BHARATH COLLEGE OF SCIENCE AND MANAGEMENT, THANJAVUR.
B.Sc INFORMATION TECHNOLOGY –II SEMESTER
(For the candidates admitted from the academic year 2016-2017 onwards)

PROGRAMMING IN C

Objective:
To impart basic knowledge of Programming Skills in C language.

Unit I
Introduction to C – Constants, Variables, Data types – Operator and Expressions.

Unit II
Managing Input and Output operations – Decision Making and Branching – Decision making and Looping.

Unit III
Arrays – Character Arrays and Strings – User defined Functions.

Unit IV
Structures and unions – Pointers – File management in C.

Unit V
Dynamic memory allocation – Linked lists- Preprocessors – Programming Guide lines.

Text Book:
1. Balagurusamy E .,Programming in ANSI C , Sixth Edition, McGraw-Hill, 2012

Reference Book:
1. R.S.Bichkar, Programming with C, University Press, 2012

*****

UNIT – I

(Introduction to C – Constants, Variables, Datatypes – Operators and Expressions.)

INTRODUCTION TO C

C is one of the most popular computer languages. It is an outgrowth of earlier languages called ALGOL , BCPL (Basic Combined programming language) and B. C language was developed by Dennis Ritchie at Bell Lab in 1972. UNIX operating system was coded almost entirely in C.

IMPORTANCE OF C:

1. It is a Robust language i.e its built -in functions and operators can be used to write any complex programs.
2. It is well suited for writing both system software and business packages.
3. Programs written in C are efficient and fast because it has variety of data types and powerful operators.
4. There are only 32 keywords and its strength lies in built-in – functions.
5. C is highly portable. C programs written for one computer can be run on another with little or no modification.
6. C is well suited for structured programming.
7. C has ability to extend itself.

BASIC STRUCTURE OF C PROGRAM:

Documentation Section
Link Section
Definition Section
Global Definition Section

main() function Section
{
Declaration Part
Executable Part
}

Sub – Program Section Function - I : : Function - n

Documentation Section: It consists of comment lines giving the name of the program, the author etc.

Link Section: It provides instructions to the compiler to link functions from the systems library.

Eg:- #include <stdio.h> , #include<math.h> , #include<stdlib.h> .

Definition Section: It defines all symbolic constants. Eg:- #define

Global Variable Declaration Section: There are some variables that are used in more than one function.Such variables are called global variables declared in this section i.e. outside of all functions.

main() function section: Every c program must have one and only one main() section.It consists of two parts. i)Declaration part: Declares all the variables that are used in the executable part. ii) Executable part: This contains all the executable statements. These two parts must appear between {}. All statements in the declaration and executable parts end with a semicolon.

Subprogram Section: It contains all the user defined functions that are called in the main function.

Let us explain the above structure using example.

```
/* printing a message */
#include<stdio.h>
main()
{
    /*….. Printing begins …….*/
    printf("I See");
   /*….. Printing ends …….*/
```

}

main : It is a special function used by C to tell the computer where the program start.Every program must have exactly one main function

( ) : the empty pair of parenthesis indicates that the function main has no arguments.

{ } : The opening and closing braces marks the beginning and end of the main function. All the statements between these two braces form the function body. It contains a set of instruction to perform the given task.

In the above program, function body contains 3 statements. Where,

i) printf is an executable statement. It is predefined i.e it is already written and compiled and linked with our program at the time of linking.
ii) The printf function causes everything between the starting and the ending quotation marks to be printed out.
iii) Every statement in C must end with semicolon (;).

/* ….. */ : comment lines. These are not executable. It is for understanding.

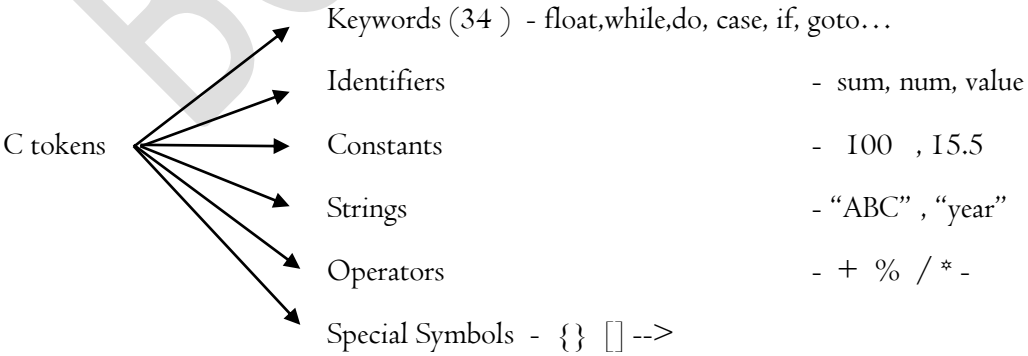
CONSTANTS, VARIABLES AND DATA TYPES

Programming languages are designed to process certain kind of data consisting of characters and strings and to provide useful output known as information. The task of processing of data is accomplished by executing a sequence of precise instructions called a program. These instructions are formed using certain symbols and words according to some rigid rules known as syntax.

Character Set: The characters in C are grouped into the following categories.

- Letters (A – Z , a-z)
- Digits (0…9)
- Special Characters (, . ; ? ‘ ! “ / \ | ~ _ + = { } [] % > < & ^)
- White Spaces.

C Tokens: In C program the smallest individual units are called tokens. There are 6 tokens.



Keywords and Identifiers:

All keywords have fixed meaning and these meanings cannot be changed. It serves as the basic building blocks for program statement. The following words are reserved for use of Keywords. We must not use them as variables or identifiers.

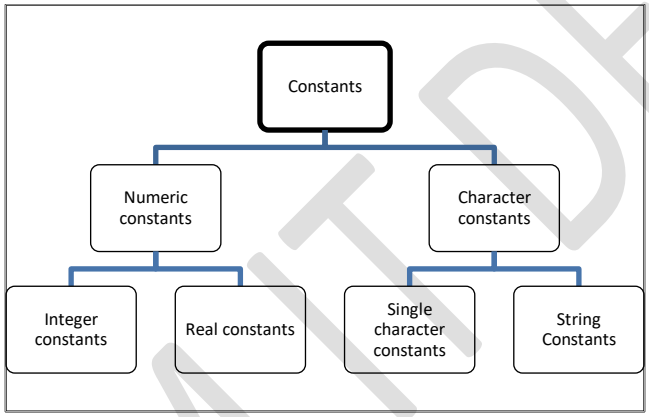
Eg/- if, for, goto, else, auto, break,while, void, int, double, float, char, const, do …

Identifiers refer to the names of variables, functions and arrays. These are user-defined names. The rules for identifiers are,

1. First character must be an alphabet.
2. Must consist of only letters, digits or underscore.
3. Cannot use a keyword.
4. Must not contain white space.

CONSTANTS:

Constants in C refer to fixed values that do not change during the execution of a program.



I. Integer Constants: An integer constant refers to a sequence of digits. They do not have decimal points. There are 3 types of integers.

a. Decimal Integer Constant: It can consist of any combination of digits taken from the set 0 through 9. Spaces, commas and non- digit characters are not permitted between digits.
Eg:- 0 743 -321 +76

b. Octal Integer Constant: It consists of digits taken from 0 through 7. The first digit must be 0. Eg:- 0743 037 0

c. Hexadecimal Integer Constant: It must begin with either 0x or 0X . It can be followed by combination of digits from 0 through 9 and A through F. Eg:- 0x 0xI 0x7FF 0xabcd

II. Real constants or Floating Point constants: Floating Point constants are decimal notation, having a whole number followed by a decimal point and the fractional part.

Eg:- 0.008 -0.75

A real number may also be expressed in *exponential* or *Scientific notation.*

Eg:- 215.65 may be written as 2.1565e2 in exponential notation. e2 means multiply by 10^2 .

The general form is :

mantissa e exponent

The mantissa is either a real number or an integer. The exponent is an integer number.

Eg:- 7500000000 =7.5e9 or 75e8

III. Single Character Constants: One character surrounded by single quotes denotes a character constant. Eg:- 'A' '5' 'x' ';' '?'
Character constants have integer values known as ASCII values.
Eg:- printf("%d" , 'a'); = printf("%d" , '97');

IV. String Constants: A String constant is sequence of characters enclosed in double quotes. The characters may be letters, numbers,special characters and blank space.
Eg:- "Hello" "1979" "well done" " ?....!" "x"

Backslash Character constants: (Escape Sequences)

These constants are used in output function.

'\n'	newline	'\\'	backslash
'\t'	horizontal tab	'\"	single quote
'\b'	backspace	'\0'	null character
'\r'	carriage return	'\"'	double quote
'\f'	form feed		

VARIABLES:

A variable is a data name that may be used to store data value.Unlike constants that remain unchanged during the execution of the program, a variable may take different values at different times during execution. A variable can be of letters, digits and underscore subject to the following conditions.

1. They must begin with a letter or an underscore.
2. The maximum length of the variable name can be 31 characters. But length should not exceed 8 characters because only first 8 characters are treated as significant.
3. Uppercase and lowercase are significant.
4. It should not be a keyword.
5. White space is not allowed between characters.

Eg:- for *valid* variables: value , T_raise, x1 , ph_value

Eg:- for *invalid* variables: 123 (area) % 25th

Types of variables:

Depending upon the data contained in the memory location, the variables are classified as follows.

- Integer Variables
  - Long interger
  - Short integer
  - Unsigned interger
  - Integer
- Real Variables
  - Floating point

- Double
- Character Variables
  - Signed character
  - Unsigned character
- String Variable

A variable which may have only integer values is called *integer variable*, that has real values is called *real variable,* that has character data is called *character variable* and that has string data is called *string variable*.

Variable type declaration:

It is compulsory that at the beginning of the program the variable type must be declared. After designing a suitable variable name, we must declare them to the compiler.

Declaration does two things;

i) It tells the computer what the variable name is.
ii) It specifies what type of data the variable will hold.

The declaration of the variable must be done before they are used in the program.

Syntax: `datatype list of variables;`

A) Integer Variable:

The word int is used to store integer values.

Eg:- int i , sum , area;
short int i, sum, area;
long int sum;

According to the size of the numbers we can declare it as short or long etc.

B) Real Variable:

The word float and double is used to determine real variable. Double has more precision.

Eg:- float side, circum;
double side, cir;
long float side;

double and long float have the same effect.But double is used widely.

C) Character Variable:

The word char is used to declare character data.

Eg:- char a, b;
char text;

D)String Variable:

A string is stored as an array of character . Suppose we want to store "PEPSI" to the variable drink , it is done in the following manner.

drink[0] = 'P'
drink[1] = 'E'
drink[2] = 'P'
drink[3] = 'S'
drink[4] = 'I'

drink =

P	E	P	S	I	\0
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]

String variables can be declared in the following format.

```
char  var1[n] , var2 [m] ;
```

where n,m are integer constants which represent the maximum size of the string.

