ROGRAMMING FOR PROBLEM

SOLVING USING C

(RPL2B001)

Mechanical Engineering

B.Tech 2nd Semester

Prepared By

Ratnam Dodda

Assistant Professor

Department. of CSE

Govt. College of Engineering Kalahandi,

Course Outcomes

The student will learn

- To formulate simple algorithms for arithmetic and logical problems.
- To translate the algorithms to programs (in C language).
- To test and execute the programs and correct syntax and logical errors.
- To implement conditional branching, iteration and recursion.
- To decompose a problem into functions and synthesize a complete program using divide and conquer approach.
- To use arrays, pointers and structures to formulate algorithms and programs.
- To apply programming to solve matrix addition and multiplication problems and searching and sorting problems.
- To apply programming to solve simple numerical method problems, namely root finding of function, differentiation of function and simple integration.

Contents

UNIT-1

INTRODUCTION

Computer

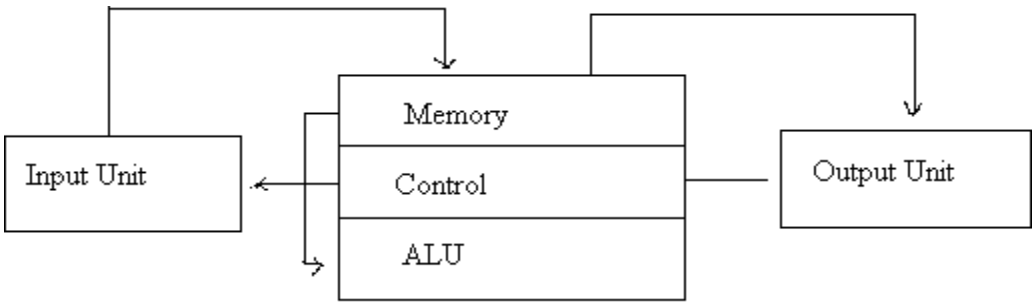
Basically it is a fast calculating machine that can be defined as an electronic device that is designed to accept data as input, perform the required mathematical and logical operations and output the result. It is now a days used for variety of uses ranging from house hold works to space technology. The credit of invention of this machine goes to the English Mathematician Charles Babbage.

Types of Computers:

Based on nature, computers are classified into Analog computers and Digital computers. The former one deals with measuring physical quantities (concerned with continuous variables) which are of late rarely used. The digital computer operates by counting and it deals with the discrete variables. There is a combined form called Hybrid computer, which has both features.

Based on application computers are classified as special purpose computers and general computers. As the name tells special computers are designed to perform certain specific tasks where as the other category is designed to cater the needs of variety of users.

Basic structure of a digital computer



Block Digram of a computer

The main components of a computer are Input unit (IU), Central Processing unit (CPU) and Output unit (OU). The information like data ,programs etc are passed to the computer through input devices. The keyboard, mouse, floppy disk, CD, DVD, joystick etc are certain input devices. The output device is to get information from a computer after processing. VDU (Visual Display Unit), Printer, Floppy disk, CD etc are output devices.

The brain of a computer is CPU. It has three components- Memory unit, Control unit and Arithmetic and Logical unit (ALU)- Memory unit also called storage device is to store information. Two types memory are there in a computer. They are RAM (random access memory) and ROM (read only memory). When a program is called, it is loaded and processed in RAM. When the computer is switched off, whatever stored in RAM will be deleted. So, it is a temporary memory. Whereas ROM is a permanent memory, where data, program etc. are stored for future use. Inside a computer there is storage device called Hard disk, where data are stored and can be accessed at any time.

The control unit is for controlling the execution and interpreting of instructions stored in the memory. ALU is the unit where the arithmetic and logical operations are performed.

The information to a computer is transformed to groups of binary digits, called bit. The length of bit varies from computer to computer, from 8 to 64. A group of 8 bits is called a Byte and a byte generally represents one alphanumeric (Alphabets and Numerals) character.

The Physical components of a computer are called hardwares. But for the machine to work it requires certain programs (A set of instructions is called a program). They are called soft wares. There are two types of soft wares – System software and Application software – System software includes Operating systems, Utility programs and Language processors.

ASCII Codes

American standard code for information interchange. These are binary codes for alpha numeric data and are used for printers and terminals that are connected to a computer systems for alphabetizing and sorting.

Operating Systems

The set of instructions which resides in the computer and governs the system are called operating systems, without which the machine will never function. They are the medium of communication between a computer and the user. DOS, Windows, Linux, Unix etc. are Operating Systems.

Utility Programs

These programs are developed by the manufacturer for the users to do various tasks. Word, Excel, Photoshop, Paint etc. are some of them.

Language Translators

These are the programs which are used for converting the programs in one language into machine language instructions, so that they can be executed by the computer.

Creating and Running a Program

This process is presented in a straightforward, linear fashion but you should recognize that these steps are repeated many times during development to correct errors and make improvements to the code. The following are the four steps in this process 1) Writing and Editing the program 2) Compiling the program 3) Linking the program with the required modules 4) Executing the program

Compiling Programs

The code in a source file stored on the disk must be translated into machine language. This is the job of the compiler. The Compiler is a computer program that translates the source code written in a high-level language into the corresponding object code of the low-level language. This translation process is called compilation. The executable program is stored in a disk for future use or to run it in another computer. The compiled programs runs the whole program

at time. The Compiler which executes C programs is called as C Compiler. Example Turbo C, Borland C, GCC etc.

Linker

It is a program that combines object modules to form an executable program. Generally, in case of large program the programmer prefer to break a code into smaller modules as this simplifies the programming task. Eventually, when the source code of all modules has been converted into object code, we need to put all the modules together. This is the job of the linker. Usually, the compiler automatically invokes the linker as the last step in compiling a program. If a source file references library functions or functions defined in other source files, the link editor combines these functions (with **main ()**) to create an executable file. The Linker assembles all functions, the program's functions and system's functions into one executable program.

Executing Programs

To execute a program we use an operating system command, such as run, to load the program from a storage device to primary memory and execute it. Getting the program into memory is the function of an operating system program known as the loader. It locates the executable program and related files into main memory from where it can be executed by the CPU.

Stored Program Concept

A stored program architecture is a fundamental computer architecture wherein the computer executes the instructions that are stored in its memory.

Languages.

These programs facilitate the users to make their own programs. User's programs are converted to machine oriented and the computer does the rest of works.

Application Programs

These programs are written by users for specific purposes.

Computer Languages

They are of three types –

1. Machine Language (Low level language)
2. Assembly language (Middle level language)
3. User Oriented language (Higher level language)

Machine language depends on the hard ware and comprises of 0 and 1 .This is tough to write as one must know the internal structure of the computer. At the same time assembly language makes use of English like words and symbols. With the help of special programs called Assembler, assembly language is converted to machine oriented language. Here also a programmer faces practical difficulties. To over come this hurdles user depends on Higher level languages, which are far easier to learn and use. To write programs in higher level language, programmer need not know the characteristics of a computer. Here he uses English alphabets, numerals and some special characters.

Some of the Higher level languages are FORTRAN, BASIC, COBOL, PASCAL, C, C++, ADA etc. We use C to write programs. Note that Higher level languages can not directly be followed by a computer. It requires the help of certain softwares to convert it into machine coded instructions. These softwares are called Compiler, Interpreter, and Assembler. The major difference between a compiler and an interpreter is that compiler compiles the user's program into machine coded by reading the whole program at a stretch where as Interpreter translates the program by reading it line by line.

C and BASIC are an Interpreter where as FORTRAN is a language for processing numerical data, but it does not lend itself very well to organizing large programs. Pascal can be used for writing well-structured and readable programs, but it is not as flexible as the C programming language. C++ goes one step a head of C by incorporating powerful object-oriented features, but it is complex and difficult to learn.

Programming Methodology

A computer is used to a solve a problem. Steps

1. Analyze the problem
2. Identify the variables involved
3. Design the solution
4. Write the program
5. Enter it into a computer
6. Compile the program and correct errors
7. Correct the logical errors if any

8. Test the program with data
9. Document the program

Algorithms

Step by step procedure in sequence for solving a problem is called algorithm.

Example

To make a coffee

Step1: Take proper quantity of water in a cooking pan Step2: Place the pan on a gas stow and light it

Step3: Add Coffee powder when it boils

Step4: Put out the light and add sufficient quantity of sugar and milk

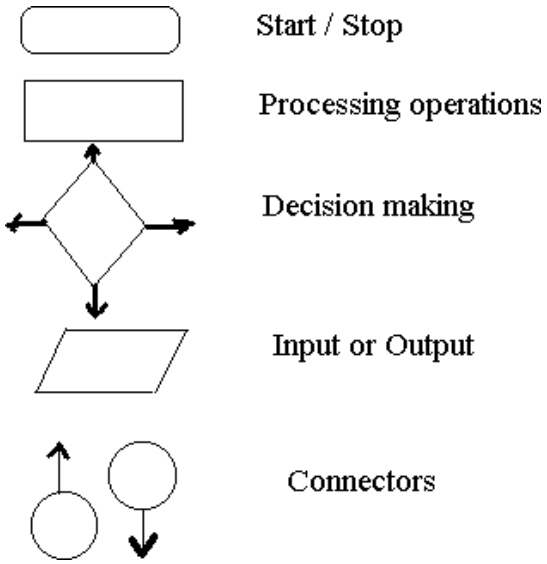
Step5: Pour into cup and have it.

To add two numbers

Step1: Input the numbers as x, y Step2: sum=x + y

Step3: print sum

For a better understanding of an algorithm, it is represented pictorially. The pictorial representation of an algorithm is called a Flow Chart. For this certain pictures are used.

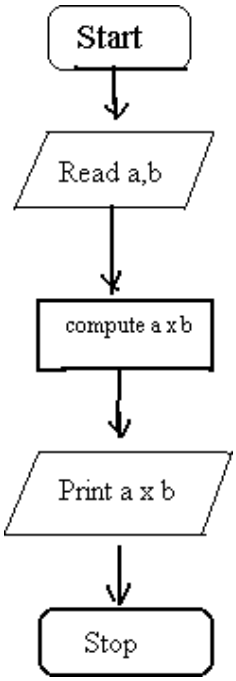


Consider a problem of multiplying two numbers

Algorithm

Step1: Input the numbers as a and b Step2: find the

product a x b Step3: Print the result



Flow chart

In the above example execution is done one after another and straight forward. But such straight forward problems occur very rarely. Sometimes we have to depend on decision making at certain stage in a normal flow of execution. This is done by testing a condition and appropriate path of flow is selected. For example, consider the following problem

To find the highest of three numbers

Algorithm

Step 1: read the numbers as x ,y and z Step 2:

