PROBLEM SOLVING AND COMPUTER PROGRAMMING

Module I Problem solving with digital computer

- **Steps in computer programming**
- **Features of a good program**
- **Modular programming**
- **Structured programming**
- **Object Oriented Programming**
- **Top-down and Bottom-up approaches**
- **Algorithms**
- **Flowchart**
- **Pseudo code**
- **Examples**

Module II C Fundamentals

- **Identifiers**
- **Keywords**
- **Data types**
- **Operators**
- **Expressions**
- **Data input and output statements**
- **Simple programming in C**

Module III Control statements and Functions

- **If – else statement**
- **For statement**
- **Do-while statement**
- **Switch statement**
- **Break and continue statement**
- **Nested loops**
- **Functions**
- **Parameter passing**
- **Void functions**
- **Recursion**
- **Macros**

Module IV Structured data types

- **Single dimensional arrays**
- **Multidimensional arrays**
- **Strings**
- **Structures and unions**
- **Program for bubble sort**

Module V Pointers and Files

- **Declaration**
- **Passing pointers to a function**
- **Accessing array elements using pointers**
- **Operations on pointers**
- **Opening and closing a file**
- **Creating and processing a file**
- **Command line arguments**

MODULE-I

Steps in Computer Programming

To process a particular set of data the computer must be given set of instructions called a program. These instructions are entered into the computer and then stored in a portion of computer's memory. These instructions will be executed one by one to get the required result of the program.

A computer program can be developed by using the following number of steps.

1. Problem definition : At this stage the problem to be solved or the task to be performed is defined. Inputs, outputs, processing requirements, system constraints such as execution time, accuracy etc. and error handling methods are specified.
2. Program design : At this stage the program is designed to meet the specified requirements according to its definition. The important design techniques are used here. Top down, structured programming, modular programming and flowcharting is used in this step. This choice will help in better documentation of the program.
3. Preparation of actual program : The instructions are written according to its design.
4. Testing : At this stage the program is tested to check whether it performs the required task or solves the given problem. This stage is also called validation.
5. Debugging : At this stage program errors are detected and corrected. This is also called verification. The programmer can use many methods to check the errors in the program.
6. Documentation : After the program is executed correctly the implementation of the program can be done. This indicates what functions are performed by the program and how these functions are carried out. It helps users to understand and maintain the program.
7. Maintenance : The programs are corrected and updated to meet the needs of changing conditions. It should be corrected or modified on the basis of new requirements.
8. Extension and redesign : a program can be extended to other tasks. If necessary it can be redesigned to get its improved version or to perform other tasks.

Features of a good program

1. Reliability : A program must work reliably. It should perform the task properly for which it has been developed.
2. Speed : A program must execute the specified task quickly. The time taken by a program to perform a given task should be minimum as possible.

3. Programming time and cost : The programming approach should be selected in such a way that the maximum output is obtained. Proper testing, debugging and documentation reduce overall cost of a program.
4. Ease of use/ Clarity : A program must be easily understood to others. A program with strictly defined and complicated data formats is difficult to use and expensive to debug and maintain.
5. Error tolerance : A program should respond to error quickly. Proper error messages should be displayed before closing the program execution.
6. Extensibility : A program that can be extended to tasks other than for which it has been designed and developed is definitely a better program.
7. Integrity : Calculations included in the program should be accurate. Otherwise extending a program may result in more errors.
8. Simplicity : The clarity and accuracy of a program can be enhanced by keeping things as simple as possible, consistent with the overall program objectives
9. Efficiency : A program should efficiently utilize memory
10. Modularity : Program can be broken down into a series of identifiable subtasks. It is a good practice to implement each of these subtasks as a separate program module.
11. Generality : considerable amount of generality can be added to a program with additional programming effort.

Different programming styles

1. Modular programming
2. Structured programming
3. Top down and bottom up design
4. Object oriented programming

1. Modular programming

When a program becomes very long and complex, it becomes a very difficult task for the programmer to design, test and debug such a program. Therefore a long program can be divided into smaller programs called modules. The division of a long program into smaller programs (modules) is called modular programming.

Advantages:

1. It is easier to design, test and debug a small module compared to an entire program.]
2. A module is convenient for using elsewhere
3. Changes if required can be done in the module
4. Previously written programs can be used again.

Disadvantages

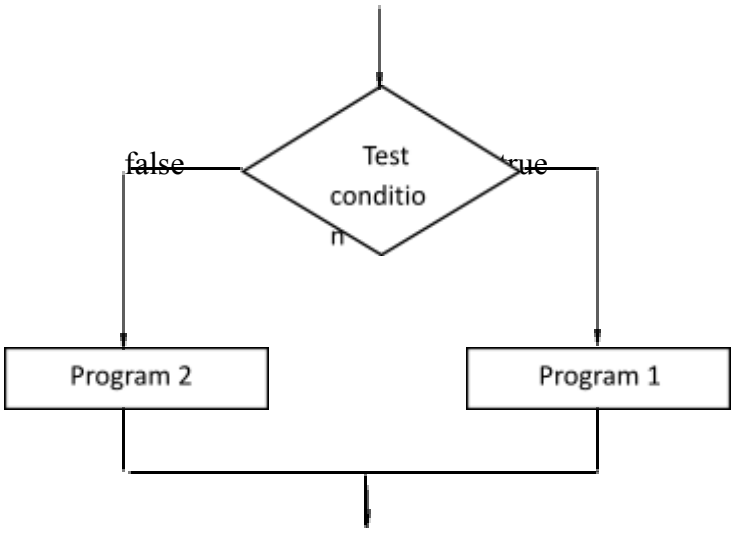
1. Since separate modules may repeat certain functions, the modular programming often need extra time and memory
2. When different persons design different modules separately combining them will be difficult.
3. While testing modular programming may require data from other modules. The development of drivers for this purpose is time consuming and extra effort is required for that.

Modules are prepared for common tasks. The programming methods used by microprocessors often use this technique. The modules of 20 to 50 lines are kept in a library and will be used later for expanding the existing program code.

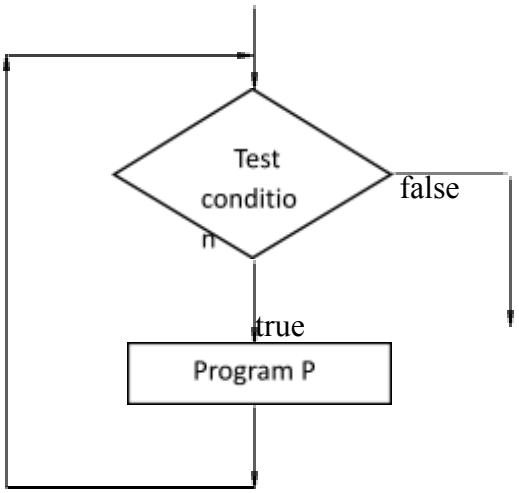
Structured programming

With the increasing capacity of memory, program also became longer. The long and complex programs may be well understood by the programmers who developed it but not by the persons who had to maintain them. The basic idea behind this technique is that any part of the program can be represented by elements from three basic logic structures. Each structure has single entry and single exit.

1. Simple sequence structure : It is a linear structure in which instructions or statements are executed consecutively in a sentence.
2. Conditional structure : In this case a condition is tested. The condition is followed by two alternative program control paths. If condition is true a section will be executed and if not another section will be executed.



3. Loop structure : in loop structure the computer checks the condition. If the condition is true, the program p is executed. It stops execution of program p when the condition is not satisfied or false.



the structured programming structure is not suitable for programs with complex data structure. The testing and debugging of a structured program is easy. This can be described with the help of flowchart or other graphic methods.

Top-down and bottom-up design

In top-down technique, the design of the system program is started at the system level. The programmer first develops the overall supervisor program which is used to outline and control subprograms. The whole system work is divided into a number of subtasks. To perform each subtask there is a subprogram. The main program is then tested to see that the logic is correct or not. The undefined subprograms are replaced by temporary programs called stubs. A stub represents a subtask. A stub may either record the entry of a subprogram or produce the result to a selected test program. Then the programmer proceeds by expanding stubs. Testing and debugging is made at each step as a stub is replaced by the working program. In this approach testing and integration is made along the way at each level. Testing is done in the actual program environment. Top-down design is compatible with modular as well as structured programming.

In bottom-up approach, inner subprograms are prepared first for specific tasks and then integrated into a complete system. This technique should be used for frequently used subprograms whose speed is critical to the speed of the whole program and whose functions are clearly understood initially. The top-down technique is better if the precise nature of the subprogram cannot be determined. In bottom-up the entire integration is to be done at the end.

Object-oriented programming(OOP)

OOP is a programming technique where data is given more importance than the function using it. It helps to model real world problems. It ties data more closely to the subprograms / functions that operate on it and protects it from accidental modifications from outside functions. This technique decompose a problem into a number of entities called objects and then builds data and functions around these entities. The data of an object can be accessed only by the functions associated with that object. Functions of one object can access the functions of other objects.

Some important features of OOP are:

1. Programs are based on objects
2. Data is hidden and cannot be accessed by external functions
3. New data and functions can be added whenever necessary
4. Follows bottom-up approach in program design.

Important OOP concepts are :

1. Objects / classes
2. Encapsulation
3. Inheritance
4. Polymorphism
5. Message passing

Problem solving methodology

In order to carry out a given task using a computer, an effective computer program must be generated. It is necessary that every instruction in a program must be in the proper sequence. However the instruction sequence of a computer program may be very complex. Hence in order to ensure that instructions in the program are appropriate for the problem and are in the correct sequence, programs must be planned before they are written.

In order to carry out the given task (problem to be solved) using a computer the following steps are followed.

1. The given task is analyzed.
2. Based on the analysis an algorithm is formulated.
3. Draw the pictorial representation of the algorithm.
4. Instruction sequence is written in any computer language.
5. The computer program is fed to the computer.

Now the computer interprets the program, carries out the instructions given in the program and computes the result. Thus one can get the answers from the computer for the given problem.

ALGORITHM

It is a set of instructions, arranged in a specific order to solve a particular problem in a finite number of steps.

An algorithm is a scientific procedure to solve a problem where solution for that particular problem is guaranteed.

CHARACTERISTICS OF A GOOD ALGORITHM

1. Each and every instruction should be precise and unambiguous
2. Each instruction can be performed in finite time
3. An algorithm should be terminated within finite number of steps
4. After the algorithm terminates, the desired results must be obtained

How to write an algorithm

Algorithms are written in simple English. It is done manually on a paper. Understanding the problem is the first and important step in any problem solving exercise. To understand any problem, we should define the problem by noting down its objective, the available data and the process to be adopted.

Example:

If we want to find total marks by adding the internal and external marks

The logic is

Objective : to find the total marks

Input data : internal marks and external marks

Process : add internal marks with external marks

Output : total marks

It is easy to write an algorithm if we could define the problem well. The algorithm for the above problem can be written as

Step1 : Start

Step2 : Input internal and external marks

Step3 : Add internal mark with external mark to get total marks

Step4 : Display total marks

Step5 : Stop

A problem can have more than one correct algorithm. A good computer algorithm should make the most efficient use of computer time as well as memory.