Eg:- char drink [5];

char line[10];

char drink[] ="pepsi";

The number in the square brackets represents the maximum number of characters allowed in each string variable. In the third example the string variable drink is assigned a word of 5 letters. But no number is mentioned in the square bracket. In such cases the variable is assigned the length of the string+1. Thus the length is 6 (5+1). The 6th character is null character which marks the end of the string.

DATA TYPES:

C language is rich in its data types. C supports 3 classes of data types.

i) Primary or fundamental data type
ii) Derived Data type
iii) User- defined data type

Primary Data types:

All C compilers support 5 fundamental data types namely integer (int), character(char), floating point (float), double precision floating point (double) and void.

A. Integer Types:

Integers are whole numbers. C has 3 classes of integer storage namely short int, int and long int in both signed and unsigned.

Type	Size (bytes)	Range
short int	2	
int	2	
long int	4	
unsigned short int	2	
unsigned int	2	
unsigned long int	4	

B. Character Types:

Char is the data type which is used to represent individual characters. The char type will generally require one byte of memory. A char data type may have the identifier signed and unsigned.

Type	Size (bytes)	Range
signed char	I	
unsigned char	I	

C. Floating Point Types:

Floating point or real numbers are stored in 32 bits,with 6 digits precision.There are two types of floating point data types namely float and double.These may also have a qualifier long. Long float may be equivalent to double.Floating point numbers are essentially signed.

Type	Description	Size	Range
float	single precision	4	
double	double precision	8	
long double	extended precision	10	

D. Void Types:

The void type has no values. This is usually used to specify the type of functions. The type of a function is said to be void when it does not return any value to the calling function.

OPERATORS AND EXPRESSIONS:

C support a rich set of operators. An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations. Operators are used in programs to manipulate data and variables. They usually form a part of the mathematical or logical expressions.

C Operators can be classified into 8 categories. They are:

i. Arithmetic Operator
ii. Relational Operator
iii. Logical Operator
iv. Assignment Operator
v. Increment and Decrement Operator
vi. Conditional Operator
vii. Bitwise Operator
viii. Special Operator

Arithmetic Operators:

There are 5 arithmetic operators.

+	addition or unary minus
-	Subtraction or unary minus
*	multiplication
/	division

%	modulo division(remainder of integer division)

Integer division truncates any fractional part. The modulo division produces the remainder of an integer division.

Eg:- a-b , a+b , a*b, a/b , a%b

Here a and b are variables and are known as operands.

- Integer Arithmetic: When both the operands in an arithmetic expression is an integer, the expression is called integer expression and the operation is called integer arithmetic. Integer arithmetic always yields an integer value.

  Eg:- if a=14 and b=4

  a-b=10 a+b=18 a*b=56 a/b=3 a%b=2

- Real Arithmetic: An arithmetic operation involving only real operands is called real arithmetic. A real operands may assume values either in decimal or exponential notation.

  Eg:- a-b=10.5 a+b=18.2 a*b=25.0 a/b=3.2

  The operator % cannot be used with real operands.

- Mixed mode Arithmetic: When one of the operands is real and the other is integer, the expression is called mixed mode arithmetic expression. If either operand is of real type, then only real operation is performed and the result is always a real number.
  Eg:- 15/10.0 = 1.5
  15/10 = 1

Relational Operators:

Comparisons can be done with the help of relational operators. The values of relational expressions is either 0 or 1 . It is 1 if the specified relation is true and 0 if the relation is false. The following are the 6 relational operators.

<	less than
<=	less than or equal to
>	Greater than
>=	Greater than or equal to
==	is equal to
!=	not equal to

Eg:- 10 < 20 is true

20 < 10 is false

Logical Operators:

The 3 logical operators in C are

Operator	Symbol
AND	&&
OR	||
NOT	!

- Logical AND: A compound expression is true when two conditions (expressions) are true.

  `

  ```
  Exp1  &&  Exp2
  ```

  The two expressions must be an integer type.

Situation	Results
True && True	True
True && False	False
False &&False	False
False && True	False

Eg:- a=4 ; b=5 ; c=6

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

(4 < 5) && (5 < 6)

True && True

True

- Logical OR: It has the form

  ```
  Exp1  || Exp2
  ```

  It evaluates to true if either exp1 or exp2 is true.

Situation	Results
True | | True	True
True | | False	True
False | | False	False
False | | True	True

Eg:- a=4 ; b=5 ; c=6

(a < b) || (b > c)

(4 < 5) || (5 > 6)

True && False

True

- Logical NOT: (Negation Operator)

The logical expression can be changed from false to true or from true to false with the negation operator. The general form of the expression is !(Expression)

Situation	Result
!(true)	False
!(false)	True

Eg:- a=4 ; b=5 ; c=6

!(a < b) | | (b < c)

!(4 < 5) | | (5 < 6)

!(True) | | (True)

False | | True

True

Assignment Operators:

It is used to assign a value to a variable. The usual assignment operator is = .

```
Variable = expression;
```

Eg: - x = y +I;

In addition C has a set of *shorthand* assignment operator.

```
Variable  operator = expression;
```

Eg:- x += y+I; i.e x = x + (y +I)

Other shorthand assignment operators are -= , *= , /= , %= .

Multiple assignment statements are also possible.

```
v1 = v2 = v3 = …expression;
```

Eg:- x=y=z=10; i.e x=10 , y=10 , z=10.

Increment and Decrement Operators:

++ and - - are the increment and decrement operators. The operator ++ adds 1 to the operand , while - - subtracts 1 from the operand. Both are unary operators and take the following form

++m; or m++;

- - m; or m- - ;

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

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

We use these operators in for and while loops. While ++m and m++ means the same thing when they form statements independently, they behave differently when they are used in expressions on the right hand side of the assignment statement. Consider the following,

m=5 ;

y= ++m;

Here, the value of y and m is 6. Suppose if we rewrite the above statement as

m=5;

y=m++;

then the value of y would be 5 and m would be 6.

A prefix operator first adds 1 to the operand and then the result is assigned to the variable on the left. On the other hand , the postfix operator first assigns the value to the variable on the left and then increments the operand.

Conditional Operator: (or Ternary operator)

?: is called the ternary operator because it uses three expressions.

```
exp1 ? exp2 : exp3;
```

Where exp1, exp2 and exp3 are expressions. exp1 is evaluated first. If it is true, then the exp2 is evaluated and becomes the value of the expression. If exp1 is false, exp3 is evaluated and its value becomes the value of the expression. Only one of the expression is evaluated (either exp2 or exp3).

Eg:- a=10 ; b=15; x= (a<b) ? a : b ;

Bitwise Operators:

It is used to manipulate data at bit level.

Operators	Meaning
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
<<	Shift left
>>	Shift right

Special Operators:

There are special operators in C to performs particular type of operations.

A. Unary Operators:.They precede their single operands.

Operators	Meaning
*	Contents of the storage field to which a pointer is pointing
&	Address of a variable
Typeof	Forced type conversion

Sizeof	Size of the subsequent data type Or type in byte.

B. Comma Operator:

C uses comma in two ways. The first uses comma as a separator in the variable declaration. Eg:- int a,b,c;

Another use is an operator in an expression for loop.

C. Parentheses for grouping expressions ( ). Eg:- (a + b) * c
D. Membership Operators: . and →

UNIT – II

(Managing input and output operations –Decision making and branching –Decision making and looping.)

DATA INPUT AND OUTPUT:

Inorder to write an interactive C program we require input and output functions.

scanf() – Simple input statement:

The scanf() function is used to read the formatted data items from the keyboard. The format is a user defined data items. It can read or written as defined by the programmer.:

Syntax:

```
scanf( " control strings" , argument list);
```

The control string is a specified format cod to be read from the keyboard and the argument is a user defined variable list.Usually, the argument list of user defined variables must include the address operator (&) as a prefix to the variable.

Eg:- int x,y;

scanf("%d%d", x,y);

The complete format code used in scanf() function for the various data variables is shown below.

Code	Meaning
%c	Read a single character
%d	Read a decimal integer
%i	Read a decimal integer

%e	Read a floating point number
%f	Read a floating point number
%h	Read a short integer
%o	Read a octal number
%s	Read a string
%x	Read a hexadecimal number

printf() - simple output statement:

The printf() function is one of the most important and useful function to display the formatted output data item on the standard output device.

Syntax:

```
printf ( " control strings" , argument list);
```

Where the control strings are user defined format code and the argument list is a set of data items to be displayed in a proper format as defined by the control strings.

The complete format code used in the printf() function for the various data variables is shown below.

Code	Meaning
%c	a single character
%d	Decimal notation
%i	Decimal notation
%e	Decimal floating point
%f	"
%g	"
%o	Octal
%s	String of characters
%u	Unsigned decimal
%x	Hexadecimal
%%	Print a percent sign

Eg:- printf("%10.4f \n",value);

This will display a number atleast 10 characters wide with 4 decimal places

getchar() and putchar() :

These two functions are included in the header file stdio.h library which supports the input and output functions.

getchar() :

Reading a single character gechar can be done using the function.We read a character from the input device usually a keyboard and write it to the output device usually a monitor.

Syntax:

```
Variable_name = getchar();
```

Eg:- #include<stdio.h>

```
main()

{

 char answer;
    printf( "Would you like to know my name ?\n");
    printf("Type Y for yes and N for No");
    answer= getchar();
    if (answer= = 'Y' | | answer = = 'y')
    printf("My name is IT\n");

    else

    printf("You are good for nothing");

 }
```

putchar():

The putchar() function is a complementary function to the getchar(). It is used for writing characters one at a time to the terminal.

Syntax: `putchar(variable_name);`

Eg:- #include<stdio.h>

```
void main()

{

 char alphabet;
  printf( "Enter an alphabet\n");

  putchar('\n');

  alphabet= getchar();

  if (islower(alphabet))

  putchar(toupper(alphabet));

  else

  putchar(tolower(alphabet));

 }
```

DECISION MAKING AND BRANCHING (CONTROL STATEMENTS)

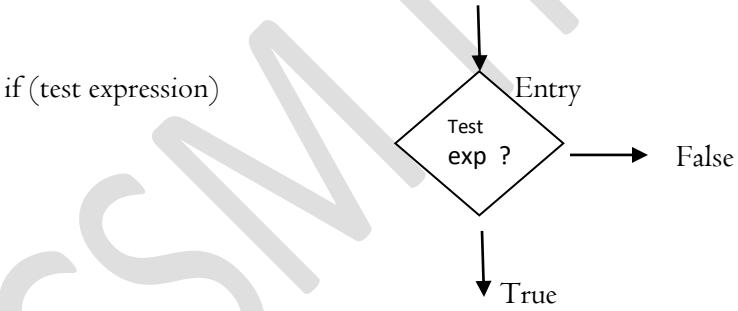
C program is a set of statements which are normally executed sequentially in the order in which they appear. This happens when no options or no repetitions of certain calculations are necessary. But there may be number of situations where we may have to change the order of execution based on certain conditions, or repeat a group of statements until certain specified conditions meet. This involves a kind of decision making to see whether a particular condition has occurred or not and then direct the computer to execute certain statements accordingly.