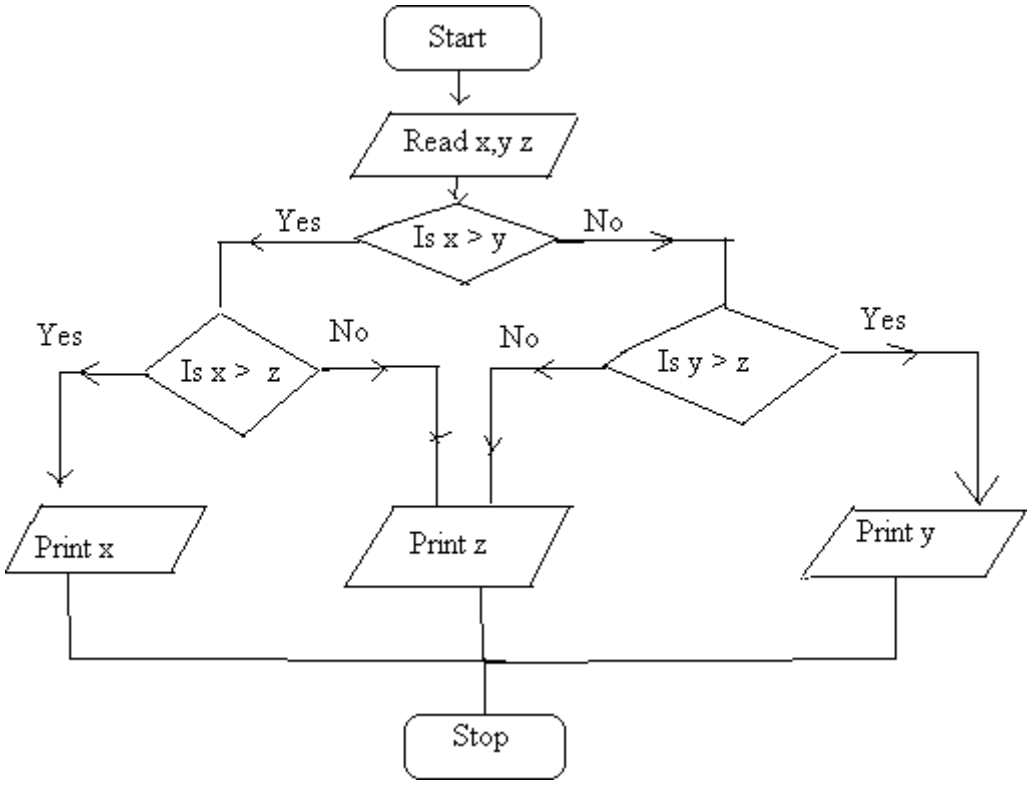
compare x and y

Step 3: if x > y then compare x with z and find the greater Step 4: Otherwise

compare y with z and find the greater

Flow Chart :

It is the graphical representation of an algorithm.



Exercise: Write Algorithm and flow chart for the solution to the problem

1. To find the sum of n, say 10, numbers.
2. To find the factorial of n , say 10.
3. To find the sum of the series $1+x+x^2+x^3+ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots + x^n$
4. To find the sum of two matrices.
5. To find the scalar product of two vectors
6. To find the Fibonacci series up to n
7. To find gcd of two numbers

A brief history of C

C devolved from a language called B, written by Ken Thompson at Bell Labs in 1970. Ken used B to write one of the first implementations of UNIX. B in turn was a descendant of the language BCPL (developed at Cambridge (UK) in 1967), with most of its instructions removed.

So many instructions were removed in going from BCPL to B, that Dennis Ritchie of Bell Labs put some back in (in 1972), and called the language C.

The famous book *The C Programming Language* was written by Kernighan and Ritchie in 1978, and was the definitive reference book on C for almost a decade.

The original C was still too limiting, and not standardized, and so in 1983 an ANSI committee was established to formalise the language definition.

It has taken until now (ten years later) for the ANSI (American National Standard Institute) standard to become well accepted and almost universally supported by compilers.

Structure of a program

Every C program consists of preprocessor commands, a global declaration section, and one or more modules called functions. One of these functions is called main. The preprocessor directives contain special instructions that indicate how to prepare the program for compilation. One of the most important and commonly used preprocessor commands is *include* which tells the compiler that to execute the program, some information is needed from the specified header file. The global declaration statements can be accessed at any part or block of a program as needed. A C program contains one or more functions where, a function is defined as a group of C statements that are written

in a logical sequence to perform a specific task. The **main()** function is the most important function and is a part of every C program . The execution of a C program begins at this function.

All the functions (including main()) are divided into two parts-the declaration section and the statement section. The declaration section precedes the statement section and is used to describe the data that will be used in the function. Note that data declared within a function are known as local declaration as that data will be visible only within that function. The life time of the data will be only till the function ends. The statement section in a function contains the code that manipulates the data to perform a specified task.

Preprocessor directives
Global declarations
main()
{
Local declarations
Statements
}
Function1()
{

Local declarations
Statements
}
…….
…….
FunctionsN()
{
Local declarations
Statements
}

STRUCTURE OF A C PROGRAM

Programmer can choose any name for functions. It is not mandatory to write Function1, Function2 etc. but with the exception that every program must contain one function that has its name as *main()*.

Writing the First C Program

To write a C program, we first need to write the code. For this, open a text editor. If you are a windows user you may use Notepad and if we want to write on UNIX/Linux we can use emacs or vi. Once the text editor is opened on your screen type the following program statements.

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

```
int main()
{
printf("\n This is your First C Program");
return 0;
}
Output:
This is your First C Program
```

After writing the code save the file in any name with a .c or .cpp extension(ex. file1.c or prog1.c file1 or prog1 are file name that can be chosen by user) To see the output we need to compile the code in any compiler like TC or DevC or GCC followed by run or execution of the code. During compilation it show error status. Because before execution the program should be error free. Compilation can be done by clicking compile option in the compiler followed by click run option. If you are a Windows user then open the command prompt by clicking Start->Run and typing 'command' and clicking OK. Using command prompt, change to the directory in which you had saved your file and then type.

c:\>tc file1.c

In case you are working on UNIX/Linux operating system, then exit the text editor and type $cc file1.c -ofile1

-o is the output file name. If you have leave out the -o then the file name a.out is used. After compilation **.exe** file creates that can be directly run by clicking **file1.exe** for Windows and typing **./file1** for UNIX /Linux operating system. When we run .exe file,

output of the program will be displayed on the screen i.e.

This is your First C Program.

#include<stdio.h>

This is a preprocessor command that comes as the first statement in our code. All preprocessor commands start with symbol hash(#). The #include statement tells the compiler to include the standard input/output library or header file (stdio.h) in the program. This file has some in-built functions. By simply including this file in our code we can use these functions directly. The standard input/output header file contains functions for input and output of data like reading values from the keyboard and printing the results on the screen.

int main()

Every C program contains a main() function which is the starting point of the program . *int* is the return value of the main(). After all the statement of the program have been written, the last statement of the program will return an integer value to the operating system . {} The two curly brackets are used to group all the related statements of the main function. All the statements between the braces form the function body. The function body contains a set of instructions to perform the given task.

printf("\n This is your First C Program");

The **printf** function is defined in the **stdio.h** file and is used to print the text on the screen. The message that has to be displayed on the screen is enclosed within double quotes and put inside the brackets. The message is quoted because in C a text is always put between inverted commas. '\n' is an escape sequence and represents a newline character. It is used to print the message on a new line on the screen. Escape Sequences are actually non-printing control characters that begin with a backlash(\).

return 0;

This is a return command that is used to return the value 0 to the operating system to give an indication that there were no errors during the execution of the program.

; (Semicolon)

Every statement in the main function ends with a semicolon(;).

Example

```
/* program to find the area pf a circle */
#include<stdio.h>
#include<conio.h> int
main( )
{
float r, a;
printf("radius");
```

```
scanf("%f", &r);
a=3.145*r*r;
printf("area of circle=%f", area);
return 0;
}
```

Library Function

They are built in programs readily available with the C compiler. These functions perform certain operations or calculations. Some of these functions return values when they are accessed and some carry out certain operations like input, output. A library functions accessed in a used written program by referring its name with values assigned to necessary arguments.

Some of these library functions are:

abs(i), ceil(d), cos(d), cosh(d), exp(d), fabs(d),floor(d), getchar( ), log(d),

pow(d,i),

printf( ), putchar(c), rand( ), sin(d), sqrt(d), scanf( ), tan(d), toascii(c),

toupper(c), tolower(c).

Note : the arguments i, c, d are respectively integer, char and double type.

Example:

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

```
#include<stdio.h>
#include<conio.h>
int main( )
{
float x, s;
printf(" \n input the values of x :");
scanf("%f ",&x);
s=sqrt(x);
printf("\n the square root is %f ",s);
return 0;
    }
```

Note that C language is case sensitive, which means ‘a’ and ‘A’ are different. Before the main program there are statements begin with # symbol. They are called preprocessor statements. Within the main program “ float r, a;” is a declaration statement. ‘include’ is a preprocessor statement. The syntax is #include<file name>. it is to tell the compiler looking for library functions, which are used in the program, included in the file, file name (like stdio.h, conio.h, math.h, etc…).

Data Input and Output

For inputting and outputting data we use library function. The important of these functions are getch( ), putchar( ), scanf( ), printf( ), gets( ), puts( ). For using these functions in a C-program there should be a preprocessor statement.

stdio.h is a header file that contains the built in program of these standard input output function.

getchar function

It is used to read a single character (char type) from keyboard. The syntax is

char variable name = getchar( ); Example:

```
char c;
c = getchar( );
```

For reading an array of characters or a string we can use getchar( ) function.

```
#include<stdio.h> int
main( )
{
char place[80]; int i;
for(i = 0;( place [i] = getchar( ))! = '\n', ++i);
return 0;
}
```

This program reads a line of text.

Putchar Function

It is used to display single character. The syntax is

putchar(char c);

Example:

```
char c; c = 'a';
putchar(c);
```

Using these two functions, we can write a very basic program to copy the input, a character at a time, to the output:

```
#include <stdio.h>
/* copy input to output */
   int main()
{
       int c;
     c = getchar();
      while(c !=   EOF)
{
putchar(c);
c = getchar();
}
     return 0;
}
```

scanf function

This function is generally used to read any data type- int, char, double, float, string.

The syntax is

scanf (control string, list of arguments);

The control string consists of group of characters, each group beginning % sign and a

conversion character indicating the data type of the data item. The conversion characters are c,d,e,f,o,s,u,x indicating the type resp. char decimal integer, floating point value in exponent form, floating point value with decimal point, octal integer, string, unsigned integer, hexadecimal integer. ie, "%s", "%d" etc are such group of characters.

An example of reading a data:

```
#include<stdio.h>
Int main( )
{
char name[30], line; int x;
float y;
……
…….…
scanf("%s%d%f", name, &x, &y); scanf("%c",line);
}
```

NOTE:

1. In the list of arguments, every argument is followed by & (ampersand symbol) except string variable.
2. s-type conversion applied to a string is terminated by a blank space character. So string having blank space like "Govt. Victoria College" cannot be read in this manner. For reading such a string constant we use the conversion string as "%[^\n]" in place of "%s".

Example:

```
char place[80];
```

……. scanf(“%[^\n]”, place);

……..

with these statements a line of text (until carriage return) can be input the variable ‘place’.

printf function

This is the most commonly used function for outputting a data of any type.

The syntax is

printf(control string, list of arguments)

Here also control string consists of group of characters, each group having % symbol and

conversion characters like c, d, o, f, x etc.

Example:

```
#include<stdio.h>
`                    int main()
{
int x; scanf(“%d”,&x); x*=x;
printf(“The square of the number is %d”,x);
return 0;
}
```

Note that in this list of arguments the variable names are without &symbol unlike in the case of scanf( ) function. In the conversion string one can include the message to be displayed. In the above example “The square of the number is” is displayed and is

followed by the value of x. For writing a line of text (which include blank spaces) the conversion string is “%s” unlike in scanf function. (There it is “[^\n]”).

