RPL2B001 PROGRAMMING FOR PROBLEM SOLVING USING C

Course Outcomes

The student will learn

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

Contact hrs : 40

Detailed contents

Unit 1:

Introduction to Programming (4 lectures)

Introduction to components of a computer system (disks, memory, processor, where a program is stored and executed, operating system, compilers etc.) - **(1 lecture).**

Idea of Algorithm: steps to solve logical and numerical problems. Representation of Algorithm: Flowchart/Pseudocode with examples. **(1 lecture)**

From algorithms to programs; source code, variables (with data types) variables and memory locations, Syntax and logical errors in compilation, object and executable code- **(2 lectures)**

Unit 2:

Arithmetic expressions, operators and precedence (2 lectures)

Conditional Branching and Loops (6 lectures)

Writing and evaluation of conditionals and consequent branching **(3 lectures)**

Iteration and loops **(3 lectures)**

Arrays (6 lectures)

Arrays (1-D, 2-D), Character arrays and Strings

Unit 3:

Function (5 lectures)

Functions (including using built in libraries), Parameter passing in functions, call by value, Passing arrays to functions: idea of call by reference

Recursion (4 lectures)

Recursion as a different way of solving problems. Example programs, such as Finding Factorial, Fibonacci series, Ackerman function etc. Quick sort or Merge sort.

Unit 4:

Pointers (2 lectures)

Idea of pointers, Defining pointers, Use of Pointers in self-referential structures, notion of linked list (no implementation). Dynamic memory allocation.

Structure (4 lectures)

Structures, Defining structures and Array of Structures, Structure vs Union.

File handling: ASCII and binary Files **(1 lecture)**

Unit 5:

Basic Algorithms (6 lectures)

Searching (Linear and Binary), Basic Sorting Algorithms (Bubble, Insertion, and Selection), Concepts of time and space complexity.

Assignments: All lab should be handled in UNIX/LINUX environment.

Minimum 3-5 problems should be implemented from Unit-2 to Unit-5 each..

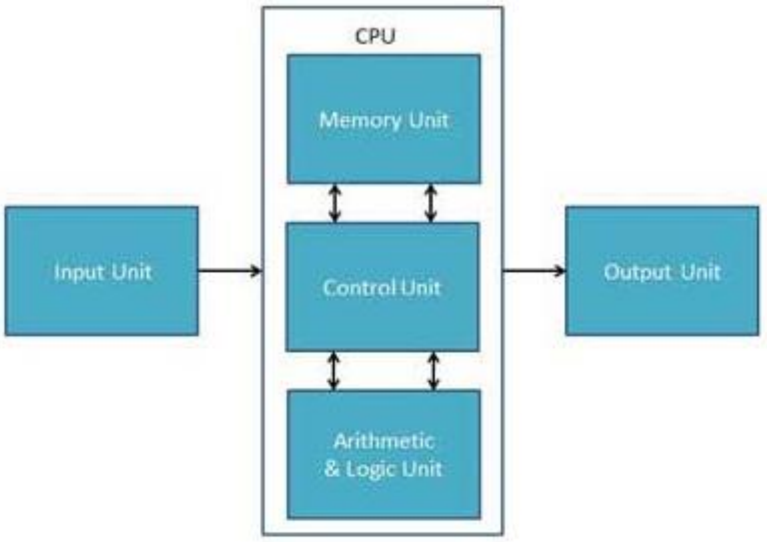
Suggested Text Books

(i) Reema Thareja, Introduction to C Programming, 2nd Edition, Oxford University Press.

(ii) E. Balaguruswamy, Programming in ANSI C, Tata McGraw-Hill

UNIT 1
INTRODUCTION TO PROGRAMMING

1. Introduction to components of a computer system



INPUT UNIT:

This unit contains devices that help to enter data into computer.

CENTRAL PROCESSING UNIT(CPU):

This is also considered as brain of computer. It has three parts 1. ALU 2. CU 3. MU

OUTPUT UNIT:

This unit contains all devices that supply information to the outside world.

STORAGE UNIT:

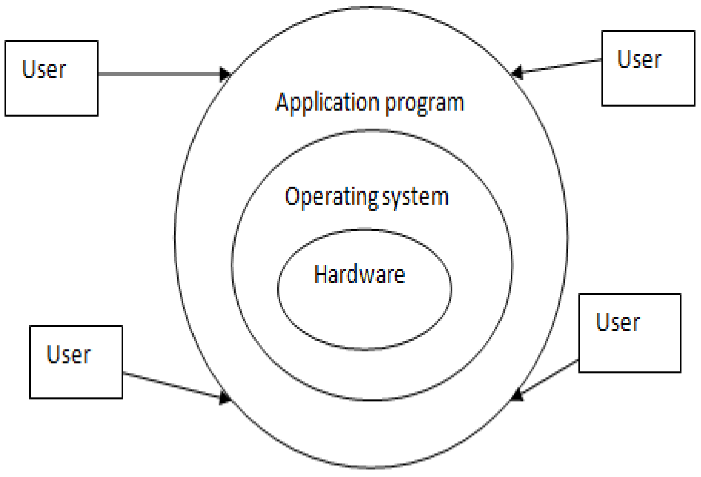
The data and instructions that are supplied to the computer and supplied to outside the world are to be stored separately in the computer system. The storage unit or main storage is an integrated part of a computer system.

Introduction to Operating System

The operating system is system software which is loaded into memory of computer at the time of booting process. It remains in main memory all the time to control all the functionality of the computer system.

These are the functionality of Operating System.

1. DISK MANAGEMENT
2. PROCESS MANAGEMENT
3. I/O MANAGEMENT
4. MEMORY MANAGEMENT
5. DEVICE MANAGEMENT
6. FILE MANAGEMENT



AVAILABLE OS TODAY

a. MS-DOS
b. WINDOWS
c. LINUX
d. ANDROID

Types of Computer Languages and Language Translators

1. Machine Language
2. Assembly Language
3. High level Language

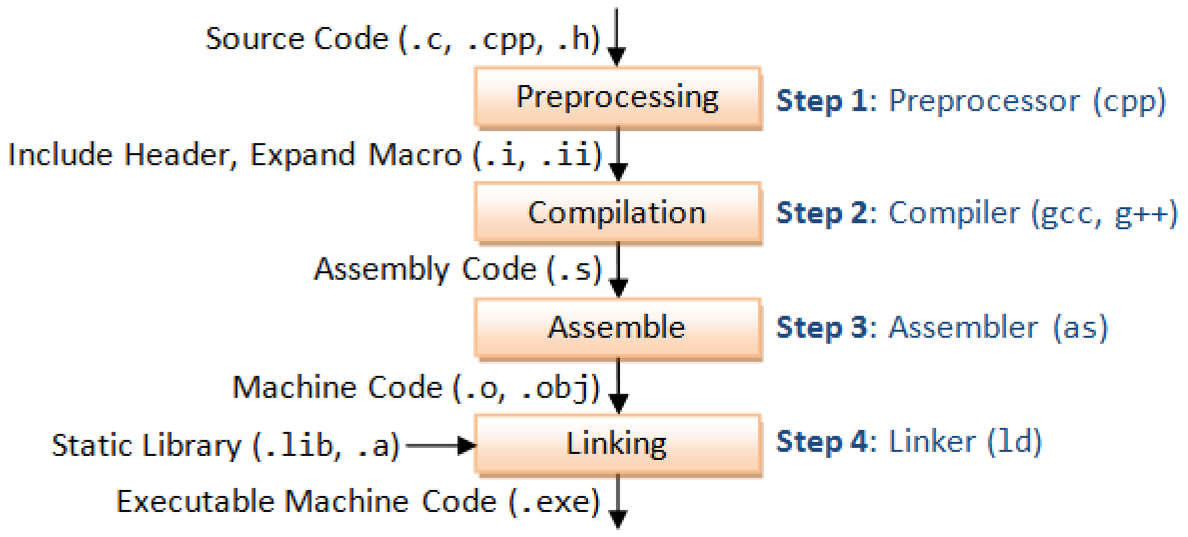
CONCEPT OF ASSEMBLER, COMPILER, LOADER, LINKER

Compiler:
The programs written in any high-level programming language (C or Pascal) needs to be converted into machine language or its equivalent. This is achieved by using a complier.

Assembler:
The program written in assembly language are symbolic representation of machine language. Assembler the language translator translate symbolic language to machine language.