C language possess such decision making capabilities by supporting the following statements.

i. if statements
ii. switch statement
iii. Conditional operator
iv. goto statement

Decision Making with if statements:

The if statement is a powerful decision making statement and is used to control the flow of execution of statements. It is usually a two-way decision statement and is used in conjunction with an expression. It allows the computer to evaluate the expression first and then depending on the value of the expression (relation or condition) is true or false, it transfers the control to a particular statement.



The program has 2 paths to follow, one for true condition and the other for false condition.

1. <u>The simple if statement</u>:

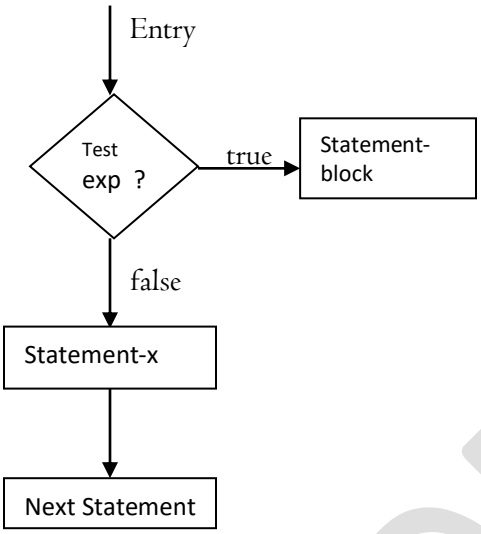
Syntax:

```
if (test expression)
 {
  Statement – block;
 }
```

The statement block may be a single statement or a group of statements. If the test expression is true, the statement- block will be executed, otherwise the statement-block will be skipped and the

execution will jump to the statement-x. When the condition is true both the statement-block and the statement-x will be executed in sequence.

Flow chart:



Example:

```
#include<stdio.h>
main()
{
 int a,b,c,d;
float ratio;
printf("Enter 4 Integer values\n");
scanf("%d%d%d%d",&a,&b,&c,&d);
if(c-d!=0)
  {
   ratio = (float)(a+b)/(float)(c-d);
   printf("Ratio=%f\n',ratio);
  }
}
```

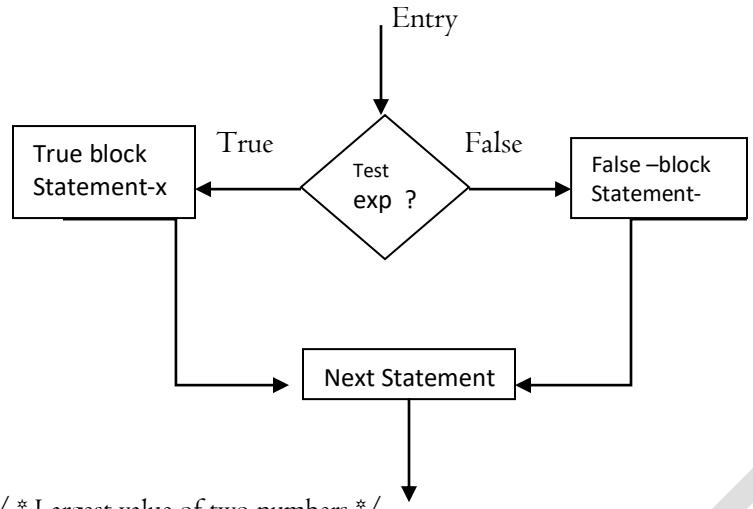
2. The if…else statement:

Syntax:

```
if (test expression)
 {
True statement – block;
 }
else
{
False statement – block;
}
Statement-x;
```

If the test expression is true , then the true –block statement , immediately following the if statements are executed; otherwise, the false block statements are executed, In either case, either true block or false block will be executed, not both. In both the cases , the control is transferred subsequently to the statement-x.

Flow Chart:



Example: / * Largest value of two numbers */

```
#include<stdio.h>
main()
{
float x,y;
printf("Enter 2 values\n");
scanf("%f%f",&x,&y);
if(x>y)
  {
      printf("Largest Value is=%f\n',x);
  }
else
   {
       printf("Largest Value is=%f\n',y);
   }
}
```

3. Nesting of if…else statements:

When a series of decisions are involved, we may have to use more than one if…else statements in nested form.

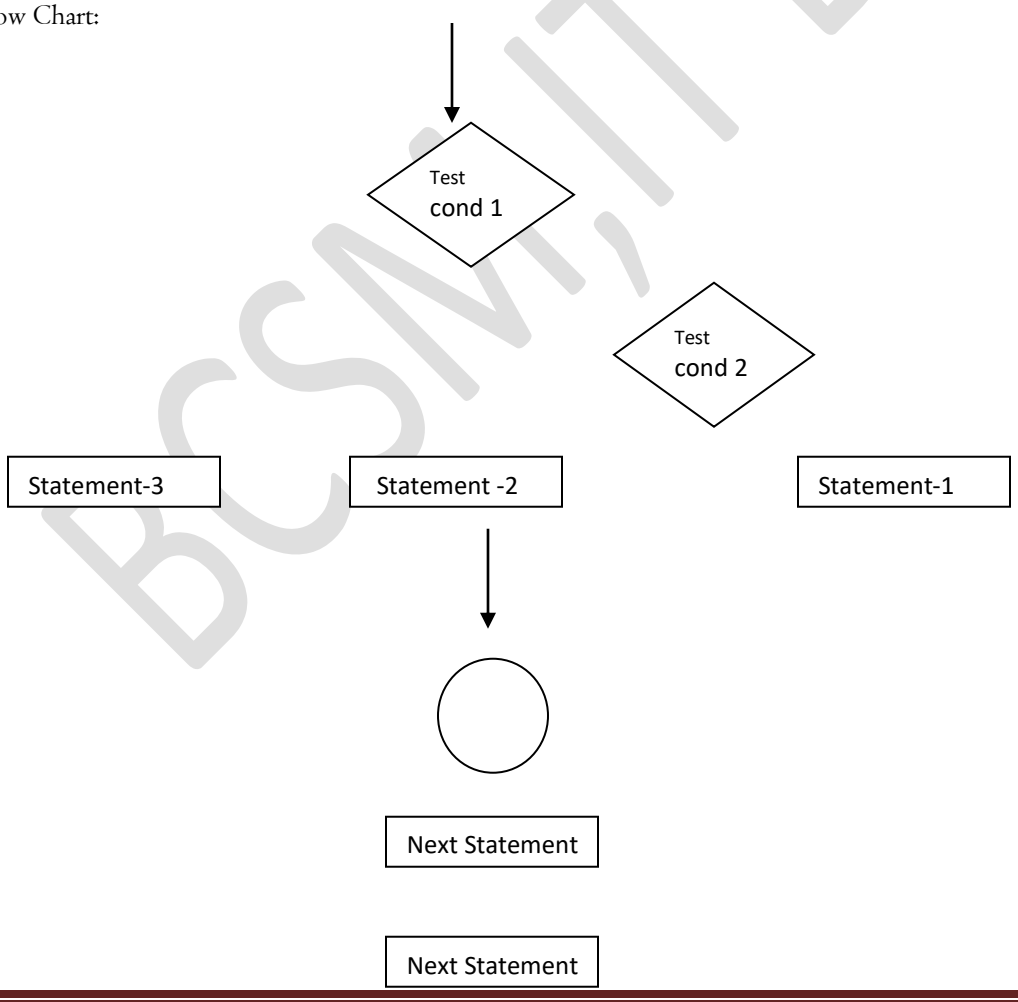
Syntax:

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

```
      {
        statement 2;
       }
}
else
 {
  Statement-3;
 }
Statement-x;
```

If the condition 1 is false, the statement-3 will be executed; otherwise it continues to perform the second task. If the condition2 is true, then statement-1 will be evaluated; otherwise statement-2 will be evaluated and then the control is transferred to statement-x.

Flow Chart:



Example: / * Largest value of three numbers */

```
#include<stdio.h>
main()
{
float x,y,z;
printf("Enter 3 values\n");
scanf("%f%f%f",&x,&y,&z);
if(x>y)
  {
  if (x>z)
       printf("Largest Value is=%f\n',x);
  else
       printf("Largest Value is=%f\n',z);
  }
else
   {
   if( y>z)
        printf( "largest value is=%f\n",y);
   else
        printf( "largest value is=%f\n",z);
   }
}
```

4. The elseif ladder:

When multipath decisions are involved we can put all ifs together.A multipath decision is a chain of ifs in which the statement associated with each else is an if.

Syntax:

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

The conditions are evaluated from top to down. As soon as a true condition is found, the statement associated with it is executed and the control is transferred to the statement-x. When all the n conditions become false, then the final else containing the default-statement will be executed.

Flowchart:

Example: /* Program to display name of the day depending upon the number entered from the keyboard*/

```
#include<stdio.h>
main()
{
int day;
printf("Enter numbers between 1 to 7 \n");
scanf("%d",&day);
if ( day==1)
printf("Monday \n");
if ( day==2)
printf("Tuesday \n");
if ( day==3)
printf("Wednesday \n");
if ( day==4)
printf("Thursday \n");
if ( day==5)
printf("Friday \n");
if ( day==6)
printf("Saturday \n");
if ( day==7)
printf("Sunday \n");
else
printf("Enter the correct number \n");
}
```

The Switch statement:

C has a built-in multiway decision statement known as switch. The switch statement tests the value of a given variable(or expression) against a list of case values and when a match is found , a block of statements associated with that case is executed.

Syntax:

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

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

When the switch is executed, the value of the expression is successfully compared against the values value-1, value-2..If a case is found whose value matches with the value of the expression, then the block of statements that follows the case are executed.

The break statement at the end of each block signals the end of a particular case and causes an exit from the switch statement, transferring the control to the statement-x following the switch.

The default is an optional case. When present, it will be executed if the value of the expression does not match with any of the case values. If not present, no action takes place if all matches fail and the control goes to the statement – x.

DECISION MAKING AND LOOPING

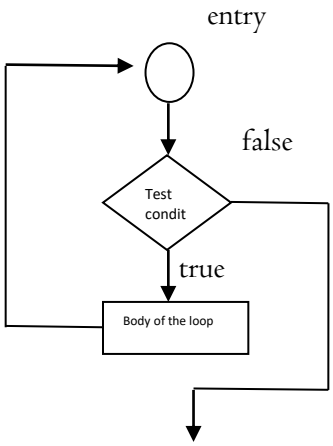
In looping, sequence of statements are executed until some conditions for termination of the loop are satisfied i.e repeating some work is called looping.