More about printf statement

There are quite a number of format specifiers for `printf`. Here are the basic ones :

```
%d                print an int argument in decimal
%ld               print a long int argument in decimal
%c                print a character
%s                print a string
%f                print a float or double argument
%e                same as %f, but use exponential notation
%g                use %e or %f, whichever is better
%o                print an int argument in octal (base 8)
%x                print an int argument in hexadecimal (base 16)
%%                print a single %
```

To illustrate with a few more examples: the call

```
printf("%c %d %f %e %s %d%%\n", '1', 2, 3.14, 56000000.,
"eight", 9);
```

would print

```
1 2 3.140000 5.600000e+07 eight 9%
```

The call

```
printf("%d %o %x\n", 100, 100, 100);
```

would print

100 144 64

Successive calls to `printf` just build up the output a piece at a time, so the calls

```
printf("Hello, ");
printf("world!\n");
```

would also print `Hello, world!` (on one line of output).

While inputting or outputting data field width can also be specified.This is included in the conversion string.(if we want to display a floating point number convert to 3 decimal places the conversion string is "%.3f").For assigning field width,width is placed before the conversion character like "%10f","%8d","%12e" and so on…Also we can display data making correct to a fixed no of decimal places.

For example if we want to display x=30.2356 as 30.24 specification may be "%5.2f" or simply "%.2f".

C Tockens

C tokens are the basic buildings blocks in C language which are constructed together to write a C program. Each and every smallest individual unit in a C program is known as C tokens. C tokens are of six types. They are

Keywords (eg: int, while),

Identifiers (eg: main, total),

Constants (eg: 10, 20),

Strings (eg: —total, —hello),

Special symbols (eg: (), {}),

Operators (eg: +, /,-,*)

The Character Set

C used the upper cases A,B,……..,Z, the lower cases a ,b,…..,z and certain special characters like **+ - * / = % & # ! ? ^ “ ‘ ~ \ < > ( ) = [ ] { } ; : . , _** blank space @ $. also certain combinations of these characters like \b, \n, \t, etc…

Identities and Key Words

Identifiers are used as the general terminology for the names of variables, functions and arrays. Identities are names given to various program elements like variables, arrays and functions. The name should begin with a letter and other characters can be letters and digits and also can contain underscore character (_) . They must consist of only letters, digits, or underscore. They must begin with a letter or underscore (_).

No other special character is allowed. It should not be a keyword. It must not contain white space. It should be up to 31 characters long as only first 31 characters are significant.

Exapmle: area, average, x12, name_of_place etc……

C keywords

These are the words that convey a special meaning to the C compiler. The keywords cannot be used as variable names. The list of C keywords is given below.

Auto, break, case, char, const, continue, default, do, double, else, enum, extern float, for, goto, if, int, long, register, return, short, signed, sizeof, static, struct , Switch, typedef, union,

unsigned, void, volatile, while. **(**Note that these words should not be used as identities**.)**

Data Type

The variables and arrays are classified based on two aspects- first is the data type it stores and the second is the type of storage. The basic data types in C language are int, char, float and double. They are respectively concerned with integer quantity, single character, numbers, with decimal point or exponent number and double precision floating point numbers (ie; of larger magnitude). These basic data types can be augmented by using quantities like short, long, signed and unsigned. (ie; long int, short int, long double etc).

Constants

There are 4 basic types of constants. they are integer constants, floating-point constants, character constants and string constants.

(a) **integer constants**: It is an integer valued numbers, written in three different number system, decimal (base 10) , octal(base8), and hexadecimal(base 16).

A decimal integer constant consists of 0,1,…..,9..

Example : 75 6,0,32, etc…..

5,784, 39,98, 2-5, 09 etc are **not** integer constants.

An octal integer constant consists of digits 0,1,…,7. with 1st digit 0 to indicate that it is an octal integer.

Example :0, 01, 0756, 032, etc…..

32, 083, 07.6 etc ..... are **not** valid octal integers.

A hexadecimal integer constant consists of 0,1, …,9,A, B, C,D, E, F. It begins with 0x.

Example: 0x7AA2, 0xAB, etc……

0x8.3, 0AF2, 0xG etc are **not** valid hexadecimal constants.

Usually negative integer constant begin with (-) sign. An unsigned integer constant is identified by appending U to the end of the constant like 673U, 098U, 0xACLFU etc. Note that 1234560789LU is an unsigned integer constant.

(b) floating point constants : It is a decimal number (ie: base 10) with a decimal point or an exponent or both. Ex; 32.65, 0.654, 0.2E-3, 2.65E10 etc.
These numbers have greater range than integer constants.

(c)character constants : It is a single character enclosed in single quotes like 'a'. '3', '?', 'A' etc. each character has an ASCII to identify. For example 'A' has the ASCII code 65, '3' has the code 51 and so on.

(d)escape sequences: An escape sequence is used to express non printing character like a new line, tab etc. it begin with the backslash (\ ) followed by letter like a, n, b, t, v, r, etc. the commonly used escape sequence are

\a : for alert \n : new line \0 : null

\b : backspace	\f : form feed	\? : question mark
\f : horizontal tab	\r : carriage return	\' : single quote
\v : vertical tab	\" : quotation mark	

(e)string constants : it consists of any number of consecutive characters enclosed in double quotes .Ex : " C program" , "mathematics" etc……

Variables and arrays

A variable is an identifier that is used to represent some specified type of information. Only a single data can be stored in a variable. The data stored in the variable is accessed by its name. before using a variable in a program, the data type it has to store is to be declared.

Example : int a, b, c,

a=3; b=4;

c=a+b

Note : A statement to declare the data types of the identifier is called declaration statement. An array is an identifier which is used to store a collection of data of the same type with the same name. the data stored is an array are distinguished by the subscript. The maximum size of the array represented by the identifier must be mentioned.

Example : int mark[100] .

With this declaration n, mark is an array of size 100, they are identified by nark[0], mark[1],.....,mark[99].

Note : along with the declaration of variable, it can be initialized too. For example int x=10; with this the integer variable x is assigned the value 10, before it is used. Also note that C is a case sensitive langauge. i.e. the variables d and D are different.

Declaration

This is for specifying data type. All the variables, functions etc must be declared before they are used. A *declaration* tells the compiler the name and type of a variable you'll be using in your program. In its simplest form, a declaration consists of the type, the name of the variable, and a terminating semicolon:

Example : int a,b,c;

float mark, x[100], average; char name[30];

```
char c; int i;
```

float f;

You may wonder *why* variables must be declared before use. There are two reasons:

1. It makes things somewhat easier on the compiler; it knows right away what kind of storage to allocate and what code to emit to store and manipulate each variable; it doesn't have to try to intuit the programmer's intentions.
2. It forces a bit of useful discipline on the programmer: you cannot introduce variables willy-nilly; you must think about them enough to pick appropriate types for them. (The compiler's error messages to you, telling you that you apparently forgot to declare a variable, are as often helpful as they are a nuisance: they're helpful when they tell you that you misspelled a variable, or forgot to think about exactly how you were going to use it.)

Expression

This consists of a single entity like a constant, a variable, an array or a function name. it also consists of some combinations of such entities interconnected by operators.

Example : a, a+b, x=y, c=a+b, x<=y etc……..

Statements

Statements are the ”steps” of a program. Most statements compute and assign values or call functions, but we will eventually meet several other kinds of statements as well. By default, statements are executed in sequence, one after another. A statement causes the compiler to carry out some action. There are 3 different types of statements – expression statements compound statements and control statements. Every statement ends with a semicolon.

Example:

```
(1) c=a + b;
(2)                 {
a=3; b=4;
c=a+b;
}

(3)                 if (a<b)
{
printf("\n a is less than b");
}
```

Statement may be single or compound (a set of statements).

Most of the statements in a C program are *expression statements*. An expression statement is simply an expression followed by a semicolon. The lines

```
i = 0;
i = i + 1;
and         printf("Hello, world!\n");
```

```
are all expression statements
```

Symbolic Constants

A symbolic constant is a name that substitutes for a sequence of characters, which represent a numeric, character or string constant. A symbolic constant is defined in the beginning of a program by using #define, without: at the end.

Example :

```
#define      pi     3.1459
#define      INTEREST     P*N*R/100
```

With this definition it is a program the values of p, n ,r are assigned the value of INTEREST is computed.

Note : symbolic constants are not necessary in a C program.

UNIT-2

OPERATORS AND EXPRESSIONS

Arithmetic Operators

The basic operators for performing arithmetic are the same in many computer languages:

```
+                   addition
-                   subtraction
*                   multiplication
/                   division
%                   modulus (remainder)
```

For exponentiations we use the library function **pow**. The order of precedence of these operators is % / * + - . it can be overruled by parenthesis. **Integer Division** Division of an integer quantity by another is referred to integer division. This operation results in truncation. i.e.When applied to integers, the division operator `/` discards any remainder, so `1 / 2` is 0 and `7 / 4` is 1. But when either operand is a floating-point quantity (type `float` or `double`), the division operator yields a floating-point result, with a potentially nonzero fractional part. So `1 / 2.0` is 0.5, and `7.0 / 4.0` is 1.75.

Example :

```
int a, b, c;
a=5; b=2;
c=a/b;
```

Here the value of c will be 2

Actual value will be resulted only if a or b or a and b are declared floating type. The value of an arithmetic expression can be converted to different data type by the statement (data type) expression.

Example :

```
int a, b;
float c;a=5;b=2; c=(float) a/b
```

Here c=2.5

Order Preference

Multiplication, division, and modulus all have higher *precedence* than addition and subtraction. The term ``precedence" refers to how ``tightly" operators bind to their operands (that is, to the things they operate on). In mathematics, multiplication has higher precedence than addition, so `1 + 2 * 3` is 7, not 9. In other words, `1 + 2 * 3` is equivalent to `1 + (2 * 3)`. C is the same way.

Unary Operator

A operator acts up on a single operand to produce a new value is called a unary operator.

(1) the **decrement and increment** operators - ++ and -- are unary operators. They increase and decrease the value by 1. if x=3 ++x produces 4 and –x produces 2.

Note : in the place of ++x , x++ can be used, but there is a slight variation. In both csse x is incremented by 1, but in the latter case x is considered before increment.

(2) **sizeof** is another unary operator

```
int x, y; y=sizeof(x);
```

The value of y is 2 . the *sizeof* an integer type data is 2 that of float is 4, that of

double is 8, that of char is1.

Relational and Logical Operator

< (less than), <= (less than or equal to), > (greater than), >= (greater than or equal to), = = (equal to) and != (not equal to) are relational operators.

A logical expression is expression connected with a relational operator. For example ‘b*b – 4*a*c< 0 is a logical expression. Its value is either true or false.

```
int i, j, k ;
i=2;
j=3 ;
k=i+j ;
```

k>4 has the value true k<=3 has the value false.

Logical Operator

The relational operators work with arbitrary numbers and generate true/false values. You can also combine true/false values by using the *Boolean operators*, which take true/false values as operands and compute new true/false values. The three Boolean operators are:

```
&&                  and
```

```
||                  or
!                   not (takes one operand; ``unary'')
```

The && (``and'') operator takes two true/false values and produces a true (1) result if both operands are true (that is, if the left- hand side is true **and** the right-hand side is true). The || (``or'') operator takes two true/false values and produces a true (1) result if either operand is true. The ! (``not'') operator takes a single true/false value and negates it, turning false to true and true to false (0 to 1 and nonzero to 0).

&& (and) and || (or) are logical operators which are used to connect logical expressions. Where as ! (not) is unary operator, acts on a single logical expression.

