i<5;i++)
 { printf("a[%d]:",i);
   scanf("%d",&a[i]);
   printf("b[%d]:",i);
   scanf("%d",&b[i]);
 }
 for(i=0;i<5;i++)
 sum=sum+a[i]+b[i];
 printf("Sum of all the elements of a&b:%d\n",sum);
 getch();
 }
```

Output

```
Enter the elements of array a then b
a[0]:10
b[0]:10
a[1]:10
b[1]:10
a[2]:10
b[2]:10
a[3]:10
b[3]:10
a[4]:10
b[4]:10
Sum of all the elements of a &b:100
```

Reversing Arrays

If a **is an array of size** n **, where** n **has been defined previously, and if we want to reverse this array for some reason, we need to interchange a[0] with a[n-1], a[1] with a[n-2] so on and so forth. If an array has 5 elements for example, the number of exchanges needed is just 5/2 i.e., 2. This is explained by the figure.**

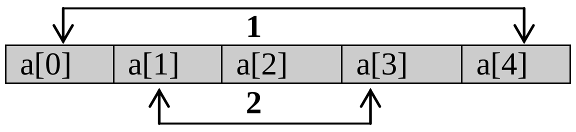


Figure 6.1: Reversing of array of 5 elements.

Example 4 will make the procedure of reversing clear.

Example 4: An array containing n number of elements is reversed in this program.

```
#include<stdio.h>

main()
{
 int a[100],i,j,temp,k,ne;
 printf("How many elements?\n");
 scanf("%d",&ne);
 printf("Enter the elements one after other\n");
  for(i=0;i<ne;i++)
  scanf("%d",&a[i]);
/*Reversing is done now*/
  for(i=0;i<ne/2;i++)
 { temp=a[i];
  /* k will generate ne-1,ne-2 ………… */
   k=ne-1-i;
   a[i]=a[k] ;
   a[k]=temp;
 }
 printf("Array after reverse\n");
 for(i=0;i<ne;i++)
 printf("%d\n",a[i]);
 getch();
 }
```

Output

```
How many elements?
7
Enter the elements one after the other
1
2
3
4
5
6
7
Array after reverse
7
6
5
4
3
2
1
```

Explanation

- **The number of the elements (ne) to be held by array** a **is input by user and the elements are taken in to the array** a**.**
- **The** for **loop does the process of exchange** ne/2 **number of times.**
- **The variable temp holds a[0] and a[0] is freed, k will have ne-1 that is 6 in this example. Then a[0] is filled with a[k] i.e., a[6], the process continues 3 times.**

Array Merging

Suppose a **and** b **are two arrays of size 5, and if we want another array** c **to contain elements such that,**
c[ ] ={ a[0],b[0],a[1],b[1],………………}
Then we say that both arrays are merged in to c**. Array** c **contains 10 elements.**

Example 5: This program will merge array a and array b in to c.

```
#include<stdio.h>
main()
{
 int a[5],b[5],c[10],i,k;
 printf("Enter five elements of array a\n");
 for(i=0;i<5;i++)
   scanf("%d",&a[i]);
  printf("Enter five elements of array b\n");
 for(i=0;i<5;i++)
   scanf("%d",&b[i]);
 /* Merging is done in this loop */
   k=0;
 for(i=0;i<5;i++)
 { c[k]=a[i];
   k++;
   c[k]=b[i];
   k++;

   }
   printf("The merged array\n");
   for(i=0;i<10;i++)
   printf("%3d",c[i]);
   getch();
   }
```

Output:

```
Enter five elements of array a
1 2 3 4 5
Enter five elements of array b
6 7 8 9 10
The merged array
```

```
  1   6   2   7   3   8   4   9   5  10
```

Explanation

- **In the first succession the elements of array** a **and** b **are taken in.**
- **Before entering in to for loop a variable** k **is initialised to 0, this variable** k **will help in indexing the elements of** c**. In the first cycle of the loop,** c[0]=a[0]**,** k **increments to 1 then** c[1]=b[0], k **further increments to 2 and the loop continues till all 10 elements gets filled into** c**.**

Before we go for other types of manipulations on arrays , we shall do few more programs which will enable us to apply the manipulations we learnt till now.

Example 6. In this program 10 integers(+ve,-ve and 0s) are read and sum of positive elements, negative elements is printed. Also the program lists the number of elements above average and below average.

```
#include<stdio.h>

main()
{
 int a[10],sum_neg=0,sum_pos=0,i,more_avg,less_avg;
 float avg;
 printf("Enter 10 integer values(+,-&0)\n");
 for(i=0;i<10;i++)
 scanf("%d",&a[i]);
 for(i=0;i<10;i++)
 { if(a[i]<0)
   sum_neg+=a[i];
   if(a[i]>0)
   sum_pos+=a[i];}
   avg=(sum_neg+sum_pos)/10.0;
   printf("The average of all the elements:%f\n",avg);
   more_avg=0;
   less_avg=0;
   for(i=0;i<10;i++)
   { if(a[i]>avg)
     more_avg++;
     else
     less_avg++;
   }
   printf("The sum of positive elements:%d\n",sum_pos);
   printf("The sum of negative elements:%d\n",sum_neg);
   printf("The average of elements:%f\n",avg);
   printf("Elements above average:%d\n",more_avg);
   printf("Elements below average:%d\n",less_avg);
 getch();
 }
```
output:
```
Enter 10 integer values(+,- &0)
```

```
10
-5
-2
3
4
0
0
-4
12
10
The sum of positive elements:39
The sum of negative elements:-11
The average of elements:2.800000
Elements above average:5
Elements below average:5
```

Explanation:

- **Ten integer numbers are taken in to the array** a.
- **All the elements are checked to verify whether they are more than zero(+ve) or less than zero(-). Positive elements will get accumulated into variable** sum_pos **while variable** sum_neg **takes care summing all negative elements.**
- **Average of all the elements is calculated . Elements are once again compared with average value to categorise them into more than average or less than average. Correspondingly, variables** more_avg **and** less_avg **will make a count of them.**

Example 7: This program inputs n real numbers and finds average, standard deviation and variance.

Standard deviation is given by,

$$\sigma = \sqrt{\frac{\sum \left(x_i - \bar{x}\right)^2}{n}}$$

variance is v = σ^2

```
#include<stdio.h>
#include<math.h>
main()
{
 float a[100],avg,std_dev,var,numer,sum=0.0;
 int ne,i;
 printf("How many elements?\n");
 scanf("%d",&ne);
 printf("Enter the elements one below other\n");
 for(i=0;i<ne;i++)
 scanf("%f",&a[i]);
 /* calculating average */
 for(i=0;i<ne;i++)
  sum+=a[i];
  avg=sum/ne;
  numer=0.0;
 /* Calculating standard deviation */
 for(i=0;i<ne;i++)
 numer+=(a[i]-avg)*(a[i]-avg);

   std_dev=sqrt(numer/ne);
   var= std_dev*std_dev;
   printf("Average of all the elements:%f\n",avg);
   printf("Standard deviation :%f\n",std_dev);
   printf("Variance :%f\n",var);
   getch();
   }
```

Output

```
How many elements?
6
Enter the elements one below the other
2
2
2
2
2
2
Average of all the elements :2.000000
Standard deviation:.000000
Variance:.000000
```

Explanation

- **Number of elements given by the user is taken by** ne.
- **Elements are taken and summed up using the accumulator sum. Average is found out.**
- **Numerator is generated by** numer **, which accumulates square of the difference between individual elements and average.**
- **Standard deviation is then calculated (std_dev=sqrt(numer/ne);). Finally variance is calculated.**

Example 8: This program reads an array of n elements and finds maximum and minimum among them and their positions.

```
#include<stdio.h>
main()
{
 int a[10],max,min,pos_max,pos_min,i,ne;
 printf("Enter number of elements\n");
 scanf("%d",&ne);
 printf("Enter elements one by one\n");
 for(i=0;i<ne;i++)
 scanf("%d",&a[i]);
 max=a[0];
 min=a[0];
 pos_max=0;
 pos_min=0;
 for(i=1;i<ne;i++)
 {
  if(a[i]>max)
    { max=a[i];
      pos_max=i;
    }
  if(a[i]<min)
    { min=a[i];
      pos_min=i;
    }
  }
  printf("Maximum of all the elements is %d in position
%d\n",max,pos_max);
  printf("Minimum of all the elements is %d in position
%d\n",min,pos_min);
  getch();
  }
```

Output:

```
Enter number of elements
5
Enter elements one by one
43
12
67
88
5
Maximum of all the elements is 88 in position 3
Minimum of all the elements is 5 in position 4
```

Explanation:

- **After having read the array, the very first element is assumed to be the maximum as well as the minimum. The position being zero in both the cases. The maximum and minimum values so assumed is compared with rest of the elements. However, the comparison is effected independently by two if statements.**

Array sorting

When we look at the raw data, our first impulse is to arrange them in order(ascending/descending) so that looking at them becomes a great deal easier.Sorting is one of the most practised form of data processing(for directories,tax table, book retrieval, identification so forth..). Arrays lend themselves to this type of manipulation. Though we have many methods of sorting, we shall look into two widely used methods. Namely,

- Bubble sort
- Selection Sort

Bubble sort

The method resembles the situation of raising of an air bubble entrapped in water column. If the air bubble is at the bottom of the water body, it replaces heavier water molecules and raises above. On the other hand, if a drop of mercury is placed on the top of water it goes down replacing lighter molecules of water. We shall see how the numbers are arranged by bubble sort method by taking an example.

If the numbers 4,5,3,2 are to be arranged in ascending order, the method goes like this;

Pass 1:

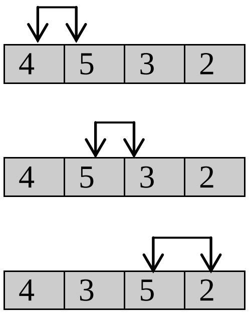


Fig 6.1 (a) Bubble sort 1st pass.

We can see from the above sketch that, in the first pass, 4,3,2,5 **comes out. In the first pass three comparisons are made. At the end of this pass, the highest number will be shifted to right most corner.**

Pass 2:

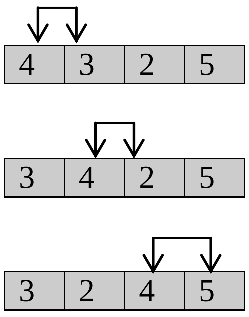


Fig 6.1 (b) Bubble sort 2nd pass.

After second pass, 3,2,4,5 **comes out. In this pass again three comparisons are being made. However last comparison was not needed.**

Pass 3:

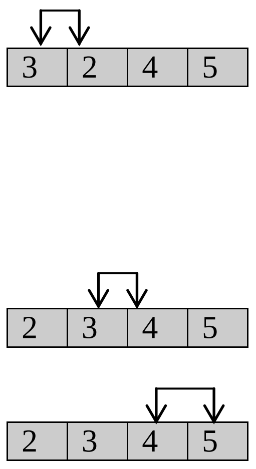


Fig 6.1c. Bubble sort 3rd pass.

Same procedure is carried out for third pass also. However , in the third pass, we only needed to compare the first pair of numbers, since the last two numbers are already in order.

Finally , it can be concluded that ,if n elements are there in an array n-1 passes are required . And if passes are counted cyclically by a variable j, then in each pass n-j comparisons are needed.

Before we write complete program to sort array of four elements in ascending order, let us just write the code to take care of just the first pass(moving the largest to the right –most position). The code for this is;

```
for( i=0; i<3 ; i++)
{ if(a[i]>a[i+1])
     temp=a[i];
     a[i]=a[i+1];
     a[i+1]=temp;
}
```

→ The upper limit for i is 2, when i is 2, the comparison takes place between a[2] and a[3] which is a[i +1]
→ a[i] and a[i +1] are inter changed if a[i +1] is less than the previous number a[i].

The above segment of the program will move the largest number in the array a **into the right most position. To sort the entire array, we need to make three passes through the array as explained previously. We use the following code to conduct three passes.**

```
for( j=0 ; j<3; j++)
 for (i=0;i<3;i++)
 { if a[i]>a[i+1])
    { temp=a[i];
a[i]=a[i+1]
```

```
      a[i+1]=temp ;  }
       }
```

The above code is very easy to remember, but this segment is not efficient. This is because, in the second pass , the program compares a[2] with a[3] when it is established that a[3] is the largest. This also happens in the third pass, a[1] is compared with a[2] when it is already known that a[2] is greater than a[1]. S o it would be better to stop the second pass at a[2] and stop third pass at a[1]. This can be achieved by making index i in the inner for loop to be 3 for first pass, 2 for the second pass and one for the third pass. To do this we have to replace for(i=0;i<3;i++) **with** for(i=0;i<3-j;i++) **since** (3-j) **will generate the numbers 3,2 and 1. The complete program is appended in example 8.**

Example 9: This program checks whether a given array of n elements is arranged in ascending or descending order or unsorted and appropriately prints messages.

```
#include<stdio.h>
main()
{
  int  a[100],i,n,asc=0,dsc=0;
printf("Enter number of elements\n");
scanf("%d",&n);
printf("Enter the elements one by one\n");
for(i=0;i<n;i++)
scanf("%d",&a[i]);
/* checking is done now*/
for(i=0;i<n-1;i++)
{ if(a[i]>a[i+1])dsc=1;
  if(a[i]<a[i+1])asc=1;
  if(asc==1 && dsc==1)
   {printf("Array is not sorted");
     exit();
    }

}/*end of for */

 if(asc==1)
 printf("Array in ascending order");
 else if (dsc==1)
 printf("Array in descending order");
 else
 printf("All the elements are equal");
 getch();
 }
```

Output:
First run

```
Enter number of elements
4
Enter elements one by one
4
```

```
5
6
7
Array in ascending order
```
Second run
```
Enter number of elements
4
Enter elements one by one
4
1
6
2
Array is not sorted
```
Explanation:

After reading the elements of the array, the for **loop takes care of comparing each element with its succeeding element. Since** n-1 **comparisons are only needed, the for loop is limited to** n-1 **iterations. The variables** asc **and** dsc **are used as check counters with initial value of zero.**

If an element is more than its immediate next , the counter dsc **becomes equal to 1. On the other hand if an element is less than its immediate next, the counter** asc **becomes 1. If both** asc **and** dsc **aquires a value equal to 1 in two or three consecutive iterations then it could be taken to mean that the array is not sorted. Therefore, the program terminates after printing the appropriate message.**

After completion of checking if asc **alone is 1 , then array must be in ascending order, if** dsc **alone is 1 then the array must be in descending order or otherwise all the elements must be equal.**

Example 10: This program reads n integer values and arranges them in ascending order using bubble sort.

```
#include<stdio.h>
main()
{
 int ne,i,j,arr[100],temp;
 printf("How many elements?\n");
 scanf("%d",&ne);
 printf("Enter the elements\n");
 for(i=0;i<ne;i++)
 scanf("%d",&arr[i]);
 printf("The array before sorting\n");

 for(i=0;i<ne;i++)
 printf("%4d",arr[i]);
 /* Bubble sorting takes place now */
 for(i=0;i<ne-1;i++)
 for(j=0;j<ne-1-i;j++)
 { if(arr[j]>arr[j+1])
    { temp=arr[j];
    arr[j]=arr[j+1];
    arr[j+1]=temp; }/* end of if*/
 } /* end of loop */
 printf("\nArray after sorting\n");
 for(i=0;i<ne;i++)
 printf("%4d",arr[i]);
 getch();
 }
```

output:

```
How many elements?
7
12 -5 33 56 2 6 28
The array before sorting
  12 -5  33  56   2   6  28
Array after sorting
  -5   2   6  12  28  33  56
```

Explanation

- **The number of elements taken is ne.**
- **Notice that the outer** for **loop is limited to less than** ne-1 **passes, while the inner** for **loop is limited to less than** (ne-1-j) **comparisons.**

Selection sort

In this method, the smallest(or biggest) element in the array is exchanged with the first element. The next small element in the array is shifted to second position and the process continues till all the elements are arranged in ascending pattern. The process is explained by means of figure 6.2.

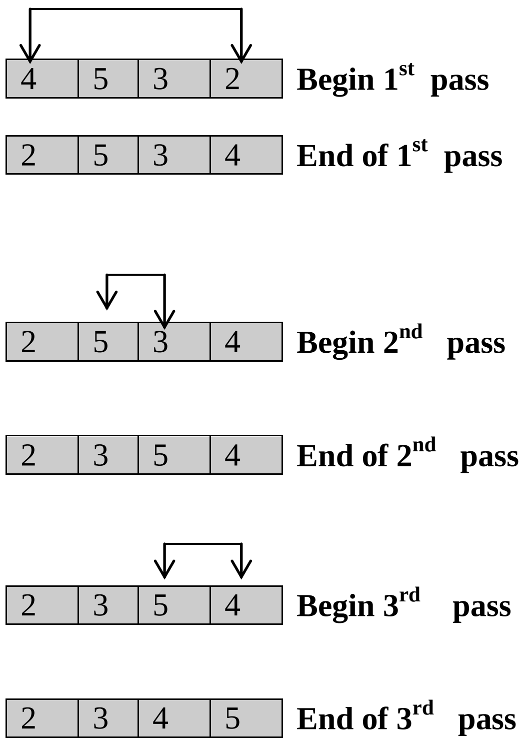


Fig: 6.2 Selection sort.

To understand the process still better, let us write the program segment which sorts four numbers in ascending order. In the first instance we shall write the code just for the first pass. The pass determines the position of the smallest score of the array and interchanges the smallest score in the first position of the array.