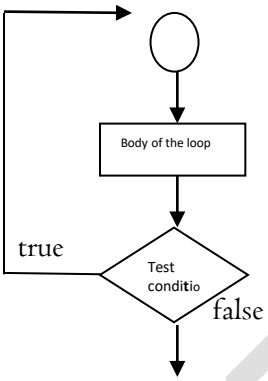
A program loop therefore consists of two segments, one known as the body of the loop and the other known as control statements.

The control statement tests certain conditions and then directs the repeated execution of the statements contained in the body of the loop.

Depending on the position of the control statement in the loop, a control structure can be classified into *entry-controlled loop* and *exit controlled loop.*

Entry control Exit control





In the entry controlled loop, the control conditions are tested before the start of the loop execution. If the conditions are not satisfied, the body of the loop will not be executed.

In exit- controlled loop the test is performed at the end of the body of the loop and therefore the body is executed unconditionally for the first time.

The entry and exit controlled loops are also called as pre-test and post-test loops.

A looping process should include the following four steps:

1. Setting and initialization of a condition variable.
2. Execution of the statements in the loop.
3. Test for a specified value of the condition variable for execution of the loop.
4. Incrementing or updating the condition variable.

C language provides three constructs for performing loop operations. They are:

1. The while statement
2. The do statement
3. The for statement

A. <u>The while statement:</u>

The basic format of the while statement is

```
while(test condition)
{
  Body of the loop
}
```

The while is an entry controlled loop statement. The test condition is evaluated and if the condition is true , then the body of the loop is executed. After execution of the body, the test condition is once again evaluated and if it is true, the body is once again executed. This process of repeated execution of the body continues until the test condition finally becomes false and the control is transferred out of the loop.

Eg:-// program to find sum of first 100 natural numbers 1+2+…100

```
main()
{
    int sum, digit;
    sum=0;
```

```
        digit =1:
        while(digit <=100)
        {
          sum+=digit;
          digit++;
        }
}
```

B. Do – while loop:

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

This takes the form:

```
do
{
  Body of the loop
}
while(test condition);
```

Eg:-

```
// displays odd numbers from 1 to 100
main()
{
        int x=1;
        do
         {
             printf("%d",x);
            x=x+2;

        }
        while(x<=100);

 }
```

C. The for statement:

The for loop is an entry-controlled loop.

The general form is:

```
for(initialization ; test condition ; increment)
{
  Body of the loop
}
```

The execution of the for statement is as follows:

1. Initialization of the control variables is done first using assignment statements such as i=0 and count =0. The variables i and count are known as *loop-control variables.*
2. The value of the control variable is tested using the test-condition. The test-condition is an relational expression, such as i<10 that determines when the loop will exit. If the condition is true,

the body of the loop is executed, otherwise the loop is terminated and the execution continues with the statement that immediately follows the loop.

3. When the body of the loop is executed , the control is transferred back to the for statement after evaluating the last statement in the loop. Now the control variable is incremented using an assignment statement such as i=i+1 and the new value of the control variable is again tested to see whether it satisfies the loop condition. If the condition is satisfied the body of the loop is again executed. This process continues till the value of the control variable fails to satisfy the test condition.

```
Eg: // display numbers from 0 to 100
main ()
{
  int i;
  for(i=0 ; i < = 100 ; i++)
  {
    printf( "%d\t" , i);
  }
}
```

Nested for loops:

C programming allows to use one loop inside another loop. The nesting may continue upto any desired level.

Syntax:

The syntax for a **nested for loop** statement in C is as follows −

```
for ( init; condition; increment ) {

   for ( init; condition; increment ) {
      statement(s);
   }
   statement(s);
}
```

The following program uses a nested for loop to find the prime numbers from 2 to 100 −

```
#include <stdio.h>

int main () {

   /* local variable definition */
   int i, j;

   for(i = 2; i<100; i++) {

      for(j = 2; j <= (i/j); j++)
        if(!(i%j)) break; // if factor found, not prime
```

```
        if(j > (i/j)) printf("%d is prime\n", i);
    }

    return 0;
}
```

BREAKING CONTROL STATEMENTS:

I. Break statement:

The general format of the break statement is: `break;`

The break is a keyword in the C program.

A. Break in switch – case structure:

```
Eg:-  switch(day)
      {
       case'1':
       printf("Monday");
       break;
      case'2'
       printf("Tuesday");
       break;
       default:
    }
```

If there is no break statement in case1, the computer will transfer the control to other cases also.To avoid undesired results normally the break statement will be used in each case section.

B. Break in while, do-while and for loops:

Exiting from a loop can be accomplished by using break statement.When a break is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop.When the loops are nested , the break would only exit from the loop containing it.That is break will exit only a single loop.

```
Eg:- main()
     {
     int I,value;
     i=0;
     while(i<=10)
     {
      printf("enter a number");
      scanf("%d",&value);
     if(value<= 0)
     {
     printf("Zero or negative value found");
     break;
     }
     i++;
```

Exit from loop

```
}
}
```

The above program segment will process only the positive integers, whenever zero or negative value is found , the computer will display the message Zero or negative value found as an error and it does not repeat the loop further.

Eg:- // using break in do-while

```
main()
{
int I,value;
i=0;
do
{
 printf("enter a number");
 scanf("%d",&value);
if(value<= 0)
{
printf("Zero or negative value found");
break;
}
i++;
}
while(i<=10);
}
```

Exit from loop

Eg:-// using break in for loop

```
main()
{
int I,value;
i=0;
for( i=0;i<=10;i++)
{
 printf("enter a number");
 scanf("%d",&value);
if(value<= 0)
{
printf("Zero or negative value found");
break;
}
}
}
```

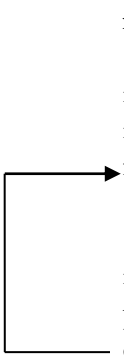
Exit from loop

II. The continue statement:

The continue statement is used to repeat the same operations once again even if it checks the error. The general form is:

```
continue;
```

The continue is a keyword. The continue is used for the inverse operation of the break statement and is used in looping statements.



```
main()
{
int I,value;
i=0;
for( i=0;i<=10;i++){
 printf("enter a number");
 scanf("%d",&value);
if(value<= 0){
printf("Zero or negative value found");
continue;}}}
```

The above program will process only the positive integers, whenever the zero or negative value is found, the computer will display the message Zero or negative value found as an error and it continues the same loop as the given condition is satisfied.

III. **The goto statement:**

The goto statement is used to alter the program execution sequence by transfer of control to some other part of the program.

Syntax:

```
goto label ;
```

Where, label is a valid C identifier used to label the destination such that control could be transferred.

Two ways of using goto statement is conditional and Unconditional goto.

A. Conditional goto:

It is used to transfer the control of execution from one part of the program to the other part under certain conditional cases.

```
Eg:- main()
{
int a,b;
if(a>b)
goto output1;
else
goto output2;
output1:
printf("largest value is%d",a);
output2:
printf("largest value is%d",b);
}
```

It can be used in all looping statements.

B. Unconditional goto:

It is used just to transfer the control from one part of the program to the other without checking any conditions.

```
Eg:- main()
{
200:
```

```
printf("unconditional goto");
goto 200;
}
```

UNIT – III

(Arrays- Character Arrays and Strings –User defined functions.)

ARRAYS

An array is a sequential collection of related data items which are stored in consecutive memory locations that share a common name. In other words an array is a group or table of values referred by the same variable name.The individual values in an array are called elements.

Arrays are sets of values of the same type,which have a single name followed by an index. In C , square brackets appear around the index right after the name. Eg:- salary[10].

Declaring the name and type of an array and setting the number of elements in the array is known as dimensioning the array.

Depending upon the number of subscripts in the array, arrays can be classified into

i. One-dimensional arrays
ii. Two-dimensional arrays
iii. Multi-dimensional arrays

One-dimensional Arrays:

A list of items can be given one variable name using only one subscript and such a variable is called a single-subscripted variable or a one dimensional-array.

An array must be declared before it ish used.

Syntax:- **data_type array_name[size];**

The data type specifies the type of the element, whether int,float or char. The size indicates the maximum number of elements that can be stored inside the array.It should be positive integer.

Eg:- int group[5];

group is an array which contains 5 integer constants.The computer reserves 5 storage locations as shown below.

	group[0]
	group[1]
	group[2]
	group[3]
	group[4]

The first element in the array is numbered 0, so the last element is 1 less than the size of the array i.e for 5 it is 0,1,2,3 and 4.

After an array is declared , its elements must be initialized. Array can be initialized either at two stages : a) At compile time b) At run time.

At compile time:

Syntax: **data_type array_name[size] ={ list of values};**

The values in the list are separated by commas.

Eg:- int group[5] = {20,30,40,10,50};

20	group[0]
30	group[1]

i.e

40	group[2]
10	group[3]
50	group[4]

If the number of values is less than the number of elements, then only that many number are initialized. The remaining elements will be set to zero automatically.
Sometime the size may be omitted , in such cases, the compiler allocates enough space for all initialized elements.

E.g:- int counter[]={2,4,1,0};

This would declare counter array to contain four elements with initial value 1.

At run time:

For large arrays e.g.

```
for(i=0;i<100;i++)
{
  if(i<50)
     sum[i]=0;
 else
     sum[i]=1.0;
}
```

The first 50 elements of the array sum is initialized to zero while the remaining 50 elements are initialized to one at run time.

We can also use scanf function to initialize an array.

Eg. int x[3];

```
scanf ("%d%d%d",&x[0],&x[1],&x[2]);
```

This will initialize array elements with the values entered through the keyboard.

Example for one-dimensional Arrays:

1.
```
main()
{
 int a[5]={11,12,13,14,15};
int i;
printf("contents of the arrya\n");
for(i=0;i<=4;i++)
{
printf("%d \t" , a[i]);
}
}
```

Output: contents of the array

11 12 13 14 15

2.
```
main()
{
int a[100];
int i,n;
printf("how many numbers are in the array? :");
```

```
scanf("%d",&n);
printf("Enter the elements");
for(i=0;i<=n-1;i++)
{
scanf("%d",&a[i]);
}
printf("contents of the array\n");
for(i=0;i<=n-1;i++)
{
printf("%d\t",a[i]);
}
}
```

Two-dimensional array:

When a table of values are to be stored, two-dimensional arrays are used. A two-dimensional will require two pairs of square brackets.