Linker:
A linker is a system program that links together several object modules to form a single executable program.

Loader:
The part of operating system that load an executable file that reside in computer storage unit into main memory and execute it,

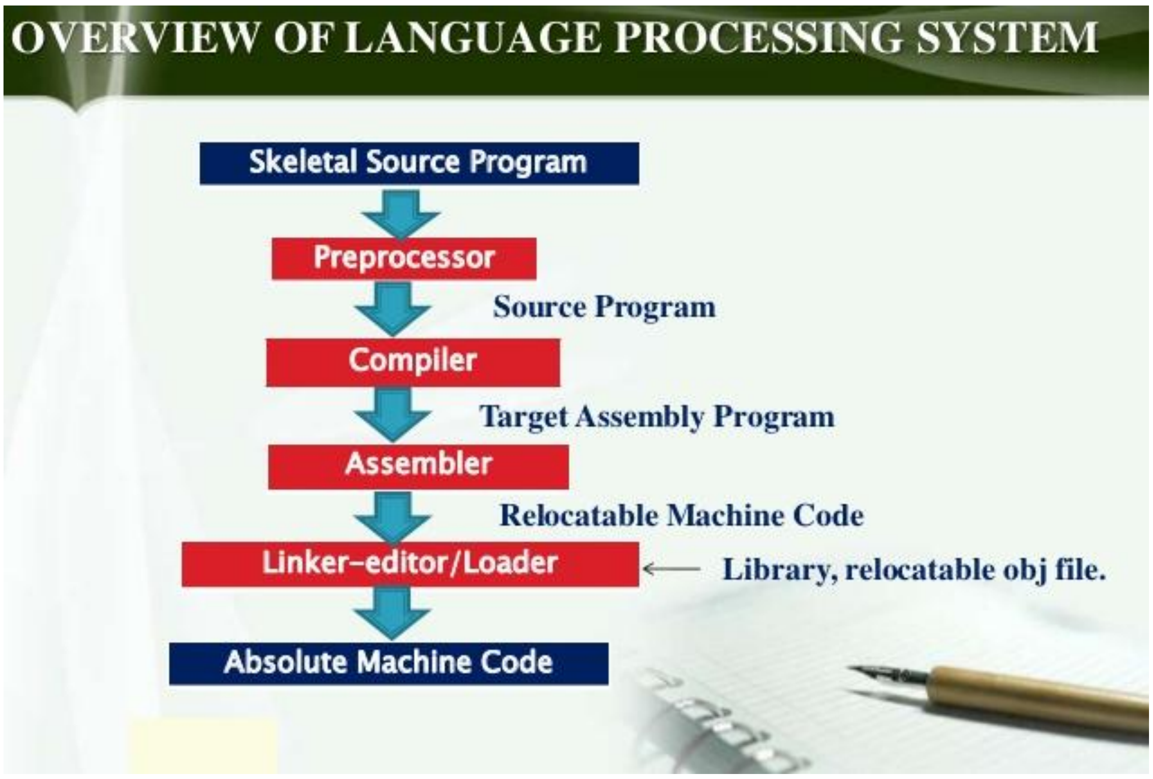

Source Code (.c, .cpp, .h)
Preprocessing
Step 1: Preprocessor (cpp)
Include Header, Expand Macro (.i, .ii)
Compilation
Step 2: Compiler (gcc, g++)
Assembly Code (.s)
Assemble
Step 3: Assembler (as)
Machine Code (.o, .obj)
Static Library (.lib, .a)
Linking
Step 4: Linker (ld)
Executable Machine Code (.exe)

Source code

A program written in a high-level language is called source code. Source code is also called source program. Computer cannot understand the statements of high-level language. The source code cannot be executed by computer directly. It is converted into object code and then executed.

Object code

A program in machine language is called objet code. It is also called object program or machine code. Computer understands object code directly.



Idea of Algorithm: steps to solve logical and numerical problems. Representation of Algorithm: Flowchart/Pseudo code with examples. (1 lecture)

An algorithm is a step by step logical method to solve a problem. Efficiency of algorithm is how fast it can produce the correct result. The efficiency of algorithm is measured through time-complexity and space-complexity. Analysis of algorithm: the time and space complexity is done through the asymptotic notation.

Characteristics of an Algorithm

1. Input: it may take zero or more input
2. Output: it should atleast produce one output
3. Definiteness: each instruction must be clear and well defined
4. Finiteness: it should be finite set of instructions
5. Effectiveness: operation must be simple and carried out in finite time

Pseudo code is a simple way of writing programming code in English. Pseudocode is not an actual programming language. It uses short phrases to write code for programs before you actually create it in a specific language.

Flowchart: A flowchart is a pictorial representation of an algorithm in which the steps are drawn in the form of different shapes of boxes and the logical flow is indicated by interconnecting arrows. The boxes represent operations and the arrows represent the sequence in which the operations are implemented. The primary purpose of the flowchart is to help the programmer in understanding the logic of the program.

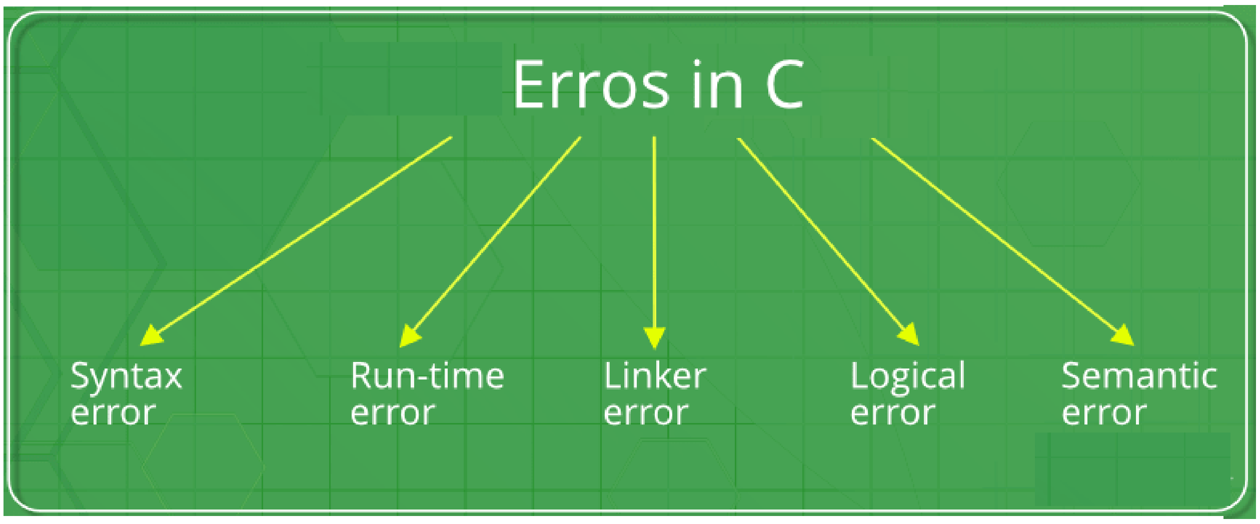
Algorithms and **flowcharts** are two tools a software developer uses when creating new programs. An algorithm is a step-by-step recipe for processing data; it could be a process an online store uses to calculate discounts, for example. A flowchart graphically represents the steps a program or set of programs takes to process data.

Program is a sequence of instructions in a particular programming language, written to perform a specified task on a computer.