FLOW CHART

Flow chart is a pictorial representation of an algorithm. It uses boxes of different shapes to denote different types of instructions. The actual instructions are written within these boxes using clear and concise statements. These boxes are connected by solid lines having arrow marks to indicate the flow of operation.

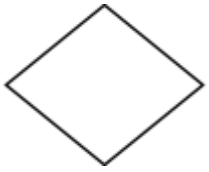
Flow chart symbols

Start / stop

Input / output

Processing

Flow lines



Decision

Connector

Sample flow chart

To find the total marks by adding the internal and external marks

The logic is

Objective : to find the total marks

Input data : internal marks and external marks

Process : add internal marks with external marks

Output : total marks

Algorithm

Step1 : Start

Step2 : Input internal as a and external marks as b.

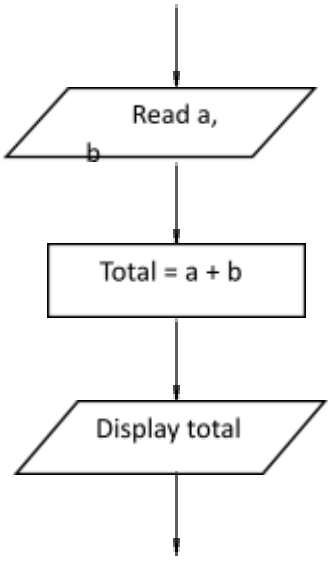
Step3 : Add a with b to get total

Step4 : Display total

Step5 : Stop

Flow chart

Start



Stop

Example:

Write an algorithm and draw the flow chart for printing n natural numbers

Algorithm

Step1 : Start

Step2 : Input the value of n.

Step3 : Assign x as zero

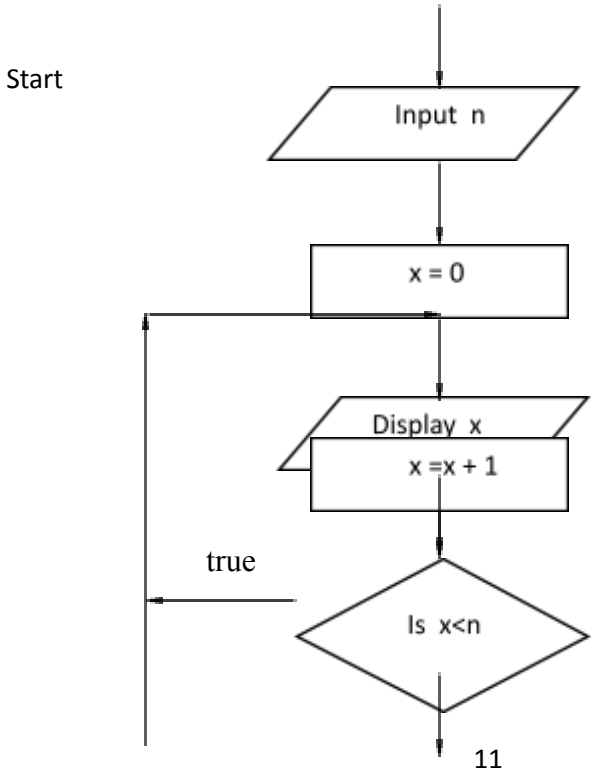
Step4 : Display x

Step5 : Increment x by 1

Step6 : Check x < n, if true go to step 4, else go to step 7

Step7 : Stop

Flow chart



false

Stop

MODULE-II

ABOUT C PROGRAMMING LANGUAGE

History

C was developed at Bell Laboratories in 1972 by Dennis Ritchie. Many of its principles and ideas were taken from the earlier language B and B's earlier ancestors BCPL and CPL. CPL (Combined Programming Language) was developed with the purpose of creating a language that was capable of both high level, machine independent programming and would still allow the programmer to control the behavior of individual bits of information. The one major drawback of CPL was that it was too large for use in many applications. In 1967, BCPL (Basic CPL) was created as a scaled down version of CPL while still retaining its basic features. In 1970, Ken Thompson, while working at Bell Labs, took this process further by developing the B language. B was a scaled down version of BCPL written specifically for use in systems programming. Finally in 1972, a co-worker of Ken Thompson, Dennis Ritchie, returned some of the generality found in BCPL to the B language in the process of developing the language we now know as C.

C's power and flexibility soon became apparent. Because of this, the Unix operating system which was originally written in assembly language, was almost immediately re-written in C (only the assembly language code needed to "bootstrap" the C code was kept). During the rest of the 1970's, C spread throughout many colleges and universities because of it's close ties to Unix and the availability of C compilers. Soon, many different organizations began using their own versions of C causing compatibility problems. In response to this in 1983, the American National Standards Institute (ANSI) formed a committee to establish a standard definition of C which became known as ANSI Standard C. The standardization process took six years. The

ANSI C standard was finally adopted in December 1989, with the first copies becoming available in early 1990. The standard was also adopted by ISO (International Standards Organization), and the resulting standard was typically referred to as ANSI/ISO Standard C, or simply ANSI/ISO C. Today C is in widespread use with a rich standard library of functions.

Significant Language Features

C is a powerful, flexible language that provides fast program execution and imposes few constraints on the programmer. It allows low level access to information and commands while still retaining the portability and syntax of a high level language. These qualities make it a useful language for both systems programming and general purpose programs.

C's power and fast program execution come from it's ability to access low level commands, similar to assembly language, but with high level syntax. It's flexibility comes from the many ways the programmer has to accomplish the same tasks. C includes bitwise operators along with powerful pointer manipulation capabilities. C imposes few constraints on the programmer. The main area this shows up is in C's lack of type checking. This can be a powerful advantage to an experienced programmer but a dangerous disadvantage to a novice.

Another strong point of C is it's use of modularity. Sections of code can be stored in libraries for re-use in future programs. This concept of modularity also helps with C's portability and execution speed. The core C language leaves out many features included in the core of other languages. These functions are instead stored in the C Standard Library where they can be called on when needed.. An example of this concept would be C's lack of built in I/O capabilities. I/O functions tend to slow down program execution and also be machine independent when running optimally. For these reasons, they are stored in a library separately from the C language and only included when necessary.

C is often called a *middle-level computer language.* This does not mean that C is less powerful, harder to use, or less developed than a high-level language such as Pascal;

nor does it imply that C is similar to, or presents the problems associated with, assembly language. The definition of C as a middle-level language means that it combines elements of high-level languages with the functionalism of assembly language. As a middle-level language, C allows the manipulation of bits, bytes, and addresses—the basic elements with which the computer functions. Despite this fact, C code is surprisingly portable. *Portability* means that it is possible to adapt software written for one type of computer to another. For example, if a program written for one type of CPU can be moved easily to another, that program is portable. All high-level programming languages support the concept of data types. A *data type* defines a set of values that a variable can store along with a set of operations that can be performed on that variable. Common data types are integer, character, and real. Although C has several basic built-in data types, it is not a strongly typed language like Pascal or Ada. In fact, C will allow almost all type conversions. For example, character and integer types may be freely intermixed in most expressions. Traditionally C performs no run-time error checking such as array-boundary checking or argument-type compatibility checking. These checks are the responsibility of the programmer. A special feature of C is that it allows the direct manipulation of bits, bytes, words, and pointers. This makes it well suited for system-level programming, where these operations are common. Another important aspect of C is that it has only 32 keywords (5 more were added by C99, but these are not supported by C++), which are the commands that make up the C language. This is far fewer than most other languages.

DATA TYPES IN C

C language is rich in data types. The fundamental data types which can be used in C are integer data types, character data types and floating point data types. Integer data type is used to represent an integer-valued number. Character data type represents one single alphabet or a single-digit integer. Each character type has an equivalent integer representation. Integer and character data types can be augmented by the use of the data type qualifiers, short, long, signed and unsigned. The range of values which belong to each category varies with respect to the qualifiers and so, the memory requirement.

Floating point data types are used to represent real numbers. Basic floating point data type offers six digits of precision. Double precision data type can be used to achieve a precision of 14 digits. To extend the precision further more, we may use the extended double precision floating point data type. Each of the above discussed data types and their corresponding keywords are given in the table below.

DATA TYPES	KEYWORD
Character	char
Unsigned Character	unsigned char
Signed Character	signed char
Signed Integer	signed int (or int)
Signed Short Integer	signed short int (or short int or short)
Signed Long Integer	signed long int (or long int or long)
Unsigned Integer	unsigned int (or unsigned)
Unsigned Short Integer	unsigned short int (or unsigned short)
Unsigned Long Integer	unsigned long int (or unsigned long)
Floating Point	float
Double Precision Floating Point	double
Extended Double Precision Floating Point	long double

A schematic representation of the various data types in C is given below

In the table below, the memory size required by each data type in bits and the range of values they can posses are listed.

DATA TYPE	SIZE in BITS	RANGE
char or signed char	8	-128 to 127.
unsigned char	8	0 to 255.
int or signed int	16	-32768 to 32767.
unsigned int	16	0 to 65535.
short int or signed short int	8	-128 to 127.
unsigned short int	8	0 to 255.
long int or signed long int	32	-2,147,483,648 to 2,147,483,647.
unsigned long int	32	0 to 4,294,967,295.
float	32	3.4e-38 to 3.4e+38.
double	64	1.7e-308 to 1.7e+308.
long double	80	3.4e-4932 to 3.4e+4932.

The syntax of the declaration of variables using data type is,

data-type vname-1, vname-2,…..vname-n;

where vname-1, vname-2, vname-n are variable names. Some examples for the variable declaration are given below.

int number;

char a;

long double big-number;

EXPRESSIONS

Expressions in C are classified according to the operators used in them. The classifications are,

- Relational expression.
- Logical expression.
- Arithmetic expression.

1. Relational expression.

An expression containing a relational operator is termed as relational expression. The list of all relational operators is given below

OPERATOR	MEANING
<	Is less than.
<=	Is less than or equal to.
>	Is greater than.
>=	Is greater than or equal to.
==	Is equal to.
!=	Is not equal to.

Format of a relation expression:

(a-expression-1) relational-operator (a-expression-2)

Where a-expression-1 and a-expression-2 are arithmetic expressions.
A relational expression always will have a value one or zero. The value of the relational expression will be one if the specified relation is true and zero if the relation is false. Relational operators are used in decision statements such as 'if', and 'while' to decide the course of action of a running program. When arithmetic expressions are used on either side of a relational operator, the arithmetic expressions will be evaluated first and then the results are compared.

2. Logical expression.

Logical expressions are the expressions where logical operators are used to combine more than one relational expression. Logical expressions are also called compound relational expressions. The logical operators used in C are && (logical AND), || (logical OR), and ! (logical NOT). Logical expressions also yields a value of one or zero, and used in decision statements.
Format of a logical expression:

(rel-expression-1) logical operator (rel-expression-2)

Where rel-expression-1 and rel-expression-2 are relational expressions. The values of the above statement when different logical operators are used is given below

Operator	Value of rel-expression-1	Value of rel-expression-2	Value of the statement
&&	0	0	0
	0	1	0
	1	0	0
	1	1	1
|	0	0	0
	0	1	1
	1	0	1
	1	1	1

ARITHMETIC EXPRESSIONS

An expression is a combination of variables constants and operators written according to the syntax of C language. In C every expression evaluates to a value i.e., every expression results in some value of a certain type that can be assigned to a variable. Some examples of C expressions are shown in the table given below.