```
pos=0;    /* assume the smallest is in position 0 */
for(i=1 ; i<4 ;i++)
if(a[pos]>a[i])
   pos=i;
 temp=a[i];
 a[i]=a[pos];
 a[pos]=temp;
```

In the above segment if a[0] **is greater than** a[1]**, then the position has to be changed. So** pos **becomes equal to 1. If the element in the position 1 is not greater than that at position 0 then position remains to be 0 only. This will be checked for all the positions. Finally when the looping is over, the swapping is done to replace the smallest element to the first place and the element in the first place shall go to the location from where the smallest is picked.**
The code explained above takes care of just the first pass. At the end of the first pass, the smallest element will be in position 0. So we start looking for the next smallest score at position 1 of the array. Hence prior to the second pass the position pos **must be set to 1. At the finish of the second pass, the interchanging should be done with the element in the second position. The process continues. The complete code is given in**

Example 11. This example sorts n elements of integer type by selection sort.

```
#include<stdio.h>
main()
{
 int arr[100],ne,temp,max,pos,i,j;
 clrscr();
 printf("How many elements?\n");
 scanf("%d",&ne);
 printf("Enter the elements\n");
 for(i=0;i<ne;i++)
 scanf("%d",&arr[i]);
 printf(" Array before sorting\n");
  for(i=0;i<ne;i++)
  printf("%4d",arr[i]);
  /* Selection sort takes place now */
   for(i=0;i<ne-1;i++)
  {   pos=i;
     for(j=i+1;j<ne;j++)
     if(arr[pos]>arr[j])
      pos=j;
     temp=arr[i];
     arr[i]=arr[pos];
     arr[pos]=temp;
   } /* end of outer for */
   printf("\nArray after sorting\n");
    for(j=0;j<ne;j++)
    printf("%4d",arr[j]);
    getch();
    }
```

Output:

```
How many elements ?
5
Enter the elements
-4
15
2
33
4
Array before sorting
  -4 15  2  33   4
Array after sorting
 -4  2  4  15  33
```

Explanation:

- **The number of elements are designated as** ne.
- **As per the method , the outer** for **loop makes** ne-2 **(less than** ne-1**) exchanges . While the inner** for **loop determines the correct position where the smallest element in that succession is residing.**

Array searching

Some times we may need to find the presence or absence of a particular element in an array. For example we may need to find those students who have got less than 35% of attendance in a subject. We may be interested in searching a particular phone number and so on. There are two important methods of searching. Namely;

- **Linear Search**
- **Binary Search**

Linear Search

Linear search uses the concept of matching the element to be searched with each element in the array. If the item to be searched matches with any of the elements then search is completed and the item is available in array. If none of the elements are matching with the item, then item is not available in the array.

Binary search

Binary search is designed to reduce the number of matching (comparisons). To adopt this search technique, the elements must be sorted in an order(ascending or descending). If a[0,1,2,3………n) is an array of ordered list of elements in ascending order. Let key **be the element to be searched in the array. The following steps are adopted to implement binary search.**

If a[mid] **is the middle element of the array, then** key **may be;**
< a[mid] **in this case, key can be searched in the subarray a[0,1,2,3…..mid]. Again mid value for this is mid/2.**
>a[mid] **in this case, key may be searched in the subarray a[mid,mid+1…..n]. Again mid value for this is (n+mid)/2**
=a[mid]**, in this case key is found. Searching is stopped.**

The above steps are repeated changing the value of mid **until** key **is found or search in the entire array is exhausted. At each stage the number of elements is decreased by half. Therefore binary search is faster than linear search.**

We shall implement both the methods in examples 10 and 11.

Example 12. This program inputs n integers, conducts a linear search for a given key number and reports success or failure.

```
#include<stdio.h>
main()
{
 int arr[100],i,key,ne,pos,scount;
 printf("How many elements?\n");
 scanf("%d",&ne);
 printf("Enter the elements\n");
 for(i=0;i<ne;i++)
 scanf("%d",&arr[i]);
 printf("Enter the element to be searched\n");
 scanf("%d",&key);
 /* Linear search takes place now */
 scount=0;
 for(i=0;i<ne;i++)
 if(arr[i]==key)
 { scount=1;
   pos=i; }
 if(scount==1)
 printf("The element is found in location:%d\n",pos);
 else
 printf("Element not found\n");
 getch();
 }
```

output:

First run:

```
How many elements?
5
Enter the elements
12 22 34  1 2
Enter the element to be searched
22
The element is found in location:1
```

Explanation

- **The number of elements as specified by** ne **is read into an array. Then the element to be searched is named as** key**.**
- **In the for loop each and every element is matched with** key **. If a particular element matches search count** scount **becomes 1 and position is also recorded by** pos**. If the element is not found the** scount **remain to be zero. The value of** scount **is tested outside the do loop after matching of all the elements with key is over.**

Example 13. This program sorts the array and searches for a given key value using binary search technique. Also it makes a count of number of probes.

```
#include<stdio.h>
main()
{
  int arr[100],ne,i,j,key,mid,first,last,temp,np;
  printf("Enter number of elements\n");
  scanf("%d",&ne);
  printf("Enter the elements\n");
  for(i=0;i<ne;i++)
  scanf("%d",&arr[i]);
  /* bubble sorting done now*/
  for(i=0;i<ne-1;i++)
  { for(j=0;j<ne-1-i;j++)
   if(arr[j]>arr[j+1])
    { temp=arr[j];
      arr[j]=arr[j+1];
      arr[j+1]=temp; } /* end of if*/
  }
   printf("Array after sort\n");
   for(i=0;i<ne;i++)
   printf("%4d",arr[i]);
   /* Binary search*/
   printf("\nEnter the element to be searched\n");
   scanf("%d",&key);
   last=ne-1;
   first=0;
   np=0;
   do
   { mid=(last+first)/2;
```

```
      np++;
      if(key<arr[mid])
      last=mid-1;
      else
      first=mid+1;
      } while(!(key == arr[mid] || first > last));

    if(key==arr[mid])
   { printf("Element is found\n");
    printf("Searching has taken %d probes\n",np);}
    else
    printf("Element not found\n");
    getch();
    }
```

Output:

```
Enter the number of elements
5
Enter the elements
12 33 45 -2 10
Array after sort
  -2 10 12 33 45
Enter the element to be searched
12
Element is found
Searching has taken 3 probes
```

Explanation

- **After taking ne number of elements, they are sorted in ascending order by bubble sort.**
- **The key element to be searched is read.**
- **Then mid value of the array is found out. The position is identified by comparing the value of** key **with the** mid **element of array. This checking is done with in the loop. The number of elements with in which searching is to be done goes on reducing since the value of** first **or** last **keeps on changing. Finally the loop will end when** key **is equal to** a[mid**] or when** first **exceeds** last**. If** key **is not present , the value of** first **will become more than the value of the** last**(This can be verified by taking a specific example).**

Two Dimensional array

We worked with arrays of single dimension. It is possible for arrays to have two or more dimensions. We shall go through two dimensional arrays only. Two dimensional array is called a matrix. A two dimensional array will posses two subscripts.

Declaration

The declaration of two dimensional array is done in the format shown below,

```
     int a[3][2];
    float x[3][3];
```

The declaration would mean that array a **has** 4 **rows and** 3 **columns and is of integer type. While array** x **has** 4 **rows and** 4 **columns and is of float type. In total array** a **has** 12**(4x3) elements while array** x **has** 16 **(4x4)elements.**

Initialisation

Two dimensional arrays may be initialised in the same way as that of one dimensional arrays. The format is shown below.

```
  Int a[3][2] = {
                   { 120, 55,56};
                   { 150,37,57};
                   { 55, 77 ,58 };
                   {44, 44 ,59 };
                 };
```

Reading and writing of matrix

We may recall that one for **loop was just enough to read elements in to array and to output one dimensional array. Here, we need two** for **loops nested one in other. The format for reading and writing is done as per the format shown.**

For reading elements

```
 for ( i=0 ; i< 3 ;i++)              /* Elements are taken row wise . Sequence is
   for(j =0; j<2 ; j++)                  a[0][0],a[0][1]
   scanf(" %d",&a[i][j]);              a[1][0],a[1][1]
                                          a[2][0],a[2][1]
```

While entering the elements can be entered side by side with a space or one below the other by hitting enter key.

For writing elements

Here again we have to use two for loops . The outer for loop for generating rows and the inner for loop for generating columns. The format is;

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

This format will print the array in matrix form with clear display of rows and columns.

We shall now go through some examples to explore the handling of two dimensional arrays.

Example 14: Let us write a program to add two matrices.

```
#include<stdio.h>

main()
{
 int a[20][20],b[20][20],c[20][20],i,j,k,l,n,m;
 printf("Enter the order of two matrices (mxn)\n");
 scanf("%d%d",&m,&n);
 printf("Enter the elements of a & b row wise\n");
 for(i=0;i<m;i++)
  for(j=0;j<n;j++)
   { printf("Enter a(%d,%d)",i,j);
     scanf("%d",&a[i][j]); }
 for(i=0;i<m;i++)
  for(j=0;j<n;j++)
   { printf("Enter b(%d,%d)",i,j);
     scanf("%d",&b[i][j]); }
/* Addition takes place now */
 for(i=0;i<m;i++)
  for(j=0;j<n;j++)
   c[i][j]=a[i][j]+b[i][j];
 printf("The resultant matrix\n");


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

getch();
}
```

Output:

```
Enter the order of two matrices (mxn)
2
3
Enter a[0,0]:1
Enter a[0,1]:2
Enter a[0,2]:3
Enter a[1,0]:4
Enter a[1,1]:5
Enter a[1,2]:6
Enter b[0,0]:1
Enter b[0,1]:2
Enter b[0,2]:3
Enter a[1,0]:4
Enter a[1,1]:5
Enter a[1,2]:6
The resultant matrix
   2   4    6
   8  10   12
```

Explanation

- **The matrix size is taken as 2 x 3 (m=2, n=3). Then the elements of both arrays** a **and** b **are taken in same style(one below the other).**
- **With in the** for **loops the** row **elements of** c **is generated by adding row elements of** a **and** b **respectively.**

Example 15. This program sums up all the row elements and column elements of an a matrix.

```
#include<stdio.h>

main()
{
  int a[10][10],ro_sum[10],co_sum[10],j,i,k,m,n;
  printf("Enter the size of matrix(mxn)\n");
  scanf("%d%d",&m,&n);
  printf("Enter the elements row wise\n");
 for(i=0;i<m;i++)
  for(j=0;j<n;j++)
  scanf("%d",&a[i][j]);
  printf("The matrix is\n");
  for(i=0;i<m;i++)
  {for(j=0;j<n;j++)
   printf("%4d",a[i][j]);
   printf("\n"); }
 /* Row sum is done now */
   for(i=0;i<m;i++)
  { ro_sum[i]=0;
   for(j=0;j<n;j++)
   ro_sum[i]=ro_sum[i]+a[i][j];}
   for(i=0;i<m;i++)
   printf("Sum of row no %d:=%d\n",i,ro_sum[i]);
  /* Column sum is done now*/
   for(j=0;j<n;j++)
  { co_sum[j]=0;
   for(i=0;i<m;i++)
   co_sum[j]=co_sum[j]+a[i][j];}
```

```
    for(j=0;j<n;j++)
    printf("Sum of coloumn no %d:=%d\n",j,co_sum[j]);
    getch();
    }
```

Output

```
Enter the size of the matrix(mxn)
3  2
Enter the elements row wise
10 11
12 13
14 15
The matix is
10 11
12 11
14 15
Sum of row no 0:=21
Sum of row no 1:=25
Sum of row no 2:=29
Sum of column no 0:=36
Sum of column no 1:=39
```

Explanation:

- **The size of the matrix is taken as (mxn i.e. 3x2). The elements are entered row wise with a space in between.**
- **The summing of rows is done with in for loops as explained below,**
  For i=0, ro_sum[0]=ro_sum[0] + a[0][0]+a[0][1]
  For i=1, ro_sum[1]=ro_sum[1] + a[1][0]+a[1][1] **so on…**
- **The column sum is done in different set of for loops as explained below,**
  For j=0 co_sum[0]= co_sum[0] + a[0][0]+ a[1][0]+a[2][0]
  For j=1 co_sum[1] = co_sum[1] + a[0][1]+ a[1][1]+a[2][1]
- **Finally the sums of all the rows and columns are displayed as a one dimensional array.**

Example 16: program generates a Pascal triangle

```
#include<stdio.h>
main()
{
 int a[20][20],i,k,j,nr;
 clrscr();
 printf("How many rows in pascal Triangle?\n");
 scanf("%d",&nr);
 for(i=0;i<nr;i++)
 { a[i][i]=1;
   a[i][0]=1;}
  for(i=0;i<nr;i++)
  { for(j=0;j<nr;j++)
    if(j!=0&&i!=j)
    a[i][j]=0;
       }
       printf("\n\n*******PASCAL TRIANGLE*******\n");
  for(i=1;i<nr;i++) /* for generating nr rows*/
 { for(j=1;j<i;j++) /* for generating i elements */
  a[i][j]=a[i-1][j-1]+a[i-1][j];
  }
/* for printing pascal triangle*/
  for(i=0;i<nr;i++)
  { for(j=0;j<=i;j++)
 printf("%5d",a[i][j]);
 printf("\n");
 }
getch();
}
```

Output:

```
How many rows in pascal Triangle?
6

*******PASCAL TRIANGLE*******
1
1    1
1    2    1
1    3    3    1
1    4    6    4    1
1    5   10   10    5   1
```

Explanation:

A Pascal triangle looks like the one shown in the output. The extent of the triangle can be limited by the number of rows of numerals. Thus the number of rows is being made the input.

The outermost for loop will generate the number of rows sought. While the inner for loop will restrict the number of elements in each row. The inner for loop also meant for generating actual numerals. This is effected by the statement;

a[i][j]=a[i-1][j-1]+a[i-1][j]

The next set of nested for loops will enable the pascal triangle to be printed.

Example 17: This program reads matrix of size m an n of a[m][n] and p and q of b[p][q], checks if they are multipliable . Further, if it is multipliable, it calculates trace of the resultant matrix, if trace really exists.

Pre-processing of the problem

For two matrices to be multipliable, we know that the number of columns of a **should match with number of rows of matrix** b.

If a[5][4] and b[4][3] are multiplied then resulting matrix will be of size 5 x 3. Matrix multiplication is recalled here.

$$\begin{bmatrix} a_{00}a_{01}a_{o2}a_{03} \\ a_{10}a_{11}a_{12}a_{13} \\ a_{20}a_{21}a_{22}a_{23} \\ a_{20}a_{31}a_{32}a_{33} \\ a_{4o}a_{41}a_{42}a_{43} \end{bmatrix} x \begin{bmatrix} b_{00}b_{01}b_{02} \\ b_{10}b_{11}b_{12} \\ b_{20}b_{21}b_{22} \\ b_{30}b_{31}b_{32} \end{bmatrix} = \begin{bmatrix} c_{00}c_{01}c_{02} \\ c_{10}c_{11}c_{12} \\ c_{20}c_{21}c_{22} \\ c_{30}c_{31}c_{32} \\ c_{40}c_{41}c_{42} \end{bmatrix}$$

[m][n] x [n][q] = [m][q]

Where,
C_{00} = a_{00} * b_{00} + a_{01} * b_{10} + a_{02} * b_{20}+ a_{03} * b_{30}
C_{10} = a_{10} * b_{00} + a_{11} * b_{10} + a_{12} * b_{20}+ a_{13} * b_{30}
- - - - - - -
- - - - - - -

The general equation for the resultant matrix is;

$$c_i k = \sum_{i=0}^{m-1}\sum_{j=0}^{q-1}\sum_{k=0}^{n-1} a_i kXb_k j$$

We need three for **loops housed one inside the other. The** outermost **will generate the** rows **of resultant matrix, the** central **for loop will generate** columns **while the** inner most **for loop will generate the** elements **of the resultant matrix.**

The trace **of the matrix is the sum of the elements of the principal diagonal of the matrix which runs from left top corner to right bottom corner. Therefore, trace can be obtained only from a square matrix.**

Program follows now.
```
#include<stdio.h>

main()
{
  int a[10][10],b[10][10],c[10][10];
  int i,j,k,l,m,n,p,q,trace;
  printf("Enter  order of the first matrix a (mxn)\n");
  scanf("%d%d",&m,&n);
  printf("Enter the order of matrix b(pxq)\n");
  scanf("%d%d",&p,&q);
  if(n!=p)
  { printf("The arrays are not multiplicable\n");
    exit(); }
  else
  { printf("The matrices are multiplicable\n");
    printf("Enter elements of the matrix a(row wise)\n");
    for(i=0;i<m;i++)
    for(j=0;j<n;j++)
    scanf("%d",&a[i][j]);
    printf("Enter the elements of matrix b(row wise)\n");
    for(i=0;i<p;i++)
    for(j=0;j<q;j++)
    scanf("%d",&b[i][j]);
    /* Matrix multiplication */
    for(i=0;i<m;i++)
    for(j=0;j<q;j++)
    { c[i][j]=0;
      for(k=0;k<n;k++)
      c[i][j]+=a[i][k]*b[k][j];
    } /* close of inner most for loop */