Syntax:- **data_type array_name[row-size][column-size];**

Eg:- int x[2][2];

It is stored in memory as follows:

	Column 0	Column 1
Row 1	[0][0]	[0][1]
Row 2	[1][0]	[1][1]

Two-dimensional arrays may be initialized by following their declaration with a list of initial values enclosed in braces. The initialization is done row by row.

```
Eg:  int x[2][2] = {0,0,1,1};
                (or)

int x[2][2] = { {0,0},{1,1}};
                (or)
int x[2][2] = {
                    {0,0},
                    {1,1}
                };
```

When an array is completely initialized with all values, explicitly we need not specify the size of the first dimension.

Eg:- int x[ ][2] = { {0,0},{1,1}};

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

Example for two-dimensional arrays:

```
1. void main()
   {
```

```
int i,j;
int a[3][4] = {
                {1,2,3,4} ,
                { 5,6,7,8},
                {9,10,11,12}
              } ;
printf( "Contents of Array:\n");
 for(i=0;i<=2;i++)
{
    for( j=0;j<=3;j++)
    {
        printf( "%d \t" , a[i][j]);
    }
printf("\n");
}
}
```

output:

```
1   2   3   4
5   6   7   8
9   10  11  12
```

2.
```
main()
{
int a[50];
int i,j,n,m;
printf("Enter how many rows and cols in the array?");
scanf("%d%d",&n,&m);
printf("Enter the elements");
for(i=0;i<=n-1;i++)
{
  for(j=0;j<=m-1;j++)
   {
    scanf("%d",&a[i][j]);
   }
}
printf("contents of the array\n");
for(i=0;i<=n-1;i++)
{
for(j=0;j<=m-1;j++)
   {
    printf("%d",&a[i][j]);
   }
printf("\n"]);
} }
```

CHARACTER ARRAYS AND STRINGS:

A string is a sequence of characters that is treated as a single data item.Any group of characters defined between double quotation is a string constant. Eg: " India".

C allows us to represent strings as character arrays.Therefore a string variable is any valid C variable name and is always declared as an array of characters.

Syntax:-

char string_name [size];

The size determines the number of characters in the string_name.

Eg:- char city[10];

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

i. char city[9] = "NEW YORK" ;

ii. char city[9] = {'N', 'E','W',' ','Y','O','R','K'};

We can also initialize a character array without specifying the number of elements.In such cases the size of the array will be determined automatically based on the number of elements initialized.

Eg: char string[]={"good"};

Here there are 5 elements in the array string.

We can also declare the size much larger than the string in the initialize.

Eg: char string[10]= {"good"};

Reading strings from terminal:

A) Using scanf function:

scanf can be used with %s format specification to read a string of characters and & is not required before the variable name.

Eg:- char address[10];
scanf('%s',address);

The scanf function terminates its input on the first white space it finds.

Eg:- "NEW YORK" . Only NEW will be read into the arrays. If we want to read the entire line"NEW YORK" ,then we must use two character arrays of appropriate sizes.

Eg: char addr1[5],addr[5];
scanf("%s%s",addr1,addr2);

This would assign the string "NEW" to addr1 and"YORK" to addr2.

scanf function will read only strings without white spaces. To read a text containing more than one word, C supports a format specification known as edit set conversion . code %[].

Eg: char line[80];
scanf("%[^\n]" , line);
printf("%s", line);

B) Using getchar and gets function:

getchar cab be used to repeatedly read successive single character from the input and place them into a character array.Thus, an entire line of text can be read and stored in an array.The reading is terminated when the newline character is entered.

Eg:- char ch;
ch=getchar();

The getchar function has no parameter.

gets function can also be used to read a text containing white spaces.This is a simple function with one string parameter and called as under:

gets(str);

str is a string variable declared properly.It reads character into str from the keyboard until a newline character is encountered and then appends a null character to the string.

Eg:- char line[80];

```
gets(line);
printf("%s",line);
```

Writing strings to screen:

A) Using printf function:

The format %s can be used to display an array of characters.

Eg:- printf("%s",name); can be used to display the entire contents of the array name.

We can also specify the precision with which the array is displayed.

Eg:- %10.4 indicates that the first 4 characters are to be printed in a field width of 10 columns. If we include minus sign (% - 10.4), the string will be left justified.

B) Using putchar and puts functions:

putchar is used to output the values of character variables. It takes the following form:

```
char ch ='A';
putchar(ch);
```

The function putchar requires one parameter. This statement is equal to printf("%c",ch);

Another way of printing string values is to use the function puts.

```
puts(str);
```

where, str is a string variable containing a string value.This prints the value of the string variable str and then moves the cursor to the beginning of the next line on the screen.

```
Eg:- char line[80];
     gets(line);
     puts(line);
```

STRING HANDLING FUNCTIONS:

C library <stdlib.h> supports many string-handling functions that can be used to carry out many of the string manipulations. The following are the most commonly used string-handling functions.

i. strcat() - concatenates (joins) two strings
ii. strcmp() - compares two strings
iii. strcpy() - copies one string over another
iv. strlen() - finds the length of a string

I. strcat () function:

The strcat function joins two strings together.

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

StrignI and string2 are character arrays.When the function strcat is executed, string2 is appended to stringI.It does so by removing the null character at the end of stringI and placing string2 from there.

Eg:-

partI =

0	I	2	3	4	5	6	7	8	9
V	E	R	Y		\0				

part2=

0	I	2	3	4	5	6	7	8	9
G	O	O	D	\0					

part3=

0	1	2	3	4	5	6	7	8	9
B	A	D	\0						

strcat(part1,part2); will result in

part1 =

0	1	2	3	4	5	6	7	8	9
V	E	R	Y		G	O	O	D	\0

part2=

0	1	2	3	4	5	6	7	8	9
G	O	O	D	\0					

We must make sure that the size of the string2(to which string2 is appended) is large enough to accommodate the final string.

strcat may also append a string constant to a string variable.

Eg:- strcat(part1,"hello");

C permits nesting of string functions. Eg:- strcat(strcat(str1,str2),str3);

II. strcmp() function:

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

Syntax:

strcmp(string1,string2);

string1 and string2 may be string variables or string constants.

Eg:- strcmp(name1,name2);
strcmp(name1,"john");
strcmp("RAM",ROM");

III. strcpy() function:

This copies one string over another. It works like a string assignment operator.

Syntax:

strcpy(string1,string2);

This assigns the contents of string2 to string1. String2 may be character array variable or a string constant.

Eg:- strcpy(city,"Delhi");

This assigns Delhi to the string variable city.

IV. strlen() function:

This function counts and returns the number of characters in a string.

Syntax:

n = strlen(string);

where n is an integer variable, which receives the value of the length of the string.The counting ends at the first null character.

USER DEFINED FUNCTIONS:

C functions can be classified into two categories.

i) Library functions or Built-in functions (Eg:- printf and scanf)
ii) User-defined functions. (main())

A complex C Program can be decomposed into small or easily manageable parts. Each small module is called a function. In C main function itself is a function which invokes the other functions to perform a task.
Advantages:- 1. Writing a function avoids rewriting the same code over and over.
2.Simple to write correctly a small function.
3.Easy to read , write and debug a function.
Elements of User-defined function:

There are 3 elements:

i) Function definition
ii) Function call
iii) Function declaration

Function definition:

Syntax:

```
Function_type  Function_name (Parameter list)      } function header
{
        Local variable declaration;
        Executatble statement-1;                   } function body
        ……
        return  statement;
}
```

Function Call:

A function can be called by simply using the function name followed by a list
Of actual parameters enclosed in paranthesis.

Eg: y = mul(10,5);

The actual parameters must match the functions parameter in type,order and number.
Function Declaration: (or) Function Prototype :

All functions in a C program must be declared before they are called.

Syntax:

Function_type function_name(Parameter list);

Eg:- int mul(int m, int n);

THE return STATEMENT:

The Keyword return is used to terminate function and return a value to its caller. The return statement may also be used to exit a function without returning a value. The return statement may or may not include an expression.

Syntax:

return;
return(expression);

TYPES / CATEGORIES OF FUNCTION:

There are three categories:

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

Functions with no arguments and no return values:-

It is the simplest way of writing a user defined function in C. There is no data communication between a calling portion of a program and a called function block.The function is invoked by a calling environment without any formal arguments and the function also does not return back anything to the caller.

Eg:-
```
main()
  {
        message();
  }
 void message()
  {
        printf("hello");
  }
```

Functions with arguments and no return values.

Some formal arguments are passed to the function, but the function does not return back any value to the caller.It is a one way data communication between a calling portion of a program and the function block.

Eg:
```
main()
  {
        int i, max;
        printf("Enter Value");
        scanf("%d",&max);
        printf("Number\t Square");
        for(i=0;i<=max-1;i++)
        square(i);
  }
 square(int n);
  {
        int temp;
        temp = n * n;
        printf("%d\t%d\n",n,temp);
  }
```

Functions with argument and return values:

Here some formal arguments are passed to a function from calling portion of the program and the computed values, if any, is transferred back to the caller. Data are communicated between calling portion and a function block.

Eg:
```
main()
  {
        int i, temp,max;
        printf("Enter Value");
        scanf("%d",&max);
        printf("Number\t Square");
        for(i=0;i<=max-1;i++) {

        temp = square(i);
        printf("%d\t%d\n",i,temp);

  }
 square(int n);
  {
        int value;
        value = n * n;
        return(value);
  }
```

ACTUAL AND FORMAL ARGUMENTS:

Sometimes a function may be called by a portion of a program with some parameters and these parameters are known as actual arguments.

The formal arguments are those parameters present in a function definition. It is also called as dummy arguments or parametric variables.

Eg:
```
main()
  {
```

```
int a,b,c,sum;
…….
sum = calsum(a,b,c);  ——————→ actual arguments
…….
}
calsum( int x, int y, int z)  ——————→ formal arguments
{
……..
}
```

LOCAL AND GLOBAL VARIABLES:

Local variables are defined inside the function b**lock or compound statement and are referred only to the particular part of the block or function.The same variable name may be** given to the different part of the function or a block and each variable will be treated as **different** entity.

```
func( int i, int j)
{
        int k , m;  ——————→ local variables
}
```

The global variables are declared outside the main function block. These variables are referred to the same datatype and the same name throughout the program in both calling portion of a program and a function block.

Eg:

```
int x,y=4;
main()
{
 x = 10;
…….
 funI();
}
funI()
{
 int sum;
 sum = x + y;
 …….
}
```

CALL BY VALUE AND CALL BY REFERENCE:

Arguments can generally passed to functions in one of the two ways.

1. Sending the values of the argument.
2. Sending the address of the argument.

In the first method the value of each of the actual arguments in the calling function is copied into corresponding formal arguments of the called function. The changes made to the formal arguments in the called function have no effect on the values of the actual arguments in the calling function.

Eg:-

```
main ()
{
int a=10,b=20;
swap(a,b);
printf("a=%d b=%d",a,b);
}
swap(int x, int y)
{
int t;
t = x;
x = y;
y = t;
printf("x = %d y=%d",x,y);
}
```

The value of a and b remain unchanged even after exchanging the values of x and y.