ALGORITHM:

```
Find largest number out of three numbers A, B, and C
Step 1:         Start
Step 2:  Read three numbers say A,B,C
Step 3 : Find the larger number between A and B and store it in MAX_AB.
Step 4 : Find the larger number between MAX_AB and C and store it in MAX.
Step 5 : Display MAX
Step 6 : Stop
```

Errors in C



Run in Turbo C compiler(16-bit)

Syntax errors: Errors that occur when you **violate the rules** of writing C syntax. These are also called compile-time error. Most frequent syntax errors are:

Missing Parenthesis (**}**)

Printing the value of variable without declaring it

Missing semicolon like this **;**

```
/* syntax error */
#include<stdio.h>
int  main()
{
   int  x = 10;
   int  y = 15;

   printf("%d", (x, y)) // semicolon missed
return 0;
}
```

Error:

```
error: expected expression before '.' token
    while(.)
```

Run-time Errors : Errors which occur during program execution(run-time) after successful compilation are called run-time errors.

```
/* run-time error   */
#include<stdio.h>
void  main()
{
   int  n = 9, div  = 0;
   /*wrong logic
     number is divided by 0,
     so this program abnormally terminates */
   div  = n/0;

   printf("resut = %d", div);
}
```

Error:

```
warning: division by zero [-Wdiv-by-zero]
     div = n/0;
```

Linker Errors: These error occurs when after compilation we link the different object files with main's object using *Ctrl+F9* key(RUN) in turbo C.

```
/* linker error */
#include<stdio.h>

void Main() // Here Main() should be main()
{
   int a = 10;
   printf("%d", a);
}
```

Error:

```
(.text+0x20): undefined reference to `main'
```

Logical Errors : On compilation and execution of a program, desired output is not obtained when certain input values are given.

```
/* logical error */
int  main()
{
   int  i = 0;

   /* logical error : a semicolon after loop */
   for(i = 0; i < 3; i++);
   {
      printf("loop ");
      continue;
   }
   getchar();
   return  0;
}
```

No Output

Semantic errors : This error occurs when the statements written in the program are not meaningful to the compiler.

```
/* semantic error */
void main()
{
  int a, b, c;
  a + b = c; //semantic error
}
```

Error

```
error: lvalue required as left operand of assignment
a + b = c; //semantic error
```

UNIT 2
DATA TYPE, EXPRESSIONS, OPERATORS AND PRECEDENCE (2 LECTURES)
CONDITIONAL BRANCHING AND LOOPS (6 LECTURES)
Arrays (6 lectures)

VARIABLE, CONSTANT, DATA TYPE, EXPRESSION, OPERATORS AND PRECEDENCE (ANSI)

Variables and constants are the basic data objects manipulated in a program. Declarations list the variables to be used, and state what type they have and perhaps what their initial values are. Operators specify what is to be done to them. Expressions combine variables and constants to produce new values. The type of an object determines the set of values it can have and what operations can be performed on it. These building blocks are the topics of this chapter. The ANSI standard has made many small changes and additions to basic types and expressions. There are now signed and unsigned forms of all integer types, and notations for unsigned constants and hexadecimal character constants. Floating-point operations may be done in single precision; there is also a long double type for extended precision. String constants may be concatenated at compile time. Enumerations have become part of the language, formalizing a feature of long standing. Objects may be declared const, which prevents them from being changed. The rules for automatic coercions among arithmetic types have been augmented to handle the richer set of types.

Variable Names:

There are some restrictions on the names of variables and symbolic constants. Names are made up of letters and digits; the first character must be a letter. The underscore ‘‘_’’ counts as a letter; it is sometimes useful for improving the readability of long variable names. Don’t begin variable names with underscore, however, since library routines often use such names.
Upper and lower case letters are distinct, so x and X are two different names. Traditional C practice is to use lower case for variable names, and all upper case for symbolic constants. Keywords like if, else, int, float, etc., are reserved: you can’t use them as variable names.

Data Types and Sizes:

There are only a few basic data types in C:

char	a single byte, capable of holding one character in the local character set
int	an integer, typically reflecting the natural size of integers on the host machine
float	single-precision floating point
double	double-precision floating point

In addition, there are a number of qualifiers that can be applied to these basic types. short and long apply to integers:

```
short int sh;
long int counter;
```

Constants:

An integer constant like 1234 is an int. A long constant is written with a terminal l (ell) or L, as in 123456789L; an integer constant too big to fit into an int will also be taken as a long. Unsigned constants are written with a terminal u or U, and the suffix ul or UL indicates unsigned long. Floating-point constants contain a decimal point (123.4) or an exponent (1e-2) or both; their type is double, unless suffixed. The suffixes f or F indicate a float constant; l or L indicate a long double. The value of an integer can be specified in octal or hexadecimal instead of decimal. A leading 0 (zero) on an integer constant means octal; a leading 0x or 0X means hexadecimal. For example, decimal 31 can be written as 037 in octal and 0x1f or 0x1F in hex. Octal and hexadecimal constants may also be followed by L to make them long and U to make them unsigned: 0XFUL is an unsigned long constant with value 15 decimal. A character constant is an integer, written as one character within single quotes, such as ’x’. Certain characters can be represented in character and string constants by escape sequences like \n (newline); these sequences look like two characters, but represent only one.

\a	alert (bell) character	\\	backslash
\b	backspace	\?	question mark
\f	formfeed	\'	single quote
\n	newline	\"	double quote
\r	carriage return	*ooo*	octal number
\t	horizontal tab	\x*hh*	hexadecimal number
\v	vertical tab		

The character constant ’\0’ represents the character with value zero, the null character. ’\0’ is often written instead of 0 to emphasize the character nature of some expression, but the numeric value is just 0.

Expression

A constant expression is an expression that involves only constants. Such expressions may be evaluated at during compilation rather than run-time, and accordingly may be used in any place that a constant can occur, as in

```
#define MAXLINE 1000
char line[MAXLINE+1];
```

A string constant, or string literal, is a sequence of zero or more characters surrounded by double quotes, as in

"I am a string"

The quotes “ “ or ‘’are not part of the string, but serve only to delimit it.

The standard library function strlen(s) returns the length of its character string argument s, excluding the terminal ’\0’. Here is our version:

```
/* strlen: return length of s */
int strlen(char s[])
{
int i;
while (s[i] != ’\0’)
++i;
return i;
}
```

strlen and other string functions are declared in the standard header <string.h>.

There is one other kind of constant, the enumeration constant. An enumeration is a list of constant integer values, as in

```
        enum boolean { NO, YES };
```

The first name in an enum has value 0, the next 1, and so on, unless explicit values are specified. If not all values are specified, unspecified values continue the progression from the last specified value, as the second of these examples:

```
        enum escapes { BELL = '\a', BACKSPACE = '\b', TAB = '\t', NEWLINE = '\n', VTAB = '\v',
                       RETURN = '\r' };
        enum months { JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC };
/* FEB = 2, MAR = 3, etc. */
```

Names in different enumerations must be distinct. Values need not be distinct in the same enumeration.

Declarations:

All variables must be declared before use, although certain declarations can be made implicitly by content. A declaration specifies a type, and contains a list of one or more variables of that type, as in

```
 int lower, upper, step;
 char c, line[1000];
```

Variables can be distributed among declarations in any fashion; the lists above could well be written as

```
 int lower;
 int upper;
 int step;
 char c;
 char line[1000];
```

A variable may also be initialized in its declaration. If the name is followed by an equals sign and an expression, the expression serves as an initializer, as in

```
 char esc = '\\';
 int i = 0;
 int limit = MAXLINE+1;
 float eps = 1.0e-5;
```

The qualifier const can be applied to the declaration of any variable to specify that its value will not be changed. For an array, the const qualifier says that the elements will not be altered.

```
        const double e = 2.71828182845905;
        const char msg[] = "warning: ";
```

The const declaration can also be used with array arguments, to indicate that the function does not change that array:

```
        int strlen(const char[]);
```

Arithmetic Operators

The binary arithmetic operators are +, -, *, /, and the modulus operator %. Integer division truncates any fractional part. The expression

```
        x % y
```

produces the remainder when x is divided by y.

For example, a year is a leap year if it is divisible by 4 but not by 100, except that years divisible by 400 are leap years. Therefore

```
if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0)
printf("%d is a leap year\n", year);
else
printf("%d is not a leap year\n", year);
```

The % operator cannot be applied to a float or double.

The binary + and - operators have the same precedence, which is lower than the precedence of *, / and %, which is in turn lower than unary + and -. Arithmetic operators associate left to right.

Relational and Logical Operators

The relational operators are

```
>    >=    <    <=
```

They all have the same precedence. Just below them in precedence are the equality operators:

```
==    !=
```

Relational operators have lower precedence than arithmetic operators, so an expression like i < lim-1 is taken as i < (lim-1), as would be expected. More interesting are the logical operators && and ||. Expressions connected by && or || are evaluated left to right, and evaluation stops as soon as the truth or falsehood of the result is known. The precedence of && is higher than that of ||, and both are lower than relational and equality operators, so expressions like

```
i < lim-1 && (c=getchar()) != ’\n’ && c != EOF
```

need no extra parentheses. But since the precedence of != is higher than assignment, parentheses are needed in

```
(c=getchar()) != ’\n’
```

to achieve the desired result of assignment to c and then comparison with ’\n’.