For example : 1. (a<5) && (a>-2)

2. (a<=3) || (b>2)

In the first example if a= -3 or a=6 the logical expression returns true.

Assignment Operator

These operators are used for assigning a value of expression to another identifier.

=, + =, - = , * =, /= and %= are assignment operators.

a = b+c results in storing the value of b+c in ‘a’. a += 5 results

in increasing the value of a by 5

a /= 3 results in storing the value a/3 in a and it is equivalent a=a/3

Note : **1**. if a floating point number is assigned to a integer type data variable, the value will be

truncated.

nple float a=5.36; int b;

b=a

It results in storing 5 to b.

Similarly if an integer value is a assigned to a float type like float x=3 the value of x stored is 3.0.

Conditional Operator

The operator ?: is the conditional operator. It is used as

variable 1 = expression 1 ? expression 2 : expression 3.

Here expression 1 is a logical expression and expression 2 and expression 3 are expressions having numerical values. If expression 1 is true, value of expression 2 is assigned to variable 1 and otherwise expression3 is assigned.

Example:

```
int a,b,c,d,e
a=3;b=5;c=8;
d=(a<b)? a : b;
     e=(b>c) ? b : c;
     Then d=3 and e=8
```

Example Program For Conditional/Ternary Operators in C

```
#include <stdio.h>
int main()
{
int x=1, y ;
y = ( x ==1 ? 2 : 0 ) ;
printf("x value is %d\n", x);
```

printf("y value is %d", y);

}

OUTPUT:

x value is 1

y value is 2

Bit Wise Operators in C

These operators are used to perform bit operations. Decimal values are converted into binary values which are the sequence of bits and bit wise operators work on these bits. Bit wise operators in C language are & (bitwise AND),

| (bitwise OR),

~ (bitwise OR),

^ (XOR),

<< (left shift) and

>> (right shift).

Consider x=40 and y=80. Binary form of these values are given below.

x = 00101000

y= 01010000

All bit wise operations for x and y are given below.

x&y = 00000000 (binary) = 0 (decimal) x|y = 01111000 (binary) = 120 (decimal) ~x = 11111111111111111111111111 11111111111111111111111111111010111 .. ..= -41 (decimal) x^y = 01111000 (binary) = 120 (decimal) x << 1 = 01010000 (binary) = 80 (decimal) x >> 1 = 00010100 (binary) = 20 (decimal)

Note: Bit wise NOT: Value of 40 in binary

is00000000000000000000000000000000000000000000000000010100000000000. So, all 0‘s are converted into 1‘s in bit wise NOT operation. Bit wise left shift and right shift : In left shift operation —x << 1 —, 1 means that the bits will be left shifted by one place. If we use it as —x << 2 —, then, it means that the bits will be left shifted by 2 places.

Example Program, For Bit Wise Operators in C

In this example program, bit wise operations are performed as shown above and output is displayed in decimal format.

```
#include <stdio.h>
int main()
{
int m = 40,n = 80,AND_opr,OR_opr,XOR_opr,NOT_opr ;
AND_opr = (m&n);
OR_opr = (m|n);
NOT_opr = (~m);
XOR_opr = (m^n);
printf("AND_opr value = %d\n",AND_opr );
printf("OR_opr value = %d\n",OR_opr );
printf("XOR_opr value = %d\n",XOR_opr );
printf("left_shift value = %d\n", m << 1);
printf("right_shift value = %d\n", m >> 1);
}
```

OUTPUT:

```
AND_opr value = 0
OR_opr value = 120
NOT_opr value = -41
```

XOR_opr value = 120

left_shift value = 80

right_shift value = 20

Type Conversion in Expressions

When variables and constants of different types are combined in an expression then they are converted to same data type. The process of converting one predefined type into another is called type conversion. Type conversion in c can be classified into the following two types: Implicit Type Conversion When the type conversion is performed automatically by the compiler without programmer‘s intervention, such type of conversion is known as implicit type conversion or type promotion. The compiler converts all operands into the data type of the largest operand. The sequence of rules that are applied while evaluating expressions are given below: All short and char are automatically converted to int, then, If either of the operand is of type long double, then others will be converted to long double and result will be long double. Else, if either of the operand is double, then others are converted to double.

Else, if either of the operand is float, then others are converted to float. Else, if either of the operand is unsigned long int, then others will be converted to unsigned long int. Else, if one of the operand is long int, and the other is unsigned int, then if a long int can represent all values of an unsigned int, the unsigned int is converted to long int. otherwise, both operands are converted to unsigned long int. Else, if either operand is long int then other will be converted to long int. Else, if either operand is unsigned int then others will be converted to unsigned int. It should be noted that the final result of expression is converted to type of variable on left side of assignment operator before assigning value to it. Also, conversion of float to int causes truncation of fractional part,

conversion of double to float causes rounding of digits and the conversion of long int to int causes dropping of excess higher order bits.

Explicit Type Conversion The type conversion performed by the programmer by posing the data type of the expression of specific type is known as explicit type conversion. The explicit type conversion is also known as type casting. Type casting in c is done in the following form: (data_type)expression; where, data_type is any valid c data type, and expression may be constant, variable or expression. For example, x=(int)a+b*d; The following rules have to be followed while converting the expression from one type to another to avoid the loss of information: All integer types to be converted to float. All float types to be converted to double. All character types to be converted to integer.

Control Statements

When we run a program, the statements are executed in the order in which they appear in the program.Also each statement is executed only once. But in many cases we may need a statement or a set of statements to be executed a fixed no of times or until a condition is satisfied. Also we may want to skip some statements based on testing a condition. For all these we use control statements .

Control statements are of two types – branching and looping.

Branching

It is to execute one of several possible options depending on the outcome of a logical test ,which is carried at some particular point within a program.

Looping

It is to execute a group of instructions repeatedly,a fixed no of times or until a specified condition is satisfied.

Conditional Branching

1. if else statement

It is used to carry out one of the two possible actions depending on the outcome of a logical test.The else portion is optional.

The syntax is

if (expression) statement1 [if there is no else part]

Or

if (expression)

statement1;

else

statement2;

Here expression is a logical expression enclosed in parenthesis. if expression is true ,statement 1 or statement 2 is a group of statements

,they are written as a block using the braces { }

e:

```
1. if(x<0) printf("\n x is negative"); 2. if(x<0)
printf("\n x is negative"); else
    printf("\n x is non negative");

                                3.if(x<0)
                                        {
                                        x=-x;
    s=sqrt(x);
    }
    else
    x=sqrt(x);
```

2. **nested if statement**

Within an if block or else block another if – else statement can come. Such statements are called nested if statements.

The syntax is

if (expression1)

statement1

if (expression2)

statement2 else

statement3

3. **Ladder if Statement**

Inorder to create a situation in which one of several courses of action is executed we use ladder – if statements.

The syntax is

If (expression1)

statement1

else if (expression2)

statement2

else if (expression3)

statement3

……….

……….

else statementn

Example:

```
if(mark>=90) printf("\n excellent");
 else if(mark>=80) printf("\n very good"); else if(mark>=70)
    printf("\n good");
else if(mark>=60) printf("\n average"); else
    printf("\n to be improved");
```

Switch Statement

It is used to execute a particular group of statements to be chosen from several available options. The selection is based on the current value of an expression with the switch statement.

The Syntax is:

switch(expression){

case value1:

statement1;

break;

case value2:

statement2;

break;

……..

……..

default:

statement;

}

All the option are embedded in the two braces { }.Within the block each group is written after the label case followed by the value of the expression and a colon. Each group ends with '***break'*** statement. The last may be labeled ***'default'.*** This is to avoid error and to execute the group of statements in default if the value of the expression does not match value1, value2,……..

Looping

1. ***The while statement***

This is to carry out a set of statements to be executed repeatedly until some condition is satisfied.

The syntax is:

While (expression) statement

The statement is executed so long as the expression is true. Statement can be simple or compound.

Example 1:

```
#include<stdio.h>

main()

while(n > 0)

{

printf("\n"); n = n - 1;

}
```

Example2:

```
#include<stdio.h>

int main()

{

                        int i=1;

       while(x<=10)

  {

      printf("%d",i);

      ++i;

      }

      return 0;

      }
```

2. **do while statement**

This is also to carry out a set of statements to be executed repeatedly so long as a condition is true.

The syntax is:

do statement while(expression)

Example:

```
#include<stdio.h>
int main()
{
int i=1; do
{
printf("%d",i);
++i;
                        }while(i<=10);
                           Return 0;
                       }
```

Difference between while loop and do – while loop

1) In the while loop the condition is tested in the beginning whereas in the other case it is done at the end.
2) In while loop the statements in the loop are executed only if the condition is true. Whereas in do – while loop even if the condition is not true the statements are executed atleast once.

3. for loop

It is the most commonly used looping statement in C. The general form is

```
for(expression1;expression2;expression3)
{
statement;
}
```

Here expression1 is to initialize some parameter that controls the looping action. expression2 is a condition and it must be true to carry out the action. expression3 is a unary expression or an assignment expression.

Example:

```
#include<stdio.h>
int main()
{
int i; for(i=1;i<=10;++i) printf("%d",i);
return 0;
}
```

Here the program prints *i* starting from 1 to 10.First *i* is assigned the value 1 and than it checks whether *i*<=10 If so i is printed and then *i* is increased by one. It continues until *i*<=10.

An example for finding the average of 10 numbers;

```
#include<stdio.h> int
main()
{
int i;
float x,avg=0; for(i=1;i<=10;++i)
{
scanf("%f",&x); avg += x;
}
avg /= 10;
printf("\n average=%f",avg);
return 0;
```

```
}
```

Note: Within a loop another for loop can come

Example :

```
for(i=1;i<=10;++i)
for(j=1;j<=10;++j);
```

The break statement

The break statement is used to terminate4 loop or to exit from a switch. It is used in for, while, do-while and switch statement.