  } /* close of else */
  printf("The resultant matrix c \n");
  for(i=0;i<m;i++)
  { for(j=0;j<q;j++)
    printf("%8d",c[i][j]);
    printf("\n");
  }
  /* Calculating trace of the matrix */
  if(m!=q)
 { printf("Trace does not exit\n");
  exit();}
  else
  { printf("Trace exists\n");
    trace=0;
    for(i=0;i<m;i++)
    trace=trace+c[i][i];
    printf("Trace is=%d\n",trace);
  }

  printf("Thank You!\n");
  getch();
}
```

Output

First run

```
Enter the order of the matrix a(mxn)
5  4
Enter the order of matrix b(pxq)
4  3
The matrices are multipliable
Enter the elements of matrix a(row wise)
1 2 3 4
6 7 8 9
5 4 3 2
```

```
2 3 4 5
1 2 3 4
Enter the elements of matrix b(row wise)
1 2 3
4 5 6
7 8 7
3 4 5
The resultant matrix
  42   52   56
 117  147  161
  48   62   70
  57   71   77
  42   52   56
Trace does not exist
```

Second run

```
Enter the order of the matrix a(mxn)
3   2
Enter the order of matrix b(pxq)
2   3
The matrices are multipliable
Enter the elements of matrix a(row wise)
1 2
3 4
5 6
Enter the elements of matrix b(row wise)
2 2 2
3 3 3
The resultant matrix
     8      8      8
    18     18     18
    28     28     28
Trace exists
Trace is =54
Thank you !
```

Explanation

- **Two matrices of size mxn and pxq are read , after confirming that they are multipliable. Multiplication is done as per the explanation given prior to program.**
- **The resultant matrix is printed in matrix form.**
- **To find the trace, a verification is made as to find whether the resultant matrix is square or not. If it is a square matrix, trace is calculated and printed . Otherwise the program terminates with an appropriate message as displayed in the first run.**

Points to rehearse

- **An array is a collection of similar type of elements(integers,real values, charactrers). We shall see Character arrays when we deal with strings.**
- **The first element of an array is numbered 0, so that the last element is 1 less than the size of the array. In case of two dimensional array of size [m][n], the first element is [0][0] while the last element is[m-1][n-1].**
- **An array is also called as a subscripted variable.**
- **Before using an array its type and dimension must be declared.**
- **Following examples have wrong coding because,**

```
i.   main()
       { int x(10),j;
            for(j=1;j<=10;j++)
              { scanf("%d",x[j]);
                printf("%d", x[j]);
              }
       }
```

In the for loop j has been initialised to 1, but it should be zero. In declaration [ ] should be used.

```
ii.   main()
       { int j,ne;
             scanf("%d",&ne);
             int arr[100];
             for(j=1;j<=ne;j++)
             { scanf("%d",arr[j]);
               printf("%d",arr[j]);
             }
       }
```

Array can not be declared after scanf.
In for loop the condition must be j<ne and not j<=ne.

- **In an array** int num[5] **represents its size, while** num[5]=11 **represents value of the 6th element.**
- **The rules for naming arrays are the same as that of naming regular variables. However an array can not have the same name as a variable with in the same program.**

Points to Ponder

We may wonder when it is appropriate to use an array in a program and when it is not. The answer is not clear cut. It is some times difficult, even for an experienced programmer to be certain that it will be easier or more efficient to use an array than some other method in a particular program. The answer is determined by the type of processing that is involved, the size of the array and the amount of memory available for storing the array. A general rule of thumb is that the use of arrays is beneficial when we must process a known volume of data of the same type and in the same way many times with in the program.

Arrays can be used almost any where, but they are often used to produce reports. This is because large amounts of similar data can be either collectively handled as in the reading or writing of groups of data items , as in searches or updates.

Placing the data in an array would make it a relatively simple matter to locate a particular data item

Theoretically, there is no limit for the dimensions of an array,. The only practical limits are memory size and the restrictions imposed by the compiler being used.

If array1 and array2 are dimensioned as array1[10] and array2[10] respectively and if array1 is initialised with elements, it is wrong to write as array1=array2 **as it is a gross violation of syntax. One array can not be assigned to another using the assignment operator. Instead each element may be assigned individually.**

Try Yourself

1. **Generate fibbonecci series 1,1,2,3,5,8,13,21,34,…….. up to n terms.**
2. **Read an array of n elements and insert an extra element in the kth position.**
3. **Read a two-dimensional array of size m x m and find its transpose.**
4. **Read a two-dimensional array and find its determinant.**

Session 10

Using Pointers

We have seen variable types that store characters, integers, floating point numbers and so on. These variables are stored in computer memory, which has address. Addresses are mere numbers just in the same way as that of numbers provided for Houses in a town. The numbers start at 0 and go up there. If we have 640 KB of memory, the highest address is 655,359. For 1 MB, it is 1,048,575. Our programs when loaded into memory occupies certain range of these addresses. Hence every variable in a program has a particular address. The **pointers** are special types of variables that hold the address value.

In this session we will see how the address of a variable can be determined. Actually, we have used this technique in previous session also. The **ampersand(&)** immediately preceding a variable name in **scanf** function returns the address of the variable attached to it. This operator is known to be address operator. This operator may be glued only to variable name or array element. Addresses can just be stored in variables. Thus in C, there are variables that tell a computer where data is placed and these variables are pointers.

Declaring a pointer variable

To declare and refer a pointer variable, C provides two special operators **&** and ***** **.** The declaration is done as done below;

```
char *cptr;
int   *iptr;
float  * fptr;
```

The declarations may look funny. The **asterisk** means **pointer to**. Thus the statements written above are understood as follows;

cptr is a pointer to character type variables.
iptr is a pointer to integer type variables.
fptr is a pointer to float type variables.

The ***** mark is not part of the variable name, it is part of the variable type.

The address operator

The address of any variable can be obtained by putting **&** (ampersand) proceeding the variable name. For example, in **&num**, **num** is a variable and **&num** is the address in which the value of **num** is stored. We can see another example to make the concept still clear. Consider the segment of the program;

```
int var;
scanf("%d" ,&var);
```

- Here **var** is an integer type variable.

- Its value is taken through key board which was facilitated by **scanf** function. s**canf** also associates **var** to a particular address.
- **If we have pressed 55 for var , it can be imagined to be seated in its pride of place(memory location) as displayed in figure 7.1.**

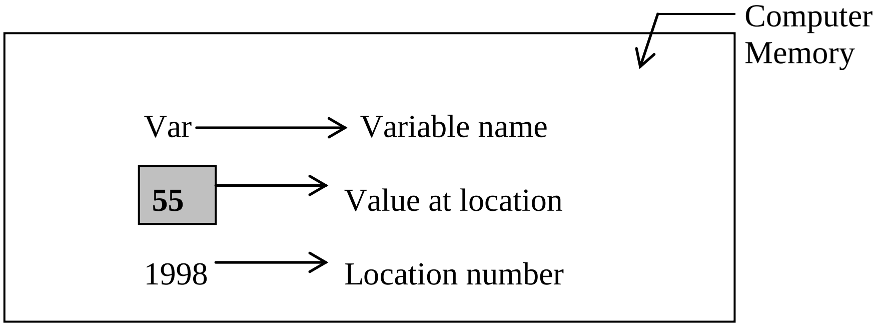


Fig 10.1 : Variable and its address.

- We see that the computer has selected memory location **1998** as the place to store the value **55**. Here, it is important to note that variable **var's** address in memory is mere a number.

We shall go through examples to dig at nut and bolts of pointers.

Example 1: This example explains the pointer operators & and *

```
#include<stdio.h>

main()
{ int *x,y; /* x is a pointer to integer variable*/
   y=9;
   x=&y;  /* Address of y is stored in x*/
  printf(" Address of y=%u\n",&y);
  printf(" Value of y =%d",y);
  printf("Address of y=%u\n",x);
  printf("Value of y=%d",*x);
  getch();
}
```

Output

```
Address of  y=65524
Value of y=9
Address of  y=65524
Value of y=9
```

Explanation

- Two integer variables are declared, out of which **x** is a pointer type(pointer to integer). And **y** is an ordinary integer variable.
- Variable **y** is assigned with constant 9.
- The pointer variable **x** is assigned to the address of **y**.
- First printf prints address of **y** directly, to get the address **&y** is used. The address happens to be 65524. Since 65524 represents an address , it should always be positive(it is always unsigned)., hence **%u** is used to print it. The **%u** is a format specifier for unsigned integer.
- The second printf is well known to us.
- The third printf is simply clear. Since **x** stores address of **y**. We have address of **y** printed using **x.**
- The last printf statement prints value of **y** indirectly by using ***x**. the operator ***** just means value at address. And ***x** means value at address stored by **x**. therefore ***** is called indirection operator when used in conjunction with pointers.

Example 2. This example explains * and & again.

```
#include<stdio.h>

main()
{ int x,y,*ipt; /* ipt is a pointer to integer variable*/
   x=7;
   ipt=&x;  /* Address of x is stored in ipt*/
   y=*ipt;  /*content of pointer goes to y */
   printf(" The value of y is=%d\n",y);
   getch();
}
```

Output

The value of y is=7

Explanation

- Ordinary integer variables are **x** and **y** while **ipt** is a pointer to integer variable.
- The pointer **ipt** is assigned to the address of **x**.
- Variable **y** is assigned to value at the address stored in **ipt**. Since **ipt** contains address of **x**, the value at address of **x** is 7, so ,*ipt is equal to 7.

Example 3. This program explains how arithmetic operations can be done with pointers in the same way as that of ordinary variables.

```
#include<stdio.h>
main()
{ int f,*ipt /* ipt is a pointer to integer variable*/
      k,l,m;
  f=555;
 ipt=&f;  /* address of f is assigned to pointer */
  k= (*ipt)++;
  l= (*ipt)-- ;
  m= (*ipt)++;
printf("k=%d\n", k);
printf("l=%d\n",l);
printf("m=%d\n",m);
getch();
}
```

Output

```
k=555;
l=556;
m=555;
```

Explanation

- Pointer **ipt** is assigned to address of **f**.
- The content of the pointer **ipt** is post incremented and stored in **k**. Since it is post incrementation, **k** gets just **555**. Incrementation is going to happen only in the next step.
- The content of the pointer is post decremented and stored in **l**, this has no effect, because, incrementation will be effected in this line. So what comes to **c** is earlier incremented value i.e **556**.
- The content of the pointer is post incremented again but for no avail, only earlier decrementation becomes effective now and what gets stored **m** is **555** again!.

Pointers and Arrays

There is a close association between pointers and arrays. In session on arrays we saw how array elements are accessed. We shall recollect yet again by a review program.

```
#include<stdio.h>
main()
{ int i;
  int arr[]={37,43,55,77};
  for(i=0;i<4;i++)
  printf("Element position:%d , its vale:%d\n",I,arr[I]);
  getch();
}
```

output

Element position:0, its value:37

```
Element position:1, its value:43
Element position:2, its value:55
Element position:3, its value:77
```

Array elements can be accessed using pointer notation as well as array notation. To do this consider the above example in which array declaration int arr[] is made and initialised. By this declaration four memory locations each occupying 2 bytes (because elements are of integer type) are allocated consecutively for the array variable arr as shown in the figure 10.2.

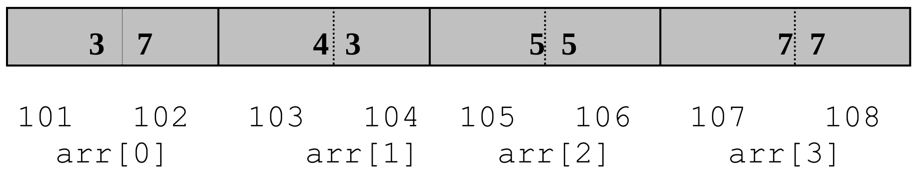


Figure 10.2 : Array elements in memory

The address of array begins with 101(assumed address value). The first element i.e., arr[0] consumes 2 bytes of memory. The **address** of the second element is at 103. The name of the array it self will represent its **address**. Thus if we say just **arr** , we are referring to a constant pointer. In fact, **arr** represents the very first element of the array**.** If we write **arr+i** , we are adding integer i to the address of array. The C compiler is intelligent enough to take the size of the data into account when it does arithmetic on data address. It understands that **arr** is an array of integer type , because it is declared that way in the beginning of the program. Thus if **arr** were to represent data address of the first element, **arr+1** will represent the address of second one, **arr+2** will represent address of third and so on… We shall go through example 4 to make points clear.

Example 4. This program reads 10 array elements and prints the elements back using pointer technique.

```
#include<stdio.h>
main()
{ int arr[10],*arpt,i;
  printf("Enter 10 elements of array\n");
  for(i=0;i<10;i++)
  scanf("%d",&arr[i]);
  /*arrpt points to array */
  arpt=arr;
  /* printing by technique 1 */
   for(i=0;i<10;i++)
  printf("%5d",*(arpt+i));
   /*printing by technique2*/
   for(i=0;i<10;i++)
  printf("%5d",*(arpt++));
  getch();
  }
```

output

```
Enter 10 elements of array
10 20 30 40 50 60 70 80 90 100
  10  20   30   40   50   60   70   80   90  100
  10  20   30   40   50   60   70   80   90  100
```

- The array pointer arpt is assigned to the address of the first element(**arr**).
- Now **arpt** becomes a pointer variable, in the first technique, in the **for** loop ***(arpt+i**) becomes ***(arpt+0**)in the first cycle which is just ***(arpt**) , which would mean the contents of address pointed by **arpt** which is 10. In the second cycle, ***(arpt+i)** becomes ***(arpt+1)** which means that the pointer jumps to point the next element which is 20.
- In the second technique also same mechanism happens , however, here post increment operator is used with the pointer instead of adding loop index to the pointer.

Example 5 : This program reads an array of 10 elements and finds the biggest element among them.

```
#include<stdio.h>
main()
{
 int arr[10],*arpt,i,big;
 printf("Enter 10 elements of array\n");
 for(i=0;i<10;i++)
 scanf("%d",&arr[i]);
/*the first element address stored in arpt */
 arpt=arr;
 big=*arpt; /*contents of arpt stored in big */
 for(i=1;i<10;i++)
 if(big < *(arpt+i))
  big=*(arpt+i);
 printf("The biggest among the elements=%d\n",big);
 getch();
 }
```

output

```
Enter 10 elements of array
10 22 44 77 88 52 -23 13 33 11
The biggest among the elements=88
```

Explanation

- The address of the first element is stored in pointer **arpt** when **arpt** is assigned to **arr**.
- Variable **big** then holds the contents of the pointer **arpt** which is the **first** element of the array.

- In the **for** loop this first element stored in **big** gets compared with the second element (which is represented by the contents of the pointer **arpt+1**. If big is less than second element then, second element will become big. Procedure continues till all the elements are compared.

Example 6. This program performs the multiplication of two matrices by pointer technique.