Type Conversions:

When an operator has operands of different types, they are converted to a common type according to a small number of rules. In general, the only automatic conversions are those that convert a ‘‘narrower’’ operand into a ‘‘wider’’ one without losing information, such as converting an integer into floating point in an expression like f + i. Expressions that don’t make sense, like using a float as a subscript, are disallowed. Expressions that might lose information, like assigning a longer integer type to a shorter, or a floating-point type to an integer, may draw a warning, but they are not illegal.

```
/* atoi: convert s to integer */
int atoi(char s[])
{
int i, n;
n = 0;
for (i = 0; s[i] >= ’0’ && s[i] <= ’9’; ++i)
n = 10 * n + (s[i] - ’0’);
return n;
}
```

As we discussed the expression

```
s[i] - ’0’
```

gives the numeric value of the character stored in s[i], because the values of ’0’, ’1’, etc., form a contiguous increasing sequence.

Another example of char to int conversion is the function lower,

```
/* lower: convert c to lower case; ASCII only */
```

```
int lower(int c)
{
if (c >= ’A’ && c <= ’Z’)
return c + ’a’ - ’A’;
else
return c;
}
```

Relational expressions like i > j and logical expressions connected by && and || are defined to have value 1 if true, and 0 if false. Thus the assignment

d = c >= ’0’ && c <= ’9’

sets d to 1 if c is a digit, and 0 if not

Finally, explicit type conversions can be forced (‘‘coerced’’) in any expression, with a unary operator called a cast. In the construction

(type name) expression

the expression is converted to the named type

Increment and Decrement Operators:

C provides two unusual operators for incrementing and decrementing variables. The increment operator ++ adds 1 to its operand, while the decrement operator -- subtracts 1.

Bitwise Operators

C provides six operators for bit manipulation; these may only be applied to integral operands, that is, char, short, int, and long, whether signed or unsigned.

&	bitwise AND
|	bitwise inclusive OR
^	bitwise exclusive OR
<<	left shift
>>	right shift
~	one’s complement (unary)

The bitwise AND operator & is often used to mask off some set of bits, for example

n = n & 0177;

sets to zero all but the low-order 7 bits of n.

Assignment Operators and Expressions:

An expression such as

i = i + 2

in which the variable on the left side is repeated immediately on the right, can be written in the compressed form

i += 2

The operator += is called an assignment operator.

Most binary operators (operators like + that have a left and right operand) have a corresponding assignment operator op=,

where op is one of

+ - * / % << >> & ^ |

If expr1 and expr2 are expressions, then

expr1 op= expr2

is equivalent to

expr1 = (expr1) op (expr2)

except that expr1 is computed only once. Notice the parentheses around expr2:
x *= y + 1
means
x = x * (y + 1)

Conditional Expressions

The statements

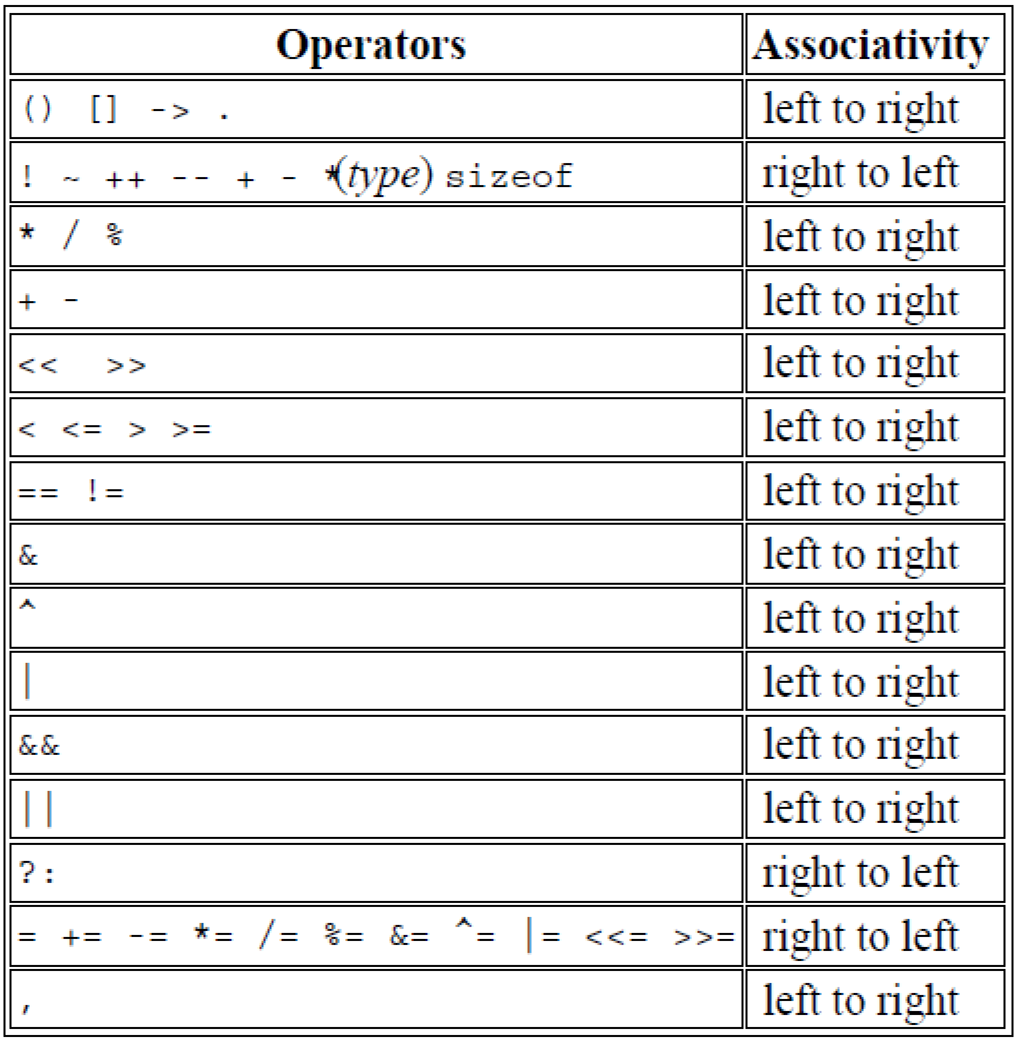
```
if (a > b)
z = a;
else
z = b;
```

compute in z the maximum of a and b. The conditional expression, written with the ternary operator ‘‘?:’’, provides an alternate way to write this and similar constructions. In the expression

expr1 ? expr2 : expr3

the expression expr1 is evaluated first. If it is non-zero (true), then the expression expr2 is evaluated, and that is the value of the conditional expression. Otherwise expr3 is evaluated, and that is the value. Only one of expr2 and expr3 is evaluated.

Precedence and Order of Evaluation:



Operators	Associativity
() [] -> .	left to right
! ~ ++ -- + - *(*type*) sizeof	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
?:	right to left
= += -= *= /= %= &= ^= |= <<= >>=	right to left
,	left to right

Multiple Choice Questions

1. Which byte order is followed by Motorola Processors?

   a. Little-Endian b. Big-Endian
   c. Bi-Endian d. None

2. The relational operator == (equality) always returns

   a. 0
   b. 1
   c. 0 or 1
   d. None

3. which modifier doubles range of data types

   a. signed
   b. unsigned
   c. short
   d. long

4. Find out the output

```
void main()
{
        printf("%d",4.5);
}
```

a. 4
b. 5
c. garbage
d. None of these

5. main()

```
{
        float x=4.6;
        if(x==4.6)
                printf("hello");
        else
                printf("bye") ;
}
```

Ans : bye

6. Find out the output

```
void main()
{
    char x=-130;
    char y=-5;
    printf("%d",x+y);
}
```

   a. -135
   b. -3
   c. 7
   d. 121

7. Find out the output

```
void main()
{
        unsigned int x=500;
        int y=-5;
        if(x>y)
```

```
                printf("Hello");
        else
                printf("Hi");
}
Ans: hi
```

8. Find the output

```
void main()
{
        int a=8,x;
        x=++a + ++a + ++a;
        printf("%d",x);
}
```

a. 33
b. 32
c. 31
d. None

9. Bitwise operators are applicable only on

a. integers
b. integers and characters
c. integers and floats
d. integers,floats,and doubles