The syntax is ***break;***

Example 1: A program to read the sum of positive numbers only

```
#include<stdio.h>
int main()
{
int x, sum=0; int n=1;
while(n<=10)
{

      scanf("%d",&x); if(x<0) break;
sum+=x;
}
printf("%d",sum);
return 0;
}
```

Example 2 :A program for printing prime numbers between 1 and 100:

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

int main()
{
int i, j;
printf("%d\n", 2);
for(i = 3; i <= 100; i = i + 1)
{
for(j = 2; j < i; j = j + 1)
{
if(i % j == 0)
        break; if(j > sqrt(i))
{
printf("%d\n", i); break;
}
}
}

return 0;
}
```

Here while loop breaks if the input for x is –ve.

The continue statement:

It is used to bypass the remainder of the current pass through a loop. The loop does

not terminate when continue statement is encountered, but statements after continue are skipped and proceeds to the next pass through the loop. In the above example of summing up the non negative numbers when a negative value is input, it breaks and the execution of the loop ends. In case if we want to sum 10 nonnegative numbers, we can use *continue* instead of *break.*

Example :

```
#include<stdio.h>
int main()
{
int x, sum=0, n=0; while(n<10)
{
          scanf("%d",x);
          if(x<0) continue;
          sum+=x;
                ++n;
  } printf("%d",sum);
   return 0;
  }
```

goto statement

It is used to alter the normal sequence of program execution by transferring control to some other part of the program .The syntax is goto *label* ;

Example :

```
 #include<stdio.h>
 int  main( )
{
int n=1,x,sum=0;
while(n<=10)
{
scanf("%d" ,&x);
if(x<0)goto error; sum+=x;
++n;
}
error:
printf("\n the number is non negative");
return 0;
}
```

Arrays

An array is an identifier to store a set of data with common name. Note that a variable can store only a single data. Arrays may be one dimensional or multi-dimensional.

Defining an array one dimensional arrays

Definition: Arrays are defined like the variables with an exception that each array name must be accompanied by the size (i.e. the max number of data it can store).For a one dimensional array the size is specified in a square bracket immediately after the name of the array.

The **syntax** is

data-type array name[size];

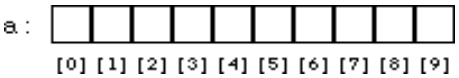
So far, we've been declaring simple variables: the declaration

```
int i;
```

declares a single variable, named `i`, of type `int`. It is also possible to declare an *array* of several elements. The declaration

```
int a[10];
```

declares an array, named `a`, consisting of ten elements, each of type `int`. Simply speaking, an array is a variable that can hold more than one value. You specify which of the several values you're referring to at any given time by using a numeric *subscript*. (Arrays in programming are similar to vectors or matrices in mathematics.) We can represent the array `a`



above with a picture like this:

eg: `int x[100];`

`float mark[50];`

`char name[30];`

Note: With the declaration int x[100],computer creates 100 memory cells with name x[0],x[1],x[2],……,x[99].Here the same identifier x is used but various data are distinguished by the subscripts inside the square bracket.

Array Initialization

Although it is not possible to assign to all elements of an array at once using an assignment expression, it is possible to initialize some or all elements of an array when the array is defined. The syntax looks like this:

```
int a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

The list of values, enclosed in braces `{}`, separated by commas, provides the initial values for successive elements of the array.

If there are fewer initializers than elements in the array, the remaining elements are automatically initialized to 0. For example,

```
int a[10] = {0, 1, 2, 3, 4, 5, 6};
```

would initialize `a[7]`, `a[8]`, and `a[9]` to 0. When an array definition includes an initializer, the array dimension may be omitted, and the compiler will infer the dimension from the number of initialisers. For example,

```
int b[] = {10, 11, 12, 13, 14};
```

Example

```
int x[ ] ={0,1,2,3,4,5}; or

int x[6]={0,1,2,3,4,5};

Even if the size is not mentioned (former case) the values

0,1,2,3,4 are stored in x[0],x[1],x[2],x[3],x[4],x[5].If the

statement is like

int x[3]={0,1,2,3,4,5};

then x[0],x[1],x[2] are assigned the values 0,1,2.
```

Note: If the statement is like

int x[6]={0,1,2};

then the values are stored like x[0]=0, x[1]=1, x[2]=2, x[3]=0, x[4]=0 and x[5]=0.

Processing one dimensional array

1) Reading arrays: For this normally we use for- loop.

If we want to read n values to an array name called 'mark' , the statements look like

```
int mark[200],i,n;

for(i=1;i<=n;++i)

scanf("%d",&x[i]);
```

Note: Here the size of array declared should be more than the number of values that are intended to store.

2) Storing array in another:

To store an array to another array. Suppose a and b are two arrays and we want to store that values of array a to array b. The statements look like

```
float a[100],b[100]; int

i; for(i=1;i<=100;++i)
```

```
b[i]=a[i];
```

Problem: To find the average of a set of values.

```
#include<stdio.h>
int main( )
{
int x,i;
float x[100],avg=0;
printf("\n the no: of values ");
scanf("%d",&n);
printf("\n Input the numbers");
for(i=1;i<=n;++i)
{
scanf("%f",&x[i]); avg=avg+x[i];
}
avg=avg/n;
printf("\n Average=%f",avg); return 0;
}
```

Passing Arrays To Function

Remember to pass a value to a function we include the name of the variable as an argument of the function.Similarly an array can be passed to a function by including arrayname (without brackets) and size of the array as arguments.In the function defined the arrayname together with empty square brackets is an argument.

Ex:

(calling function)-avg=average(n,x); where n is the size of the data stored in the array x[].

(function defined)- float average(int n,float x[]);

Now let us see to use a function program to calculate the average of a set of values.

```
#include<stdio.h>
 float average(int n,float y[]); int
main()
{
int n;
float x[100],avg;
printf("\n Input the no: of values");
scanf("%d",&n);
printf("\n Input the values");
for(i=1;i<=n;++i)
scanf("%f",&x[i]); avg=average(n,x);
printf("\n The average is %f",avg); return 0;
}
float average(int n, float y[]);
{
float sum=0; int i;
for(i=1;i<=n;++i)
sum=sum+y[i]; sum=sum/n;
return(sum);
}
```

Note:

1) In the function definition the array name together with square brackets is the argument. Similarly in the prototype declaration of this function too, the array name with square brackets is the argument

2) We know that changes happened in the variables and arrays that are in function will not be reflected in the main (calling)

program even if the same names are usual. If we wish otherwise the arrays and variables should be declared globally. This is done by declaring them before the main program.

Ex:

```
#include<stdio.h>
void arrange(int n,float x[]); main();
{
……..
arrange(n,x);
……..
}
arrange(int n,float x[]);
{
…….
return;
}
```

Problem : Write a program to arrange a set of numbers in ascending order by using a function program with global declaration.

MULTI-DIMENSIONAL ARRAYS

Multi-dimensional arrays are defined in the same manner as one dimensional arrays except that a separate pair of square brackets is required to each subscript.

Example: float matrix[20][20] (two dimensional)

Int x[10][10][5] (3-dimensional)

Initiating a two dimensional array we do as int x[3][4]={1,2,3,4,5,6,7,8,9,10,11,12}

Or

```
int x[3][4]={
{1,2,3,4};
{5,6,7,8};
{89,10,11,12};
}
```

NOTE: The size of the subscripts is not essential for initialization. For reading a two dimensional array we use two for-loop.

Example

```
for(i=1;i<=2;++i) for(j=1;j<=3;++j)
   scanf("%f",&A[i][j]);
```

NOTE: If `x[2][3]` is a two dimensional array, the memory cells are identified with name `x[0][0],x[0][1],x[0][2],x[1][0],x[1][1] and x[1][2].`

Array and Strings

A string is represented as a one dimensional array of character type. A string is a collection of characters. A string is also called as an array of characters. A String must access by %s access specifier in c and c++. A string is always terminated with \0 (Null) character. Example of string: —"Gaurav" · A string always recognized in double quotes. A string also consider space as a character.

Example: " Gaurav Arora" · The above string contains 12 characters.

Example: Char ar[20] · The above example will store 19 character with I null character.

Example : char name[20];

Here name is an array that can store a string of size 20.

If we want to store many strings(like many names or places) two dimensional array is used. Suppose we want to store names of 25 persons, then declare name as char name[25][ ]. Note that the second square bracket is kept empty if the length of string is not specified.

If the declaration is char name[25][30], 25 names of maximum size 30 can be stored. The various names are identified by name[0], name[1], name[2], ,
name[24]. These names are read by the command.

```
for( i=0; i<25;++i)
scanf( "%[^\n]",name(i));
```

Example: Program based upon String.

WAP to accept a complete string (first name and last name) and display in the output.

```
 # include<stdio.h>
 #include<conio.h>
 #include<string.h>
 void main ()
 {
char str1[20];
char str2[20];
printf("Enter First Name");
 scanf("%s",&str1);
printf("Enter last Name");
scanf("%s",&str2);

 puts(str1); puts(str2);
 }
```

String Functions in C: Our c language provides us lot of string functions for manipulating the string. All the string functions are available in string.h header file.

1. strlen().
2. strupr().
3. strlwr().
4. strcmp().
5. strcat().
6. strapy().
7. strrev().

1. strlen(). This string function is basically used for the purpose of computing the length of string.

```
Example: char str="Gaurav Arora";
              int length= strlen(str);
                printf("The length of the string is =",str);
```

2. strupr(). This string function is basically used for the purpose of converting the case sensitiveness of the string i.e. it converts string case sensitiveness into uppercase.

```
Example: char str = "Gaurav"      strupr(str);
              printf("The uppercase of the string is : %s",str);
```

3.strlwr (). This string function is basically used for the purpose of converting the case sensitiveness of the string i.e it converts string case sensitiveness into lowercase.

Example: char str = “Gaurav” strlwr(str);

printf(“The Lowercase of the string is :%s “,str);

4.strcmp ().

This string function is basically used for the purpose of comparing two string. This string function compares two strings character by characters. Thus it gives result in three cases:

Case 1: if first string > than second string then, result will be true.

Case 2: if first string < than second string then, result will be false.

Case 3: if first string = = to second string then, result will be zero.

Example: char str1= “Gaurav”; char str2= “Arora”; char str3=strcmp(str1,str2); printf(“%s”,str3);

5. strcat(). This string function is used for the purpose of concatenating two strings ie.(merging two or more strings)

Example: char str1 = “Gaurav”; char str2 = “Arora”; char str3[30]; str3=strcat(str1,str2); printf(“%s”,str3);

6. strcpy() This string function is basically used for the purpose of copying one string into another string. char str1= “Gaurav”; char str2[20]; str2 = strcpy(str2,str1); printf(“%s”,str2);

6. strrev() This string function is basically used for the purpose of reversing the string. char str1= “Gaurav”; char str2[20];

str2= strrev(str2,str1); printf(“%s”,str2);

Example: Program based upon string functions.

WAP to accept a string and perform various operations: 1. To convert string into upper case. 2. To reverse the string . 3. To copy string into another string. 4. To compute length depending upon user choice.

```
# include<stdio.h>
# include<conio.h>
 #include<string.h>
  int main() {
char str[20];
char str1[20];
int opt,len;
 printf(“\n  MAIN MENU”);
 printf(“\n 1. Convert string into upper case”);
 printf(“\n 2. Reverse the string”);
 printf(“\n 3. Copy one string into another string”);
 printf(“\n  4.Compute length of string “);
 printf(“Enter string “); scanf(“%s”, &str);
 printf(“Enter your choice”);
scanf(“%d”,&opt);
 switch(opt) {
 case 1:   strupr(str);
           printf(“The string in uppercase is :%s “,str);
             break;
```

```
case 2:   strrev(str);
          printf("The reverse of string is : %s",str);
           break;
case 3:  strcpy(str1,str);
         printf("New copied string is : %s",str1);
         break;
 case 4: len=strlen(str);
         printf("The length of the string is :  %s",len);
         break;
   default:
         printf("You have entered a wrong choice.");
 }

return 0;
}
```

UNIT-3

FUNCTIONS

Functions are programs .There are two types of functions- library functions and programmer written functions. We are familiarised with library functions and how they are accessed in a C program.

The advantage of function programs are many

1) A large program can be broken into a number of smaller modules.

2) If a set of instruction is frequently used in program and written as function program, it can be used in any program as library function.

Defining a Function

Generally a function is an independent program that carries out some specific well defined task. It is written after or before the main function. A function has two components-definition of the function and body of the function.

Generally it looks like

datatype function name(list of arguments with type)

{

statements return;

}

If the function does not return any value to the calling point (where the function is accessed) .The syntax looks like

function name(list of arguments with type)

{

statements return;

}

If a value is returned to the calling point, usually the return statement looks like return(value). In that case data type of the function is executed.

Note that if a function returns no value the keyword **void** can be used before the function name.