```
#include<stdio.h>
main()
{
 int ar[10][10],br[10][10],cr[10][10],i,j,k,m,n,p,q;
 printf("Enter size of matrix ar\n");
 scanf("%d%d",&m,&n);
 printf("Enter size of matrix br\n");
 scanf("%d%d",&p,&q);
 if(n!=p)
 {
  printf("Matrices are not multiplicable\n");
  exit();
  }
  printf("Enter the elements of matrix ar\n");
  for(i=0;i<m;i++)
  for(j=0;j<n;j++)
  scanf("%d",&ar[i][j]);
  printf("Enter the elements of matrix br\n");
  for(i=0;i<p;i++)
  for(j=0;j<q;j++)
  scanf("%d",&br[i][j]);
  for(i=0;i<m;i++)
  for(j=0;j<q;j++)
   { *(cr[i]+j) =0;
    for(k=0;k<n;k++)
    *(cr[i]+j) = *(cr[i]+j)+ *(ar[i]+k)* *(br[k]+j);
    }
    printf("The resultant matrix\n");
    for(i=0;i<m;i++)
     { for(j=0;j<q;j++)
       printf("%8d",*(cr[i]+j));
       printf("\n");
     }
    getch();
    }
```

Output:

```
Enter size of the matrix ar
5
4
Enter size of the matrix br
4
3
Enter the elements of matrix ar
```

```
1 2 3 4
6 7 8 9
5 4 3 2
2 3 4 5
1 2 3 4
Enter the elements of matrix br
1 2 3
4 5 6
7 8 7
3 4 5
The resultant matrix
  42      52       56
 117     147      161
  48      62       70
  57      71       77
  42      52       56
```

Explanation

- Two matrices of size m x n and p x q are read, a check is done to verify whether they are multipliable or not. Rest of the procedure is same (the reader is requested to go through the same problem done in session 9) except the fact that we are referring the contents of pointers.
  Pointers can be used to refer elements of two dimensional **array** also .We know that
- two dimensional array can be thought of as an **array** of arrays. Since number of columns are specified in declaration, If we write ar[0], it is interpreted as (ar+0) which is the address of first element i.e., ar[0][0]. Similarly ,ar[1] is interpreted as (ar+1) which is ar[1][0]. If we want to refer an element ar[3][1] using pointers, all the following expressions will refer the same element,
  ar[3][1]
  ***(ar[3]+1)**
  ((ar+2)+1)

Points to Rehearse

- Pointers are the variables that hold address value of other variables.
- Address is simply a numeric value associated with every memory location.
- By use of & operator, variable name gets translated to its corresponding address.
- A variable's contents may change during the execution of the program but not it's address. And at any given moment, each variable has a unique address.
- The & operator can be used before a discrete variable, before a subscripted array element. Thus,
  &a is correct if a is a variable.
  &a[j] is correct as a[j] is subscripted array variable.
  &arr is wrong if arr is an array name.
  &22 is wrong because 22 is a constant.
  &(k+22) is wrong because (k+22) is an expression.

- The declarations and assignments written below has the meaning as explained
  int *x,z; x is a pointer variable, z is an integer.
  float *y,p; y is a pointer to float, p is a float type variable.
  x=&z; x is assigned to address of z.
  y=&p; y is assigned to address of p.

- The indirection operator is the asterisk(*). When this is placed before a pointer variable, it means contents of the address pointed by a pointer variable or simply contents of pointer.

- The name of the array is a pointer constant, it represents the address of very first element of an array.

- If **arr** is name of an array ,
  (arr+1) denotes the address of 2nd element.
  *(arr+1) denotes the contents of 2nd location.

- If scores is the name of array and x is a variable then, the expression **scores=&x** is invalid because **score** is a **constant pointer**.

Points to Ponder

Why pointers? Are the names of the variables not sufficient?. One of the answers for this questions is, that the memory address is global to all the functions(functions are on the way coming in next session). While an ordinary variable name are local to functions. With pointers as tools, the programmer can visit the address of variables and can manipulate the data. Substantial memory saving can be achieved by using pointers.

It is possible to have an array of pointers in a program just as we had array of integers and floats. The following declaration creates an array named **marks** containing 10 elements each of which is of type int*

int *marks[10];

Each element of this array would point 10 integer variables some where in memory.

C permits pointer to pointers, pointer to pointers to pointers and so on… If we declare

int **pint1, *pint ;

pint1 should be treated as a pointer to integer pointer.

If k is a variable which holds a value 35 and has address say 1225, then pint2 holds address of k , that is 1225. The pint2 itself has an address say 3227 then pint1 will hold the address of pint2. Also pint1 will have an address say 7897. The link is displayed by figure 10.3

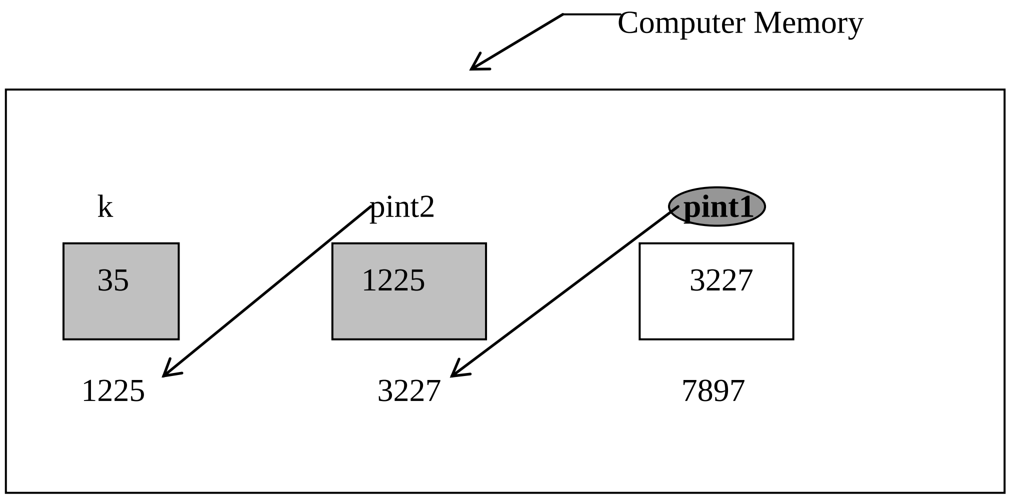


Figure 10.3 Pointer to pointer

Try Yourself

1. Read two variables x and y and exchange their contents using pointers.
2. Read an array of 10 elements and make a linear search for a key element. Use pointer technique.

Session 11.

Creating functions

A function groups a number of program statements into a unit and gives it a name. We have been dealing with **main**() function all along the sessions. A **main**() function is a complete program unit which executes by itself. A function on the other hand, is not a complete program and cannot be executed by itself. Thus far, all our programs have been **main** programs

A **function** can be thought of as a sequence of instructions that performs one task and whose result, if any is incorporated into a **main** function. Writing functions for a big program is similar to a **co-ordinator** assigning parts of a bigwork to different individuals and completing it. Here big work is big program, co-ordinator is main() function, individuals are functions. Listed below are some of the advantages of functions.

- Main advantage of using function is that they allow us to break a large, complex task into series of smaller ones.
- Smaller functions are easier to code because, entire effort can be directed to the task at hand.
- A function may be used any number of times by any number of other programs requiring similar tasks to be performed.
- Debugging is generally simpler with functions that are compiled separately because syntax errors can be removed from one small function at a time.
- Use of function generates a modular program. This is one of the objectives of a good structured programming.
- Use of function tends to make more efficient use of memory.

C has plenty of built in functions like Arithmetic functions, data conversion functions, character classification functions, string manipulation functions, searching and sorting functions, Input and output functions, file handling functions, memory allocation functions and many more. When we are using a library function, the function has already been defined and compiled for us. But the real programming pleasure comes with writing our own functions. Therefore we shall write a simple function to start with.

First function:

```
#include<stdio.h>
main()
{
  printf("This function is written \n");
  function();
  printf("Thank you \n");
}

function()
{
  printf("Because we want to start with a \n");
  printf("simple function like this \n");
}
```

Explanation:

- Here, the **main**() function is calling the function **function**(). To call a function what is needed is just the function name. The syntax of the call is very similar to declaration. The call is terminated by a semicolon. When function is called, the control is transferred to the function, the statements in the function are executed and then control returns to **main**() .
- What happened in the above program is just this, the first printf function was executed, then control passed to **function**() and the two printf statements are displayed, control returned to **main** and the printf in the **main** worked to produce the last line of output.

  We shall write one more function, which prints result of a student.

Example 2.This program creates functions, whose job is to print the result based on percentage of marks.

```
#include<stdio.h>
main()
{  ‶
  int mark1, mark2, mark3, mark4, tot;
  float per;
  printf(‶Enter the marks in 4 subjects \n‶);
  scanf(‶%d%d%d%d‶, &mark1,&mark2,&mark3,&mark4);
  tot = (mark1+mark2+mark3+mark4);
  per = ( tot/400.0 * 100.0);
  if(per < 35.0) res_declared1();
if(per >= 35.0 && per < 60.0) res_declared2();
if(per >= 60.0 && per < 70.0) res_declared3();
if(per >= 70.0) res_declared4();
getch();
}

res_declared1()
{
  printf(‶FAILS \n‶);
  printf(‶Sorry! Don’t get disheartened \n‶);
  printf(‶Work more this time \n‶);
}
res_declared2()
{
  printf(‶SECOND CLASS \n‶);
  printf(∀Fine! But scope is there for improvement \n∀);
  printf(∀Try for Ist class next time  \n∀);
}
res_declared3()
{
  printf(∀FIRST CLASS \n∀);
  printf(∀Very Good! Try for still more! \n∀);
}
res_declared4()
{
  printf(∀DISTINCTION \n∀);
  printf(∀Excellent! SWEETS? \n∀);
  printf(∀Keep it up \n∀);
}
Output:
90 100 99 97
DISTINCTION
Excellent! SWEETS?
Keep it up
```

Explanation:

- Here, the percentage marks is tested to identify the range with in which it falls. If the percentage is less than 35, the function **res_declared1** is called. This function when called neatly does its job of declaring the results.

Example 3: We shall introduce function in a function.

```
#include<stdio.h>
main()
{
  float per;
  printf("What is your percentage of marks in PU ? :\n");
  scanf("%f",&per);
  if(per < 60.0)try();
  if(per >= 60.0 && per < 70.0)wish();
  if(per >= 70 )congrats();
  getch();
}
try()
{
  printf("Try next time\n");
  conclude();
}

wish()
{
 printf("Wonderful! But, You may have to work harder
\n");
 conclude();
}
congrats()
{
  printf("Congratulations \n");
  conclude();
}
```

```
conclude()
{
  printf("Thanks for responding \n");
}

Ouput:
What is your percentage of marks in PU ? :
66
Wonderful! But, You may have to work harder
Thanks for responding
```

Explanation

- Here percentage is the input. The percentage is tested to see under what range it comes. If it comes under a particular range mentioned, corresponding function is called. When the respective function is invoked it does the job of displaying some remarks. Soon after this , the function calls another function **conclude**() , the control will shift to this function and this function flags the message thanking the user.

After having gone through examples 1, 2 and 3 we can conclude following points about functions.

- Any C program on earth will have one function which is none other than **main**().
- If there are more than one function in a C program, the **main**() function will be the controller(or co-ordinator), because, it is from **main**() the execution always begins.
- Functions are called by **main**() in a specified sequence.
- After function performs its specified task, the control goes back to main.
- Any function can be called from any other function. The **main**() function can also be called by other function in a program.

Passing Values:

As the name itself would convey, we can pass values of variables to functions for intended work to be done by them. The values of variables are sent to function through "**arguments**". We shall go through example 4 to understand this feature.

Example 4:

This program explains sending values to functions and receiving from function.

```
#include<stdio.h>

#include<math.h>
main()
{
  int x,y,z,fact;
  clrscr(); /* clears the screen */
  printf("Enter two integer numbers :\n");
  scanf("%d%d",&x,&y);
  z = cal_diff(x,y);
  fact = cal_fact(z);
  printf("Difference between two numbers = %d \n",z);
  printf("The factorial of %d is %d \n ",z,fact);
  printf("Press any key to close");getch();
}

cal_diff(a,b)
int a,b;
{
 int diff;
 diff = abs(a-b);
 return(diff);
}

cal_fact(q)
int q;
{

  int f,i;
  f=1;
  for(i=1; i <= q;i++)
   f = f*i;
  return(f);
}
Output:
Enter two integer numbers :
1 5
Difference between two numbers = 4
The factorial of 4 is 24
 Press any key to close
```

Explanation

- From the function **main**() the value of **x** and **y** are passed to function **cal_diff(x,y)**. The values **x** and **y** were sent to the function by enclosing them in parenthesis separated by commas. The variables **x** and **y** are called **arguments**.
- The values so sent, will get copied to **a** and **b** when function is declared. The values sent are called **arguments** or **parameters**. While the corresponding name in the function declaration (**a** and **b**) are called **formal** or **dummy** arguments.
- Value **z** is passed to function **cal_fact(z)**.
- After receiving the actual values of variables in **q** , the function **cal_fact** perform task of calculating factorial of **q**.
- The calculated values of difference (**diff**) and factorial(**f**) are sent to main function by return statement. Because **return** statement transfers the control to calling program. It **returns** the value present in the parenthesis.

Example 5:

This program reads integer numbers n and r and evaluates

$$\frac{n!}{(n-r)!}$$

$$\frac{n!}{(n-r)!n!}$$

using functions.

```
/* program to calculate factorials n!, n-r! */
#include<stdio.h>

main()
{
  int n,r;
  long int factn,factnr,factr,f1,f2;
   long int factorial (long int);
       clrscr();
  printf("Enter n and r values ");
  scanf("%d %d",&n,&r);
  factn =  factorial(n);
  factnr =  factorial(n-r);
  factr =  factorial(r);
  f1 = factn/factnr;
  f2 = factn/(factnr*factr);
  printf("\n n!/ (n-r)!    = %ld",f1);
  printf("\n n!/ (n-r)! r! = %ld",f2);
  printf("\nPress any key to continue");getch();

}
int factorial(m)
int m;
{
 long int f;
 int k;
 f = 1;
 for(k=1;k <= m;k++)  f=f*k;
 return(f);
}
Output:
Enter n and r values 5 4
 n!/ (n-r)!    = 120
 n!/ (n-r)! r! = 5
Press any key continue
```

Explanation:

Here, **n** and **r** are read, and function **factorial**() is invoked for **three** times to collect the value it returns to **variables** declared in the main function. Finally, f1 and f2 are calculated. One can notice the name of the function being mentioned in the main program after all the type declaration statements are made. This is called a **prototype** of the function. If a function is returning a value other than an integer (long int, character , float) , prototypes are to be made available to the compiler right in the beginning of the main faction. In the function prototype

Long int factorial (long int) , function name is factorial it is preceded by what type of value it returns to main function . Function name is followed by the type of arguments sent by the main fuction.

Example 6: This function determines whether triangle can be constructed with the three sides given and determines its area.