Algebraic Expression	C Expression
a x b – c	a * b – c
(m + n) (x + y)	(m + n) * (x + y)
(a x b / c)	a * b / c
$3x^2+2x + 1$	3*x*x+2*x+1
(x / y) + c	x / y + c

Evaluation of Expressions

Expressions are evaluated using an assignment statement of the form

Variable = expression;

Variable is any valid C variable name. When the statement is encountered, the expression is evaluated first and then replaces the previous value of the variable on the left hand side. All variables used in the expression must be assigned values before evaluation is attempted. Example of evaluation statements are

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

Precedence in Arithmetic Operators

An arithmetic expression without parenthesis will be evaluated from left to right using the rules of precedence of operators. There are two distinct priority levels of arithmetic operators in C.

High priority * / %

Low priority + -

Rules for evaluation of expression

- First parenthesized sub expression left to right are evaluated.
- If parentheses are nested, the evaluation begins with the innermost sub expression.
- The precedence rule is applied in determining the order of application of operators in evaluating sub expressions.
- The associability rule is applied when two or more operators of the same precedence level appear in the sub expression.
- Arithmetic expressions are evaluated from left to right using the rules of precedence.
- When Parenthesis are used, the expressions within parenthesis assume highest priority.

Type conversions in expressions

Implicit type conversion:
C permits mixing of constants and variables of different types in an expression. C automatically converts any intermediate values to the proper type so that the expression can be evaluated without loosing any significance. This automatic type conversion is known as implicit type conversion.
During evaluation it adheres to very strict rules and type conversion. If the operands are of different types the lower type is automatically converted to the higher type before the operation proceeds. The result is of higher type. The following rules apply during evaluating expressions:
All short and char are automatically converted to int then

1. If one operand is long double, the other will be converted to long double and result will be long double.
2. If one operand is double, the other will be converted to double and result will be
   double.

3. If one operand is float, the other will be converted to float and result will be float.

4. If one of the operand is unsigned long int, the other will be converted into unsigned long
int and result will be unsigned long int.

5. If one operand is long int and other is unsigned int then

a. If unsigned int can be converted to long int, then unsigned int operand will be
converted as such and the result will be long int.

b. Else both operands will be converted to unsigned long int and the result will be
unsigned long int.

6. If one of the operand is long int, the other will be converted to long int and the result
will be long int.

7.If one operand is unsigned int the other will be converted to unsigned int and the
result will be unsigned int.

Explicit Conversion:

Many times there may arise a situation where we want to force a type conversion in a way that is different from automatic conversion.

Consider for example the calculation of number of female and male students in a class.

$$\textbf{Ratio} = \frac{\textbf{Female students}}{\textbf{Male students}}$$

Since if female students and male students are declared as integers, the decimal part will be rounded off and its ratio will represent a wrong figure. This problem can be

solved by converting locally one of the variables to the floating point as shown below.

Ratio = (float) female students / male students;

The operator float converts the female students to floating point for the purpose of evaluation of the expression. Then using the rule of automatic conversion, the division is performed by floating point mode, thus retaining the fractional part of the result. The process of such a local conversion is known as explicit conversion or coasting a value. The general form is,

(type_name) expression

Operator precedence and associativity.

Each operator in C has a precedence associated with it. The precedence is used to determine how an expression involving more than one operator is evaluated. There are distinct levels of precedence and an operator may belong to one of these levels. The operators of higher precedence are evaluated first.

The operators of same precedence are evaluated from right to left or from left to right depending on the level. This is known as associativity property of an operator.

The table given below gives the precedence of each operator.

Order	Category	Operator	Operation	Associativity
1	Highest precedence	( ) [ ] →	Function call	L → R Left to Right
2	*Unary*	! ~ + - ++ - - & *	Logical negation (NOT) Bitwise 1's complement Unary plus Unary minus Pre or post increment Pre or post decrement Address Indirection	R → L Right -> Left

		Size of	Size of operant in bytes	
3	Member Access	.* →*	Dereference Dereference	L → R
4	Multiplication	* / %	Multiply Divide Modulus	L → R
5	Additive	+ -	Binary Plus Binary Minus	L → R
6	Shift	<< >>	Shift Left Shift Right	L → R
7	Relational	< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	L → R

8	Equality	== !=	Equal to Not Equal to	L → R
9	Bitwise AND	&	Bitwise AND	L → R
10	Bitwise XOR	^	Bitwise XOR	L → R
11	Bitwise OR	|	Bitwise OR	L → R
12	Logical AND	&&	Logical AND	L → R
14	Conditional	? :	Ternary Operator	R → L
15	Assignment	= *= %= /= += -= &= ^= |= <<= >>=	Assignment Assign product Assign reminder Assign quotient Assign sum Assign difference Assign bitwise AND Assign bitwise XOR Assign bitwise OR Assign left shift Assign right shift	R → L
16	Comma	,	Evaluate	L → R

VALUES TO VARIABLES

In C, there are two different methods through which a variable can be assigned values. They are,

1. Through assignment statements.
2. Using built-in functions.

ASSIGNMENT STATEMENTS

Assignment statements are the simple means by which variables are assigned values. The syntax is given below.

variable-name=expression;

Consider the below assignment statement.

a=b*c;

Here, the product of ‘b’, and ’c’ is calculated and it is stored in ‘a’.

Accepting single characters from the keyboard

getchar()

The following program illustrates this:

```
#include <stdio.h>

int main(void)
{

int i;
int ch;

for (i = 1; i<=5; ++i)
{
   ch = getchar();
   putchar();
}
```

```
return 0;

}
```

Sample program output

```
AACCddEEtt
```

The program reads five character (one for each iteration of the for loop) from the keyboard. Note that getchar() gets a single character from the keyboard, and putchar() writes a single character (in this case, ch) to the console screen.

Reading Strings

```
#include <stdio.h>

int main(void)
{

char name[25];
printf("Input a character string, up to 25 characters. \n");
gets(name):
printf("The string is %s\n", name);
printf("End of program.\n");

return 0;   }
```

gets collects a string of characters terminated by a new line from the standard input stream and puts it into name. It replaces the new line by a null character (\0) in name; it also allows input strings to contain certain characters (spaces, tabs).

MODULE-III

IF_ELSE

This lesson will show you how to:

- Use an if statement
- Use an else statement
- Use an else if statement

If Statement

The if statement is used to conditionally execute a block of code based on whether a test condition is true. If the condition is true the block of code is executed, otherwise it is skipped.

SYNTAX

```
if (condition)
        {
        true_statements ;
         }
        OR
if (condition)
true_statement ;
        OR
 if (condition)
        {
        true_statements ;
        }
else
```

```
{
false_statements ;
}
```

PROGRAM FLOW

1. First, the boolean expression *condition* is evaluated.
2. If *condition* evaluates to true, the *true_statement(s)* are executed.
3. If *condition* evaluates to false and an else clause exists, the *false_statement(s)* are executed.
4. Flow exits the if-else structure.

EXAMPLE

```
#include <stdio.h>
int main(void)
 {
        int number = 75;
         int mark;
        printf("Your examination mark\n");
        printf("Enter your score, please\n");
        scanf("%d",&mark);
         if (mark >= number)
                {
                printf("Incredible, you passed with a merit\n");
                 }
        return 0;
 }
```

The "==" is called a *relational operato*r. Relational operators, ==, !=, >, >=, <, and <=, are used to compare two values.

Else Statement

The *else* statement provides a way to execute one block of code if a condition is true, another if it is false.

Example:

```
#include <stdio.h>
int main(void)
{
        int number = 75;
         int mark;
        printf("Your examination mark\n");

        printf("Enter your score, please\n");

        scanf("%d",&mark);

        if (mark >= number)

        {
                printf("Incredible, you have passed with a merit\n");
         }
        else
        {
                printf("You failed, unlucky\n");
        }
        return 0;
}
```

FOR LOOP

SYNTAX

for (*initialization* ; *test* ; *increment*)

{

statements ;

}

OR

for (*initialization* ; *test* ; *increment*)

statement ;

PROGRAM FLOW

1. The *initialization* is first executed. This is typically something like int i=0, which creates a new variable with initial value 0, to act as a counter. Variables that you declare in this part of the for loop cease to exist after the execution of the loop is completed. Multiple, comma separated, expressions are allowed in the initialization section. But declaration expressions may not be mixed with other expressions.
2. The boolean expression *test* is then evaluated. This is typically something like i<10. Multiple, comma separated, expressions are not allowed. If *test* evaluates to true, flow contiues to step 3. Otherwise the loop exits.
3. The *statement(s)* are executed.
4. Then the statement *increment* is executed. It is typically something like i++ , which increments i by one or i+=2 , which increments i by two. Multiple, comma separated, expressions are allowed in the increment section.
5. Flow returns to step 2.

EXAMPLE

The following code will print hello ten times:

```
for (t=0; t<10; t++)
        printf("Hello\n");
```

WHILE STATEMENT

The simplest of all the c looping structures in C is the **while** statement . The **while** statement is an entry -controlled loop statement. The test condition is evaluated and if the condition is true then the body of the loop is executed . After the execution of the body the test condition is once again evaluated and if it is true ,the body is executed once again. This process of repeated execution of the body is continued

until the test condition becomes false and the control is transferred out of the loop. On exit the program is continued with the statement after the body of the loop.
The body of the loop may have one or more statements. The braces are needed only if the body contains two or more statements.

The format of while statement is

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

Example of while statement

```
/*******************************************************/
/* Program to compute x to the power n using while loop */
/*******************************************************/

# include<stdio.h>
main()
{
        int count,n;
        float x,y;
        printf("Enter the values of x and n:");
        scanf("%f %d", &x, &n);
        y= 1.0;
        count = 1;
        /* LOOP BEGINS*/
        while(count<= n)
                {
                y = y * n;
                count ++;
                }
        /*End of loop*/
        printf(" x = %f ; n = %d ; to power n = %f n ", x , n, y);
}
```

DO STATEMENT

In while statement there is a chance of skipping the entire loop without execution if the value is not satisfying the condition . To avoid such a condition we use do while statement. In **do .. while** statement the body of the loop will be executed at least once . On reaching the **do** statement , the program proceeds to evaluate the body first . At the end of the loop the *test condition* in the in the **while** statement is evaluated if the condition is true the program continues to evaluate the body of the loop once again . This process continues as long as the condition is true . When the condition becomes false , the loop will be terminated and the control of the program goes to the statement that appears after the while statement .

Since *the test-condition* is evaluated at the bottom of the loop , the **do** construct provides an *exit-controlled* loop and therefore the body of the loop is *always executed once*.