Example:

```
(1)       writecaption(char x[] );
           {
       printf("%s",x); return;
          }
(2)       int maximum(int x, int y)
          {
                int z ;
   z=(x>=y)? x: y ; return(z);
   }
(3)       maximum( int x,int y)
   {
   int z;
   z=(x>=y) ? x : y ; printf("\n maximum =%d",z);
   return ;
   }
```

Note: In example (1) and (2) the function does not return anything.

Advantage of Function

1. It appeared in the main program several times, such that by making it a function, it can be written just once, and the several places where it used to appear can be replaced with calls

to the new function.

2. The main program was getting too big, so it could be made (presumably) smaller and more manageable by lopping part of it off and making it a function.

3. It does just one well-defined task, and does it well.

4. Its interface to the rest of the program is clean and narrow

5. Compilation of the program can be made easier.

Accessing a Function

A function is accessed in the program (known as calling program)by specifying its name with optional list of arguments enclosed in parenthesis. If arguments are not required then only with empty parenthesis.

The arguments should be of the same data type defined in the function definition.

Example:

1). int a,b,y;

y=maximum(a,b);

2). char name[50];

writecaption(name);

If a function is to be accessed in the main program it is to be defined and written before the main function after the preprocessor statements.

Example:

```
#include<stdio.h>
int maximum (int x,int y)
{
int z ;
```

```
z=(x>=y) ? x : y ; return
(z);
}
int main( )
{
int a,b,c; scanf("%d%d",&a,&b);
c=maximum(a,b);
printf("\n maximum number=%d",c);
return 0;
}
```

Function Prototype

It is a common practice that all the function programs are written after the main( ) function. When they are accessed in the main program, an error of prototype function is shown by the compiler. It means the computer has no reference about the programmer defined functions, as they are accessed before the definition. To overcome this, i.e to make the compiler aware that the declarations of the function referred at the calling point follow, a declaration is done in the beginning of the program immediately after the preprocessor statements. Such a declaration of function is called prototype declaration and the corresponding functions are called function prototypes.

Example1:

1)

```
#include<stdio.h>
int maximum(int x,int y);
int main( )
{
```

```
 int a,b,c; scanf("%d%d",&a,&b);
c=maximum(a,b);
printf("\n maximum number is : %d",c);
return 0;
}
int maximum(int x, int y)
{
int z;
z=(x>=y) ? x : y ;
return(z);
    }
```

Example-2

```
#include<stdio.h>
void int factorial(int m);
int main( )
{
int n;
scanf("%d",&n);
factorial(n);
return 0;
}
void int factorial(int m)
{
 int i,p=1;
for(i=1;i<=m;++i)
```

```
p*=i;
printf("\n factorial of      %d  is  %d ",m,p);
return( );
}
```

Note: In the prototype declaration of function, if it return no value, in the place of data-type we use void.

Eg: void maximum(int x, int y);

Passing arguments to a function

The values are passed to the function program through the arguments. When a value is passed to a function via an argument in the calling statement, the value is copied into the formal argument of the function (may have the same name of the actual argument of the calling function). This procedure of passing the value is called passing by value. Even if formal argument changes in the function program, the value of the actual argument does not change.

Example

```
#include<stdio.h> void
square (int x); int
main( )
{
int x;
scanf("%d",&x);
square(x):
return 0;
```

```
}
void square(int x)
{
x*=x ;
printf("\n the square is %d",x);
return;
}
```

In this program the value of x in the program is unaltered.

Call by Value

In which values of variables are passed by the calling function to the called function. The programs that we have written so far call functions using call-by-value method of passing parameters. In call by value method, the called function creates new variables to store the value of the arguments passed to it. Therefore, the called function uses a copy of the actual arguments to perform its intended task.

If the called function is supposed to modify the value of the parameters passed to it, then the change will be reflected only in called function. In the calling function no change will be made to the value of variables .This is because all the changes were made to copy of the variables and not to the actual variables.

```
#include<stdio.h>
Void add(int n)
Int main(){
 Int num=2;
Printf("\n The value of num before calling the function=%d", num);
Add(num);
Printf("\n The value of num after calling the function=%d", num);
```

```
Return 0;

Void add(int n)

{

N=n+10;

Printf("\n The value of num in the called function =%d",n)

}

Output: The value of num before calling the function =2

The value of num in the called function =12

The value of sum after calling the function =2
```

In this program the add() accepts an integer variable num and adds to it. In the calling function, the value of num=2. In add(), the value of num is modified to 12 but in the calling function the change is not reflected.

Since the called function uses a copy of num, the value of num in the calling function remains untouched .

Call by Reference

In which address of variables are passed by the calling function to the called function. When calling function passes arguments to the called function using call-by-value method, the only way to return the modified value of the argument to the caller is explicitly using the **return** statement. The better option when a function wants to modify the value of the argument is to pass arguments using call-by-reference technique. In call by reference, we declare the function parameters as references rather than normal variables.When this is done by any change made by function to the arguments it receives are visible in the calling function. To indicate that an argument is passed using call by reference, an asterisk(*) is placed after the type in the parameter in the called function will then be reflected in the calling function.

Hence, in call-by-reference method, a function receives an implicit reference to the argument, rather than a copy of its value. Therefore, the function can modify the value of the variable and that change will be reflected in the calling function as well.

Example:

#include<stdio.h>

```
void add(int *n)
int main()
{
int num=2;
printf("\n The value of num before calling the function=%d", num);
add(&num);
printf("\n The value of num after calling the function=%d",num);
return 0;
}
void add(int *n)
{ *n=*n+10;
printf("\n The value of num in the called function=%d",*n);
}
Output:
The value of num before calling the function=2
The value of num in the called function=12
The value of num after calling the function=12
```

Advantages

The advantages of using the call-by-reference technique of passing arguments are as follows.

Since arguments are not copied into new variables, it provides greater time and space

efficiency.

The called function can change the value of the argument and the change is reflected in the calling function.

A return statement can return only one value. In case we need to return multiple values, pass those arguments by reference.

Disadvantages

However side effect of using this technique is that when an argument is passed using call by address, it becomes difficult to tell whether that argument is meant for input, output o

Storage Classes

Storage classes defines the scope(visibility) and lifetime of variables and/or functions declared within a C program. In addition to this, the storage class gives the following information about the variable or the function.

The storage class of a function or a variable determines the part of memory where storage space will be allocated for that variable or function (whether the variable/function will be stored in a register or in RAM).

It specifies how long the storage allocation will continue to exit for that function or variable.
It specifies the scope of variable or functions i.e. the storage class indicates the part of the C program in which the variable name is visible or the part in which it is accessible. In other words, whether the variable/function can be referenced throughout the program or only within the function, block, or source file where it has been defined.

It specifies whether the variable or function has internal, external or no linkage.
It specifies whether the variable or function will be automatically initialized to zero or to any indeterminate value.

There are 4 different storage classes specification in C – automatic, register, , external and static. The general syntax for specifying the storage class of a variable can be given as:

<storage_class_specification> <data type> <variable name>

Automatic Storage Class

The ***auto*** storage class specifier is used to explicitly declare a variable with automatic storage. It is local and its scope is restricted to that block or function. They are called so because such variables are created inside a function and destroyed automatically when the function is exited. Any variable declared in a function is interpreted as an automatic variable unless specified otherwise. So the keyword auto is not required at the beginning of each declaration.

Ex. **auto** int x;

Here x is an integer that has automatic storage. It is deleted when the block in which x is declared is exited.

Memory for the variable is automatically allocated upon entry to a block and freed automatically upon exit from that block.

The scope of the variable is local to the block in which it is declared. These variable may be declared within a nested block.

Every time the block is entered, the variable is initialized with the values declared.

If auto variables are not initialized at the time of declaration, then they contain some garbage value.

Register Storage Class

When a variable is declared using registered as its storage class, it is stored in a CPU register as its storage class, it is stored in a CPU register instead of RAM. Since the variable is stored in a register, the maximum size of the variable is equal to the the register size. One drawback of using a register variable is that they cannot be operated using the unary '&' operator because it does not have a memory location associated with it. A register variable is declared in the following manner.

register int x;

Registered variables are used when quick access to the variable is needed. It also stored in registered depending on the hardware and implementation restrictions. Like auto variable, *register* variables also have automatic storage duration. Each time a block is entered, the *register* variables defined in that block are accessible and the moment that block is existed, the variables becomes no longer accessible for use.

Example:

I

Extern Storage Class (Global Variable)

The variables which are alive and active throughout the entire program are called external variables. It is not centered to a single function alone, but its scope extends to any function having its reference. The value of a global variable can be accessed in any program which uses it. For moving values forth and back between the functions, the variables and arrays are declared globally i.e., before the main program. The keyword *external* is not necessary for such declaration, but they should be mentioned before the main program. Incase the **extern** variable is not initialiazed, it will be initialized to zero by default.

```
Example:
//FILE1.c
#include<stdio.h>
#include<FILE2.c> //programmer's own header file
int x;
 void print(void)
int main()
{
x=10;
printf("\n x in FILE1=%d", x);
print();
return 0;
}

//End of FILE1.c
```

```
//FILE2.c

#include<stdio.h>

extern int x;


void print()

{

Printf("\n x in FILE2=%d",x);

}

main(){

//statements

}

//end of FILE2.c

Output:

X in FILE1=10
```

In the program, we have used two files-FILE1 and FILE2.FILE1 has declared a global variable x. FILE1 also includes FILE2 which has print function that uses the external variable x to print its value on the screen.

Static Storage Class

It is, like automatic variable, local to functions is which it is defined. Unlike automatic variables static variable retains values throughout the life of the program, i.e. if a function is exited and then re-entered at a later time the static variables defined within the function will retain their former values. Thus this feature of static variables allows functions to retain information permanently throughout the execution of the program. Static variable is declared by using the keyword static. To declare an integer x as static , write

static int x=10; where x is a local static variable.

When a *static* variable is not explicitly initialized by the programmer, then it is automatically initialized to zero when memory is allocated for it. Although *static* automatic variables exist even after the block in which they are defined terminates, their scope is local to the block in which they are defined.

Static storage class can be specified for *auto* as well as *extern* variables.

Example : static float a ;

static extern int x;

When we declare a variable as *extern static* variable, then that variable is accessible from all the functions in the source file.

Look at the following code which clearly differentiates between a *static* variable and a normal variable.

```
#include<stdio.h>
void print(void);

int main()
{

      printf("\n First call of print()");
     print();
      printf("\n\n Second call of print()");
      print();
      printf("\n\nThird call of print()");
      print();
```

```
        return 0;
}
        void print()
{
        static int x;
        int y=0;
        printf("\n Static Integer variable,x=%d",x);
        printf("\n Integer variable,y=%d",y);
        x++;
        y++;
}
```

Output:

First call of print()

Static integer variable, x=0

Integer variable, y=0

Second call of print()

Static integer variable, x=1

Integer variable, y=0

RECURSION

It is the process of calling a function by itself, until some specified condition is satisfied. It is used for repetitive computation (like finding factorial of a number) in which each action is stated in term of previous result

Example:

```
#include<stdio.h>
long int factorial(int n);
int main( )
{
int n;
long int m;
scanf("%d",&n);
m=factorial(n);
printf("\n factorial is : %d", m);
return 0;
}
long int factorial(int n)
{
        if (n<=1)
return(1);
else
return(n*factorial(n-1));
}
```

In the program when n is passed the function, it repeatedly executes calling the same function for n, n-1, n-2, ...1.