```
#include<stdio.h>
#include<math.h>
main()
{
 float a,b,c,area_tri;
 int flag;
 float area(float a,float b,float c);/*prototype*/
 clrscr();
 printf("Enter side lengths of a triangle\n");
 scanf("%f%f%f",&a,&b,&c);
 flag=check(a,b,c);
 if(flag)
 {printf("Triangle can be constructed\n");
  area_tri=area(a,b,c);
  printf("The area of triangle=%f\n",area_tri);
  }
  else
  printf("Triangle can not be constructed\n");
  getch();
  }

  check(a,b,c)
  float a,b,c;
  { int token;
   if(a+b<=c || b+c<=a || c+a<=b)
    { token=0;
      return(token);
     }
   else
    { token=1;
      return(token);
     }
    }

   float area(float a, float b, float c)
    {
     float s,ar;
     s=(a+b+c)/2;
     ar= sqrt(s*(s-a)*(s-b)*(s-c));
     return(ar);
     }
```

Output:

```
Enter side lengths of a triangle
3
4
5
Triangle can be constructed
The area of triangle =6.000000
```

Explanation:

- This program uses two functions, function **check** and function **area .** Function check carries the side lengths of the triangle as arguments and performs a check as to whether triangle can be constructed. If constructed, it sends value 1 to main function otherwise it sends 0 to main function.

- If it is possible to construct a triangle, the function area is invoked. The function **area** calculates area and returns area value which is of float type. Whenever functions return values otherthan integer type, its declaration should specify what type of value the function is returning. Thus we see the function area being ascribed as **float area (float a, float b, float c)**
- Some compilers demand that the name of such functions to be declared right in the beginning also. Such initial declaration of function is called a **prototype**. Such prototypes will give clue to the compiler before they being processed.

Example 7.
This program is used to solve a system for simultaneous equations (3 unknowns) using Cramer's rule.

Pre-processing of the problem

Three simultaneous equations can be solved using cramers rule as follows,
If the equations are,

a1x+a2y +a3z=co1
b1x+b2y+b3z=co2
c1x +c2y+c3z=co3

If matrix a is

$$\begin{bmatrix} a1a2a3 \\ b1b2b3 \\ c1c2c3 \end{bmatrix}$$

and constant vector is

$$\begin{bmatrix} co1 \\ co2 \\ co3 \\ \end{bmatrix}$$

The matrix b is,

$$\begin{bmatrix} co1 a2 a3 \\ co2 b2 b3 \\ co3 c2 c3 \end{bmatrix}$$

The matrix c is

$$\begin{bmatrix} a1 co1 a2 \\ b1 co2 b3 \\ c1 co3 c3 \end{bmatrix}$$

The matrix d is

$$\begin{bmatrix} a1 a2 co1 \\ b1 b2 co2 \\ c1 c2 co3 \end{bmatrix}$$

Now the value of x is = Determinant of b/ Determinant of a
Value of y = Determinant of c/ Determinant of a
Value of z = Determinant of d/ Determinant of a

If the determinant value of a is zero then the coefficient matrix becomes singular and solution is not possible.

```
/* Program to solve three simultaneous Equations by
Cramer's Rule */
#include<stdio.h>

main()
{
 float a[3][3],b[3][3],c[3][3],d[3][3],con[3];
 float x,y,z,det;
 int i,j;
 float det(float a[3][3]); /* prototype of  function*/
 clrscr();/* clears the screen */
 printf("Enter the Co-efficient of x,y,z of 3
Equations:\n ");
 for(i=0;i < 3;i++)
 {
  for(j=0;j < 3;j++)
  scanf("%f",&a[i][j]);
 }
printf("Entered Co-efficients are: \n");
  for(i=0;i < 3;i++)
 {
  for(j=0;j < 3;j++)
  printf("%f ",a[i][j]);
  printf("\n");
 }
 printf("Enter the Constants of 3 Equations:\n ");
 for(i=0;i < 3;i++)
 {
  scanf("%f",&con[i]);
 }
 /* copying the contents of a to b,c,d arrays */
  for(i=0;i < 3;i++)
 {
  for(j=0;j < 3;j++)
  {b[i][j] = a[i][j];
   c[i][j] = a[i][j];
   d[i][j] = a[i][j];
  }
 }
 /* copying the constant vector to b,c,d arrays in
suitable columns */
  for(j=0;j < 3;j++)
 { b[j][0] = con[j];
   c[j][1] =con[j];
   d[j][2] =con[j];
 }

  /* call function to claculate determinants */
  det = del(a);
  if(det == 0.0) printf(" Solution is not possible");
  else
  {
    x=del(b)/det;
    y=del(c)/det;
    z=del(d)/det;
    printf("\n The Values of X, Y, Z are :\n");
    printf("X= %f \n",x);
    printf("Y= %f \n",y);
    printf("Z= %f \n",z);
  }
  printf("\nPress any key to close");getch();
}

float del(float e[3][3])
{
 float cdet;

 cdet =   e[0][0] * ( e[1][1] * e[2][2] - e[2][1] *
e[1][2])
     - e[0][1] * ( e[1][0] * e[2][2] - e[2][0] * e[1][2])
     + e[0][2] * ( e[1][0] * e[2][1] - e[2][0] *
e[1][1]);

 return(cdet);
}
```

```
Output:
Enter the Co-efficient of x,y,z of 3 Equations:
 1 -2 1
2 2 -1
1 -1 2
Entered Co-efficients are:
1.000000 -2.000000 1.000000
2.000000 2.000000 -1.000000
1.000000 -1.000000 2.000000
Enter the Constants of 3 Equations:
 -2 5 1

The Values of X, Y, Z are :
X= 1.000000
Y= 2.000000
Z= 1.000000

Press any key to close
```

Explanation:

- The coefficient matrix a and constant vector con are read. The coefficient matrix is duplicated in to **b**, **c** and **d** matrices. The first column o**f b** , second column of **c**, third column of **d** is replaced by constant vector.
- A function **del** is created to find the determinenent value of matrices. Each time when this function is called(invoked) it returns the determinant of the matrix.
- It is important to notice in this program that the function **det** is returning float type values to main function. Some compilers demand **prototype** of the function be delared at the beginning of the program.

Example 8.

This program Read a matrix A(M x N) and finds the following using function.

i) Sum of elements of each of m rows.
ii) Sum of elements of each of n columns
iii) Find sum of all elements of matrix using function in i)

```
/* Program to find the Rowsum, Columnsum and Matrixsum
using functions */
#include<stdio.h>
main()
```

```
{
 int a[10][10],i,j,m,n;
 clrscr();/*clears the screen*/
 printf("Enter No. of M-rows and N-columns : ");
 scanf("%d%d",&m,&n);
 printf("Enter Matrix elemental values: ");
 for(i=0;i < m;i++)
  {
   for(j=0;j < n;j++) scanf("%d",&a[i][j]);
  }
   printf("Entered  Matrix elemental values are :\n ");
 for(i=0;i < m;i++)
  {
   for(j=0;j < n;j++) printf("%d ",a[i][j]);
   printf("\n ");
  }
  rowsum(a,m,n);
  columnsum(a,m,n);
printf("\nPress any key to close");getch();
}

rowsum(a,m,n)
int a[10][10],m,n;
{
  int k,l,sumr;
  for(k=0;k < m;k++)
    { sumr=0;
      for(l=0;l < n;l++)  sumr= sumr + a[k][l];
      printf("\nSum of Row %d = %d",k,sumr);
    }
arraysum(a,m,n);
}

columnsum(a,m,n)
int a[10][10],m,n;
{
  int k,l,sumc;
  for(k=0;k < n;k++)
    { sumc=0;
      for(l=0;l < m;l++)  sumc= sumc + a[l][k];
      printf("\nSum of Column %d = %d",k,sumc);
    }
}

arraysum(a,m,n)
int a[10][10],m,n;
{
  int k,l,sum;
  sum=0;
  for(k=0;k < m;k++)
    {
       for(l=0;l < n;l++)   sum= sum + a[k][l];
     }
  printf("\nSum of All the elements of Matrix  =
%d",sum);
}
Output:
Enter No. of M-rows and N-columns : 2 3
Enter Matrix elemental values: 1 1 1 1 1 1
Entered  Matrix elemental values are :
 1 1 1
 1 1 1

Sum of Row 0 = 3
Sum of Row 1 = 3
Sum of Column 0 = 2
Sum of Column 1 = 2
Sum of Column 2 = 2
Sum of All the elements of Matrix = 6
Press any key to close
```

Explanation

- In this program a matrix of size **m** and **n** is read. Main function calls two functions namely, **rowsum** and **columnsum .** When control is passed to **rowsum** it prints the sum of all the rows and calls function **arraysum.** When **arraysum** is invoked it calculates sum of all the elements and prints the same. Then the **columnsum** gets invoked and it calculates the sum of all the columns and prints there only.
- Notice that the array **a** is sent to the functions by name itself.

Example 9.

This program is used to input n integers and sort them using selection sort. Write the following functions and use them in selection sort.

i) Find the max of n elements.

ii) Exchange two elements.

```
#include<stdio.h>

main()
{
 int a[100],i,j,n,loc;
 clrscr();
 printf("Enter No. of elements to be sorted \n");
 scanf("%d",&n);
 printf("Enter elemental values to be stored for sorting
\n");
 for(i=0;i<n;i++)
 scanf("%d",&a[i]);
```

```
  for(i=0;i<n-1;i++)
  {  loc=pickmax(a,n,i);
     exch(a,i,loc);
  }
 printf("Sorted array\n");
 for(i=0;i<n;i++)
 printf("\n %d",a[i]);
 printf("\nPress any key to close");getch();
}

pickmax(a,n,i)
int a[100],n,i;
{ int j,loc;
  loc=i;
 for(j=i+1;j<n;j++)
 if(a[loc] >a[j]) loc=j;

 return(loc);
}

exch(a,i,loc)
int a[100],i,loc;
{
  int temp;
  temp=a[i];
  a[i]=a[loc];
  a[loc]=temp;
  return;
}
```

```
 Output:
Enter No. of elements to be sorted
5
Enter elemental values to be stored for sorting
5
123
34
-90
5
444
 Sorted array

-90
 5
 34
 123
 444
Press any key to close
```

Explanation:

- The array **a** is read in the **main** function. The **for** loop in the main will facilitate sending control to functions **pickmax** and **exch**. In **pickmax** function, the position or location of the largest element in a particular cycle is determined and it is sent to main through the variable **loc**. In the main function .
- Upon obtaining the location , exchange will take place through function **exch**.
- The intricate details of the program can be recalled by referring the same program done is session on arrays.

Call by reference:

We can make a function to return more than one value at a time, by making call by reference. Pointers will come in a bigway to help a function by enabling it to send more than one value at a time.

Example 10.This program calculates surface area, volume of a box.

```
#include<stdio.h>

main()
{
  float len,br,ht,su_area,vol;
  clrscr();/* clears the screen */
  printf("Enter length, breadth and height of box:\n");
  scanf("%f%f%f",&len,&br,&ht);
  area_vol(len,br,ht,&su_area,&vol);
  printf("Surface Area of box = %f \n",su_area);
  printf("Volume of box = %f \n",vol);
  printf("\nPress any key to close");getch();
}
area_vol(l,b,h,s,v)
float l,b,h,*s,*v;
{
  *s = 2.0 * (l*b + b*h + h*l);
  *v = l*b*h;
}
Output:
Enter length, breadth and height of box:
2.0 3.0 4.0
Surface Area of box = 52.000000
Volume of box = 24.000000

Press any key to close
```

Explanation:

- Value of length, breadth and height are passed along with address of surface area and volume.

- **s** and **v** are pointers to address of **su_area** and **vol** respectively.
- The function **area_vol** makes changes in the values stored at address contained in pointers **s** and **v** .
- When control returns from the function, the values are displayed by printf statements, thus avoiding return.

Points to Rehearse

- Functions will make a program modular.
- While naming functions, following points are important
  - The name should begin with a letter or underscore
  - The name may contain only letter or digits after the first letter
  - The name should be preferably in all small letters
  - The name should be meaningful so that it would convey the role of the function
- A function can be called any number of times.
- A function gets invoked when its name is specified, followed by a pair of parenthesis depending on the context.
- A return only one value at a time. Following statements are thus invalid,

  Return(x,y);

  However by use of pointers it is possible to return more than one value.

Points to Ponder

- A modular programming is one where the big task is spilt into small sub-tasks, which in turn broken down again. The lower level functions in a modular programming must be as simple as possible.
- The variables declared with in the body of the function are called local variables. These local variables disappear once the function complete execution. While the variables declared outside the function are called global variables. These global variables can be referred to from any function. They exist till the entire execution of the program.
- If the value of a formal argument is changed in the called function, the corresponding changes does not take place in the calling function. For example:

```
#include<stdio.h>
main()
{
  int x = 55;
  change(x);
  printf("x = %d \n",x);
}
change(y)
int y;
{
  y = 110;
  printf("y = %d \n",y);
}
```
Output:
```
y = 110
x = 55
```

Thus, though the value of x is changed in function change(), the value of x in main() remains unchanged.

- It is possible for functions to call themselves. These type functions which calls themselves are called **recursive** functions. A simple example will explain this:

```
#include<stdio.h>

main()
{
  int x;
  long int fact;
  clrscr(); /* clears the screen */
  printf("Enter a number \n");
  scanf("%d",&x);
  fact = self_call(x);
  printf("Factorial of %d = %ld",x,fact);
  printf("\nPress any key to close");getch();
}

self_call(y)
int y;
{
  long int facto;
  if(y == 1)
  return(1);
  facto = y * self_call(y-1);
  return(facto);
}
Ouput:
Enter a number
3
Factorial of = 6
Press any key to close
```

Explanation:
- The recursion is effected by the statement
  ```
  facto = y * self_call(y-1);
  ```
- In first cycle recursion y is 3 i.e. fcato = 3 * self_call(2)
- In first cycle recursion y is 3 i.e. fcato = 3 * 2 * self_call(2-1)
- In first cycle recursion y is 3 i.e. fcato = 3 * 2 * 1
  Because if y = 1 the function return 1;

Try Yourself

1.Write a program to calculate all the roots of a quadratic equation using functions.
2.Read n elements in to an array and calculate mean and standard deviation using functions.

3.Write a program to conduct linear search for finding the presence of a key element using functions (use passing by reference technique).

Session 12.

Working with Strings

As we have acquired some familiarity with arrays, we can look in to **strings** in this session. **String** is a term used to denote **character arrays.** Character arrays are used by programming languages to manipulate text such as words and sentences. A **string** is typically one dimensional array of characters terminated by a **null(\0')** character. For example the string **semester** is stored as shown in figure 9.1.

s	**e**	**m**	**e**	**s**	**t**	**e**	**r**	**\0**

Figure 9.1 String as An array of characters.

The length of the string is determined by the number of characters exclusive of null character. The terminating **null character** is the only way to know where the string ends. A string can be initialised as shown below;

```
Char name[ ] = " RAJENDRA " ;
```

As with other data types, strings can be variables or constants. A string constant is called as a **Literal**. String constants are always enclosed in double quotes.

```
Ex:     " 12345 "
        " SEMESTER"
        " Enter values of a and b\n";
```

String variables are declared as shown below;

```
        char  name[30];  code[5];
```

We shall go through some examples to see how we can work with strings.

Example 1 This program shows one of the methods to print a string.

```
#include<stdio.h>
main()
{
  char name[]="Jayaram";
  int i;
  for(i=0;i<7;i++)
  printf("%c",name[i]);
  getch();
  }
```

Output

Jayaram

Explanation

- Character array is **name** is initialised with a string.

- Elements are printed with in a for loop letter by letter. From 0th position ,it requires 6 places to place all the letters of the name hence the conditional statement in for is put at less than 7.

Example 2. In this example the string is printed making use of position of null character.

```
main()
{ char sem[]="Semester1";
  int i=0;
  /*print letter by letter till \0 is reached*/
  while(sem[i]!='\0')
  { printf("%c",sem[i]);
    i++;
  }
  getch();
}
```

Output

```
                         Semester1
```

Explanation

- **Here length of the string is not fixed instead the target '\0' character is fixed.**
- The letter constituting the string are printed one after the other till the position reaches just before null character.

Example 3: This program uses pointer technique to print the string

```
#include<stdio.h>

main()
{ char sub[ ]="Mechanics";
  char *cptr;
  cptr=sub; /* cptr holds the address of first location*/
  while(*cptr!='\0')
  { printf("%c",*cptr);
    cptr++;
  }
 getch();
}
```

Output

```
Mechanics
```

Explanation

- Here, ***cptr** is a pointer to character. It holds address of the first location of the array **sub**.
- Till the pointer points to last character which is **null**, the contents of pointer is printed.

Example 4: This example shows a very simple way of printing a string

```
#include<stdio.h>
main()
{ char sub[]="Computers";
  printf("%s",sub);
  getch();
}
```

Output

```
Computers
```

Explanation

- Here , the format specifier **%s** has been used to print the string. This is a straight forward approach

Example 5: This program reads a name of the student and prints it.