The format of do while statement is as follows

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

Example for *do while* **statement**

```
/* Printing of multiplication table */
#define colmax 10
#define rowmax 12
main()
{
```

```
        int row,column,y;
        row = 1;
        printf(“ MULTIPLICATION TABLE \n”);
        printf(“_____\n”);
        /* OUTER LOOP BEGINS */
        do {
        column = 1;
        /* INNER LOOP BEGINS */
        do
        {
        column = 1;
        do
        {
        y = row * column;
        printf(“%4d”,y);
        column = column +1;
        }
        while (column <= COLMAX);
        /* INNER LOOP BEGINS */
        printf(“\n”);
        row = row + 1;
        }
        while( row <=ROWMAX);
        /* OUTER LOOP ENDS */
        printf(“_____”)
}
```

SWITCH STATEMENT

The switch statement is another form of the multi way decision. It is well structured, but can only be used in certain cases where;

- Only one variable is tested, All branches must depend on the value of that variable. The variable must be an integral type.(int long, short or char).
- Each possible value of the variable can control a single branch. A final ,catch all default branch may optionally be used to trap all unspecified cases.

Syntax

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

    case value-2:
                    block-2
                    break;

   default:
                   default-block
                   break;
}
```

Hopefully an example will clarify things. This is a program which accepts four mathematical symbols + , - , * , / .Then the program accepts two integers a & b & displays the output as the value of a + b , a – b , a * b or a / b depending upon the symbol entered.

```
#include<stdio.h>

main()

{

  int a,b;

  char c;

  printf("Enter the symbol \n");

  scanf("%c",&c);
  switch(c)

  {

   case '+' : printf("Enter the values for a & b\n");

              scanf("%d %d",&a,&b);

              printf("a + b = %d\n",a+b);

              break;


   case '-' : printf("Enter the values for a & b\n");

              scanf("%d %d",&a,&b);

              printf("a - b = %d\n",a-b);

              break;


   case '*' : printf("Enter the values for a & b\n");

              scanf("%d %d",&a,&b);

              printf("a * b = %d\n",a*b);

              break;
```

```
    case '/' : printf("Enter the values for a & b\n");
               scanf("%d %d",&a,&b);
               printf("a / b = %d\n",a/b);
               break;

    default : printf("Invalid operator\n");
    }
}
```

Here, each interesting case is listed with a corresponding action. The control passes to the statement whose 'case constant- expression' matches the value of switch (expression). The **switch** statement can include any number of **case** instances, but no two case constants within the same **switch** statement can have the same value. Execution of the statement body begins at the at the selected statement and proceeds until the end of the body or until a **break** statement transfers control out of the body. The **break** statement prevents any further statements from being executed by leaving the **switch**. The default statement is executed if no **case** constant-expression is equal to the value of switch(expression). If the default statement is omitted and no **case** match is found, none of the statements in the switch body are executed. There can be at most one default statement. The default statement need not come at the end, it can appear anywhere in the body of the switch statement. A **case** or **default** label can only appear inside a switch statement. Switch statement can be nested. A single statement can carry multiple case labels.

FUNCTIONS IN C

What Is a Function?

This chapter approaches the question "What is a function?" in two ways. First, it tells you what functions are, and then it shows you how they're used.

"Scope and Storage Classes in C," you might have noticed that a function definition is always given first, before the function is called from a main() function. In fact, you can put a function definition anywhere you want, as long as you keep the function declaration at the first place before the function is called. You'll learn about many function features from the following topics covered in this lesson:

- **Function declarations**
- **Prototyping**
- **Values returned from functions**
- **Arguments to functions**

In addition, several C library functions and macros, such as time(), localtime(), asctime(), va_start(), va_arg(), and va_end() are introduced in this hour.

Declaring Functions

As you know, you have to declare or define a variable before you can use it. This is also true for functions. In C, you have to declare or define a function before you can call it.

How a Function Works

A C program doesn't execute the statements in a function until the function is called by another part of the program. When a function is called, the program can send the function information in the form of one or more arguments. An argument is program data needed by the function to perform its task. The statements in

the function then execute, performing whatever task each was designed to do. When the function's statements have finished, execution passes back to the same location in the program that called the function. Functions can send information back to the program in the form of a return value.

Declaration Versus Definition

According to the ANSI standard, the declaration of a variable or function specifies the interpretation and attributes of a set of identifiers. The definition, on the other hand, requires the C compiler to reserve storage for a variable or function named by an identifier.

A variable declaration is a definition, but a function declaration is not. A function declaration alludes to a function that is defined elsewhere and specifies what kind of value is returned by the function. A function definition defines what the function does, as well as gives the number and type of arguments passed to the function.

A function declaration is not a function definition. If a function definition is placed in your source file before the function is first called, you don't need to make the function declaration. Otherwise, the declaration of a function must be made before the function is invoked.

For example, the printf() function is used in almost every sample program in this book. Each time, it include a header file, stdio.h, because the header file contains the declaration of printf(), which indicates to the compiler the return type and prototype of the function. The definition of the printf() function is placed somewhere else. In C, the definition of this function is saved in a library file that is invoked during the linking states.

Specifying Return Types

A function can be declared to return any data type, except an array or function. The return statement used in a function definition returns a single value whose type should match the one declared in the function declaration.

By default, the return type of a function is int, if no explicit data type is specified for the function. A data type specifier is placed prior to the name of a function like this:

```
data_type_specifier  function_name();
```

Here data_type_specifier specifies the data type that the function should return. function_name is the function name that should follow the rule of naming in C.

In fact, this declaration form represents the traditional function declaration form before the ANSI standard was created. After setting up the ANSI standard, the function prototype is added to the function declaration.

Using Prototypes

Before the ANSI standard was created, a function declaration only included the return type of the function. With the ANSI standard, the number and types of arguments passed to a function are allowed to be added into the function declaration. The number and types of an argument are called the function prototype.

The general form of a function declaration, including its prototype, is as follows:

```
data_type_specifier  function_name(
    data_type_specifier argument_name1,
    data_type_specifier argument_name2,
    data_type_specifier argument_name3,
    .
    .
    .
    data_type_specifier argument_nameN,
);
```

The purpose of using a function prototype is to help the compiler check whether the data types of arguments passed to a function match what the function expects. The compiler issues an error message if the data types do not match.

Although argument names, such as argument_name1, argument_name2, and so on, are optional, it is recommended that you include them so that the compiler can identify any mismatches of argument names.

Making Function Calls

When a function call is made, the program execution jumps to the function and finishes the task assigned to the function. Then the program execution resumes after the called function returns.

A function call is an expression that can be used as a single statement or within other statements.

An example of declaring and defining functions, as well as making function calls is given below

Prototyping Functions

In the following subsections, we're going to study three cases regarding arguments passed to functions. The first case is a function that takes no argument; the second one is a function that takes a fixed number of arguments; the third case is a function that takes a variable number of arguments.

Functions with No Arguments

The first case is a function that takes no argument. For instance, the C library function getchar() does not need any arguments. It can be used in a program like this:

```
int c;

c = getchar();
```

As you can see, the second statement is left blank between the parentheses ((and)) when the function is called.

In C, the declaration of the getchar() function can be something like this:

int getchar(void);

Note that the keyword void is used in the declaration to indicate to the compiler that no argument is needed by this function. The compiler will issue an error message if somehow there is an argument passed to getchar() later in a program when this function is called.

Therefore, for a function with no argument, the void data type is used as the prototype in the function declaration.

Local and global variables

Local:

These variables only exist inside the specific function that creates them. They are unknown to other functions and to the main program. As such, they are normally implemented using a stack. Local variables cease to exist once the function that created them is completed. They are recreated each time a function is executed or called.

Global:

These variables can be accessed (ie known) by any function comprising the program. They are implemented by associating memory locations with variable names. They do not get recreated if the function is recalled. To declare a global variable, declare it outside of all the functions. There is no general rule for where outside the functions these should be declared, but declaring them on top of the code is normally recommended for reasons of scope, as explained below. If a variable of the same name is declared both within a function and outside of it, the function will use the variable that was declared within it and ignore the global one.

Defining global variables:

/* Demonstrating global variables */

```
#include <stdio.h>

int add_numbers( void); // ANSI function prototype

/* These are global variables and can be accessed by functions from this point on */

int value1, value2, value3;

int add_numbers (void)
{

   auto int result;
   result = value1 + value2 + value3;

   return result;

}

int main(void)
{

   auto int result;
   value1 = 10;
   value2 = 20;
   value3 = 30;
   result = add_numbers();
   printf("The sum of %d + %d + %d is %d\n", value1, value2, value3, final_result);

   return 0;

}
```

PARAMETR_PASSING_BETWEEN FUNCTIONS

The mechanism used to convey information to the function is called argument or parameter. The format string and the list of variables used inside the parameter in these functions are arguments. These are of two types.

1.Call by value

2.Call by reference.

Call by value

In this method the value of the actual arguments in the calling functions are copied in to

corresponding formal arguments of called function. With this method the changes made to

to the formal arguments in called function have no effect on the values of actual arguments in calling function.

Consider the following program ,

/*Sending and receiving values between function*/

```
main( )
 {
  int a  b  product;
  printf("Enter two numbers");
  scanf("%d%d",&a,&b);
  product=calproduct(a,b);
   printf("\n  %d   %d   product=%d",a,b  product); /here values of a &b remains unchanged/
}
 int calproduct( int x , int y)
      {
  int d;
 d=x*y;
```

```
return (d);
}
```

In this program, in main( ) we receive the values of a , b through the keyboard and then output the product of a,b .However the calculation of product is done in a different function called calproduct( ).The variables a ,b are called 'actual argument', whereas the variables x,y are called formal arguments. Any number of arguments can be passed to a function being called . However the type, order and number of the actual and formal arguments must always be same. Instead of using different variable names x ,y ,we could have used the same variable names a,b. But the compiler would still treat them as different variables. So if the value of a formal argument is changed in called function, the corresponding changes do not take place in the calling function.

A function can return only one value so return(a,b) is invalid statement.

There are two methods of declaring the formal arguments .

```
calproduct( x,y)
int x,y;
```

Another methode is ,

```
calproduct(int x,int y)
```

Call by reference

In call by reference method we are passing the address of the data as argument. This means that using these address we would have an access to actual argument and hence we would able to manipulate them. The following program illustrate this fact

```
main( )
```

```
{
  int a=10,b=20;
  swapr(&a,&b);
  printf("\na=%d  b = %d",a,b);
}
swapr(int *x,int *y)
{
  int t ;
    t =*x ;
   *x =*y ;
    *y=t ;
}
```

the output of this program would be a=20   b=10

Using void in function declarations.