10. Find the output

```
void main()
{
        int x,y;
        x=10;
        y=sizeof(++x);
        printf("x=%d  y=%d",x,y);
}
```

a. x=10 y=2
b. x=10 y=4
c. Compilation error
d. None of these

11. Which operator is used both as an operator and keyword?

a. Right shifting operator
b. cast operator
c. sizeof operator
d. Token pasting operator

12. Find the output

```
void main()
{
int i,j=1;
for(i=1;i<=5;i++)
{
        while(j<=5)
        {
                printf("%d",i+j);
                if(i==j)
                        break;
                j++;
        }
}
}
```

a. 2 3 4 5 6 7 8,9,10

b. 2 3 4 5 5 6 ,7,8
c. Compilation error
d. None of these

13. Find the output

```
void main()
{
        int num=345,m,sum=0;
        do
        {
                m=num%10;
                num=num/10;
                continue;
                sum=sum*10+m;
        } while(num!=0);
        printf("%d",sum);
}
```

a. 3 4 5
b. 5 4 3
c. 0
d. None of these

14. main()

```
{
        int x;
        int a=5;
        x= 30 || --a;
        printf("%d",a);
}
```

Ans: 5

15. main()

```
{
        int a=5,b=5,c=5;
        if(a==b==c)
                printf("delhi challo");
        else
                printf("do or die");
}
```

Ans: do or Die

16. Which of the following is correct declaration?
a) int age;
b) short age;
c) Long age;
d) All the above

17. Comment on the below statement

```
while(0==0)
{

}
```

a. It has a syntax error
b. It will run for ever
c. It will compare 0 with 0 and they are equal so its exit the loop
d. none

18. main()

```
{
        int i=1;
        while(i<=5)
        {
                printf("%d",i);

                if(i==2)
                        continue;
                i++;
        }
}
Out:1 2 2 2 2 2   …
```

CONDITIONAL BRANCHING AND LOOPS:

1. Statements and Blocks
2. If-Else
3. Else-If
4. Switch
5. Loops - While and For
6. Loops - Do-While
7. Break and Continue
8. Goto and labels

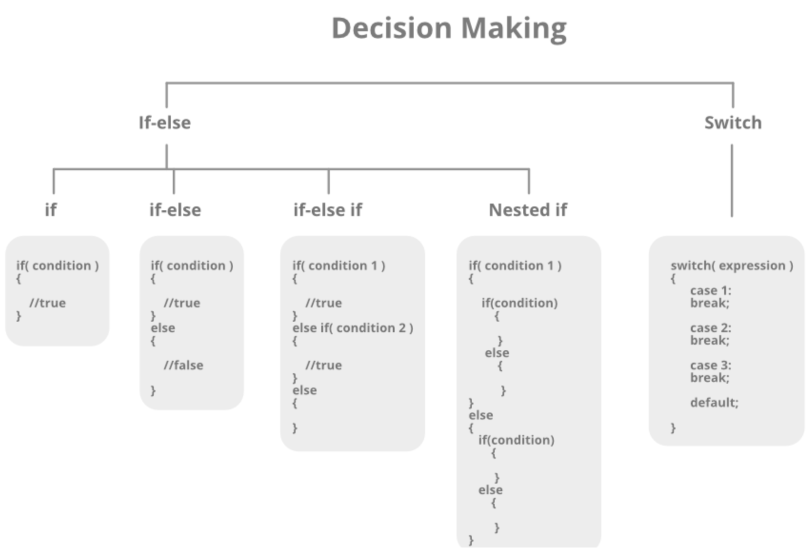
C provides the fundamental control-flow constructions required for well-structured programs: statement grouping, decision making (if-else), selecting one of a set of possible values (switch), looping with the termination test at the top (while, for) or at the bottom (do), and early loop exit (break).

Statements and Blocks:
An expression such as x = 0 or i++ or printf(...) becomes a statement when it is followed by a semicolon, as in

```
x = 0;
i++;
printf(...);
```

In C, the semicolon is a statement terminator.
Block: Braces { and } are used to group declarations and statements together into a compound statement, or block, so that they are syntactically equivalent to a single statement.

Decision Making:



```
if( condition )
{
   //true
}
```

```
if( condition )
{
   //true
}
else
{
   //false
}
```

```
if( condition 1 )
{
   //true
}
else if( condition 2 )
{
   //true
}
else
{
}
```

```
if( condition 1 )
{
   if(condition)
     {
     }
    else
     {
     }
}
else
{
  if(condition)
    {
    }
  else
    {
    }
}
```

```
switch( expression )
{
    case 1:
    break;

    case 2:
    break;

    case 3:
    break;

    default;
}
```

IF STATEMENT IN C

if statement is the most simple decision making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statement is executed otherwise not.

Syntax:

```
if(condition)
{
   // Statements to execute if
   // condition is true
}
```

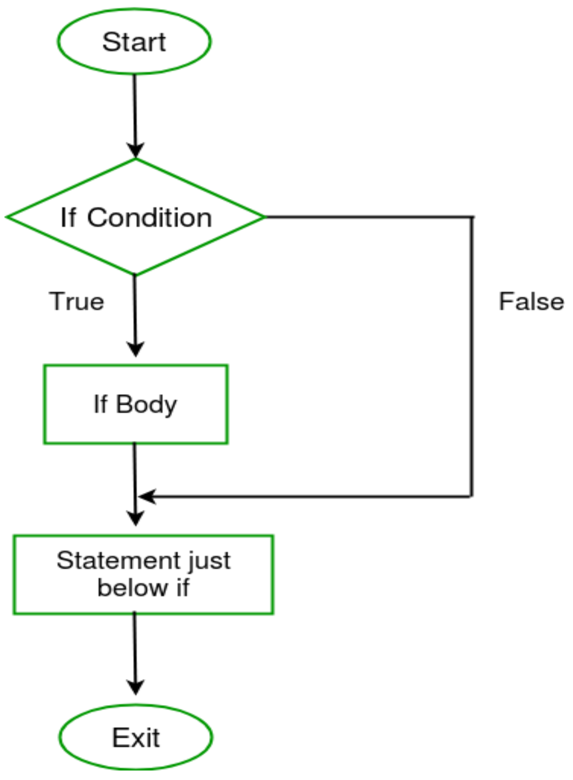
Here, **condition** after evaluation will be either true or false. C if statement accepts boolean values – if the value is true then it will execute the block of statements below it otherwise not. If we do not provide the curly braces '{' and '}' after if(condition) then by default if statement will consider the first immediately below statement to be inside its block.

Example:

```
if(condition)
   statement1;
   statement2;

/*Here if the condition is true, if block
    will consider only statement1 to be inside
    its block. */
```

Flowchart



```
/* C program to illustrate If statement */
#include <stdio.h>

int main() {
    int i = 10;

    if (i > 15)
    {
       printf("10 is less than 15");
    }
    printf("I am Not in if");

return 0;

}
```

Output:

I am Not in if

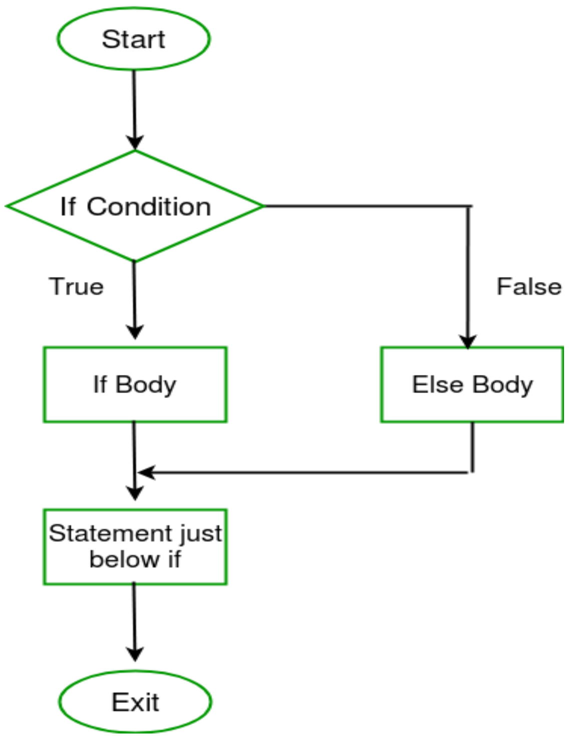
IF-ELSE IN C

The *if* statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But what if we want to do something else if the condition is false. Here comes the C *else* statement. We can use the *else* statement with *if* statement to execute a block of code when the condition is false.