In the second method the address of actual arguments in the calling function are copied into formal arguments of the called function.

```
Eg:- main ()
        {
        int a=10,b=20;
        swap(&a , &b);
        printf("a=%d b=%d",a,b);
        }
     swap(int  *x, int *y)
        {
        int t;
        t = *x;
        *x = *y;
        *y = t;
        printf("x = %d y=%d",x,y);
        }
```

Using addresses we would have an access to the actual arguments and hence we would be able to manipulate them. We cannot alter the actual arguments, if desired it can be achieved through call by reference.

RECURSION:

The function which calls itself directly or indirectly again and again is known as the recursive function. There is much difference between the normal function and the recursive function. The normal function will be called by the main function whenever the function name is used. On the other hand the recursive function will be called by itself directly or indirectly as long as the given condition is satisfied.

```
Eg:-  main()
      {
              int a,fact;
              printf("Enter a number");
              scanf("%d", &a);
              fact = rec (a);
              printf("Factorial =%d",fact);
      }
      rec ( int x)
      {
              int f;
              if (x = =1)
              return;
              else
              f=x*rec(x-1);
              return f;
      }
```

UNIT-IV

(Structures and Unions – Pointers – File Management in c.)

POINTERS

Definition:

A pointer is a variable which holds the address of another variable. The Pointer is a powerful technique to access the data by indirect reference because it holds the address of that variable where it has been stored in the memory.

Accessing the address of a variable:-

Declaring pointer variable:

Like any other variable, a pointer variable must be declared as pointer before we use them:

General form: **datatype *pt_name;**

The asterisk (*) tells that the variable pt_name is a pointer variable. pt_name points to a variable of type datatype. pt_ name needs a memory location.

Eg/- int *p;
float *p;

Initialization of pointer variable:-

The process of assigning the address of a variable to a pointer variable is known as initialization. Accessing The address of a variable is done through &(ampersand) operator i.e address of operator. The pointer variable must be initialized before they are used in the program. We can initialize the Variable using assignment operator.

```
Eg\-    int quantity;
        int *p;    /* declaration */
        P=&quantity;   /* initialization */
```

Accessing a variable through its pointer:-

After assigning the address of a variable to a pointer, We can access the value the value of a variable using the pointer. This is done by using another *operator or indirection operator or dereferencing operator.

```
Eg\-    int quantity, *p,n;
        quantity=179;
        p= & quantity;
        n=*p;    /* use of indirection operator */
```

The last line contains the indirection operator *. When the operator * is placed before a pointer variable in an expression (on the right hand side of the equal sign), the pointer returns the value of the variable of which the pointer value is the address.

Here *p returns the value of the variable quantity i.e the value of n would be 179. The * can be remembered as "value at address".

Valid pointer declaration:-

```
int x,y;
int *p1,*p2;
```

1) p1=&x;
The memory address of x is assigned to a pointer variable p1.
2) y=*p1;
The contents or value of the pointer variable *p1 is assigned to the variable y, not the memory address.
3) p1=&x;

```
p2=p1;   /* address of p1 is assigned to p2 */
```

The address of p1 is assigned to pointer variable p2. The contents of both p1 and p2 will be same.

Invalid pointer Declaration:-

1)
```
int x;
int x_ptr;  /* error */
x_ptr =&x;
```

Error: pointer declaration must have prefix of *.

2)
```
float y;
float *y-ptr;
y-ptr=y;     /* error */
```

Error: While assigning variable to the pointer variable the address operator (&) must be used along with the variable y.

3) int x;
char *c-ptr;
c-ptr =&x;

Error: Mixed data type is not permitted

Eg/-

```
#include <stdio.h>
main()
{
int x=10;
int *ptr;
ptr=&x;
printf(" Address of x=%u",&x);
Printf("Address of x=%u",ptr);
Printf("Address of ptr=%u",&ptr);
Printf("Address of ptr=%u",ptr);
Printf("Address of x=%d",x);
Printf("Address of x=%d",*(&x));
Printf("Address of x=%d",*ptr);
}
```

o/p:

```
Address of x=6485
Address of x=6485
Address of ptr=5276
Address of ptr=6485
Address of x=10
Address of x=10
Address of x=10
```

Pointer Arithmetic:

Some arithmetic operation can be performed with pointers. The C language four arithmetic operators that may be used with pointers such as,

addition +
subtraction –
incrementation ++
decrementation –

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

p1+4, p2=2 and p1=p2 are valid but p1+p2 is illegal because no two pointers can be added .Pointers can be compared using relational operator.

p1>p2,p1==p2 and p1!=p2 are valid.

The following pointers variables can be used in expression. If p1 and p2 are properly declared and initialized, then the following statement are valid.

```
y=*p1 X  *p2;
sum=sum+*p1;
Z=5* - *p2/*p1;    // (5*(-(*p2)))/(*p)
```

In case of pointer incrementation.

Eg/-

```
int value,*p1;
value=100;
P1=&value;
P1++;
```

If the address of value is assigned as 2800 to p1, then after p1=p1+1.the value of p1 will be 2802, that is when we increment, a pointer, its value is increased by "length" of the datatype that is points to. This length is called "*scale factor*".

Character 1byte
Integer 2bytes
Float 4bytes
Longint 4bytes
Double 8bytes

Pointer and arrays:

In C, there is a close correspondence between array datatype and pointers. An array name in C is very much like pointer but there is difference between each others. The pointer is a variable that can appears on the left side of an assignment operator. The array name is a constant and can't appear on the left side of an assignment operator

The following declaration is valid

```
int value [20];
int*ptr;
```

can be declared as

```
value [0]
```

which holds the address of the Zeroth element of the array value,

the pointer variable pointer is also an address, so the declaration value[0]and pointer is same because both holds the address.

The following statement is value [0];

The address of the zeroth element is assigned to a pointer variable pointer.If the pointer is incremented to the next data element then the following expression of the equality operator is same.

Eg/-

```
Ptr++  == value[1]
The following equalities are valid
Ptr+6  =&value[6];
Ptr ==&value[0];
*(ptr+6)==&value[6];
Ptr++==&value[1];
```

Array subscripting is defined in terms of pointer arithmetic.

(or) The expression a[i] is defined to be the same as *((a) + (i)) which is to say the same as *(&(a)[0]+(i)).

Eg\- /* to display the contents of the array using a pointer arithmetic */

```
#include<stdio.h>
main()
{
int a[4]={11,12,13,14};
int I,n,temp;
n=4;
printf("contents of the array \n");
for(i=0;i<=n-1;i++)
{
temp=*((a)+(i));
printf("value=%d\n",temp);
}
}
```

o/p:

```
Contents of the array,
Value =11
Value=12
Value=13
```

Value=14

Array of pointers:

Pointers may be arrayed like any other datatype.The declaration for an integer pointer array of size 10 is,

```
int*ptr[10];
```

makes ptr[0],ptr[1],ptr[2]….ptr[n] an array of pointer.

Eg/-

```
#include<stdio.h>
main()
{
char*ptr[3];
ptr[0]="hai";
ptr[1]="hello";
ptr[2]="bye";
printf("Contents of printer1=%s\n",ptr[0]);
printf("Contents of pointer2=%s\n",ptr[1]);
printf("Contents of pointer3=%s\n",ptr[2]);
}
```

STRUCTURES:

C supports a constructed data type known as structures. Structure consist of a group of variable of different datatypes placed in a common name. A structure is a heterogeneous datatype where as an array is a homogeneous data type.

Defining a structure:

The format of a structure must be defined first.

Syntax:

```
struct tag_name
{
        datatype member1;
        datatype member2;
        ……
};
```

Here, struct is a keyword and it declares a structure to hold different data fields.Each field is called a structure element or member. Each member may belong to different data types. The tag_name is the name of the structure and is called structure tag. The tag_name is used for declaring variables. The template should end with a semicolon.

Eg:-

```
struct book_bank
{
        char title[20];
        char author[15];
        int pages;
        float price;
};
```

book_bank is the name of the structure. It holds four data fields namely title,author,pages and price. Each of the members belong to different type of data.

Declaring Structure variables:

The above definition has not declared variables. We can declare structure variables like any other variable. Declaring includes

i. The keyword struct

ii. The structure tag name
iii. List of variable names separated by commas
iv. A terminating semicolon

Eg:-

```
struct book_bank book1,book2,book3;
```

This statement declares book1,book2 and book3 as variables of type struct book_bank. The complete definition is as follows.

```
struct book_bank
{
char title[20];
char author[15];
int pages;
float price;
} b1,b2b3;
```

(or)

```
struct book_bank
{
char title[20];
char author[15];
int pages;
float price;
}
struct book_bank b1,b2b3;
```

Accessing Structure members:

The members are not variables.So they have to be linked to the structure variables in order to make them meaningful. The link is established using the member operator '**.**' which is also known as dot operator or period operator. eg:- b1.price

Structure Initialization:

Like any other variable structure be can also be initialized.

At Compile time:

```
main()
{
 struct st_record
 {
      int weight;
      int height;
 };
struct st_record s1={50,70,80);
struct st_record s2={55,75,85);
……
}
```

We can also initialize a structure variable outside the function.

```
struct st_record
{
      int weight;
      int height;
 };
main()
{
 struct st_record s1={50,70,80);
 struct st_record s2={55,75,85);
 ……
}
```

We can also assign values as follows.

Eg:- i. strcpy(s1.name,"Ram"); ii. s1.weight = 60;

At Run time:

We can also use scanf to give values through keyboard at runtime.

Eg:- scanf("%s%d",s1.name, &s1.weight);

Example:

```
struct person
{
 char name[20];
 int day;
 char month[10];
 int year;
 float salary;
};
main ( )
 {
 struct person p;
 printf("Input values");
 scanf("%s%d%s%d%f " , p.name, &p.day, p.month, &p.year, &p. salary);
 printf("%s%d%s%d%f " , p.name, &p.day, p.month, &p.year, &p. salary);
}
```

Copying and Comparing Structure Variables:

Two variables of the same structure types can be copied the same way as the ordinary variables. For example , if b1 and b2 belong to the same structure then

b1 = b2;

b2 = b1; are valid statements.

C does not permit any logical operations on structure variables.

b1!=b2;

b2==b1 are invalid.

We can compare them only by comparing the members individually.

b1.author==b2.author

Arrays of Structures:

Similar type of structures placed in a common heading or a common variable name is called an array of structures.

Eg:-

```
struct book
{
char name[10];
int pages;
float price;
} b[100];
```

In the above example b is an array of 100 elements b[0] …..b[99]. Each element is defined to be of type struct book.