Write a Program to calculate GCD using recursive functions.

```
#include<stdio.h>
int GCD(int, int);
int main(){
int  num1, num2, res;
printf("\n Enter the two numbers:");
scanf("%d %d",&num1,&num2);
res=GCD(num1,num2);
printf("\n GCD of %d and %d=%d",num1,num2,res);
return 0;
}
int GCD(int x, int y)
{
int rem;
rem=x%y;
if(rem==0)
return y;
else
return GCD(y,rem);
}
```

```
Output:


Enter two numbers :10 4
GCD of 10 and 4=2
```

Types of Recursion

Recursion is a technique that breaks a problem into one or more sub-problems that are similar to the original problem. Any recursive function can be characterized based on:

- Whether the function calls itself directly or indirectly(direct or indirect recursion).
- Whether any operation is pending at each recursive call (tail-recursive or not) , and
- the structure of the calling pattern (linear or tree-recursive).

Direct Recursion

A function is said to be directly recursive if it explicitly calls itself. For example .

```
int fun(int n)
{  if(n==0)
 return 0;
else
return (fun(n-1));
}
```

Here fun() calls itself for all positive values of n, so it is said to be a directly recursive function.

Indirect Recursion

A function is said to be indirectly recursive if it contains a call to another function which

ultimately calls it. These two functions are indirectly recursive as they both call each other.

```
int fun1(int n)
{ if(n==0)
   return n;
else
return fun2(n);
}
int fun2(int x)
{
  return fun2(x-1);
}
```

Tail Recursion

A recursive function is said to be tail recursive if no operations are pending to be performed when the recursive function returns to its caller.When the called function returns, the returned value is immediately returned from the calling function. Tail recursive functions are highly desirable because they are much more efficient to use as the amount of information that has to be stored on the system stack is independent of the number of recursive calls.

For example , the factorial function that we have written is a non-tail recursive function, because there is a pending operation of multiplication to be performed on return from each recursive call.

```
int Fact(int n)
 {
        if(n==1)
        return 1;
        else
```

```
return (n*Fact(n-1));
}
```

Whenever there is a pending operation to be performed, the function becomes non-tail recursive. In such a non-tail recursive function, information about each pending operation must be stored, so the amount of information directly depends on the number calls.

However, the same factorial function can be written a tail-recursive manner as follows.

```
int Fcat(n)
{
return Fact1(n,1);
}
int Fact1(int n, int res)
{
   if(n==1)
  return(n==1)
else
return Fact1(n-1, n*res);
}
```

Converting Recursive Functions to Tail Recursive

A non-tail recursive function can be converted into a tail-recursive function by using an auxiliary parameter as we did in case of the Factorial function. The auxiliary parameter is used to form the result. When we use such a parameter, the pending operation is incorporated into the auxiliary parameter so that the recursive call no longer has a pending operation. We generally use an auxiliary function while using the auxiliary parameter. This is done to keep the syntax clean and hide the fact that auxiliary parameters are needed.

UNIT-4

POINTERS

A pointer is a variable that represents the location or address of a variable or array element.

Uses

1. They are used to pass information back and forth between a function and calling point.
2. They provide a way to return multiple date items.
3. They provide alternate way to access individual array elements.

When we declare a variable say x, the computer reserves a memory cell with name x. the data stored in the variable is got through the name x. another way to access data is through the address or location of the variable. This address of x is determined by the expression &x, where & is a unary operator (called address operator). Assign this expression &x to another variable px(i.e. px=&x).this new variable px is called a pointer to x (since it points to the location of x. the data stored in x is accessed by the expression *px where * is a unary operator called the indirection operator.)

Ex: if x=3 and px=&x then *px=3

Declaration and Initialisation

Data type *pointer variable

int *p declares the variable p as pointer variable pointing to a an integer type data. It is made point to a variable q by p= &q. In p the address of the variable q is stored.

The value stored in q is got by *p.

If y is a pointer variable to x which is of type int, we declare y as int *y ;

Ex: float a;

float *b; b=&a;

Note : here in 'b' address of 'a' is stored and in '*b' the value of a is stored.

Passing pointers to a function

Pointers are also passed to function like variables and arrays are done. Pointers are normally used for passing arguments by reference (unlike in the case if variable and arrays, they are passed by values). When data are passed by values the alteration made to the data item with in the function are not carried over to the calling point; however when data are passed by reference the case is otherwise. Here the address of data item is passed and hence whatever changes occur to this with in the function, it will be retained through out the execution of the program. So generally pointers are used in the place of global variables.

Ex:)

```
#include<stdio.h>
void f(int*px,int *py)
 int main()
{
int x = 1;
```

```
int y=2;
f(&X,&y);
printf("\n %d%d", x,y);
return 0;
}
void f(int *px, int *py);
*px=*px+1;
*py=*py+2;
return;
}
```

Note:

1. here the values of x and y are increased by 1 and 2 respectively.
2. arithmetic operations *, +, -, / etc can be applied to operator variable also.

Pointer and one dimensional arrays

An array name is really a pointer to the first element in the array i.e. if x is a one dimensional array, the name x is &x[0] and &x[i] are x + i for i= 1,2,……. So to read array of numbers we can also use the following statements

```
int x[100],n;
for (i=1 ; i<=n;            ++i)
scanf ("%d", x + i )
```

(in the place of scanf (“%d”,&x[i]))

Note : the values stored in the array are got by * (x + i) in the place x[i].

Dynamic memory allocation

Usually when we use an array in C program, its dimension should be more than enough or may not be sufficient. To avoid this drawback we allocate the proper (sufficient) dimensions of an array during the run time of the program with the help of the library functions called memory management functions like ‘malloc’, ‘calloc’, ‘realloc’ etc. The process of allocating memory at run time is known as dynamic memory allocation.

Ex:

to assign sufficient memory for x we use the following statement

x= (int *) malloc (n* sizeof (int)) , for in the place of initial declaration int x[n]

Similarly in the place of float y [100] we use

y = (float *) malloc (m* sizeof (float));

Example to read n numbers and find their sum

```
int main()
{
int *x, n, i, sum=0;
printf("\n Enter number of numbers");
scanf("%d", &n);
```

```
x=(int *)malloc(n * sizeof(int));
for(i=1;i<=n,++i)
{
scanf("%d", x+i); sum +=
                    *(x+i);
}
printf("\nThe sum is %d ", sum);
return 0;
}
```

Passing function to other function

A pointer to a function can be passed to another pointer as an assignment. Here it allows one function to be transferred as if the function were a variable.

STRUCTURE

We know an array is used to store a collection of data of the same type. But if we want to deal with a collection of data of various type such as integer, string, float etc we use structures in C language. It is a method of packing data of different types. It is a convenient tool for handling logically related data items of bio-data people comprising of name, place, date etc. , salary details of staff comprising of name, pay da, hra etc.

Defining a structure.

In general it is defined with the syntax name **struct** as follows

Struct structure_name

{

Data type variable1; Data type variable2;

…

}

For example

```
1    struct account
     {
     int accountno char
     name[50]; float balance;
     }customer[20]
```

Note :1here accountno, name and balance are called members of the tructure

2

```
struct date
{
int month; int day; int
year;
}dateofbirth;
```

In these examples customer is a structure array of type account and dateofbirth is a structural type of date.

Within a structure members can be structures. In the following example of biodata structure date which is a structure is a member.

For example

```
struct date
{
int day; int month; int
year;
}
struct          biodata
{
Name char[30]; int age ;
Date birthdate;
}staff[30];
```

Here staff is an array of structure of type biodata

Note: we can declare other variables also of biodata type structure as follows.

Struct biodata customer[20]; , Struct biodata student; etc

Processing a structure

The members of a structure are themselves not variable. They should be linked to the structure variable to make them meaningful members. The linking is done by period (.)

If staff[] is structure array then the details of first staff say staff[1] is got by staff[1].name, staff[1].age, staff[1].birthdate.day, staff[1].birthdate.month, staff[1].birthdate.year . we can assign name, age and birthdate of staff[1] by

Staff[1].name=”Jayachandran”

staff[1].age=26

staff[1].birthdate.day=11

staff[1].birthdate.month=6

staff[1].birthdate.year=1980

If ‘employee’ is a structure variable of type biodata as mentioned above then the details of ‘employee’ is got by declaring ‘employee as biodata type by the statement

biodata employee;

The details of employee are got by employee.name, employee.age, employee.birthdate.year etc.

Structure initialisation

Like any other variable or array a structure variable can also be initalised by using syntax static

```
struct record
{
```

```
char name[30]; int
age;
int weight;
}
```

Static struct record student1={“rajan”, 18, 62}

Here student1 is of record structure and the name,age and weight are initialised as “rajan”, 18 and 62 respectively.

Write a c program to read biodata of students showing name, place, pin, phone and grade

```
#include<stdio.h> int
main()
{
struct biodata
{
char     name[30];     char
Place[40] int pin;
long Int phone; char grade;
};
struct biodata student[50];
int n;
prinf(“\n no of students”);
scanf(“%d”,n);
for(i=1;i<=n;++i)
```

```
{
scanf("%s",student[i].name);
scanf("%s",student[i].place);
scanf("%d",student[i].pin); scanf("%ld",student[i].phone);
scanf("%c",student[i].grade);
}return 0;
}
```

User Defined Datatype

This is to define new data type equivalent to existing data types. Once defined a user-defined data type then new variables can be declared in terms of this new data type. For defining new data type we use the syntax typedef as follows.

typedef type new-type.

Here type refers to existing data type

For example Ex1:

```
typedef int integer;
```

Now integer is a new type and using this type variable, array etc can be defined as

```
integer x;
integer mark[100];
```

Ex2:

```
typedef struct
{
```

```
int accno; char name[30];
float balance;
}record;
```

Now record is structure type using this type declare customer, staff as record type

Record customer;

Record staff[100];

Passing structures to functions

Mainly there are two methods by which structures can be transferred to and from a function.

1 Transfer structure members individually

2 Passing structures as pointers (ie by reference)

Example 1

```
#include<stdio.h>
typedef struct
{
int accno; char name[30];
float balance;
     }record;
int main()
{
…..
Record customer;
```

```
 . . . .
     Customer.balance=adjust(customer.name,customer.accn
o,balance)
. . . . .
return 0;
}

float adjust(char name[], int accnumber, float bal)
{
float x;
. . . . . X=
. . . . .
return(x);
}
```

Example 2

```
#include<stdio.h>
typedef struct
{
int accno; char name[30];
float balance;
}record;

int main()
{
```

```
record customer;


void adjust(record *cust)

. . . . . .

adjust(&customer);

printf("\n %s\t%f",coustomer.name,customer.balance)

return 0;

}


void adjust(record *cust)

{

float x;

. . . .

cust->balance=...

. . . .

return;

}
```

In the first example structure members are passed individually where as in the second case customer is passed entirely as a pointer named cust. The values of structure members are accessed by using -> symbol like cust->.name, cust->balance etc.

Union

Union is a concept similar to a structure with the major difference in terms of storage. In the case of structures each member has its own storage location, but a union may contain many members of different types but can handle only one at a time. Union is also defined as a structure is done but using the syntax union.

union var

{

```
int m; char c;

float a;
```

}

union var x;

Now x is a union containing three members m,c,a. But only one value can be stored either in x.m, x.c or x.a

FILE HANDLING

Data Files are to store data on the memory device permanently and to access whenever is required.

There are two types of data files

1. Stream Oriented data files
2. System Oriented data files

Stream oriented data files are either text files or unformatted files. System oriented data files are more closely related to computer's operating system and more complicated to work with. In this session we go through stream oriented data files.