```
1:  /*  Functions with no arguments */
2:  #include <stdio.h>
3:  #include <time.h>
4:
5:  void GetDateTime(void);
6:
7:  main()
```

```
8:  {
9:     printf("Before the GetDateTime() function is called.\n");
10:    GetDateTime();
11:    printf("After the GetDateTime() function is called.\n");
12:    return 0;
13: }
14: /* GetDateTime() definition */
15: void GetDateTime(void)
16: {
17:    time_t now;
18:
19:    printf("Within GetDateTime().\n");
20:    time(&now);
21:    printf("Current date and time is: %s\n",
22:       asctime(localtime(&now)));
23: }
```

OUTPUT

The following output will be obtained when the above program is run

```
Before the GetDateTime() function is called.
```

Within GetDateTime().

Current date and time is: Sat Apr 05 11:50:10 1997

After the GetDateTime() function is called.

Functions with a Fixed Number of Arguments

int function_1(int x, int y);

contains the prototype of two arguments, x and y.

To declare a function with a fixed number of arguments, you need to specify the data type of each argument. Also, it's recommended to indicate the argument names so that the compiler can have a check to make sure that the argument types and names declared in a function declaration match the implementation in the function definition.

Prototyping a Variable Number of Arguments

The syntax of the printf() function is

int printf(const char *format[, argument, ...]);

Here the ellipsis token ... (that is, three dots) represents a variable number of arguments. In other words, besides the first argument that is a character string, the printf() function can take an unspecified number of additional arguments, as many as the compiler allows. The brackets ([and]) indicate that the unspecified arguments are optional.

The following is a general form to declare a function with a variable number of arguments:

```
data_type_specifier  function_name(
    data_type_specifier argument_name1, ...
);
```

Note that the first argument name is followed by the ellipsis (...) that represents the rest of unspecified arguments.

For instance, to declare the printf() function, you can have something like this:

```
int printf(const char *format, ...);
```

Void functions:

The functions that do not return any values can be explicitly defined as void. This prevents any accidental use of these functions in expressions.

Recursion

The term recursion refers to a situation in which a function calls itself either directly or indirectly. Indirect recursion occurs when one function calls another function that then calls the first function. C allows recursive functions, and they can be useful in some situations.

For example, recursion can be used to calculate the factorial of a number. The factorial of a number x is written x! and is calculated as follows:

x! = x * (x-1) * (x-2) * (x-3) * ... * (2) * 1

However, you can also calculate x! like this:

x! = x * (x-1)!

Going one step further, you can calculate (x-1)! using the same procedure:

(x-1)! = (x-1) * (x-2)!

You can continue calculating recursively until you're down to a value of 1, in which case you're finished. The following program uses a recursive function to calculate factorials. Because the program uses unsigned integers, it's limited to an input value of 8; the factorial of 9 and larger values are outside the allowed range for integers.

This is Using a recursive function to calculate factorials.

```
 1:   /* Demonstrates function recursion. Calculates the */
2:   /* factorial of a number. */
3:
4:   #include <stdio.h>
5:
6:   unsigned int f, x;
7:   unsigned int factorial(unsigned int a);
8:
9:   main()
10:  {
11:      puts("Enter an integer value between 1 and 8: ");
12:      scanf("%d", &x);
13:
14:      if( x > 8 || x < 1)
15:      {
16:          printf("Only values from 1 to 8 are acceptable!");
17:      }
18:      else
19:      {
20:          f = factorial(x);
```

```
21:         printf("%u factorial equals %u\n", x, f);
22:     }
23:
24:     return 0;
25: }
26:
27: unsigned int factorial(unsigned int a)
28: {
29:     if (a == 1)
30:         return 1;
31:     else
32:     {
33:         a *= factorial(a-1);
34:         return a;
35:     }
36: }
```

Out Put:-

```
Enter an integer value between 1 and 8:
6
6 factorial equals 720
```

The basic philosophy of function is divide and conquer by which a complicated tasks are successively divided into simpler and more manageable tasks which can be

easily handled. A program can be divided into smaller subprograms that can be developed and tested successfully.

A function is a complete and independent program which is used (or invoked) by the main program or other subprograms. A subprogram receives values called arguments from a calling program, performs calculations and returns the results to the calling program.

There are many advantages in using functions in a program they are:

1. It facilitates top down modular programming. In this programming style, the high level logic of the overall problem is solved first while the details of each lower level functions is addressed later.

2. the length of the source program can be reduced by using functions at appropriate places. This factor is critical with microcomputers where memory space is limited.

3. It is easy to locate and isolate a faulty function for further investigation.

4. A function may be used by many other programs this means that a c programmer can build on what others have already done, instead of starting over from scratch.

5. A program can be used to avoid rewriting the same sequence of code at two or more locations in a program. This is especially useful if the code involved is long or complicated.

6. Programming teams does a large percentage of programming. If the program is divided into subprograms, each subprogram can be written by one or two team members of the team rather than having the whole team to work on the complex program

We already know that C support the use of library functions and use defined functions. The library functions are used to carry out a number of commonly used operations or calculations. The user-defined functions are written by the programmer to carry out various individual tasks.

Functions are used in c for the following reasons:

1. Many programs require that a specific function is repeated many times instead of writing the function code as many timers as it is required we can write it as a single function and access the same function again and again as many times as it is required.

2. We can avoid writing redundant program code of some instructions again and again.

3. Programs with using functions are compact & easy to understand.

4. Testing and correcting errors is easy because errors are localized and corrected.

5. We can understand the flow of program, and its code easily since the readability is enhanced while using the functions.

6. A single function written in a program can also be used in other programs also.

Preprocessor directives:

Directive	**Function**
#define	Defines a macro substitution
#undef	Undefines a macro
#include	Specifies a file to be included
#ifdef	Tests for macro definition
#endif	Specifies the end of #if
#ifndef	Tests whether the macro is not def
#if	Tests a compile time condition
#else	Specifies alternatives when # if test fails

The preprocessor directives can be divided into three categories

1. Macro substitution division
2. File inclusion division
3. Compiler control division

Macros:

Macro substitution is a process where an identifier in a program is replaced by a pre defined string composed of one or more tokens we can use the #define statement for the task.

It has the following form

#define identifier string

The preprocessor replaces every occurrence of the identifier int the source code by a string. The definition should start with the keyword #define and should follow on identifier and a string with at least one blank space between them. The string may be any text and identifier must be a valid c name.

There are different forms of macro substitution. The most common form is

1. Simple macro substitution
2. Argument macro substitution
3. Nested macro substitution

Simple macro substitution:

Simple string replacement is commonly used to define constants example:

#define pi 3.1415926

Writing macro definition in capitals is a convention not a rule a macro definition can include more than a simple constant value it can include expressions as well. Following are valid examples:

#define AREA 12.36

Macros as arguments:

The preprocessor permits us to define more complex and more useful form of replacements it takes the following form.

define identifier(f1,f2,f3…..fn) string.

Notice that there is no space between identifier and left parentheses and the identifier f1,f2,f3 …. Fn is analogous to formal arguments in a function definition.

There is a basic difference between simple replacement discussed above and replacement of macro arguments is known as a macro call

A simple example of a macro with arguments is

define CUBE (x) (x*x*x)

If the following statements appears later in the program,

volume=CUBE(side);

The preprocessor would expand the statement to

volume =(side*side*side)

Nesting of macros:

We can also use one macro in the definition of another macro. That is macro definitions may be nested. Consider the following macro definitions

define SQUARE(x)((x)*(x))

Undefining a macro:

A defined macro can be undefined using the statement

undef identifier.

This is useful when we want to restrict the definition only to a particular part of the program.

File inclusion:

The preprocessor directive "#include file name" can be used to include any file in to your program if the function s or macro definitions are present in an external file they can be included in your file

In the directive the filename is the name of the file containing the required definitions or functions alternatively the this directive can take the form

#include< filename >

Without double quotation marks. In this format the file will be searched in only standard directories.

The c preprocessor also supports a more general form of test condition #if directive. This takes the following form

#if constant expression

{
statement-1;
statemet2'
….
….
}
#endif

the constant expression can be a logical expression such as test < = 3 etc

If the result of the constant expression is true then all the statements between the #if and #endif are included for processing otherwise they are skipped. The names TEST LEVEL etc., may be defined as macros.

MODULE-1V

ARRAYS

Introduction

An array is a group of related data items that share a common name. For instance, we can define an array name salary to represent a set of salaries of a group of employees. A particular value is indicated by writing a number called index number or subscript in brackets after the array name. For example,

Salary [10]

represents the salary of the 10^{th} employee. While the complete set of values is referred to as an array, the individual values are called elements. Arrays can be of any variable type.

The ability to use a single name to represent a collection of items and to refer to an item by specifying the item number enables us to develop concise and efficient programs.

One dimensional arrays

A list of items can be given one variable name using only one subscript and such a variable is called a single subscripted variable or a one dimensional array. In mathematics, we often deal with variables that are single subscripted. For instance, we use the equation.

To calculate the average of n values of x. The subscripted variable x_i refers to the i^{th} element of **x.** In C, single- subscripted variable x^i can be expressed as

x[1], x[2], x[3], …….. x[n]

The subscript can begin with number 0. That is

x [0]

is allowed

Declaration of Arrays

Like any other variable, arrays must be declared before they are used. The general form of array declaration is

type variable name [size];

The type specifies the type of element that will be contained in the array, such as int, float, or char and the size indicates the maximum number of elements that can be stored inside the array. For example,

Float height [50];

Declares the height to be an array containing 50 real elements. Any subscripts 0 to 49 are valid. Similarly,

Int group [10];

Declares the group as an array to contain a maximum of 10 integer contents.

The C language treats character strings simply as arrays of characters. The size in a character string represents the maximum number of characters that the string can hold. For instance,

Char name [10];

Declares the name as a character array (string) variable that can be hold maximum of 10 characters.

Initialization of Arrays

We can initialize the elements of arrays in the same way as the ordinary variable when they are declared.

The general form of initialization of arrays is:

Static type array- name [size] = {list of values};

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

Static int number [3] = {0,0,0};

Will declare the variable number as an array of size 3 and will assign zero to each element. If the number of values in the list is less than the number of elements, then

only that many elements will be initialized. The remaining elements will be set to zero automatically. For instance,

Static float total [5] = {0.0, 15.75, -10};

Will initialized the first three elements to 0.0, 15.75, and -10.0 and the remaining two elements to zero.

Initialization of arrays in C suffers two drawbacks.

1. There is no convenient way to initialize only selected elements.
2. There is no shortcut method for initializing a large number of array elements like the one available in FORTRAN.

Program

```
/*****************************************************/
/*    PROGRAM SHOWING ONE DIMENSIONAL ARRAY    */
/*****************************************************/
Main ( )
{
   Int     I;
   Float x [10], value, total;
/*……Reading values into array……*/
 Print f(“ENTER 10 REAL NUMBERS\n”);
 For (I = 0 ; I < 10; I + +)
 {
  Scan f (“% f”, & value);
```

```
  X[i] = value;
 }
/* ……..COMPUTATION OF TOTAL ……*/
  Total = 0.0;
  For (I = 0; I < 10; I + + )
  Total = total + x[i] * x[i]
/* ……PRINTING OF x[i] VALUES AND TOTAL ……*/
  Print f (“\n”);
  for (I = 0; I < 10; I + + )
  Print f ( “x [ % 2d] = % 5.2f\n”, I +1, x[i]);
  Print f (“\ntotal = % 2f\n”, total);
 }
```

Output

Enter 10 real numbers

1.1 2.2 3.3 4.4 5.5 6.6 7.7 8.8 9.9 10. 10

X[1] = 1.10

X[2] = 2.20

X[3] = 3.30

X[4] = 4.40

X[5] = 5.50

X[6] = 6.60

X[7] = 7.70

X[8] = 8.80

X[9] = 9.90

X[10] = 10.10

Total = 446.86

TWO DIMENSIONAL ARRAYS

So far we have discussed the array variable that can store a list of values. There will be situations where a table of values will have to be stored. Consider the following data table, which shows the value of sales of three items by four salesgirls:

STRUCTURES AND UNIONS IN C

STRUCTURE

A structure is a collection of one or more variables, possibly of different types, grouped together under a single name. It is a convenient tool for handling a group of logically related data items.

Structure Definition

A structure definition creates a format that may be used to declare structure variables. Let us use an example to illustrate the process of structure definition and the creation of structure variables. Consider a book database consisting of book name, author, number of pages, and price. We can define a structure to hold this information as follows:

```
struct book_bank
 {
    char titile[20];
 char author[15];
int pages;
```

```
    float price;
};
```

The keyword **struct** declares a structure to hold the details of four fields, namely **titile, author, pages, and price.** These fields are called **structure elements or members.** Each member may belong to a different type of data. **book_bank** is the name of the structure and is called the **structure tag.** The tag name used subsequently to declare variables that have the tag's structure.

The general format of a structure definition is as follows:

```
struct tag_name
{
  data_type  member 1;
  data_type  member 2;
  …….     ……..
  …….     ……..
};
```

Declaring a structure

We can declare structure variable using the tag name anywhere in the program.For example the variables **b1, b2, b3** can be declared to be of the type **struct book_bank,** as

```
struct book_bank b1, b2, b3;
```

Each one of these variables has four members as specified by the template.The complete declaration might look like this:

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

struct book_bank b1,b2,b3

Remember that the members of a structure themselves are not variables. They do not occupy any memory until they are associated with the structure variables such as **b1.**

In defining a structure you may note the following syntax:

1. The template is terminated with a semicolon.
2. While the entire declaration is considered as a statement, each member is declared independently for its name and type in a separate statement inside the template.
3. The tag name such as **book_bank** can be used to declare structure variable of its type, later in program.

Accessing Structure Element

Having declared the structure type and the structure variable. Let us see the element of the structure can be accessed.

In arrays we can access individual element of an array using a subscript. Structure use a different scheme. They use a dot(.) operator. So refer to **pages** of the structure defined in above example we have to use

b1.pages

similarly to refer to **price** we would use, **b1.price.**

Note that before the dot there must always be a structure variable and after the dot there must always be a structure element.

For example,

```
main( )
 {
   struct book_bank
    {
      char name[15];
   float price;
  int pages;
};
        struct book_bank b1,b2,b3
        printf(" Enter names, prices & no. of pages of 3 books\n");
        scanf("%c %f %d",&b1.name,&b1.price,&b1.pages);
        scanf("%c %f %d",&b2.name,&b2.price,&b2.pages);
        scanf("%c %f %d",&b3.name,&b3.price,&b3.pages);

        printf(" Details You Entered\n");
        printf("\n%c    %f   %d",b1.name,b1.price,b1.pages);
        printf("\n%c    %f   %d",b2.name,b2.price,b2.pages);
```

```
        printf("\n%c     %f     %d",b3.name,b3.price,b3.pages);
}
```

Sample Input and Out put,

Enter names, prices & no. of pages of 3 books

A 100.00 345

B 256.50 682

K 233.70 512

Details You Entered

A 100.00 345

B 256.50 682

K 233.70 512

How Structure Elements are Stored

Whatever be the elements of a structure, they are always stored in contiguous memory location.

```
main( )
  {
    struct book_bank
     {
       char name[15];
```

```
    float price;
    int pages;
};
        struct book_bank b1={'B',130.00,550};

        printf(" \nAddress of name=%u",&b1.name);
        printf(" \nAddress of price=%u",&b1.price);
        printf(" \nAddress of page=%u",&b1.page);

    }
```

Output:

Address of name = 65518

Address of price = 65519

Address of pages = 65523

Arrays of Structures

We use structures to describe the format of a number of related variables.We can declare an array of structures, each element of the array representing a structure variable. For example,

struct class student[100];

defines an array called **student,** that consist of 100 elements. Each element is defined to be of the type **struct class**. Consider the following declaration.

```
struct marks
 {
    int subject 1;
     int subject 2;
    int subject 3;
 };
main( )
                {
     static struct marks student[3] = {{45,68,46},{65,83,92},{75,83,39}};
```

This declares the **student** as an array of three element **student[0], student[1],** and **student[2]** and initializes their members as follows:

```
                    student[0].subject1 = 45;
                    student[0].subject1 = 68;
…….
…….
                    student[2].subject1 = 39;
```

Additional Features of Structure

Following are the major additional features of structure,

1. The value of a structure variable can be assigned to another structure variable of the

same type using the assignment operator.

2. One structure can be nested within another structure.

3. Like an ordinary variable, a structure variable can also be passed to a function.

4. Pointer can be used in structure. Such pointers are known as ‘structure pointers’.

Structure within a structure:

A structure may be defined as a member of another structure. In such structures the declaration of the embedded structure must appear before the declarations of other structures.

```
struct date
{
int day;
int month;
int year;
};
struct student
{
int id_no;
char name[20];
char address[20];
char combination[3];
int age;
structure date def;
structure date doa;
}oldstudent, newstudent;
```

The sturucture student constains another structure date as its one of its members.

UNIONS

Unions are a concept borrowed from structures and therefore follow the same syntax as structures. However, there is major distinction between them in terms of storage. In structures, each member has its own storage location,whereas all the members of a union use the same location. This implies that, although a union may

contain many members of different types, it can handle only one member at a time. Like structures, a union can be declared using the keyword **union** as follow:

```
union item
{
    int m;
    float x;
    char c;
} code;
```

This declares a variable code of type **union item.** The union contain three members, each with a different data type. However we can use only one at a time. This is due to the fact that only one location is allocated for a union variable, irrespective of its size. The compiler allocates a piece of storage that is large enough to hold the largest variable type in the union. To access a union member, we can use the same syntax that we use for structure members. That is

code.m

code.x

code.c

are all valid member variables. During accessing, we should make sure that we are accessing the member whose value is currently stored. For example, the statement such as

```
code.m = 379;
code.x = 786.63
printf("%d",code.m);
```

would produce erroneous output.

In effect, a union creates a storage location that can be used by any one of its members at a time. When a different member is assigned a new value supercedes the previous member's value.

Unions may be used in all places where a structure is allowed.

Module V

The scope and lifetime of variables in functions:

The scope and lifetime of the variables define in C is not same when compared to other languages. The scope and lifetime depends on the storage class of the variable in c language the variables can be any one of the four storage classes:

1. Automatic Variables
2. External variable
3. Static variable
4. Register variable.

The scope actually determines over which part or parts of the program the variable is available. The lifetime of the variable retains a given value. During the execution of the program. Variables can also be categorized as local or global. Local variables are the variables that are declared within that function and are accessible to all the functions in a program and they can be declared within a function or outside the function also.

Automatic variables:

Automatic variables are declared inside a particular function and they are created when the function is called and destroyed when the function exits. Automatic variables are local or private to a function in which they are defined by default all variable declared without any storage specification is automatic. The values of variable remains unchanged to the changes that may happen in other functions in the same program and by doing this no error occurs.

```
/* A program to illustrate the working of auto variables*/
#include
void main()
```

```
{
int m=1000;
function2();
printf("%d\n",m);
}

function1()
{
int m=10;
printf("%d\n",m);
}
function2()
{
int m=100;
function1();
printf("%d\n",m);
}
```

A local variable lives through out the whole program although it accessible only in the main. A program with two subprograms function1 and function2 with m as automatic variable and is initialized to 10,100,1000 in function 1 function2 and function3 respectively. When executes main calls function2 which in turns calls function1. When main is active m=1000. But when function2 is called, the main m is temporarily put on the shelf and the new local m=100 becomes active. Similarly when function1 is called both previous values of m are put on shelf and latest value (m=10) become active, a soon as it is done main (m=1000) takes over. The output clearly shows that value assigned to m in one function does not affect its value in the other function. The local value of m is destroyed when it leaves a function.

External variables:

Variables which are common to all functions and accessible by all functions of aprogram are internal variables. External variables can be declared outside a function.

Example

```
int sum;
float percentage;
main()
{
…..
…..
}
function2()
{
….
….
}
```

The variables sum and percentage are available for use in all the three functions main, function1, function2. Local variables take precedence over global variables of the same name.

For example:

```
int i = 10;
void example(data)
int data;
{
int i = data;
}

main()
{
```

```
example(45);
}
```

In the above example both the global variable and local variable have the same name as i.
The local variable i take precedence over the global variable. Also the value that is stored in integer i is lost as soon as the function exits.

A global value can be used in any function all the functions in a program can access the global variable and change its value the subsequent functions get the new value of the global variable, it will be inconvenient to use a variable as global because of this factor every function can change the value of the variable on its own and it will be difficult to get back the original value of the variable if it is required.

Global variables are usually declared in the beginning of the main program ie., before the main program however c provides a facility to declare any variable as global this is possible by using the keyword storage class extern. Although a variable has been defined after many functions the external declaration of y inside the function informs the compiler that the variable y is integer type defined somewhere else in the program. The external declaration does not allocate storage space for the variables. In case of arrays the definition should include their size as well. When a variable is defined inside a function as extern it provides type information only for that function. If it has to be used in other functions then again it has to be re-declared in that function also.

Example:

```
main()
{
int n;
out_put();
```

```
extern float salary[];
……
…..
out_put();
}

void out_put()
{
extern float salary[];
int n;
….
…..
}
float salary[size];
```

a function when its parameters and function body are specified this tells the compiler to allocate space for the function code and provides type info for the parameters. Since functions are external by default we declare them (in calling functions) without the qualifier extern.

Multi-file programs:

Programs need not essentially be limited into a single file, multi-file programs is also possible, all the files are linked later to form executable object code. This approach is very useful since any change in one file does not affect other files thus eliminating the need for recompilation of the entire program. To share a single variable in multiple programs it should be declared, as external variables that are shared by two or more files are obviously global variables and therefore we must declare them accordingly in one file and explicitly define them with extern in other file. The example shown below illustrates the use of extern declarations in multi-file programs

File1.c

```
main()
{
extern int j;
int k;
}

function1()
{
int z;
…
….
}
```

file2.c

```
function2()
{
int k;
}
function3()
{
int num;
…
….
}
```

the function in main file1 reference the variable j that is declared as global in file 2. Here function1() cannot access j if the statement extern int k is places before main then both the functions could refer to j. this can also be achieved by using extern int j statement inside each function in file1.

The extern specifier tells the compiler that the following variables types and names have already been declared elsewhere and no need to create storage space for them. It is the responsibility of the linker to resolve the reference problem. It is important to note that a multi-file global variable should be declared without extern in one of the files.

Static variables:

The value given to a variable declared by using keyword static persistes until the end of the program. A static variable is initialized only once, when the program is compiled. It is never initialized again. During the first call to stat in the example shown below x is incremented to 1. because x is static, this value persists and therefore the next call adds another 1 to x giving it a value of 2. The value of x becomes 3 when third call is made. If we had declared x as an auto then output would here been x=1 all the three times.