Syntax:

```
if (condition)
{
    /* Executes this block if
       condition is true*/
}
else
{
    /* Executes this block if
       condition is false */
}
```

Flowchart:



Example:

```
/* C program to illustrate If statement */
#include <stdio.h>

int main() {
    int i = 20;

    if (i < 15)
        printf("i is smaller than 15");
    else
        printf("i is greater than 15");

    return 0;
}
```

Output:

i is greater than 15

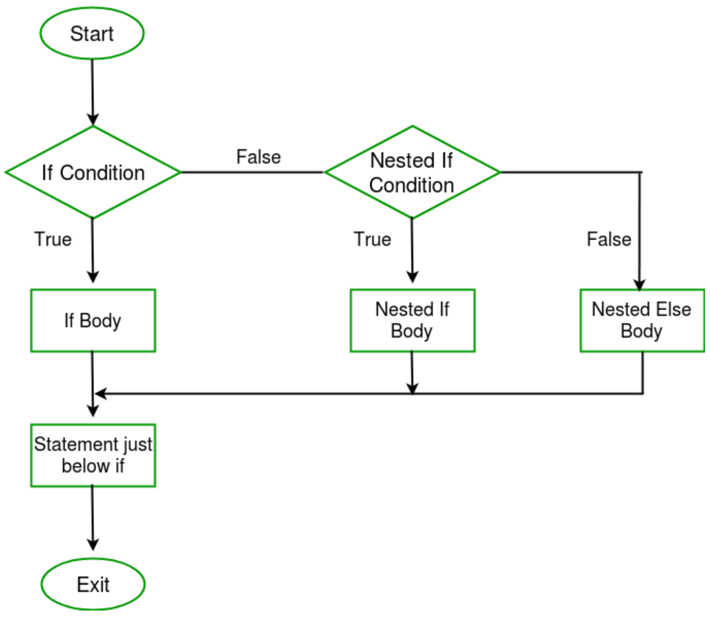
NESTED-IF IN C

A nested if in C is an if statement that is the target of another if statement. Nested if statements means an if statement inside another if statement. Yes, C allows us to nested if statements within if statements, i.e, we can place an if statement inside another if statement.

Syntax:

```
if (condition1)
{
  // Executes when condition1 is true
  if (condition2)
  {
    // Executes when condition2 is true
  }
}
```

Flowchart



Example:

```
/* C program to illustrate nested-if
statement */
#include <stdio.h>

int main() {
    int i = 10;

    if (i == 10)
    {
        /* First if statement */
        if (i < 15)
           printf("i is smaller than 15\n");

        /* Nested - if statement
           Will only be executed if statement above
           is true  */
        if (i < 12)
            printf("i is smaller than 12 too\n");
        else
            printf("i is greater than 15");
    }

    return 0;
}
```

Output:

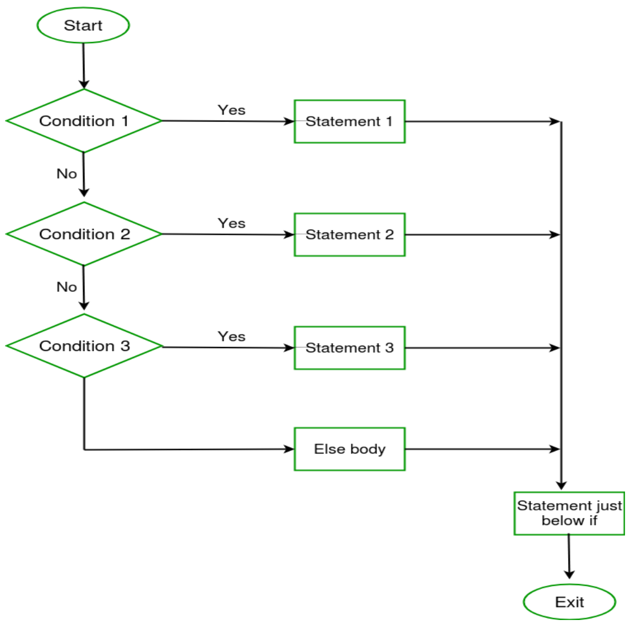
```
i is smaller than 15
i is smaller than 12 too
```

IF-ELSE-IF LADDER IN C

Here, a user can decide among multiple options. The C if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the C else-if ladder is bypassed. If none of the conditions are true, then the final else statement will be executed.

Syntax:

```
if (condition)
    statement;
else if (condition)
    statement;
.
.
else
    statement;
```



```
/* C program to illustrate nested-if statement */
#include <stdio.h>

int main() {
    int i = 20;

    if (i == 10)
        printf("i is 10");
    else if (i == 15)
        printf("i is 15");
    else if (i == 20)
        printf("i is 20");
    else
        printf("i is not present");

return 0;
}
```

Output:

```
i is 20
```

Loop Structure

Iteration statements specify looping.

iteration-statement:

1. while (*expression*) *statement*
2. do *statement* while (*expression*);
3. for (*expression1*; *expression2*; *expression3*) *statement*

In the `while` and `do` statements, the sub-statement is executed repeatedly so long as the value of the expression remains true.

In the for statement, the first expression is evaluated once, and thus specifies initialization for the loop. There is no restriction on its type. The second expression must have arithmetic or pointer type; it is evaluated before each iteration, and if it becomes equal to 0, the for is terminated. The third expression is evaluated after each iteration, and thus specifies a re-initialization for the loop.

for (*expression1*; *expression2*; *expression3*) *statement*

is equivalent to

expression1;
while (*expression2*) {
statement
expression3;
}

Note: expressions are optional in for loop; ***expressionopt***

UN-CONDITIONAL CONTROL FLOW STATEMENT

Jump statements

Jump statements transfer control unconditionally.

jump-statement-type:

goto *identifier*;
continue;
break;
`return` *expressionopt*;

A `continue` statement may appear only within an iteration statement. It causes control to pass to the loop-continuation portion of the smallest enclosing such statement i.e beginning of loop. More precisely, within each of the
statements

```
while (...) {        do {                        for (...) {
...                         ...                           ...
continue ;                  continue ;                    continue ;
        }           } while (...);                }
```

A `break` statement may appear only in an iteration statement or a `switch` statement, and terminates execution of the smallest enclosing such statement; control passes to the statement following the terminated statement.

A function returns to its caller by the `return` statement. When `return` is followed by an expression, the value is returned to the caller of the function. The expression is converted, as by assignment, to the type returned by the function in which it appears.

Q.1

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

```
include "stdio.h"
main()
{
	int i,j,n,sp,st=1;
	printf("Enter total rows");
	scanf("%d",&n);
	sp=n-1;
	for(i=1;i<=n;i++)
	{
		for(j=1;j<=sp;j++)
			printf(" ");
		for(j=1;j<=st;j++)
			printf("*");
		printf("\n");
		sp=sp-1;
		st=st+2;
	}
}
```

Q.2

```
    1
   121
  12321
 1234321
123454321
```

```
#include "stdio.h"
main()
{
	int i,j,n,sp;
	printf("Enter total rows");
	scanf("%d",&n);
	sp=n-1;
	for(i=1;i<=n;i++)
	{
		for(j=1;j<=sp;j++)
			printf(" ");
		for(j=1;j<=i;j++)
			printf("%d",j);
		for(j=i-1;j>=1;j--)
			printf("%d",j);
		printf("\n");
		sp=sp-1;
	}
}
```

1	How many statements falls under the default scope of a control structure?
Ans	A single statement falls under the default scope of the control structure.
2	Which loop is called an exit control loop
Ans	Do while
3	What is 'hanging if' statement in C?
Ans	The statement which is neither part of "if" nor part of "else" statement, but it is 'on fly' is known as 'hanging if' statement. 'hanging if' statement is not allowed in C.
4	In case of switch and if else statement , which is faster execution
Ans	'Switch' statement is faster than the 'if else' statement, because in 'switch' statement execution starts from the match case. The 'if else' execution starts from the beginning of 'if'.
5	What is the difference between for loop and while loop
Ans	For loop is suitable if number of repetition is known , but while loop is suitable if number of repetition is unknown in advance
6	What is the difference between break and return statement
Ans	Break statement transfer cursor just outside loop and switch case, but return statement transfer cursor to end of the function
7	What is the difference between return and exit statement
Ans	Return and exit statement, both are same inside main function , but both are different in other functioin. In other function return terminate the function but exit terminate the program
8	Why "goto" statement is not used in a program
Ans	"goto" statement is an unconditional jump control structure. It is suggested that not to use in a program , because it affects other control structure in a program
9	What is the difference between break and continue statement
Ans	Break statement transfer cursor to outside loop and switch case ,but continue statement transfer control to beg of the loop
10	How to write a program in c , that a loop will continue 2^100 times
Ans	According to range of data types in C, a variable maximum range is 2^64 , if the variable value beyond maximum range, then the loop becomes infinite. So to overcome this type of program , nested loop is required