```
#include<stdio.h>
main()
{
  char stud[20];
  printf("Enter your good name please\n");
  scanf("%s", stud);
  printf("You are :%s !\n",stud);
  getch();
  }
```

Output

```
Enter your good name please
Jagadeesh
You are : Jagadeesh !
```

Explanation

- The first line char **stud[20**] is the declaration. This declaration sets aside **20 bytes** for array **stud**.
- The **scanf** takes the characters typed through the key board in to the array until the enter key is hit.
- If we hit the enter key, scanf places **null** character ('\0') in the array at end.
- Therefore we should be careful while entering characters to see that we will not exceed limits. If we exceed , the characters will be overwritten on some important data.

Example 6 . This program reads a string which has more than one word.

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

```
main()
{ char words[30];
  printf("Enter a string with two words\n");
  gets(words);
  puts("Fun with");
  puts(words);
  getch();
}
```

Output

```
Enter a string with two words
Computer Laboratory
Fun with
Computer Laboratory
```

Explanation

- Here , two string functions are used. The standard input out put function **gets(**get string**)** and **puts(**put string**)** has been used. These two functions are defined in stdio.h header file. As the name would imply, gets inputs the string while puts outputs the string.

Use of some standard string functions

There are host of standard library functions available in C pertaining to string manipulations. These functions offer a straight forward procedure to manipulate strings. Using standard string functions one can find the length of the string, copy one string variable to another, concatenate (join) two strings, compare two strings, change case of the letters in the string so on and so forth. We shall go through few such functions and see how they work for us.

Example 7. This program checks whether a given word is palindrome or not.

{ A palindrome is a word which when reversed reads the same ex. Madam, Malayalam }

```
#include<stdio.h>

#include<string.h>
main()
{ int j,k;
  char poly[25];
  printf("Type a word for testing palindrome feature\n");
  scanf("%s",poly);
  j=strlen(poly);
  j--;
  for(k=0;k<=j;k++)
  {if(poly[j]!=poly[k])
  { printf(" The given word is not a palindrome\n");
```

```
    break;}
    else
    j--;
    }
  if(k>j)
  printf("The given word is a palindrome\n");
  getch();
  }
```

Output

```
Type a word for testing palindrome feature
malayalam
The given word is a palindrome
```

Explanation

- The word to be tested is read by scanf (notice that & should not be used when strings are read).
- The length of the string is stored in j by using **strlen()** function. This function counts number of characters present in the string excluding null character. Since the array starts from zero, the **j** value obtained is lessened by one using decrement operator.
- The for loop will facilitate comparison of first character with the last character and if even one of them do not match, the loop will break. Otherwise the loop continues comparing next set.
- If the for loop successfully completes, **k** will have gone for more than j, therefore the string comes out as a palindrome.

Example 8. This program copies contents of one string to other string.

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

main()
{ char original[]="Rajendra";
  char copy[20];
  strcpy(copy,original);
  printf("Now after copying:%s",copy);
  getch();
}
```

Output

```
Now after copying: Rajendra
```

Explanation

- In this example **strcpy()** function is used in this program. This function receives base address of string array **copy** and **original .** It goes on copying till it does not encounter the null string('\0') placed at the end. Therefore we must take care to see that size of copy be at least equal to the size of original.

Example 9. In this program two words are joined.

```
#include<stdio.h>

#include<string.h>
main()
{ char str1[30],str2[10];
  printf("Enter string 1\n");
  gets(str1);
  printf("Enter string 2\n");
  gets(str2);
  /* concatenation takes place */
  strcat(str1,str2);
  printf("Now string 1 becomes\n");
  puts(str1);
    getch();
  }
```

Output

```
Enter the string1
Fun With
Enter string2
Computers
Now string 1 becomes
Fun With Computers
```

Explanation

- Here standard function **strcat**() is being used. The string1 holds string2 when they are concatenated. Therefore the size of the stringe1 should be large enough to accommodate string2.

Example10: This program compares to find whether two strings are identical or not.

```
#include<stdio.h>
#include<string.h>
main()
{ char word1[20], word2[15];
  int i,j;

  printf("Enter the first string\n");
  scanf("%s",word1);
  printf("Enter the second string\n");
  scanf("%s",word2);
  i=strcmp(word1,word2);
  if(i==0)
  printf("The words are identical\n");
  else
  printf("words are not identical\n");
  getch();
  }
```

output

```
Enter the first string
Jagadeesh
Enter the second string
Jagadeesh
The words are identical
```

Explanation

- Here **strcmp()** function is used. This function compares two strings to find out whether they are same or different. Comparison takes place character by character until there is a mismatch or end of the string is reached which ever is first to occur. If the strings are identical, it will return **0** to **i**. If they are not identical , it returns the numerical difference between **ascii** values of non matching characters.

Example 11. This program converts a word in small letters to capital letters to small letters and vice versa.

```
#include<stdio.h>
#include<string.h>
main()
{ char word[20];
  int i,k;
  printf("Type a word in full capitals or small case\n");
  scanf("%s",word);
  k=strlen(word);
  printf("Word before conversion\n");
  printf("%s\n",word);
  /* conversion starts now*/
  /* ASCII character set difference between 'a'&'A' is
32*/
  for(i=0;i<k;i++)
  { if(word[i]>='a'&&word[i]<='z')
      word[i]=word[i]-32;
    else
      word[i]=word[i]+32;
  }
  printf("Word after conversion is\n");
  printf("%s\n",word);
  getch();
  }
```

Output

```
Type a word in full capitals or small case
pooja
Word before conversion
pooja
Word after conversion
POOJA
```

Explanation

- In this problem the **ASCII** character set values are made use of. In the **for** loop each character in the word is tested to see if it is one among the letters from '**a**' to '**z**'. If it is true than , that character is changed with respective **capital** letter. Otherwise it is converted in to respective **small** letter.

Example 12 This program reads a sentence and changes the uppercase letters to lower case letters and vice versa.

```
#include<stdio.h>
#include<ctype.h>
main()
{
 int i=0,j=0;
 char sentence[80],character;
 clrscr();
 printf("Type a Sentence and press Enter key at end \n");
 do
 {
   character = getchar();
   sentence[i] = character;
   i++;
 } while(character != '\n');


 printf("\n"); /* New Line */

 do
  {
    if(islower(sentence[j]))
putchar(toupper(sentence[j]));
    else
       putchar(tolower(sentence[j]));

       j++;
  } while(j < i);
  printf("\nPress any key to close");getch();
}
```

Output

```
Type a sentence and press enter key at the end
Engineering Students

eNGINEERING sTUDENTS
```

Explanation

- Here in this example , first the sentence string is read character by character till the enter key is hit(Till the character is not '\n'). Reading a single character can be done using getchar() function. It gets the character from the **keyboard** and assigns it to **character** named **character** variable. Otherwise it gets printed as lowercase letter by the help of function **tolower().**

- Then in the while loop each letter in the sentence is verified whether it is lower by function **islower() .** If the letter is found to be lower , it gets converted in to upper case character by character with the help of the function **toupper().** The two functions **islower()** and **toupper**() are also defined in **ctype.h.h** header file. Therefore this file is included.

- The functions **getchar()** will get a character from the key board while the function **putchar()** puts back the character on the screen. These two functions are defined in **stdio.h** header file.

Alternatively,

```
#include<stdio.h>
main()
{
 char c;
 clrscr();
 printf("\n Enter a sentence\n");
 while((c=getchar())!='\n')
 {
       if(c>='A'&& c<='Z')
       putchar(c+32);
       else if( c>='a' && c<='z')
       putchar(c-32);
       else
       putchar(c);
 }

getch();
 }

 Output:
 Enter a sentence
 Welcome to COMPUTERS
 wELCOME TO computers
```

Explanation:

- The alternative program seems to be less verbose and more simple. Here, ASCII character values has been made use in converting upper case letter to lower case and vice-versa. The logic is just simple, if the character is uppercase letter, 32 will be added to it to make it upper case. Similarly 32 will be negated if the character is of lower case. If the character is blank space, nothing will happen and it will be printed as it is.

Example 13: This program reads a sentence and reverses it.

```
#include<stdio.h>
main()
 { char name[40],ch;
    int l,i,k;
    clrscr();
    printf("Enter a sentence\n");
    gets(name);
    l=strlen(name);
    l--;
    for(i=l;i>=0;i--)
    printf("%c",name[i]);
    getch();
 }
```

```
Output:
Enter a sentence
Jayaram rao!
!oar marayaJ
```

Explanation:

- After reading a sentence, its length is determined. Using the for loop the sentence is printed backwards character by character.

Example 14 : This program counts number of words, lines and characters in a given sentence/ paragraph.

```
#include<stdio.h>
#include<ctype.h>
main()
{
 char c;int lc,wc,cc;
 lc=1;
 wc=0;
 cc=0;
 clrscr();
 printf("\n Enter a text and press Ctrl+Z to terminate
the input\n");
 while((c=getchar())!=EOF)
 { if(c =='\n')lc++;
   if(c== '\n' || c ==' ')wc++;
   else  ++cc;

 }

printf("\n No. of Lines = %d\n No. of Words = %d \n No.
of Characters =  %d",\
lc,wc,cc);
getch();
```

```
 }
```

Output:

```
 Enter a text and press Ctrl+Z to terminate the input
 Jack and Jill
 Went up the hill
 to fetch ^Z
 No. of Lines = 3
 No. of Words = 9
 No. of Characters = 31
```

Explanation:

- A paragraph is read character by character till end of file is reached. End of file can be created by pressing Ctrl key and Z key simultaneously. Counters for words, lines and characters are initialised to zero out side the for loop and accumulated in the for loop.

Example 15: This program counts number of newline characters, tabs and white spaces.

```
/* Program to count number of \n, \t and spaces in a
given text */
#include<stdio.h>
#include<ctype.h>
main()
{
 char c;int lc,wc,ct;
 lc=wc=ct=0;
 clrscr();
 printf("\n Enter a text and press Ctrl+Z to terminate
the input\n");
 while((c=getchar())!=EOF)
 { if(c == '\n')lc++;
   if( c == ' ')wc++;
   if(c== '\t')ct++;
 }

printf("\n No. of New Line Characters = %d\n No. of Tabs
= %d \n No. of Spaces =  %d",\
lc,ct,wc);
getch();
 }
```

Output:

```
Enter a text and press Ctrl+Z to terminate the input
Work while
      you
work
 ^Z
 No. of New Line Characters = 3
 No. of Tabs = 1
 No. of Spaces = 1
```

Explanation:

- This program is very much similar to previous one. Each and every character of the input statement are compared with the appropriate symbols like \n, \t and if matching is found, the respective counters are incremented.

Example 16. This program inputs 10 names each of length at least 8 characters, and sorts them in alphabetical order.

```
#include<stdio.h>
#include<string.h>
main()
{
   char temp[10], name[10][10];
 int i=0,j=0;
 clrscr();
 printf("Enter 10 names: \n");
  for(i=0;i <=9; i++)
  scanf("%s", name[i]);
  /* Bubble sorting */
  for(i=1;i <10; i++)
   { for(j=1;j <=10-i;j++)
      {
     if( strcmp(name[j-1],name[j]) > 0)
      {
       strcpy( temp, name[j-1]) ;
       strcpy( name[j-1], name[j]) ;
       strcpy( name[j],temp);
      }
      }
   }

  printf("Sorted names are: \n");
  for(i=0;i < 10; i++)
  {
    printf("%s \n", name[i]);
  }
  printf("press any key to  close");getch();
}
```

output

```
Enter 10 names:
Shiva
Rudra
Rajendra
```

```
Jayaram
Jagadeesh
Kalyani
Pooja
Sharanya
Aishwarya
Padmini
Sorted names are:
Aishwarya
Jagadeesh
Jayaram
Kalyani
Padmini
Pooja
Rajendra
Rudra
Sharanya
Shiva
Press any key to close
```

Explanation

- Here bubble sorting is used. In the inner **for** loop , each name(string) in the list is compared with its following name(string). If the numerical difference between the **ascii** value(which is sum total of values of all the characters) is more than zero, then it would mean that the name preceding is higher than the following name, so swapping of the names is done. While swapping string copy technique is used.
- In the program, the declaration name[10][10] should not be confused with two dimensional array , name[10] represents 11 characters(including null), name[10][10] represents 11 names each name having a length of 11 characters.

Example 17: This program reads a string and prints a substring out of it.

```
#include<stdio.h>
#include<string.h>
#include<dos.h>
main()
{
  char str[30],*stp1,*stp2;
  int i,j,k;
  clrscr();
  printf("Enter the string\n");
  gets(str);
  printf("Enter the position of beginning and end of the 
substring\n");
  printf("From what position?\n");
scanf("5d",&j);
printf("Up to what position\n");
scanf("%d",&k)
 stp1=str+j;
stp2=str+k;
put_sub(stp1,stp2); /*pointers are sent to function*/
}

put_sub(stp1,stp2)
char *stp1,*stp2;
{ char str2[20];
  int i=0,j;
  j=stp2-stp1;




 do
  {
    str2[i]=*stp1++;
    ++i;
   }while(i<=j);
str2[i]='\0';
printf("The substring is\n");
puts(str2);
getch();
}
```

Output:

```
Enter the string
Twinkle Twinkle
Enter the position of beginning and end of the substring
From what position
2
Up to what position
11
The substring is
Inkle Twinkl
```

Explanation

- This program uses pointer technique to separate a portion of the string(Sub string) out of a full length string. Pointer **stp1** is set to beginning position of the substring while the pointer **stp2** is set to the last position of the substring. These pointers are sent to the function **put_sub** . This function will use length of the substring to print the substring **str2** by collecting the contents of memory location of each character in the substring.

Example 18: This program converts binary number to its hexadecimal equivalent

```
#include<stdio.h>
#include<math.h>
#include<string.h>
main()
{
  int dec=0,hex,l,dig,i,j;
char bin[20],bin1[20],bin3[20],ch;
clrscr();
printf("Enter a binary number\n");
scanf("%s",bin);
l=strlen(bin);
l--;
i=0;
j=1;
while(i<=l)
{
 dig=(bin[j]-48);
 dec+= dig*pow(2.0,i);
 i++;
j--;
 }
printf("The corresponding decimal number is=%d\n",dec);
i=0;
do
{
 dig=dec%16;
if(dig<=9)
bin1[i]=48+dig;
else
bin1[i]=55+dig;
dec=(dec-dig)/16;
i++;
} while(dec!=0);
bin1[i]='\0';
strrev(bin1);
printf("The corresponding Hexadecimal Number is\n");
puts(bin1);
getch();
}
```

Output:

```
Enter a binary number
11110110
The corresponding decimal number is=246
The corresponding Hexadecimal number is
F6
```

Explanation:

- In order to avoid the limitations to an integer value, the binary number is read as a string. Its length is calculated using **strlen**() function. Then with in the while loop the characters of this string **bin** is converted in to their digital values. And these digital values are multiplied by appropriate power of 2.

- The decimal equivalent of the binary number is processed to obtain the hexadecimal equivalent.For this the decimal number is divided by 16 and the remainder is stored in **dig**. If the value of the digit is with in 9, it is converted in to the appropriate number, if the value of the digit is more than 9, it is converted to English letter (A to F).

Example 19: This program converts hexadecimal number into its decimal equivalent.

```
#include<stdio.h>
#include<string.h>
#include<math.h>
main()
{
 int dec=0,l,i;
char str[30];
printf("Enter a hexadecimal number\n");
scanf("%s",str);
l=strlen(str);
l--;
for(i=0;i>=0;i--)
{
 if(str[i]>='A' && str[i]<='F')
dec+=(str[i]-65+10)*pow(16,l-i);
if(str[i]>='0' && str[i]<='9')
dec+=(str[i]-48)*pow(16,l-i);
}
printf("The    corresponding    decimal    equivalent    is
=%d\n",dec);
getch();
}
```

Output:

```
Enter a hexadecimal number
F6
The corresponding decimal equivalent is = 246
```

Explanation:

- The methodology adopted here is very similar to the previous example. The hexadecimal number is read in string format and each character of this string is converted to its equivalent numerical value . So obtained value is then multiplied by 16 raised to its appropriate power.

Example 20: This program reads a small post fix expression (like 32+) and finds its value

```
#include<stdio.h>
main()
{
int f1,f2,i;
char str[3];
clrscr();
for(i=1;i<=5;i++)
```

```
{
 printf("Enter an expression (like 32+)\n");
 scanf("%s",str);
 f1=con(str[0]);
 f2=con(str[1]);
 amal(f1,f2,str[2]);
}

 con(char c)
{
  if(c>=48 && c<=57)
  return(c-48);
}

amal(int a,int b, char c)
{
  switch(c)
   {
 case '+': printf("The answer is %d\n",a+b);
                break;
 case '-' : printf("The answer is %d\n",a-b);
                break;
 case '*' : printf("The answer is %d\n",a*b);
                break;
 case '/' : printf("The answer is %d\n",a/b);
                break;
 case '%' : printf("The answer is %d\n",a%b);
                break;
 }
getch();
}
```

Output:

```
Enter an expression(like 32+)
32+
The answer is 5
Enter an expression(like 32+)
64-
The answer is 2
Enter an expression(like 32+)
23/
The answer is 0
Enter an expression(like 32+)
32%
The answer is 1
Enter an expression(like 32+)
33*
The answer is 9
```

Explanation:

- This program takes five expressions and calculates the value of the expression. The expression is taken as a string and the individual characters of the string is separated in to the variables **f1** and **f2**. The variables **f1,f2** and the operator (**str[2]**) is sent to function **amal**. Based on the type of operator, the switch statement in the function **amal** will evaluate the expression.