Opening and Closing File

The first step is to create a buffer area where information is stored temporarily before passing to computer memory. It is done by writing

File *fp;

Here fp is the pointer variable to indicate the beginning of the buffer area and called stream pointer .

The next step is to open a data file specifying the type i.e. read only file, write only file, read /write file. This is done by using the library function fopen.

The syntax is

```
fp=fopen(filename,filetype)
```

the filetype can be

1 'r' (to open an existing file for reading only)

2 'w' (to open a new file for writing only. If file with filename exists, it will be destroyed

and a new file is created in its place)

3 'a' (to open an existing file for appending. If the file name does not exist a new file with that file name will be created)

4 'r+' (to open an existing file for both reading and writing)

5 'w+' (to open a new file for reading and writing. If the file exists with that name, it will be destroyed and a new one will be created with that name)

6 'a+' (to open an existing file for reading and writing. If the file does not exist a new file will be created).

For writing formatted data to a file we use the function fprintf. The syntax is

fprintf(fp,"conversion string", value);

For example to write the name "rajan" to the file named 'st.dat'

```
File *fp; fp=fopen("st.dat",'w');
fprintf(fp,"%[^\n]","rajan");
```

The last step is to close the file after the desired manipulation. This is done by the library function fclose. The syntax is `fclose(fp);`

Example:1

To create a file of biodata of students with name 'st.dat'.

```
#include<stdio.h>
#include<string.h>
typedef struct{
                        int day;
             int month;
            int year;
        }date;
Typedef struct{
```

```
      char name(30);
 char place(30); int age;
date birthdate;
}biodata;


int main(){
File *fp; biodata student;
fp=fopen("st.dat",'w'); printf("Input data");
scanf("%[^\n]",student.name);
scanf("%[^\n]",student.place);
scanf("%d",&student.age);
scanf("%d",&student.birthdate.day);
scanf("%d",&student.birthdate.month): scanf("%d",&student.birthdate.year);
fprintf(fp,"%s%s%d%d%d%d",student.name,student.place,student.
age,student.birthdate.day, student.birthdate.month, student.birthdate.year)
fclose(fp);
return 0;
}
```

Example 2: To write a set of numbers to a file

```
      #include<stdio.h>
      int main(){
                  file *fp;
       int n;
       float x ;
       fp=fopen("num.dat",'w');
              printf("Input the number of numbers");
              scanf("%d",&n);
              for(i=1;i<=n;++i)
{ scanf("%d",&x);
fprintf(fp,"%f\n",x);
}  fclose(fp);
```

```
return 0;
    }
```

Processing Formatted Data File

To read formatted data from a file we have to follow all the various steps that discussed above. The file should be opened with read mode. To open the existing file 'st.dat' write the following syntax

file *fp; fp=fopen("st.dat",
'r+');

For reading formatted data from a file we use the function fscanf

```
typedef struct
{
char name(30); char place(30);
int age;
date birthdate;
}biodata;

typedef struct{
                  char name[30];
                  char place[30];
                  int age;
                  int birthday;
}biodata;

int main(){
            File *fp; biodata
```

```
student;
fp=fopen("st.dat",'r+'); fscanf(fp,"%s",student.name);
printf("%s",student.name); fclose(fp);
return 0;
}
```

Processing Unformatted Data files

For reading and writing unformatted data to files we use the library functions fread and fwrite in the place of fscanf and fprintf.

The syntax for writing data to file 'st.dat' with stream pointer fp is

fwrite(&student, sizeof(record),1,fp);

Here student is the structure of type biodata

Example: To write biodata to a file

```
#include<stdio.h>
typedef struct{
                int date;
                int month;
                int year;
        }date;

typedef struct
{
char name(30); char place(30);
int age;
date birthdate;
```

```
}biodata;

int main(){

        File *fp;
fp=fopen("st.dat",'a+') biodata
student; printf("Input data");
scanf("%[^\n]",student.name);
scanf("%[^\n]",student.place);
scanf("%d",&student.age);
scanf("%d",&student.birthdate.day);
scanf("%d",&student.birthdate.month):
scanf("%d",&student.birthdate.year);
fwrite(&student,sizeof(record),1,fp); fclose(fp);
return 0;
}
```

Example 2: To read the biodata from a file

```
        tpedef struct
{
int day; int month; int
year;
}date;
typedef struct
{
char name(30); char place(30);
int age;
date birthdate;
}biodata;

int main()
```

```
{
   File *fp; fp=fopen("st.dat",'a+')
biodata student;
fread(&student,sizeof(record),1,fp);
printf("%s\n",student.name); printf("%s\n]",student.place);
printf("%d\n",&student.age);
printf("%d\n",&student.birthdate.day);
printf("%d\n",&student.birthdate.month):
printf("%d\n",&student.birthdate.year); fclose(fp);
return 0;
}
```

UNIT-5

BASICS ALGORITHMS

Algorithms are mainly used to achieve software reuse. Once we have an idea or a blueprint of a solution, we can implement it in any high-level language like C, C++ or Java.

An algorithm is basically a set of instructions that solve a problem. It is not uncommon to have multiple algorithms to tackle the same problem, but the choice of a particular algorithm must depend on the time and space complexity of the algorithm.

must depend on the time and space complexity of the algorithm.

Control Structure used in Algorithms

An algorithm has a finite number of steps . Some steps may have involve decision-making and repetition. Broadly speaking an algorithm may employ one of the following control structures: (a) sequence (b)decision (c)repetition.

Sequence:

By sequence we mean that each steps of an algorithm is executed in a specified order. Let us write an algorithm to add two numbers . This algorithm performs the steps in a purely sequential order which is mentioned below.

Algorithm to add two numbers:

Step 1: Input first number as A

Step 2: Input second number as B

Step 3: SET SUM=A+B

Step 4:Print SUM

Step 5: End

Decision:

Decision statements are used when the execution of a process on the outcome of the

condition. For example if x=y, then print EQUAL. So the general form of IF construct can be given as follows.

IF condition Then process.

A condition in this context is any statement that may evaluate to either a true value or a false value. In the above example, a variable x can be either equal to yor not equal to y. However, it cannot be both true or false. If the condition is true , then the process is executed. A decision statement can also be stated in the following order.

IF condition

Then process1

ELSE process2

This form is popularly known as the IF-ELSE constructed. Here, if the condition is true , then process1 is executed , else process2 is executed

Algorithm to test for equality of two numbers.

Step 1: Input first number as A

Step 2: Input second number as B

Step 3: IF A=B

PRINT "EQUAL"

ELSE

PRINT "NOT EQUAL"

[END OF IF]

Step 4: END

Repetition

Repetition, which involves execution one or more steps for a number of times can be implemented using constructs such as while , do-while and for loops. These loops execute one or more steps until some condition is true.

Algorithm to print the first 10 natural of two numbers.

Step 1: [INITIALIZE] SET I=1, N=10

Step 2: Repeat Steps 3 and 4 while I<=N

Step 3: PRINT I

Step 4: SET I=I+1

[END OF LOOP]

Step 5: END

Time and Space Complexity

Analysis an algorithm means determining the amount of resources (such as time and memory) needed to execute it. The time complexity of an algorithm is basically the running time of a program as a function of the input size. Similarly , the space complexity of an algorithm is the amount of computer memory that is required during the program executing as a function of the input size.

In other words, the number machine instructions which a program executes is called its time complexity. This number is primarily dependent on the size of the program's input and the algorithm used.

Fixed Part: It varies from Problem to Problem. It includes the space needed for storing instructions, constants, variables and structured variables like arrays and structures).

Variable Part: It varies from program to program.It includes the space needed for recursion stack, and for structured variables that are allocated space dynamically during the runtime of a program.

Worst-case, Average-case, Best-case and Amortized Time Complexity

Worst-case running time: This denotes the behaviour of an algorithm with restpect to the worst -possible case of the input instance. The worst-case running time of an algorithm is an upper bound on the running time of an input.

Average-case running time

It is the estimate of the running time of an "average" input. It specifies the expected behavior of the algorithm when the input is randomly drawn from a given distribution. Average-case running time assumes that all inputs of a given size are equally likely.

Best-case running time

The term "Best-case performance" is used to analyze an algorithm under optimal conditions. For example, the best case for a simple linear search on an array occurs when the desired element is the first in the list.

Amortized Running time

Amortized running time refers to the time required to perform a sequence of operations averaged over all the operations performed. Amortized analysis guarantees the average performance of each operation in the worst case.

Time and Space Complexity

The time and space complexity can be expressed using a function f(n) where n it the input size for a given instance of the problem being solved. Expressing the complexity is required when We want to predict the rate of growth of complexity as the input size of the problem increases.

There are multiple algorithms that find a solution to a given problem and we need to find the algorithm that is most efficient.

The most widely used notation to express this function f(n) is the Big O notation.it provides the upper bound for the complexity.

Algorithm Efficiency:

If a function is linear (without any loop or recursion), the efficiency of an algorithm can be given as the numbers of instructions it contains. However, if an algorithm contains loops, then the efficiency of that algorithm may very depending on the number of loops and the running time of each loop in the algorithm.

Big O Notation

It has been seen that the number of statements executed in the program for n elements of the data is a function of the number of elements , expressed as f(n). Even if the expression derived for a function is complex, a dominant factor in the expression is sufficient to determine the order of the magnitude of the result and hence the efficiency of the algorithm. This factor is the Big O, and expressed as O(n). The Big O notation, where O stands for 'order of ' is concerned with what happens for very large value of n. For example, if a sorting algorithm performs n^2 operations to sort just n elements , then that algorithm would be described as an $O(n)^2$algorithm.

If (n) and g(n) are the functions defined on a positive integer n, then

f(n)=Og(n).

That is f on n is Big-O of g of n if and only if positive constants c and n exists such that f(n)<=cg(n). It means that for large amounts of data, f(n) will grow no more than a constant factor than g(n). Hence , g provides an upper bound.

Example:

```
For(i=0;i<100;i++)
        Statement block;
```

Here 100 is the loop factor. We have already said that efficiency is directly proportional to the number of iterations. Hence, the general formula in the case of linear loops may be given as

f(n)=n

```
for (i=0;i<100;i=i+2)
statement block;
```

hence the number of iterations is half the number of the loop factor. So here the efficiency can be given as f(n)=n/2.

Omega Notation(Ω)

The omega notation provides a tight lower bound for f(n).This means that the function can never do better than the specified value but it may do worse.

Ω notation is simple written as f(n) ε Ω(g(n)) where n is the problem size and Ω(g(n))={h(n): ∃ positive constants c>0, n_0 such that 0<=cg(n)<=h(n), for all n>=n_0}. Hence we can say that Ω(g(n)) comprises a set of all the functions h(n) that are greater than or equal to c(g(n)) for all values of n >=n_0.

If cg(n)<=f(n), c>0, for all n>=n_0, then f(n) ε Ω(g(n)) and g(n) is an asymptotically tight lower bound for f(n).

Theta Notation(θ)

Theta notation provides an asymptotically tight bound for f(n).θ notation is simply written as f(n)ε θ(g(n)), where n is the problem size and θ(g(n) ={h(n):∃ positive constants c_1,c_2 and n_0 such that $0<=c_1g(n)<=h(n)<=c_2g(n)$, for all $n >=n_0$}.

Hence we can say that θ(g(n)) comprises a set of all the functions h(n) that are between c_1(g(n)) and c_2g(n) for all value of $n>=n_0$.

**********END**********