Arrays

Let is write a program to count the number of occurrences of each digit, of white space characters (blank, tab, newline), and of all other characters. This is artificial, but it permits us to illustrate several aspects of C in one program. There are twelve categories of input, so it is convenient to use an array to hold the number of occurrences of each digit, rather than ten individual variables. Here is one version of the program:

```
#include <stdio.h>
/* count digits, white space, others */
main()
{
int c, i, nwhite, nother;
int ndigit[10];
nwhite = nother = 0;
for (i = 0; i < 10; ++i)
ndigit[i] = 0;
while ((c = getchar()) != EOF)
if (c >= ’0’ && c <= ’9’)
++ndigit[c-’0’];
else if (c == ’ ’ || c == ’\n’ || c == ’\t’)
++nwhite;
else
++nother;
printf("digits =");
for (i = 0; i < 10; ++i)
printf(" %d", ndigit[i]);
printf(", white space = %d, other = %d\n",
nwhite, nother);
}
```

The output of this program on itself is

```
    digits = 9 3 0 0 0 0 0 0 0 1, white space = 123, other = 345
```

Q.1	Write a program to find the second largest element in an array

```
#include "stdio.h"
int SecondLargest(int [],int);
main()
{
        int x[8]={10,4,90,11,34,22,19,45};
        int k=SecondLargest(x,8);
        printf("%d",k);
}
int SecondLargest(int x[],int size)
{
        int max1=x[0];
        int max2=x[0];
        int i,temp;
        for(i=1;i<size;i++)
        {
                if(max1 < x[i])
                        max1=x[i];
                if(max2 < max1)
                {
                        temp=max2;
                        max2=max1;
                        max1=temp;
                }
        }
        return max1;
}
```

Q.2	Write a program to reverse a string

```
#include "stdio.h"
#include "string.h"
void reverse(char []);
main()
{
        char x[]="alok das";
        reverse(x);
        printf("%s",x);
}
void reverse(char x[])
{
        int i=0,j=strlen(x)-1;
        char temp;
        while(i<j)
        {
                temp=x[i];
                x[i]=x[j];
                x[j]=temp;
                i++;
                j--;
        }
}
```

Q.3 Write a program to convert a string into toggle case

```
#include "stdio.h"
#include "string.h"
void toggle(char []);
void lower(char []);
main()
{
        char x[]="india is the best";
        toggle(x);
        printf("%s",x);
}
void upper(char x[])
{
        int i;
        for(i=0;i<strlen(x);i++)
        {
                if(x[i]>=97 && x[i]<=122)
                        x[i]=x[i]-32;
        }
}
void toggle(char x[])
{
        upper(x);
        x[0]=x[0]+32;
        int i;
        for(i=1;i<strlen(x);i++)
        {
                if(x[i]==' ')
                        x[i+1]=x[i+1]+32;
        }
}
```

Q.4 Write a program to compare between two string without using strcmp() function

```
#include "stdio.h"
#include "string.h"
int mystrcmp(char [],char []);
main()
{
        int x;
        x=mystrcmp("abc","abc"); //0
        x=mystrcmp("akc","abc"); //1
        x=mystrcmp("abc","akc"); //-1
}
int mystrcmp(char x[],char y[])
{
        int i=0,j=0;
        while(i<strlen(x) || j<strlen(y))
```

```
	{
		if(x[i]!=y[j])
		{
			if(x[i] > y[j])
				return 1;
			else
				return -1;
		}
		i++;
		j++;
	}
	return 0;
}
```

Q.5 Write program to find gcd of all numbers present in an array

```
#include "stdio.h"
int gcd(int ,int);
main()
{
	int x[8]={10,4,90,11,34,22,19,45};
	int k=x[0];
	int i;
	for(i=1;i<8;i++)
	{
		k=gcd(k,x[i]);
	}
	printf("%d",k);
}
int gcd(int a,int b)
{
	int c;
	while((c=a%b)!=0)
	{
		a=b;
		b=c;
	}
	return b;
}
```

TWO-DIMENSIONAL ARRAY

C provides rectangular multi-dimensional arrays, although in practice they are much less used than arrays of pointers. In this section, we will show some of their properties. Consider the problem of date conversion, from day of the month to day of the year and vice versa. For example, March 1 is the 60th day of a non-leap year, and the 61st day of a leap year. Let us define two functions to do the conversions: day_of_year converts the month and day into the day of the year, and month_day converts the day of the year into the month and day. Since this latter function computes two values, the month and day arguments will be pointers:

```
month_day(1988, 60, &m, &d)
```

sets m to 2 and d to 29 (February 29th).

These functions both need the same information, a table of the number of days in each month (‘‘thirty days hath September ...’’). Since the number of days per month differs for leap years and non-leap years, it’s easier to separate them into two rows of a two-dimensional array than to keep track of what happens to February during computation. The array and the functions for performing the transformations are as follows:

```
static char daytab[2][13] = {
{0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
{0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};

/* day_of_year: set day of year from month & day */
int day_of_year(int year, int month, int day)
{
int i, leap;
leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
for (i = 1; i < month; i++)
day += daytab[leap][i];
return day;
}
/* month_day: set month, day from day of year */
void month_day(int year, int yearday, int *pmonth, int *pday)
{
int i, leap;
leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
for (i = 1; yearday > daytab[leap][i]; i++)
yearday -= daytab[leap][i];
*pmonth = i;
*pday = yearday;
}
```

Recall that the arithmetic value of a logical expression, such as the one for leap, is either zero (false) or one (true), so it can be used as a subscript of the array daytab. The array daytab has to be external to both day_of_year and month_day, so they can both use it. We made it char to illustrate a legitimate use of char for storing small non-character integers. daytab is the first two-dimensional array we have dealt with. In C, a two-dimensional array is really a one-dimensional array, each of whose elements is an array. Hence subscripts are written as

```
daytab[i][j] /* [row][col] */
```

rather than

```
daytab[i,j] /* WRONG */
```

Other than this notational distinction, a two-dimensional array can be treated in much the same way as in other languages. Elements are stored by rows, so the rightmost subscript, or column, varies fastest as elements are accessed in storage order. An array is initialized by a list of initializers in braces; each row of a two-dimensional array is initialized by a corresponding sub-list. We started the array daytab with a column of zero so that month numbers can run from the natural 1 to 12 instead of 0 to 11. Since space is not at a premium here, this is clearer than adjusting the indices. If a two-dimensional array is to be passed to a function, the parameter declaration in the function must include the number of columns; the number of rows is irrelevant, since what is passed is, as before, a pointer to an array of rows, where each row is an array of 13 ints. In this particular case, it is a pointer to objects that are arrays of 13 ints. Thus if the array daytab is to be passed to a function f, the declaration of f would be:

```
f(int daytab[2][13]) { ... }
```

It could also be

```
        f(int daytab[][13]) { ... }
```

LABORATORY EXPERIMENT

Q.1 Write a program for matrix multiplication

```
#include "stdio.h"
#include "string.h"
main()
{
      int x[3][3];
      int y[3][3];
      int z[3][3];
      int i,j,k;
      printf("input elements for matrix x\n");
      for(i=0;i<3;i++)
      {
            for(j=0;j<3;j++)
            {
                  scanf("%d",&x[i][j]);
            }
      }
      printf("input elements for matrix y\n");
      for(i=0;i<3;i++)
      {
            for(j=0;j<3;j++)
            {
                  scanf("%d",&y[i][j]);
            }
      }
      memset(z,0,sizeof(z));
      //matrix multiplication
      for(i=0;i<3;i++)
      {
            for(j=0;j<3;j++)
            {
                  for(k=0;k<3;k++)
                  {
                        z[i][j]=z[i][j]+x[i][k]*y[k][j];
                  }
            }
      }
      //traverse matrix
      for(i=0;i<3;i++)
      {
            for(j=0;j<3;j++)
            {
                  printf("%d ",z[i][j]);
            }
            printf("\n");
      }
}
```