Points to Rehearse

- A String is just an array of Characters.
- The end of the string is recognised by the presence of a null character('\0') .
- The function **strlen**() returns the length of the string. The function **strcmp**() compares two strings. It returns zero if the two strings are
- identical. A negative value returned by this function indicates that the first argument is alphabetically less than the second. A positive value means the first argument is alphabetically more than the second.
- The function **strcat**() joins two strings.
- The function **strcpy**() copies the contents of one string to other.

Points to Ponder

C- compiler has a large source of standard functions . The list is presented in the table below.

Function	What it does
strlen()	Finds length of string
strcpy()	Copies a string to another
strcat()	Joins one string at the end of another
strlwr()	Conversion in to lower case
strupr()	Conversion in to uppercase
strncat()	Places first n characters of a string at the end of the other.
strncpy()	Copies first n characters of one string to another.
strcmp()	Compares two strings.
strncmp()	Compares first n characters of two strings
strcmpi()	Compares two strings with out regard to case.
strnicmp()	Compares first n characters of two strings with out regard to case.
strev()	Reverses a string
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.
strset()	Sets all characters of a string to a given character.
strnsct()	Sets n characters of a string to a given character

Character array elements can be accessed exactly in the same way as an elements of an integer array. Thus the notations given below refers to the same element. Taking j=2,

```
 word[i];
*(j+word);
*(word+j);
 j[word];
```

It is possible to construct array of **pointers** so that, it can hold number of addresses of strings.

```
 char *address[]= {
                    "Jayaram"
                    "Jagadeesh"
                    "Rajendra"
                    "Prasad"
                   };
```

In the above example, **address[]** is an array of **pointers**. It contains base addresses of respective names. That is base address of "Jayaram" is stored in **address[0]**, base address of "Jagadeesh" is stored in **address[1]** and so on…

It is possible to construct two dimensional array of characters. For example,

```
   char name_list[4][10] = {
                             "POOJA"
                             "PADMINI"
                             "AISHWARYA"
                             "SHARANYA"
                            };
```

In storage this is how it looks,

P	**O**	**O**	**J**	**A**	**\0**					
P	**A**	**D**	**M**	**I**	**N**	**I**	**\0**			
A	**I**	**S**	**H**	**W**	**A**	**R**	**Y**	**A**	**\0**	
S	**H**	**A**	**R**	**A**	**N**	**Y**	**A**	**\0**		

Try Yourself

1. Count number of ovals present in a given word.
2. Write a program to count repetitive characters.
3. Read a binary number as a string and count number of 0’s and 1’s.

Session 13:

Using Structure, Union and Enumerations

Structure

We have seen in earlier sessions, how ordinary variables can hold only one data and how arrays hold number of data items of same type. Now we shall see in this session that it is possible to hold different data types under one name. This facility is provided by structure.

A **structure** is a collection of data items of different types that are grouped and referenced under one name. In addition to the built-in data types, you can create combinations of built-in types by using structure. **Structure** has the following advantages:

- It is the most flexible and versatile data type, that allows you to create and manipulate sets of objects of mixed types, including other structures, arrays and pointers of any kind.
- It provides way for arranging logically related data items in a group.
- It simplifies to build a complex data structures in an easily understandable way.

Declaration of a Structure

Structure begins with a keyword **Struct** followed by the **structure name** which is assigned by the user. The variables of different **data types** are then grouped together by using **curly braces**. Finally the structure should end with a semicolon.

The general form of a structure declaration is:

```
struct Struct_name
{
   datatype 1;
   datatype 2;
         .
         .
         .
};
```

For example, the following structure has 3 data items of different types:
Student_name - a String data type,
semester - an Integer data type,
percent_scored - a Floating point data type.

```
struct student
{
        char student_name[40];
        int semester;
        float percent_scored;
};
```

The structure **student,** creates a data type student similar to built-in types like int, float etc,. We can use student data type to declare variables .

```
student class[60];
student college[400];
```

The above statements declares class[60] and college[400] as variables of type student.

Accessing/Initializing the components (data items) of a structure

A **dot** operator is used to access/initialize an individual structures components if the structure is global or declared in the function that references it. Otherwise an **arrow** operator is used. For example, to access/initialize a variable student_name, semester, percent_scored , of type class , we can use:

```
strcpy(class.student_name , “RAMA”);
class.semester = 1;
class.percent_scored  = 60.8;
```

Another way of initializing structure is:

```
struct student class ={“ RAMA “, 1, 60.8};
```

We can also input/output structure data items like this:

```
scanf(“%s”, class.student_name);
printf(“%s”, class.student_name);
scanf(“%d”, &class.semester);
printf(“%f ”,class.percent_scored);
```

Example 1:

This program accepts student name, semester in which the student studying and marks scored in three subjects Eng. Mechanics, Computer program and Mathematics and print the same along with average marks scored by the student.

```
/* Example to illustrate the use of a structure */
#include<stdio.h>
#include<conio.h>
/* Global structure declaration */
struct student
{
 char name[40];
 int semester;
 int eng_mech,comp_prog,maths;
 float avg_marks;
};
main()
{
  /* declaring structure variables */
  struct student marks_card;
/* Accepting the variable value through keyboard */
  clrscr(); /* clears the screen */
  printf("Enter Student Name: ");
  scanf("%s", marks_card.name);
  printf("Enter Semester(1-8) : ");
  scanf("%d", &marks_card.semester);
  printf("Enter Marks scored in Eng. Mech, Computer
prog.and Mathematics");
  scanf("%d%d%d", &marks_card.eng_mech,
           &marks_card.comp_prog,
           &marks_card.maths);
  /* calculating Average marks */
  marks_card.avg_marks = (marks_card.eng_mech +
               marks_card.comp_prog +
               marks_card.maths) / 3.0 ;
  printf("%s %d %d %d %d", marks_card.name,
             marks_card.semester,
             marks_card.eng_mech,
             marks_card.comp_prog,
             marks_card.maths);
  printf("\nAverage Marks = %f", marks_card.avg_marks);
  printf("\nPress any key to continue");getch();
}
```

```
Output:
Enter Student Name: Aishwarya
Enter Semester : 1
Enter Marks scored in Eng. Mech, Computer prog.and
Mathematics 100 70 100
Aishwarya 1 100 70 100
Average Marks = 90.000000
Press any key to continue
```

Explanation:

- The structure is named as student, and is made up of name, semester , marks scored in 3 subjects and average marks.
- marks_card is a variable of type student.

- The marks_card structure data items are read through key board.
- Average marks is calculated and printed.

Example 2:

This program Initializes(All the Structure variables at a time) student name, semester in which the student studying and marks scored in three subjects Eng. Mechanics, Computer program and Mathematics and print the same along with average marks scored by the student.

```
/* Example to illustrate the use of structure  */
#include<stdio.h>
#include<conio.h>
/* Global structure declaration */
struct student
{
 char name[40];
 int semester;
 int eng_mech,comp_prog,maths;
 float avg_marks;
};
main()
{
  /* declaring structure variable and Initialising the
variables */
  struct student marks_card = {"Aishwarya",1,100,70,100};

     clrscr(); /* clears the screen */
  /* calculating Average marks */
  marks_card.avg_marks = (marks_card.eng_mech +
                 marks_card.comp_prog +
                 marks_card.maths) / 3.0 ;
  printf("%s %d %d %d %d", marks_card.name,
               marks_card.semester,
               marks_card.eng_mech,
               marks_card.comp_prog,
               marks_card.maths);
  printf("\nAverage Marks = %f", marks_card.avg_marks);
  printf("\nPress any key to continue");getch();
}
Output:
Aishwarya 1 100 70 100
Average Marks = 90.000000
Press any key to continue
```

Explanation:

- The structure is named as student, and is made up of name, semester , marks scored in 3 subjects and average marks.
- marks_card is a variable of type student.
- The marks_card structure data items are initialised in a block enclosed between curly braces.
- Average marks is calculated and printed.

Example 3:

This program Initializes(One structure variable at a time) student name, semester in which the student studying and marks scored in three subjects Eng. Mechanics, Computer program and Mathematics and print the same along with average marks scored by the student.

```
/* Example to illustrate the marks scored by the student
*/
#include<stdio.h>
#include<conio.h>
/* Global structure declaration */
struct student
{
 char name[40];
 int semester;
 int eng_mech,comp_prog,maths;
 float avg_marks;
};
main()
{
/* declaring structure variable and Initialising the
variables*/
  struct student marks_card;
   strcpy(marks_card.name,"Aishwarya");
  marks_card.semester = 1;
  marks_card.eng_mech = 100;
  marks_card.comp_prog = 70;
  marks_card.maths = 100 ;

     clrscr(); /* clears the screen */




  /* calculating Average marks */
  marks_card.avg_marks = (marks_card.eng_mech +
                 marks_card.comp_prog +
                 marks_card.maths) / 3.0 ;
  printf("%s %d %d %d %d", marks_card.name,
               marks_card.semester,
               marks_card.eng_mech,
               marks_card.comp_prog,
               marks_card.maths);
  printf("\nAverage Marks = %f", marks_card.avg_marks);
  printf("\nPress any key to continue");getch();
}
```

Output

```
Aishwarya 1 100 70 100
Average Marks = 90.000000
```

```
Press any key to continue
```

Explanation:

- The structure is named as student, and is made up of name, semester , marks scored in 3 subjects and average marks.
- marks_card is a variable of type student.
- The marks_card structure data items are initialised one at time and string copy function is used to initialise the character data type.
- Average marks is calculated and printed.

Example 4:

This program accepts Mailing addresses of a person and print the same using structure.

```
/* Program to Generate Address Labels using Structures */

#include<stdio.h>
/* Global declaration of a Structure and Variable to hold
Address */
struct addre{   char name[30];
          char add1[30];
          char add2[30];
          char city[30];
          char state[30];
          char pin[7];
          } address[100]; /* declaring an array variable
address which can store up to 100 addresses */


main()
{
 int i,n;
 clrscr(); /* clears the screen */
 printf("\n Enter Number of Addresses You want to enter :
");
 scanf("%d",&n);
 /* Storing Addresses in a Structure Array */
 flushall(); /* Clears the buffers for input streams */
 for(i=0;i<n;i++)
  { printf("\n Enter address No.: %d\n",i+1);
    printf("\n Name: ");
    gets(address[i].name);
    printf(" Address: ");
    gets(address[i].add1);
    printf(" Street: ");
    gets(address[i].add2);
    printf(" City: ");
    gets(address[i].city);
    printf(" State: ");
    gets(address[i].state);
    printf(" Pincode: ");
    gets(address[i].pin);
  }
 /* Printing Address */
  printf("\n Addresses You have entered are :\n");
 for(i=0;i<n;i++)
  { printf("\n Address No.: %d\n",i+1);
    puts(address[i].name);
    puts(address[i].add1);
    puts(address[i].add2);
    puts(address[i].city);
    puts(address[i].state);
    puts(address[i].pin);
  }

  printf("\n Press any key");
  getch();


}
```

```
Output:
Enter Number of Addresses You want to enter : 2

Enter address No.: 1

Name: D.S. Rajendra Prasad
Address: Sri Lingeswara Sadana
Street: 10th Cross, S.S. Puram
City: Tumkur
State: Karnataka
Pincode: 572102

Enter address No.: 2

Name: M.A. Jayaram
Address: Professor,
Street: C.E.D., S.I.T.
City: Tumkur
State: Karnataka
Pincode: 572103

Addresses You have entered are:
 Address No.: 1
D.S. Rajendra Prsad
Sri Lingeswara Sadana
10th Cross, S.S. Puram
Tumkur
Karnataka
572102
 Address No.: 2
M.A. Jayaram
```

```
Professor
C.E.D., S.I.T.
Tumkur
Karnataka
572103

 Press any key
```

Explanation:

- The structure is named as **addre**, and is made up of Characters Array of name, Address , street, City, State and Pincode.
- **address** is a variable of type addre which can store up to 100 addresses.
- The addresses (structure data items) are input through the key board.
- The stored data addresses are printed.

Union:

Union is the special form of structure, in which two or more variables share the same memory, but only one of them can be accessed at any moment.

Declaration of a Union:_ Union begins with a keyword **union** followed by the **union name** which is assigned by the user. The variables of different **data types** are then grouped together by using **curly braces**. Finally the union should end with a semicolon. Unions are declared and accessed using the same syntax as for structures.

The general form of a Union declaration is:

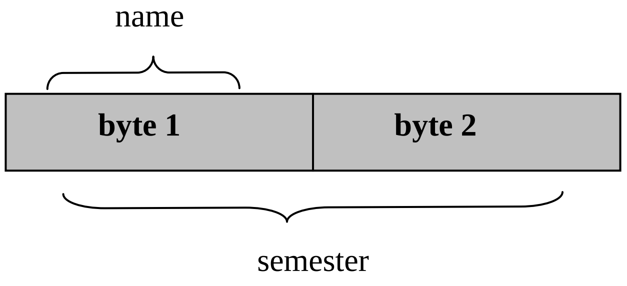
```
union  Union_name
{
   datatype 1;
   datatype 2;
          .
          .
          .
}Union_variables ;
```

The components/data items of an union overlay each other.

For example, the following Union has 2 data items of different types:
name - a Character data type,
semester - an Integer data type,

union *student*
{
    char name;
    int semester;
} *student_data***;**

In the memory it looks as shown below:



Example 5:
The following program illustrate the use of union:

```
/* Program to illustrate the use of Union data type
*/
#include<stdio.h>
#include<conio.h>
/* Global declaration of an Union */
union example
{
 char name;
 int marks;
 float avg;
};
main()
{
 union example ex;  /* define an union variable */
 clrscr(); /* clears the screen */
 ex.marks = 500; /* Initialise the union variable
one at time */
 printf("%d  ", ex.marks);
 ex.avg = 68.50;
 printf("%f  ", ex.avg);
 ex.name = 'r';
 printf("%c  ",ex.name);
 printf("\nPress any key to continue");getch();
}
 Output:
500  68.500000  r
Press any key to continue
```

Explanation:

- The union is named as an example , and is made up of name, marks and average marks.
- ex is a variable of type union.
- The ex union data items are initialised one at a time.
- The values of union data items are printed.

Enumeration:

An enumeration is a list of values of same data type defined under one name or It is simply a specification of the list of values that belongs to the enumeration.

Declaration of an Enumeration: An enumerated data type is created by using a keyword **enum**, followed by the name of the enumerated data type, followed by a list of values. The general form of enumeration type is:

enum enum_name**{**list of values**};**

For example, the following declares a enumeration data type for the months of a year:

enum months{jan,feb,mar,apr,may,jun,jul,aug,sep,oct,nov,dec);

To declare a variable year of enumerated type months:

enum months year;

The values jan,feb,mar are the only values permitted to assign to a variable year:

Year = nov;

Enumerated data types are treated internally as integers. The first name in the list is set to a value 0 and the second name is set to a value 1 etc.,

Example 6:

The following program illustrates the use of Enumerated data type.

```
/* Program to illustrate the use of Enumerated data type
*/
#include<stdio.h>
#include<conio.h>
```

```
/* declaring an enumeration data type */
enum
months{jan,feb,mar,apr,may,jun,jul,aug,sep,oct,nov,dec};
main()
{
 int sum,diff;
 enum months year1,year2;  /* define a variable */
 /* Initilize the variables */
clrscr(); /* clears the screen */
 year1 = aug;
 year2 = feb;
 /* we can perform arithematical operations */
 sum = year1 + year2;
 diff = year1 - year2;
 printf("Sum = %d ", sum);
 printf("Difference = %d ", diff);
 /* we can also perform comparions */
 if( year1 >= year2 ) printf("\n %d  >  %d",year1,year2);
 printf("\nPress any key to continue");getch();
 }

 Output:
Sum = 8 Difference = 6
 7  >  1
Press any key to continue
```

Explanation:

- The Enumeration is named as a months, and is initialised with the list of months.
- year1 and year2 are the variables of type months.
- The year1 and year2 are then initialised suitably.
- The sum and difference of year1 and yera2 are calculated and printed.
- year1 and year2 are then compared and print the result.