```
main()
{
int j;
for(j=1;j&lt;3;j++)
stat();
}
stat();
{
static int x=0;
x=x+1;
printf("x=%d\n",x);
}
```

Register variables:

A variable is usually stored in the memory but it is also possible to store a varible in the compilers register by defining it as register variable. The registers access is much

faster than a memory access, keeping the frequently accessed variables in the register will make the execution of the program faster.

This is done as follows:

register int count;

POINTERS

Every variable when declared in occupies certain memory location. In C it is possible to access and display the address of the memory location of variable using & operator with variable name. The pointer variable is needed to store the memory address of any variable. The pointer is denoted by asterisk(*) symbol.

A pointer basically is a memory variable that stores a memory address .Pointer can have any name that is legal for other variables and is declared in the same fashion like other variable but it is always denoted by * operator.

Features of pointers

- Pointer saves memory space.
- Execution time with pointer is fast because data is manipulated with address
- Memory is accessed efficiently with pointers.
- Pointers are useful for representing two dimensional and multi dimensional arrays

Pointer declaration

Pointer variable can be declared as follows

```
Int *x;

Char *y;

Float *f;
```

The first statement is an integer pointer- it holds an address of an integer variable

The second statement is an character pointer- it holds an address of a character variable

The third statement is an float pointer- it holds an address of a floating point variable

Accessing pointers

'*'is used to access the value at address and

'&' is used to access the address of the variable

%u is used with printf statement to print the address of a variable.

Address of any type of variable is whole number.

Given below is a program which use pointer to print address and value of a variable.

```
#include<stdio.h>
 main()
{
int n,*k;
printf("Enter a number");
scanf("%d",&n);
k=&n;
  printf("Address of n is %u",k);
  printf("Value of n is %d",*k);
```

Output

Enter a number 25

Address of n is 4072

Value of n is 25

In the above example address of variable n is assigned to pointer variable k. Hence k is pointing to n. Value of the variable n is displayed using the pointer *k.

Pointers and arrays

Array name by itself is an address pointer. it points to the address of the first element. The element of the array together with their address can be displayed by using array name itself. Array elements are always stored in continuous memory location. No separate pointer variable is needed to access the address of the array variable.

For a two dimensional array the first argument is taken as row number and second argument is taken as column number. To display the elements of two dimensional array it is essential to have'&' operator as pre fix with an array name followed by element numbers.

Pointer to pointer.

Pointer is known as variable containing address of another variable. The pointer variable also have an address. The pointer variable containing address of another variable is called a pointer to pointer. This chain can be continued to any extend. To represent pointer to pointer we declare the variable with two asterisks.

Example. Int **q;

Void pointers

Pointers can also be declared as void type. Void type cannot be differenced without explicit type conversion. This is because being void compiler cannot

determine the size of the object that pointer points to. Though void pointer declaration is possible, void variable declaration is not allowed.

Pointer as reference parameter

Call by reference

In call by reference method we are passing the address of the data as argument.

This means that using these address we would have an access to actual argument

and hence we would able to manipulate them.the following program illustriate this fact

```
        main( )
{
 int a=10,b=20;
 swapr(&a,&b);
 printf("\na=%d  b = %d",a,b);
        }
                swapr(int *x,int *y) {
 int t ;
   t =*x ;
 *x =*y ;
  *y=t ;
                }
```

Output

a=20 b=10

Structures and pointers

A structure is a collection of one or more variables of different data types grouped together under a single name. By using structures we can make a group of variables arrays pointers etc.

Pointers and structures

We know that pointer is a variable that holds the address of another data variable. The variable may be of any data type , Ie. Int, float or char. In the same way we can also define pointer to structures. Here starting address of the member variables can be accessed. Thus such pointers are called structure pointer.

Example:-

```
Struct book
{
        char name[25] ;
    char author[25];
    int pages;
};
    struct book  *pt;
```

in the above example*ptr is the pointer to the structure 'book'. The syntax for using pointer with member is as given below.

1) ptr → name 2)ptr → author 3)ptr → pages.

By executing these three statements starting address of each member is estimated.

Dynamic memory allocation

Dynamic memory allocation is a technique in which programs determine as they are running where to store some information. You need dynamic allocation when the amount of memory you need, or how long you continue to need it, depends on factors that are not known before the program runs.

For example, you may need a block to store a line read from an input file; since there is no limit to how long a line can be, you must allocate the memory dynamically and make it dynamically larger as you read more of the line.

Or, you may need a block for each record or each definition in the input data; since you can't know in advance how many there will be, you must allocate a new block for each record or definition as you read it.

When you use dynamic allocation, the allocation of a block of memory is an action that the program requests explicitly. You call a function or macro when you want to allocate space, and specify the size with an argument. If you want to free the space, you do so by calling another function or macro. You can do these things whenever you want, as often as you want.

Dynamic allocation is not supported by C variables; there is no storage class “dynamic”, and there can never be a C variable whose value is stored in dynamically allocated space. The only way to get dynamically allocated memory is via a system call (which is generally via a GNU C library function call), and the only way to refer to dynamically allocated space is through a pointer. Because it is less convenient, and because the actual process of dynamic allocation requires more computation time, programmers generally use dynamic allocation only when neither static nor automatic allocation will serve.

For example, if you want to allocate dynamically some space to hold a struct foobar, you cannot declare a variable of type struct foobar whose contents are the dynamically allocated space. But you can declare a variable of pointer

type struct foobar * and assign it the address of the space. Then you can use the operators `*' and `->' on this pointer variable to refer to the contents of the space:

```
{
 struct foobar *ptr
   = (struct foobar *) malloc (sizeof (struct foobar));
 ptr->name = x;
 ptr->next = current_foobar;
 current_foobar = ptr;
}
```

Dynamic memory allocation:

The process of allocating memory at run time is known as dynamic memory allocation. Although c does not inherently have this facility there are four library routines which allow this function.

Many languages permit a programmer to specify an array size at run time. Such languages have the ability to calculate and assign during executions, the memory space required by the variables in the program. But c inherently does not have this facility but supports with memory management functions, which can be used to allocate and free memory during the program execution. The following functions are used in c for purpose of memory management.

Function	**Task**
malloc	Allocates memory requests size of bytes and returns a pointer to the Ist byte of allocated space
calloc	Allocates space for an array of elements initializes them to zero and returns a pointer to the memory

free	Frees previously allocated space
realloc	Modifies the size of previously allocated space.

Memory allocations process:

According to the conceptual view the program instructions and global and static variable in a permanent storage area and local area variables are stored in stacks. The memory space that is located between these two regions in available for dynamic allocation during the execution of the program. The free memory region is called the heap. The size of heap keeps changing when program is executed due to creation and death of variables that are local for functions and blocks. Therefore it is possible to encounter memory overflow during dynamic allocation process. In such situations, the memory allocation functions mentioned above will return a null pointer.

Allocating a block of memory:

A block mf memory may be allocated using the function malloc. The malloc function reserves a block of memory of specified size and returns a pointer of type void. This means that we can assign it to any type of pointer. It takes the following form:

ptr=(cast-type*)malloc(byte-size);

ptr is a pointer of type cast-type the malloc returns a pointer (of cast type) to an area of memory with size byte-size.

Example:

x=(int*)malloc(100*sizeof(int));

On successful execution of this statement a memory equivalent to 100 times the area of int bytes is reserved and the address of the first byte of memory allocated is assigned to the pointer x of type int

Allocating multiple blocks of memory:

Calloc is another memory allocation function that is normally used to request multiple blocks of storage each of the same size and then sets all bytes to zero. The general form of calloc is: ptr=(cast-type*) calloc(n,elem-size);

The above statement allocates contiguous space for n blocks each size of elements size bytes. All bytes are initialized to zero and a pointer to the first byte of the allocated region is returned. If there is not enough space a null pointer is returned.

Releasing the used space:

Compile time storage of a variable is allocated and released by the system in accordance with its storage class. With the dynamic runtime allocation, it is our responsibility to release the space when it is not required. The release of storage space becomes important when the storage is limited. When we no longer need the data we stored in a block of memory and we do not intend to use that block for storing any other information, we may release that block of memory for future use, using the free function.

free(ptr);

ptr is a pointer that has been created by using malloc or calloc.

To alter the size of allocated memory:

The memory allocated by using calloc or malloc might be insufficient or excess sometimes in both the situations we can change the memory size already allocated with the help of the function realloc. This process is called reallocation of memory. The general statement of reallocation of memory is :

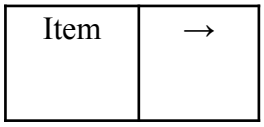
ptr=realloc(ptr,newsize);

This function allocates new memory space of size newsize to the pointer variable ptr ans returns a pointer to the first byte of the memory block. The allocated new block may be or may not be at the same region.

Linked list

A linked list is called so because each of items in the list is a part of a structure, which is linked to the structure containing the next item. This type of list is called a linked list since it can be considered as a list whose order is given by links from one item to the next.

Structure

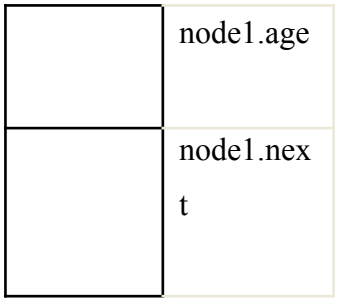


Each item has a node consisting two fields one containing the variable and another consisting of address of the next item(i.e., pointer to the next item) in the list. A linked list is therefore a collection of structures ordered by logical links that are stored as the part of data.

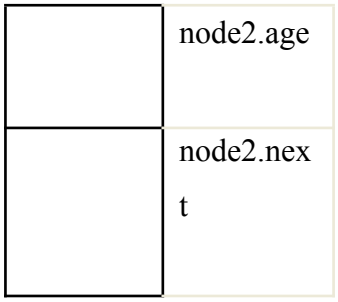
Consider the following example to illustrator the concept of linking. Suppose we define a structure as follows