Eg:-

```
struct book
{
char name[10];
int pages;
float price;
```

```
};
main ()
{
int i;
struct book b[100];
printf("enter the values for each element\n");
for(i=1;i<100;i++)
{
        scanf("%s%d%f",b[i].name,&b[i].pages, &b[i].prices);
}
printf("The structure values are \n");
for(i=1;i<100;i++)
{
         printf("%s%d%f",b[i].name,b[i].pages,b[i].prices);
} }
```

The array is declared just as it would have been with any other array. Since b is an array, we use the usual array accessing methods to access individual elements and then the member operator to access members.

Arrays within Structures:

C permits the use of arrays as structure members. We can use character arrays, single or multidimensional arrays of type int and float inside a structure.

Eg:-
```
struct marks
        {
        int number;
        float subject[3];
        };
struct marks student[2];
```

Here the member subject contains three elements, subject[0],subject[1] and subject[2]. These elements can be accessed using appropriate subscripts.

Eg: student[1] ,subject[2];

Would refer to the marks obtained in the third subject by the second student.

Eg:
```
main()
        {
        struct marks
        {
        int sub[3]; int total;
        };
        struct marks student[3]={45,67,81,0,75,53,69,0,57,36,71,0};
        struct marks total;
        inti,j;
        for(i=0;i<=2;i++)
        {
        for(j=0;j<=2;j++)
        {
        student[i] ,total+=student[i] .sub[j]; total. sub[j]+=student[i].sub[j];
        }
        total. total+=student[i] .total;
        *
        }
        printf("student Total\n"); for(i=0;i<=2;i++)
```

```
printf("student[%d] %d\n" , i+1 , student[i] .total);
printf("subject TotalVn");
for(j=0;j<=2;j++)
printf("subject %d %d\n",j+1 ,total.sub[j]); printf("grand total=%d\n", total.total);
}
O/P:
student    total
student[ 1 ]          193
student[2]            197
student[3]            164
subject total
subject 1  177
subject 2  156
subject 3  221
Grand Total = 554
```

Structures within structures :

A structure within a structure means nesting of structures. The syntax of the structure within structure is as follows.

```
struct time
{
int second;
 int minute;
 int hour;
};
struct t
{
int carno;
struct time st;
struct time et;
};
struct t player;
```

Structures and Functions:

Structures can be passed to functions. There are three methods:

i. The first method is to pass each member of the structure as the actual argument of the function call. The actual argument are then treated independently like ordinary variables.

Eg:

```
struct date
{
int day; int month; int year;
};
main()
{
struct date a; a.day=10; a.month=2; a.year=1992;
output(a.day, a.month, a.year);
}
output(n)
struct date n;
{
printf("Today's date is: %d/%d/%d/n'\n.day,n.month,n.year);
}
0/P:
Today's date is: 10/12/1992
```

ii. The second method involves passing a copy of the entire structure to the called function.

eg:

```
struct date
{
int day; int month; int year;
};
main()
{
struct date a={ 10,2,2017}; output(a);
}
output(n) struct date n;
{
printf("Todav's date is: %d/%d/%d",n.day,n.month,n.year);
}
O/P:
Today's date is: 10/2/2017
```

iii. The third method is by using pointers to pass the structure as an argument. In this case the address location of the structure is passed to the called function.

eg:

```
struct book
{
char name[35]; char author[35]; int pages;
};
main()
{
struct book bl={"Java", "P. Naughton",886}; show(&bl);
}
show(struct book *bl)
{
printf("\n%S by %S of %d pages", bl-> name, bl-> author, bl-> pages);
}
O/P:
Java by P. Naughton of 886 pages
```

UNIONS

Union is a variable which is similar to the structure. It contains a number of members like structure but it holds only one object at a time. In the structure each member has its own memory location, whereas members of union have same memory locations.The union requires bytes that are equal to the number of bytes required for the largest members. For example, if the union contains char, int and longint then the number of bytes reserved in the memory for the union is 4 bytes.

eg:

```
main()
{
union result
{
int marks; char grade;
};
struct res
{
char name[15]; int age;
union result perf;
}data;
printf("size of union:%d\n", sizeof(data.perf)); printf("size of structure:%d\n", sizeof(data));
}
O/P:
size of union: 2 size of structure: 19
```

FILE MANAGEMENT IN C

Introduction:

The console oriented I/O operations done through terminals(keyboard and screen) using printf and scanf functions works fine as long as the data is small .But,

- handling larger volumes of data is difficult and time consuming
- data is lost when either the program is terminated or the computer is turned off

These drawbacks can be overcome using files, where we can store data and read it whenever necessary.

Definition:

"A file is a place on disk where a group of related data is stored".

Basic file operations:

naming a file
opening a file
reading data from a file
writing data to a file
closing a file

File handling functions:

The above said file operations can be performed by using the file handling functions available in the C library. They are,

i. fopen()-creates a new file for use and opens an existing file for use.
ii. fcloseQ-closes a file which has been opened for use.
iii. getc()-reads a character from a file.
iv. putc()-writes a character to a file.
v. getw()-reads a integer from a file.
vi. putw()-writes an integer to a file.
vii. fprintf()-writes a set of data values to a file.
viii. scanf()-reads a set of data values from a file.
ix. fseek()-sets the position to a desired point in the file.
x. ftell()-gives the current position in the file.
xi. rewind()-sets the position to the beginning of the file.

Defining and opening a file:

If we want to store data in a file,we have to specify the

- filename
- data structure and
- purpose

Filename:

It is a stimg of characters.lt contains two parts

i. a primary name
ii. an optional period with the extension.

Data structure:

Data structure of a file is FILE. All files should be declared as type FILE before they are used.

Purpose:

When we open a file,we must specify whether to read or write a file.

The general format for declaring and opening a file is

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

- The first statement declares the variable fp as a pointer to the data type FILE.
- The second statement opens the file named filename and assings an identifier to the FILE type pointer fp.
- The second statement also specifies the purpose of opening this file.The mode does this job.

r - open the file for reading only,
w - open the file for writing only,
a - open the file for adding or appending data to it.
r+ - the existing file is opened to the beginnig for both,reading and writing.
w+ - same as w except both for reading and writing.
a+ - same as a except both for reading and writing.

When trying to open a file,one of the following things may happen:

i. When the mode is 'writing',a file is with the specified name is created if the file does not exist.The contents are deleted,if the file already exists

ii. When the purpose is 'appending',the file is opened with the current contents safe.A file with the specified name is created if the file does not exist.

iii. If the purpose is 'reading',and if it exists,then the file is opened with the curret contents safe otherwise an error occurs.

Closing a file:

A file must be closed as soon as all operations on it have been completed.

The general form is

```
fclose(file_pointer);
```

Example:

```
FILE *pl,*p2;
pl=fopen("data","w");
p2=fopen("results","r");



fclose(pl);
fclose(p2);
```

Input and Output Operations:

Once a file is opened,reading out of or writing to it is accompolishes using the standard I/O routines.There are three functions.

- The getc and putc functions
- The getw and putw functions
- The fscanf and fprintf functions,

i.The getc and putc functions:

These functions can handle one character at a time. If a file is opened in "w" mode and the file pointer is fpl,then **putc(c,fpl);**

writes the character contained in the character variable C to the file associated with FILE pointer fpl.

Similarly getc is used to read a character from a file that has been opened in read mode. For example

```
c=getc(fp2);
```

This statement would read a character from the file whose file pointer is fp2.

Example:

```
# include<stdio.h> mainQ
{
FILE*fl; char c;
printf("Data Input\n");
fl =fopen("Input","w");
while((c=getchar())!=EOF)
putc(c,fl);
fclose(fl);
printf("Data OutputVn");
fl=fopen("Input","r");
wh i le((c=getc(f I)) !=EOF)
printf("%c",c);
 fclose(fl); »
}
```

Output:

Data Input

This is an example to demonstrate file handling in C.

Data Output

This is an example to demonstrate file handling in C.

ii. The getw and putw functions:

The getw and putw are integer oriented functions.It is used to read and write integer values.

The general forms are

putw(integer,fp);

getw(fp);

iii. The fprintf and fscanf functions:

It is similar to printf and scanf except that they work on files.They can handle group of mixed data simultaneously.

General form of printf is,

fprintf(fp,"control string",list);

Example:

fprintf(fl,"%s%d%f",name,age,7.5);

General form of scanf is,

fscanf(fp,"control string",list);

Example:

fscanf(f2,":%s%d",item,&quantity);

Error handling during I/O operations:

Error may occur during I/O operations on a file like

- trying to read beyond end-of-file.
- device overflow.
- trying to use a file that is not opened.
- opening a file with invalid file name.
    - trying to perform an operation on a file,when the file is opened for another type of operation.
- attempting to write a write-protected file.

There are 2 functions to detect I/O errors in files.They are,

i. feof
ii. ferror

feof:

1. Used to test for ann end of file condition.
2. it takes file pointer as its argument
3. returns non-zero integer value if all data is read and zero otherwise.

Example:

```
if(feof(fp))
printf("end of data");
```

ferror:

i. reports the status of the file.
ii. it takes file pointer as its argument.
iii. returns a non-zero integer if an error has been detected and zero otherwise.

Random access to files:

Accessing only a particular part of a file can be achieved using

fseek()
ftell( )
rewind()

ftell() :

It gives the current position in the file.

```
n=ftell(fp);
```

i. ftell takes a filepointer as its arguments
ii. it returns a number of type long
iii. this function will be useful for saving the current position of a file

rewindQ:

This function sets the position to the beginning of the file.

```
rewind(fp);
```

it takes the filepointer as its argument and resets the position to the start of the file.

fseek():

This function is used to move the file position to a desired location within the file.

```
fseek(file_ptr,offset,position);
```

i. file_ptr is the file pointer to the file concerned.
ii. offest is a number/variables of type long,it specifies the number of position (bytes) to be moved from the location specified by position.
iii. position can take one of the following 3 values.

0 - beginning of file
1 - current position
2 - end of file

iv. if offset is positive then moves forward and if negative moves backward
v. operations on fseek function:

i. fseek(fp,ol,0); - go to the begining
ii. fseek(fp,ol,l); - stay at current position
iii. fseek(fp,ol,2); - go to the end of the file

iv. fseek(fp,m,0); - move to m+1th byte in the file
v. fseek(fp,m,1); - go forward by m bytes
vi. fseek(fp,-m,1); - go backward by m bytes from the current position.
vii. fseek(fp,-m,2); - go backward by m bytes from the end

vi. fseek returns zero if the operation is successful.-1 otherwise.

Command Line Arguments:

A command line argument is a parameter supplied to a program when the program is invoked.

For example,if we want to execute a program to copy the contents of a file named x-file to another one named y-file,then we may use a command line like

c > program x-file y-file

where,

program -> is the filename where the executable code of the program is stored.

Advantages:

This would eliminate the need for the program to request the user to enter the filenames during execution.

These parameters can be supplied to the program by passing arguments to the main functions.