Points to Rehearse

Structures are powerful and versatile way of grouping variables of different data types into single entity. Structures are used to build complex logically related data types in a simple understandable way.
Structure components are referenced with a dot operator or with an arrow operator.

Student_class.name

Student_pointer-> member_variable

The following structure declaration is wrong because:

Struct Mystr;
{

```
  int physics;
  flaot chemistry;
}
```

- ➢ The keyword starts with a capital letter.
- ➢ It ends with a semicolon at the begging of the structure it self.
- ➢ Spelling mistake of the keyword float.
- ➢ Missing semicolon at the end of a structure.

Unions are used to save memory, since two or more variables share the same memory space.

Unions are effectively used in situations where only one data item is active at any given time.

Enumerated data type can be advantageously used in a situation where there is a specific list of values to be assigned to a variable of enumerated types.

Points to Ponder

Structure variables can be used to perform mathematical, relational and logical operations as usual as ordinary data's.
Arrays of structures can be defined and used to create a database of related information.
For example:

```
Struct student
{
  char name[40];
  int subject1;
  int subject2;
}student_marks[50];
```

Defines an array of structure called student_marks, that consists of 50 elements.

Structures can be nested with another structure. Structures can be passed to a function as usual as passing of ordinary data's.
Structures of the same type can be assigned:

```
struct_var1 = struct_var2;
```

Unions are declared and accessed in a same way as that of structures. Union allows to share the same memory space for different data types that are declared in an union.

Enumerated data types allow us to declare a list values of same data type under a single name.

Try Yourself

1. Using a structure, writes a program to accept item number, item name, items cost and print the same.

2. Using a structure, writes a program to accept item number, item name, items cost for 5 items. Calculate the total cost of 5 items and print the result.
   (Hint: use array of structure)

SESSION 14

File Handling

Introduction

We know that computers have amazing capacity of easily storing and analyzing huge quantities of data. The data to be processed by CPU is stored in Main memory or Random Access Memory(RAM). In RAM, the data handling takes place at very high speeds.
But the disadvantage here is that, the RAM cannot be used to store information for ever, because when the computer switched off, data in RAM is lost. Then how is data stored in a computer ? it is by use of permanent storage devices like hard disks, floppy disks and CD-ROM's. The system can write data to a floppy disk or hard disk, from the RAM of the computer to store it permanently. When processing is to be done, data stored in the permanent storage can be transferred back to the RAM. Thus when process is on, there is something in RAM.

What are files?

In every permanent storage device mentioned above(for ex. Floppy disk, hard disk etc.,) data is stored in the form of files. To grasp this concept, we can compare the floppy disk or hard disk of a computer to a bag containing several subject notes, each notes thus forms a " Folder " or " Directory " and each notes contains several sheets containing information, each sheet may be treated as " Files ". So files are storehouses of information. In the above example directory is a synonym for note book and a page/sheet containing some write-up is a synonym for a file.

To extend further, it is possible to generate or create different types of files like Program files containing Programs, Textual documents like memo, letter, class notes etc. , sound files, videos, picture files, data files, so on. In this session we shall learn to create a file, write into a file, read a file, manipulate a file and close a file. We shall deal with all these file related tasks one by one.

File Related Operations

To Open a file

It is required that, before reading from a file or writing into a file, we should first open a file. This process of opening of a file builds a connection between the file and the program. The input/output processes can be done if and only if the file is opened. It is common to use a function to open a file. The general syntax is:

```
FILE *fptr, *fopen();
```

The above declaration line is a must when we decide to use files in a program. Here, FILE(Should always in Uppercase letters) represents data type. fptr is called file pointer. File pointer enables to identify the to read or write. fopen() is a function which reforms a file type pointer. We may now open a file writing students data into a file.

```
fptr = fopen("students.dat", "w");
```

Thus we can see that the function fopen has two arguments, the first argument is name of the file and the second argument is the mode of file.

File Open Modes

Write only mode: For writing into a file, this mode is to be used. C compiler checks for the existence of the file specified, if file is not created earlier(first creation) the file the specified name will be created. If file already exists, the old data gets erased and new(fresh) data will be written in to the file. It is not possible for the system create a file for reasons such as exhaust of secondary storage, fopen will return NULL to file pointer. This NULL is defined stdio.h.

```
Ex: fptr=fopen("marks.dat", "w);
    fptr=fopen("salary.dat","w");
```

Here "w" is used to designate write only mode

Append mode: The append mode is specified if there is a need to add some extra information into already existing file. By using this mode some more information may be added at the end of the file.

```
Ex: fptr=fopen("marks.dat", "a");.
    fptr=fopen("salary.dat", "a");
```

Here "a" is used to designate append mode

Read only mode: Only read operation. Compiler checks if not found returns NULL pointer.

```
Ex: fptr=fopen("marks.dat", "r");.
    fptr=fopen("salary.dat", "r");
```

Here "r" is used to designate read only mode.

Write Plus read mode: It is similar to write mode but using this mode, it is possible to read the data stored in the, without closing and opening the file. If a file does not exists, a file name specified will be created, if it already exists, it gets destroyed.

Ex: fptr=fopen("Student.dat", "w+");.
      fptr=fopen("sales.dat","w+");
Here "w+" will enable above said working

Append Plus Write mode: It works very similar to the above, but, it does not destroy the data if the file say "student.dat" already exists.

Ex: fptr=fopen("marks.dat", "a+");
Here "a+" is used to designate append plus write mode.

To close a file

After all the file related tasks are over, the opened files should be closed. This is done to avoid corruption of data.

Syntax is fclose(fptr);

No further file related work can be done after appearance of the above line in a program. If files are to accessed again they should again opened.

Filling data into a file and retrieving data out of a file

There are special functions that would facilitate filling data into a file. They are; Getc() and fscanf(). Similarly the output operations can be carried out by using functions putc() and fprintf(). It may be noted that input functions getc() and fscanf() is similar to getch() and scanf(). In the same token, the output functions putc() and fprintf() are similar to putch() anf printf(). We shall explore these functions one by one.

getc() : This function has a single parameter which is the pointer to FILE. This function is meant to read one character at a time from the file pointed by the file pointer. The syntax is :

```
        getc(file-pointer);
```

Assignment operations can also be done using getc().
For example; ch=getc(fp); makes one character to be read from the data file referenced by file pointer fp and is assigned to variable ch.

fscanf() : As told earlier, this works in the same way as that of scanf(). But this function has one extra parameter placed as first one, this extra parameter is file pointer. The syntax is;

```
          fscanf(filepointer , data specifier, variable list);
```

Ex: fscanf(fpt, "%f%f%f",&num1,&num2,&num3);

putc() : This function puts/prints character by character in to the file. This function needs two parameters ie., a character expression and the file pointer which is referencing the file in question. The syntax is;

```
                      putc(ch,fptr);
```

fprintf(): This function is akin to printf(). In syntax, it is similar to fscanf(). It looks like this:

```
  fprintf(fptr, data specifiers,variable list);
```

As an example;

```
 fprintf(fptr,"%d%d%d",&mark1,&mark2,&mark3);
```

To find the end of the file:

In order to find out whether the end of the file is reached or not a function feof() is made use of. This function has only one parameter which is file pointer referencing a file under operation. This function returns true if end of the file is actually reached else it returns false. Also it is customary to use EOF to check the end of the file.

We shall work through the examples to familiarize with file handling techniques.

Example 1:

This program creates a file student.dat and fills it with marks and average marks.

```
/* Program to create a file student.dat and fills with
marks and
   avearge */
#include<stdio.h>
main()
{
 int marks[10],i,tot=0;
 float avg;
 FILE *fptr, *fopen();
 /* a data file now opened in 'a' mode */
 fptr = fopen("student.dat", "a");
 clrscr();
 for(i=0;i<10;i++)
 { printf("\nEnter the marks in subjects : %d \n",i+1);
   scanf("%d",&marks[i]);
   fprintf(fptr,"%d\n",marks[i]);
   tot+= marks[i];
 }
 avg = tot/10.0;
 fprintf(fptr," The average marks is = %f", avg);
 fclose(fptr);
 printf("\n Now open a file name Student.dat to see the
results");
 printf("\n Press any key");
 getch();
 }
```

```
 Output:
Enter the marks in subjects : 1
100
Enter the marks in subjects : 2
90
Enter the marks in subjects : 3
45
Enter the marks in subjects : 4
67
Enter the marks in subjects : 5
89
Enter the marks in subjects : 6
90
Enter the marks in subjects : 7
45
Enter the marks in subjects : 8
56
Enter the marks in subjects : 9
67
```

```
Enter the marks in subjects : 10
89

Press any key
```

Explanation:
The marks in 10 subjects is entered in to a file named student.dat. Then the average is calculated the average is also printed in the file. When the file is opened this is how it looks like.

```
100
90
45
67
89
90
45
56
67
89
 The average marks is = 73.800003
```

Example 2:A file created with the following details of a salesman and display the commission based on sales value.

```
/* Program to read the records from the data file */
#include<stdio.h>
main()
{
FILE *fp;
 /* Local declaration of a structure */
 struct sale{
          int sale_code;
          float sales;
         } sal; /* declaring avariable sales */


 float comm;
 clrscr();
 /* The data file that was already created is opened in
read mode */
 fp = fopen("Sales.dat","r");

 fscanf(fp,"%d %f",&sal.sale_code,&sal.sales);
 printf("\n\n Salesman-Id         Sales
Commission\n");
```

```
 /* Till the end of file is reached, all the records are
read
    from the data file and report is printed on the
screen */
 while(!feof(fp))
 { if(sal.sales <= 1000) comm = 0.00;
   else if(sal.sales >1000 && sal.sales <=10000) comm =
0.05 * sal.sales;
   else comm = 0.1 * sal.sales;
   /* the report is printed now */
   printf("%4d           %10.2f         %10.2f
\n",sal.sale_code,sal.sales,comm);
   fscanf(fp,"%d %f", &sal.sale_code,&sal.sales);
 } /* end of while */

 fclose(fp);
 printf("\n Press any key");
 getch();

 }

 Output:
 Salesman-Id            Sales              Commission
   1                  1000.00                 0.00
   2                 10000.00               500.00
   3                   100.00                 0.00
   4                100000.00             10000.00
 Press any key
```

Explanation:

- The structure is named as sal is used to include salesman data. A flle named **sales.dat** is created and information from it is read to calculate the commission.

Example 3:
This program makes use getc and writes text into a file displays it back.

```
/* Program make use of getc and writes text into a file,
displays it back */

#include<stdio.h>
main()
{
 FILE *fp;
 int ch;
 clrscr();
 fp=fopen("Text.fil","w");
 printf("\nEnter text and press Ctrl + Z to Terminate:\n
");
 /* untill the end of file get characters and write them
in file */
 while( (ch=getc(stdin)) != EOF) putc(ch,fp);
 fclose(fp);
 /* File is reopened for reading */
 fp=fopen("text.file","r");
 /* Display the text, character by character */

 printf("\n Text Back Again \n");
 while(  (ch=getc(fp)) != EOF) putchar(ch);
 fclose(fp);
 printf("\n Press any key");
 getch();
 }

Output:
 Enter text and press Ctrl + Z to Terminate:
 I have been to London ^Z

 Text Back Again
 I have been to London
 Press any key
```

Explanation:

A file named Text.Fil is opened in write mode. A text matter is entered and closed by pressing Ctrl and Z keys. The function **getc** captures the text letter by letter from the standard input **stdin**.

Example 4:
This program creates a file and writes 10 numbers entered by the user and writes back on to the screen from the file.

```
/* Program to illustrate file handling */
#include<stdio.h>
main()
{
 FILE *fp; /* declaring a File pointer */
 int i,a,n;
 clrscr();
 fp = fopen("Input.dat", "w"); /* Open a file named
Input.dat in a
                      write mode */
 printf("\n Enter 10 numbers\n");
 for(i=0;i<10;i++)
```

```
  { scanf("%d",&a); /* Accepts numbers from the keyboard
*/
    fprintf(fp,"%d ",a); /* Writes the numbers into a
Input.dat file */
  }
  fclose(fp); /* Close the Input.dat file */
  flushall(); /* Clears the input streams */
 printf("\n Data Output from a file\n");
 fp = fopen("Input.dat", "r"); /* Open the file in Read
mode */
   /* Read the number from a file and stored in n */
  while( (fscanf(fp,"%d",&n)) != EOF)
  {

    printf("%d\n",n);/*  Prints the number on the screen
*/
  }


  fclose(fp);
  printf("\n Press any key");
  getch();

 }
```

Output:

```
Enter 10 numbers
1
2
3
4
5
6
7
8
9
10

Data Output from a file

1
2
3
4
5
6
7
8
9
```

```
10
```

Press any key

Explanation:

- The file named input.dat first opened in write mode to enter 10 numbers. It is closed and again it is opened in read mode only to display its contents.

Example 5:This program creates a file to store address labels using Structures.

```
/* Program to Create a file store Address Labels using
Structures */

#include<stdio.h>
/* Global declaration of a Structure and Variable to hold
Address */
struct addre{   char name[30];
          char add1[30];
          char add2[30];
          char city[30];
          char state[30];
          char pin[7];
          } address[100];


main()
{
 int i,n;
 FILE *fp;
 clrscr(); /* clears the screen */
 fp=fopen("Address.dat","w");
 printf("\n Enter Number of Addresses You want to enter :
");
 scanf("%d",&n);
 /* Storing Addresses in a Structure Array */
 flushall();
 for(i=0;i<n;i++)
  { printf("\n Enter address No.: %d\n",i+1);
    printf("\n Name: ");
    gets(address[i].name);
    printf(" Address: ");
    gets(address[i].add1);
    printf(" Street: ");
    gets(address[i].add2);
    printf(" City: ");
    gets(address[i].city);
    printf(" State: ");
    gets(address[i].state);
    printf(" Pincode: ");
    gets(address[i].pin);
    /* copy addresses in to a file */
    fprintf(fp,"%s %s %s %s %s %s
\n",address[i].name,address[i].add1,

address[i].add2,address[i].city,address[i].state,address[
i].pin);

   }
   flushall();
  /* Printing Address */
   printf("\n Reading Addresses from file are :\n");
   fp=fopen("Address.dat","r");
   i=0;
  while( (fscanf(fp,"%s %s %s %s %s
%s\n",address[i].name,address[i].add1,

address[i].add2,address[i].city,address[i].state,address[
i].pin)) != EOF)

   { printf("\n Address No.: %d\n",i+1);
     puts(address[i].name);
     puts(address[i].add1);
     puts(address[i].add2);
     puts(address[i].city);
     puts(address[i].state);
     puts(address[i].pin);
     i++;
   }

   printf("\n Press any key");
   getch();


}
```

```
Output:
Enter Number of Addresses You want to enter : 2

Enter address No.: 1

Name: D.S. Rajendra Prasad
Address: Sri Lingeswara Sadana
Street: 10th Cross, S.S. Puram
City: Tumkur
State: Karnataka
Pincode: 572102

Enter address No.: 2
```

```
Name: M.A. Jayaram
Address: Professor,
Street: C.E.D., S.I.T.
City: Tumkur
State: Karnataka
Pincode: 572103

Reading Addresses from file are :
 Address No.: 1
D.S. Rajendra Prsad
Sri Lingeswara Sadana
10th Cross, S.S. Puram
Tumkur
Karnataka
572102
 Address No.: 2
M.A. Jayaram
Professor
C.E.D., S.I.T.
Tumkur
Karnataka
572103

 Press any key
```

Explanation:

- The process is very similar to previous example . But here a structure is used and address labels are entered into a file named address.dat and the contents are again displayed back by opening the file in read mode.

Points to rehearse

- The file is to be opened in write mode if information/data is to be fed. The file is to be opened in read mode if the information is to be read. When file is opened in append mode, it is possible to read, write and update the file.

Points to Ponder

A file is always better than an array because, a file stores good amount of data permanently even if the computer is turned off. While the data stored in an array will not be available once the computer is switched off.

The manner in which end of file indicated is a matter of storage device used and computer.

It is possible to position the file's marker at any specified location in the file. This concept is prevalent in random access of file. The reader advised to go through listed references.

The specialty of **a+** mode lies in the fact that**,** file for appending also permits the file to be read without the need to close and reopen the file.

Try Yourself

3. Using a structure, write a program to accept item number, item name, items cost for 5 items and writes to a file. Calculate the total cost of 5 items and print the result by reading a file. (Hint: use array of structure)