```
struct linked_list
{
float age;
struct linked_list *next;
}
struct Linked_list node1,node2;
```

this statement creates space for nodes each containing 2 empty fieldsnode1



node2



The next pointer of node1 can be made to point to the node 2 by the same statement.
node1.next=&node2;

This statement stores the address of node 2 into the field node1.next and this establishes a link between node1 and node2 similarly we can combine the process to create a special pointer value called null that can be stored in the next field of the last node

Advantages of Linked List:

A linked list is a dynamic data structure and therefore the size of the linked list can grow or shrink in size during execution of the program. A linked list does not require any extra space therefore it does not waste extra memory. It provides flexibility in rearranging the items efficiently.

The limitation of linked list is that it consumes extra space when compared to a array since each node must also contain the address of the next item in the list to search for a single item in a linked list is cumbersome and time consuming.

Types of linked list:

There are different kinds of linked lists they are
Linear singly linked list
Circular singly linked list
Two way or doubly linked list
Circular doubly linked list.

Applications of linked lists:

Linked lists concepts are useful to model many different abstract data types such as queues stacks and trees. If we restrict the process of insertions to one end of the list and deletions to the other end then we have a mode of a queue that is we can insert an item at the rear end and remove an item at the front end obeying the discipline first in first out. If we restrict the insertions and deletions to occur only at one end of list the beginning then the model is called stacks. Stacks are all inherently one-dimensional. A tree represents a two dimension linked list. Trees are frequently encounters in every day life one example are organization chart and the other is sports tournament chart.

FILES

Many applications require that information be written to or read from any auxiliary memory device. Such information is stored on the memory device is in the form of a ***data file.*** Thus, data files allow us to store information permanently, and to access and alter that information whenever necessary .There are two different types of data files, called ***stream-oriented*** data files ,and ***system oriented*** data files.

Stream oriented data files can be subdivided into two categories. In the first category are ***text files*** consisting of consecutive characters. These characters can be interpreted as individual data items, or as components of strings or numbers. The

second category of stream-oriented data files, often referred to as ***unformatted data files***, organizes data in to blocks containing contiguous bytes of information. These blocks represent more complex data structures, such as arrays and structures.

System oriented data files are more closely related to the computer's operating system than stream oriented data files. They are some what more complicated to work with ,though their use may be more efficient for certain kinds of applications.

Different file operations in C programming are as follows-

- Open a file
- Read the file or write the data in the file
- Close the file.

File operation functions in C:

Function Name	**Operation**
fopen()	Creates a new file for use Opens a new existing file for use
fclose	Closes a file which has been opened for use
getc()	Reads a character from a file
putc()	Writes a character to a file
fprintf()	Writes a set of data values to a file
fscanf()	Reads a set of data values from a file
getw()	Reads a integer from a file
putw()	Writes an integer to the file
fseek()	Sets the position to a desired point in the file
ftell()	Gives the current position in the file

rewind()	Sets the position to the begining of the file

OPENING OF A FILE

When we store a record in the file then at first we need a temporary area in the memory where we store the data/records then we transfer it to file.For storing these records in the memory, we use the pointer which points the starting address where this data/record is stored. We write this as-

FILE *p;

Here p is a pointer of file type. For declaring any variable to file type pointer ,it is necessary to write FILE in capital and then pointer variable name.

For opening a file we use library function fopen().First we declare pointer variable and fopen() as file type pointer.We write this as –

FILE *p,*fopen();

Then p=fopen(“filename”,mode);

Here filename is the name of data file where data/record is stored. Mode decides which operation(read,write or append) is to be performed with the data file.

MODES

1·write(w)

This mode open a new file for writing a record, if the filename already exists then using this mode, the previous data/records are erased and the new data/record entered is written in to the file.

Ex-

p=fopen(“rec.dat”,”w”);

Here rec.dat is the filename and w is the mode.

2·append(a)

This mode open a file for appending a data/record. If the file does not exist then the work of this mode is same as “w” mode.

Ex-

p=fopen(“rec.dat”,”a”);

Here rec.dat is the filename and can be already exist or a new file.

3·read(r)

This mode is used for opening a file for reading purpose only.

Ex-

p=fopen(“rec.dat”,”r”);

If the file rec.dat does not exist then compiler return NULL to the file pointer.

4·write+read(w+)

This mode is used both for reading and writing purpose.This is same as the “w” mode but can also read the record which is stored in the file.

Ex-

p=fopen(“rec.dat”,”w+”);

5·append+read(a+)

This mode is used both for reading and appending purpose. This is same as the “a” mode but can also read the record which is stored in the file.

Ex-

p=fopen(“rec.dat”,”a+”);

6·read+write(r+)

This mode is used both for reading and writing purpose.We can read the record and also write the record in the file.

Ex-

p=fopen(“rec.dat”,”r+”);

CLOSING A FILE

The files which are opened from the fopen() function must be closed at end of the program.this is written as-

fclose(p);

if the opening file is more than one then we close all the file.

fclose(p1);

fclose(p2);

etc…

\FILE INPUT –OUTPUT FUNCTIONS

1·fprintf()

This function is same as the printf() function but it writes the data into the file,so it has one more parameter that is the file pointer.

Syntax-

fprintf(fptr,”contol character”,variable-names);

/* program to understand the use of fprintf() */

```
main()
{
 FILE *fopen(),*p;
 char name[10];
```

```
        p=fopen("rec.dat","w+");
            printf("Enter your name");
            scanf("%s",name);
    fprintf(p,"My Name is %s ",name);
    fclose(p);
     }
```

2·fscanf()

This function is same as the scanf() function but this reads the data from the file ,so this has one or more parameter that is the file pointer.

Syntax- fscanf(fptr,"control character",&variable-names);

```
/*Program to understand the use of fscanf() */
    main()
    {
     FILE *fopen(),*p;
     char name[10];int sal;
        p=fopen("rec.dat","r");
            fscanf(p,"%s %d",name,&sal);
    printf("NAME\t  SALARY\n");
    while(! feof (p))
    {
    printf("%s\t%d\n",name,sal);
```

```
        fscanf(p,"%s%d",name,&sal);
        }
        fclose(p);

  }
```

3·fgetc()

This function is same as the getc() function.It also read a single character from a given file and increments the file pointer position. It returns EOF,If the end of the of the file is reached or it encounters an error.

Syntax-

```
        fgetc(fptr);
        ch=fgetc(fptr);
        where fptr is a file pointer
```

```
/*Program to understand the use of fgetc() function */
 main()
        {   FILE *fopen(),*p;
         int ch;
         if((p=fopen("rec.dat","r"))!=NULL)
              {while((ch=fgetc(p))!=EOF)
           printf("%c",ch);
        }
        fclose(p);
        }
```

4·fputc()

This function writes the character to the specified stream at the current file position and increments the file position indicator.

Syntax-fputc(ch,fptr);

where fptr is a file pointer and ch is a variable written to the file which is pointed by the file pointer.

```
 /*Program to understand the use of fputc() function */
main()
        {   FILE *fopen(),*fptr,*fptr1;
                char name[10],ch;
         printf("Enter the file name");
                scanf("%s",name);
         if((fptr=fopen(name,"r"))!=NULL)
              {  fptr1=fopen("rec.txt","w");
                 while(ch=fgetc(fptr)!=EOF)
          fputc(ch,fptr1);
        }
        fclose(fptr);
        fclose(fptr1);
        }
```

5·fgets()

This function is used to read a string from a given file and copies the string to a memory location which is referenced by an array.

Syntax-fgets(sptr,max,fptr) ;

When sptr is a string pointer,which points to an array,max is the length of the array and fptr is a file pointer which points to a given file.

```
/*program to understand the use of fgets() */
main()
        {
           FILE *fopen(),*fptr;
                char name[10],arr[50];
         int i=0;
         printf("Enter the file name");
                scanf("%s",name);
         if((fptr=fopen(name,"r"))!=NULL)
              {  if(fgets(arr,50,fptr)!=NULL)
                        while(arr[i]!='\0'){
           putchar(arr[i]);
           i++;}
        }
        fclose(fptr);
        }
```

6·fputs()

This function is used to write a string to a given file.

Syntax-fputs(sptr,fptr)

Where sptr is a string pointer,which points to an array and fptr is a file pointer which points to a given file.

```
/*program to understand the use of fputs() */
        main()
        {   FILE *fopen(),*fptr;
                char name[10],arr[50];
         printf("Enter the file name");
                scanf("%s",name);
         if((fptr=fopen(name,"w"))!=NULL)
             {
                printf("The string is");
                gets (arr);
                fputs(arr,fptr);
        }
        fclose(fptr);
        }
```

Block read/write

It is useful to store the block of data in to the file rather than individual elements. Each block has some fixed size,it may be structure or of an array.It is possible that a

data file has one or more structures or arrays .So it is easy to read the entire block from file or write entire block to the file .there are two useful function for this purpose-:

fread()

This function is used to read an entire block from a given file.

Syntax-fread(ptr,size,nst,fptr);

Where ptr is a pointer which points to the array which receives the structure ,size is the size of the structure,nst is the number of structure and fptr is a file pointer.

```
 /*program to understand the use of fread() */
main()
{
 struct rec{
 int code;char name[20];
        }person[10];
  FILE *fptr;
  int i=0,j;
  char str[15];
                printf("Enter the file name");
                scanf("%s",str);
         if((fptr=fopen(str,"r"))!=NULL)
              {
                while(!feof(fptr))
```

```
{fread(&person,sizeof(person),1,fptr);
 i++;}
}
for(j=0;j<I;j++)
{
 printf("Code %d\t",person[j].code);
 printf("Name %s\t",person[j].name);
}
close( fptr);
```

fwrite()

This function is used for writing an entire block to a given file.

Syntax-fwrite(ptr,size,nst,fptr);

Where ptr is a pointer which points to the array of structure in which data is written, size is the size of the structure.nst is the number of structure and fptr is the file pointer.

/* Program to understand the use of fwrite() */

```
main()
{
struct rec{
 int code;char name[20];
}person[10];
 FILE *fptr;
 int i,j=0,n;
```

```
  char str[15];
                    printf("Enter the file name");
                    scanf("%s",str);
           if((fptr=fopen(str,"w")!=NULL)
                {
            printf("How many records");
                    scanf("%d",&n);
                    for(i=0;i<n;i++)
                    {printf("code");
                    scanf("%d",&person[i].code);
                    printf("Name");
                    scanf("%s",&person[i].name);}
                    while(j<n)
                    {
                      fwrite(&person,sizeof(person),1,fptr);
                              j++;
                    }
                      }
                      fclose(fptr);
 }
```

OTHER FILE FUNCTIONS

feof()

The macro feof() is used for detecting whether the file pointer is at the end of file or not.It returns nonzero if the file pointer is at the end of file, otherwise it returns zero.

ferror()

The macro ferror() is used for detecting whether an error occur in the file on file pointer or not. It returns the value nonzero if an error,otherwise it returns zero.

Syntax-ferror(fptr);

```
/*Program to understand the use of ferror() */
        main()
        {
  FILE *fptr;
  char name[15],ch;
                printf("Enter the file name");
                scanf("%s",name);
         if((fptr=fopen(name,"r")!=NULL)
             while((ch=getc(fptr))!=EOF)
           {
          printf("%c",ch);
                          if(ferror(fptr))
          {
           printf("Errorin file");
           exit(1);
                              }
                    }
```

```
        fclose(fptr);
    }
```

unlink()

This function is used for deleting the file from the directory.

Syntax-unlink(filename);

Ex-

```
    if((fptr=fopen(name,"r")!=NULL)
    unlink(name);
```

RADOM ACCESS TO THE FILE

There is no need to read each record sequentially,if we want to access a particular record. C supports these functions for random access file processing-

-fseek()

-ftell()

-rewind()

fseek()

This function is used for setting the pointer position in the file at the specified byte

Syntax-fseek(file pointer,displacement,pointer position);

File pointer

Here file pointer is the pointer which points to the file

Displacement

Displacement is positive or negative.This is the number of bytes which are skipped backward (if negative) or forward (if positive) from current position

Ex-

fseek(p,10L,0)

0 means pointer position is beginning of the file ,from this statement pointer position is skipped 5 bytes forward from the current position.

ftell()

This function returns the value of the current pointer position in the file.The value is count from the beginning of the file.

Ex-

```
p=fopen("text","r");
fseek(p,38L,0);
        c=fget(p);
        while(!feof(p))
        {printf("%c",c);
        printf("%d",ftell(p));
        c=fgetc(p);
        }fclose(p);
```

rewind()

This function is used to move the file pointer to the beginning of the given file.This can be written as

Syntax-rewind(fptr);

Ex-

```
p=fopen("text","r");
fseek(p,2L,0);
        rewind(p);
        c=fgetc(p);
        while(!feof(p))
        {printf("%c",c);
        c=fgetc(p);

        }fclose(p)
```