The main functions can take two arguments i)argc and ii)argv

i. argc is the argument counter that counts the number of arguments on the command line.The above example has 3 arguments.
ii. argv is an arguments vector and represents an array of character pointers that point to the command line arguments.The size of the array will be equal to the value of argc.

for the above example,

argv[0]- program
argv[1]- x-file
argv[2]- y-file

In order to access the command line arguments,we must declare the main functions and its parameters as follows:

In order to access the command line arguments,we must declare the main functions and its parameters as follows:

```
main(int argc,char*argv[])
```

Example:

```
# include<stdio.h>
main(int argc,char*argv[])
{
FILE* fs,* ft;
char ch;
if(argc!=3)
  {
puts("Insufficient arguments");
 exit();
  }
fs=fopen(argv[1],"r");
if(fs==NULL)
  {
```

```
puts("cannot open source file");
exit();
   }
ft=fopen (argv [2]," w");
 if(ft==NULL)
   {
puts("cannot open target file");
   exit();
   }
   }
```

UNIT-V

(Dynamic memory allocation – Linked lists- preprocessors-programming guidelines.)

DYNAMIC MEMORY ALLOCATION:

The process of allocating memory at runtime is known as dynamic memory allocation. There are four library routines called as "memory management functions" that can be used for allocating and freeing memory during program execution.

Function	Task
malloc	Allocates requested size of bytes and returns a pointer to the first byte of the allocated space.
calloc	Allocates space for an array of elements, initializes them to zero and then returns a pointer to the memory.
Free	Frees previously allocated space.
realloc	Modifies the size of previously allocated space.

1. Allocating a block of memory: **malloc**

A block of memory may be allocated using the function malloc. The malloc function reserves a block of memory of specified size and returns a pointer of type void.

Syntax: **ptr =(cast-tyye*)malloc(byte-size);**

Ptr is a pointer of type cast-type. The malloc returns a pointer to an area of memory with size byte-size.

Example: X=(int *) malloc (100 x sizeof(int));

A memory space equivalent to 100 times the size of an int bytes is reserved and the address of the first byte of the memory all ocated is assigned to the pointer X of type int.

2. Allocating multiple blocks of memory: **calloc**

Calloc is a memory allocation function that is used for requesting memory space at runtime for storing derived data types such as arrays and structures. Calloc allocates multiple blocks of storage, each of the same size, and then sets all bytes to zero.

Syntax: **ptr =(cast-tyye*)calloc(n, elem-size);**

This allocates continous space for n blocks, each of size elem-size bytes. All bytes are initialized to zero and a pointer to the first byte of the allocated region is returned.If there is not enough space, a NULL pointer is returned.

3. Releasing the used space: **free**

When the data is stored in a block of memory is no longer needed , we can release that memory for future use , using thee free function.

Syntax: **free(ptr);**

Ptr is a pointer to a memory block, which has already been created by malloc and calloc.

4. Altering the size of the block: **realloc**

   The memory size already allocated can be changed with the help of realloc function. Memory size can be reduced or extended using this function. For example if the original allocation is done by the statement **ptr=malloc(size);** then reallocation of space may be done by the statement

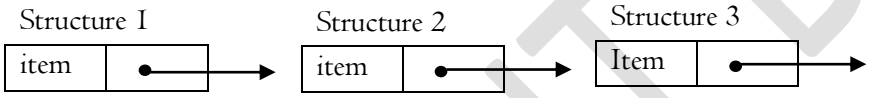
   ptr=realloc (ptr,newsize);

   This function allocates a new memory space of size newsize to the pointer variable ptr and returns a pointer to the first byte of the new memory block.The newsize may be longer or smaller than the size.

LINKED LISTS:

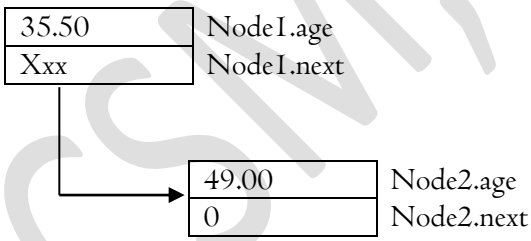
List is a set of items organized sequentially.Eg:-Arrays. In arrays we use the index for accessing and manipulation of array elements. But the major problem with the arrays is that , the size of the array must be specified precisely at the beginning.

Linked list is a dynamic data structure. Each item in the list is a part of a structure and also contains a "link" to the structure containing the next item. This type of list is called a linked list because it is a list whose order is given by links from one item to the next.



Each structure of the list is called a node and consists of two fields, one containing the item and the other containing the address of the next item in the list.

Example:



Example:

```
struct link_list
{
float age;
struct link_list *next;
};
struct  link_list node1, node2;
node1.next = &node2;
node1.age=35.50;
node2.age=49.00;
node2.next=0;
```

Advantages of Linked List:

1. Linked Lists can grow or shrink in size during the execution of a program.
2. It does not waste memory space.
3. It provides flexibility by allowing the items to be rearranged efficiently.
4. It is easier to insert or delete items by rearranging the links

Disadvantages:

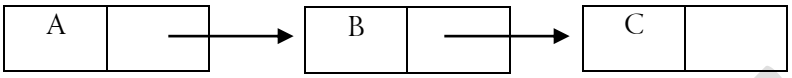
1. Accessing an arbitrary item is time consuming.
2. It uses more storage than an array with the same number of items.

Types of linked lists:

The following are the different types of linked list.

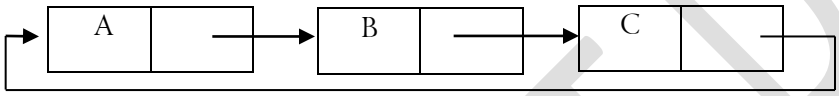
i. Linear singly linked list.
ii. Circular linked list.
iii. Two-way or Doubly linked list.
iv. Circular Doubly linked list.

Linear singly linked list.



Circular linked list.

It has no beginning and no end. The last item points back to the first item.

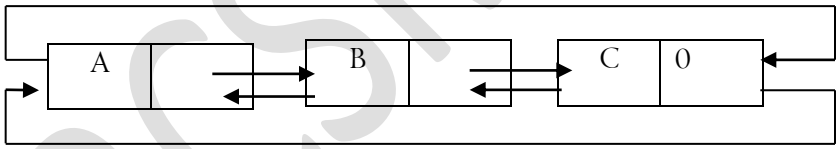


Doubly linked list.

It uses double set of pointers, one pointing to the next item and other pointing to the preceding item. This allows us to traverse the list in both directions.

Circular Doubly linked list.

It employs both the forward pointer and backward pointer in circular form.



GUIDELINES FOR DEVELOPING A C PROGRAM:

The program development process includes three stages:

i. Program design
ii. Program Coding
iii. Program Testing

Program Design:

Before coding a program the following steps should be followed.

a. program analysis
b. outlining the program structure
c. algorithm development
d. Selection of control structure.

Program Coding:

The algorithm developed during program design must be translated into a set of instructions that a computer can understand. The program created should be readable and simple to understand. Complex logic and tricky coding should be avoided.

Program Testing and Debugging:

Testing and debugging refers to the task of detecting and removing errors in a program, so that the program produces desired result.

TYPES OF ERRORS:

Errors can be classified under four types: Syntax errors, runtime errors, logical errors and latent errors.

a. Syntax Errors: Any violation of the rules of the language results in syntax errors. The compiler can detect and isolate such errors. When syntax errors are present, the compilation fails and is terminated after listing the errors and the line numbers in the source program.
b. Run-time Errors: Errors such as mismatch of data types or referencing an out of-range array element go undetected by the compiler.A program with these mistakes will run, but produces no erroneous results.
c. Logical Errors: These errors are related to the logic of the program execution like taking a wrong path, failure to consider a particular condition, incorrect order of evaluation etc.,
d. Latent Errors: It is a hidden error that shows up only when a particular set of data is used.

COMMON PROGRAMMING ERRORS:

The following are some of the common mistakes while coding a program.

i. missing semicolon: every C statement must end with a semicolon.
ii. misuse of semicolon : eg/- for(i=1;i<10;i++); while(x<10);
iii. use of = instead of = =
iv. missing braces with multiple statements.
v. missing quotes for string and characters.
vi. improper comment characters.Nesting of comments is not allowed.
vii. undeclared variables.
viii. forgetting the precedence of operators.
ix. forgetting to declare function parameters.
x. mismatching of actual and formal parameter types in function calls.
xi. non-declaration of function.
xii. missing & in scanf parameters.
xiii. crossing the bounds of array.
xiv. forgetting a space for null characters in a string.

xv. using uninitialized pointers.

PREPROCESSORS:

The preprocessor is a program that modifies the C source program according to directives supplied in the program. An original source program is stored in a file.The preprocessor does not modify this program file, but creates a new file that contains the processed version of the program.This new file is then submitted to the compiler.

Preprocessor does the following actions.

1. replacement of defined identifiers by macros.
2. inclusion of other files.
3. renumbering and renaming of source files.

The general rules for defining a preprocessor are

1. all preprocessor directives begin with **#.**
2. they must start in the first column and no space between # and the directive.
3. the preprocessor is not terminated by a semicolon.
4. only one preprocessor directive can occur in a line.
5. it may appear in any place in the source file, outside functions, inside functions or inside compound statements.

The common preprocessor directives and their uses are:

S.No.	DIRECTIVE	USES
1.	#include	Inserts text from another file.
2	#define	Defines preprocessor macros.
3	#undef	Removes macro definitions.
4	#if	Conditionally include some text, based on the value of the constant expression.
5	#error	Terminate processing easily.
6	#ifdef	Conditionally include some text based on whether a macro name is defined.
7	#else	Alternatively include some text.
8	#endif	Combination of #if and #else.
9	#line	Gives a line number for compiler message.

MACROS:

The macro definition is the use of the #define directive to define constants in a C program. The macros can be classified into two groups. They are

a. Simple macro definition
b. macro definition with arguments.

The advantages of using macros are;

1. Easy to read and write
2. Easy to check string constants.
3. Easy to transfer from one machine to another
4. gives good look to the program.

Simple macro definition:

A macro is simply a substitution string that is placed in the program.

Eg:-
```
#define MAX 100
main()
{
char name[ MAX];
for (i=0;i<MAX-1;i++)
…….
}
```
The MAX is internally replaced with the following program.
```
main()
{
char name[100];
for (i=0;i<100-1;i++)
…….
}
```
Each occurrence of the identifier MAX as a token is replaced with the string 100 that follows the identifier in #define line.

MACRO WITH PARAMETERS:

The more complex form of macro definition declares the names of formal parameters within paranthesis, separated by commas.

Syntax: **#define Name(var1,var2,…var n) substitution string**

Example: #define PRODUCT (a,b) ((a) x (b))

#define MIN(a,b) ((a)<(b) ?(a): (b))

****