Q.2 Write a program to transpose a matrix

```
#include "stdio.h"
#include "string.h"
main()
{
        int x[3][5];
        int y[5][3];
        int i,j,k;
        printf("input elements for matrix x\n");
        for(i=0;i<3;i++)
        {
            for(j=0;j<5;j++)
            {
                scanf("%d",&x[i][j]);
            }
        }
    //transpose
      for(i=0;i<3;i++)
        {
            for(j=0;j<5;j++)
            {
                y[j][i]=x[i][j];
            }
        }
        //traverse matrix y
        for(i=0;i<5;i++)
        {
            for(j=0;j<3;j++)
            {
                printf("%d ",y[i][j]);
            }
            printf("\n");
        }
}
```

Q.3 Write a program to check a matrix is sparse matrix or not

```
//zero dominated matrix

#include "stdio.h"
#include "string.h"
main()
{
    int x[5][5];
    int i,j,counter=0;
    int size=sizeof(x)/sizeof(int);

    printf("input elements for matrix x\n");
    for(i=0;i<5;i++)
    {
        for(j=0;j<5;j++)
        {
            scanf("%d",&x[i][j]);
        }
    }
    for(i=0;i<5;i++)
    {
        for(j=0;j<5;j++)
        {
            if(x[i][j]==0)
                counter++;
        }
        }
        if(counter > size/2)
                printf("It is sparse matrix");
        else
                printf("Not a sparse matrix");
}
```

Q.4 Write a program remove vowel from a string

```
#include "stdio.h"
#include "string.h"
main()
{
        char x[]="india is the best";
        int i=0,k;
        while(i<strlen(x))
        {
                if(x[i]=='a' || x[i]=='e' || x[i]=='i' || x[i]=='o' || x[i]=='u')
                {
                        for(k=i;k<strlen(x);k++)
                                x[k]=x[k+1];
                }
                else
                        i++;
        }
        printf("%s",x);
}
```

Q.5 write a program to remove duplicate element from an array

```
#include "stdio.h"
main()
{
        int x[8]={10,20,10,10,30,20,40,10};
        int i,j,k,n=8;
        for(i=0;i<8;i++)
        {
                j=i+1;
                while(j<n)
                {
                        if(x[i]==x[j])
                        {
                                for(k=j;k<n;k++)
                                        x[k]=x[k+1];

                                n--;
                        }
                        else
                                j++;
                }
        }

        for(i=0;i<n;i++)
                printf("%d ",x[i]);
}
```

UNIT 3
FUNCTION (5 LECTURES)
RECURSION (4 LECTURES)

FUNCTION

In C, a function is equivalent to a subroutine or function in Fortran, or a procedure or function in Pascal. A function provides a convenient way to encapsulate some computation, which can then be used without worrying about its implementation. With properly designed functions, it is possible to ignore *how* a job is done; knowing *what* is done is sufficient. C makes the sure of functions easy, convenient and efficient; you will often see a short function defined and called only once, just because it clarifies some piece of code.

A function definition has this form:

```
return-type function-name(parameter declarations, if any)
{
declarations
statements
}
```

Function definitions can appear in any order, and in one source file or several, although no function can be split between files. If the source program appears in several files, you may have to say more to compile and load it than if it all appears in one, but that is an operating system matter, not a language attribute. For the moment, we will assume that both functions are in the same file, so whatever you have learned about running C programs will still work.

We will generally use *parameter* for a variable named in the parenthesized list in a function. The terms *formal argument* and *actual argument* are sometimes used for the same distinction.

The declaration

```
                    int power(int base, int n);
```

just before main says that power is a function that expects two int arguments and returns an int. This declaration, which is called a *function prototype*, has to agree with the definition and uses of power. It is an error if the definition of a function or any uses of it do not agree with its prototype. parameter names need not agree. Indeed, parameter names are optional in a function prototype, so for the prototype we could have written

```
        int power(int, int);
```

Q.1	What is function ? what is the application of the function ?
Ans	A function is set of statements which are encapsulated together to perform a specific task. It is suitable to modularize the program , that's why language able to create a library and recursively use the code.
Q.2	What is function recursion ? what is the application of function recursion
Ans	When a function call to itself is known as function recursion and it is suitable for quick sort,heap sort, merge sort, tower of Hanoi and tree traversal programming
Q.3	What is the difference between header file and library?
Ans	Header file contains function declaration , but library contains function definition
Q.4	What is user define function?
Ans	The function which has no header and library is called a user define function
Q.5	What is the advantage of modular style over monolithic style c programming
Ans	Modular style gives clear image of a program but monolithic style doesn't give clear image of a program. Function recursion and library only possible if code is written in modular style.
Q.6	What is the difference between actual parameters and formal parameters
Ans	Actual parameters allocate memory , but formal parameters doesn't allocate memory

Q.7	What is the difference between call by value and call by address
Ans	In case of call by value , change the value in function definition , that doesn't change value in it's main function. But in case of call by address ,change the value in function definition that will change in main function.
Q.8	What is the difference between function and method
Ans	When a function is define inside a class is called method. All methods are function but all function are not method. (Method is a c++ concept)
Q.9	What is the difference between loop control structure and function recursion
Ans	Loop control structure doesn't allocate memory , but funtioin recursion allocate memory
Q.10	What is function over-heading and function over-loading
Ans	When the control transfer to a function it takes time, when a function is push in stack also takes time, when control return from a function it takes time, when function is pop also takes time, all together is called function over-heading. When multiple functions are defined having same name is called function over loading which is a c++ concept.

PROBLEMS

Q.1 Write a program to find GCD of any two numbers using function

```
#include "stdio.h"
int gcd(int,int);
main()
{
        int a,b,x;
        printf("enter any two numbers");
        scanf("%d%d",&a,&b);
        x=gcd(a,b);
        printf("%d",x);
}
int gcd(int a, int b)
{
        int c;
        while((c=a%b)!=0)
        {
                a=b;
                b=c;
        }
        return b;
}
```

Q.2 Write a program to reverse a number using function

```
#include "stdio.h"
int reverse(int);
main()
{
        int n,x;
        printf("enter any two numbers");
        scanf("%d",&n);
        x=reverse(n);
        printf("%d",x);
}
int reverse(int n)
{
        int s=0;
        while(n>0)
        {
```

	`s=s*10 + n%10;` `n=n/10;` `}` `return s;` `}`
Q.3	Write a program to find sum of digits of a number using function
	see code below
Q4.	Write a program to check a given number is prime or not using function
	see code below
Q.5	Write a program count no of even digits present in a number using function
	see code below

```
                s=s*10 + n%10;
                n=n/10;
        }
        return s;
}
```

Q.3 Write a program to find sum of digits of a number using function

```
#include "stdio.h"
int sumofdigits(int);
main()
{
        int n,x;
        printf("enter any two numbers");
        scanf("%d",&n);
        x=sumofdigits(n);
        printf("%d",x);
}
int sumofdigits(int n)
{
        int s=0;
        while(n>0)
        {
                s=s + n%10;
                n=n/10;
        }
        return s;
}
```

Q4. Write a program to check a given number is prime or not using function

```
#include "stdio.h"
int prime(int);
main()
{
        int n,x;
        printf("enter any two numbers");
        scanf("%d",&n);
        x=prime(n);
        if(x==2)
                printf("Prime");
        else
                printf("Not prime");
}
int prime(int n)
{
        int i,count=0;
        for(i=1;i<=n;i++)
        {
                if(n%i==0)
                        count++;
        }
        return count;
}
```

Q.5 Write a program count no of even digits present in a number using function

```
#include "stdio.h"
int CountEvenDigits(int);
main()
{
        int n,x;
        printf("enter any two numbers");
        scanf("%d",&n);
        x=CountEvenDigits(n);
        printf("%d",x);
}
int CountEvenDigits(int n)
{
        int r,count=0;
        while(n>0)
        {
                r=n%10;
                if(r%2==0)
                        count++;
                n=n/10;
        }
        return count;
}
```

Function Recursion:
C functions may be used recursively; that is, a function may call itself either directly or indirectly.

For example Consider printing a number as a character string.

Another good example of recursion is quicksort, a sorting algorithm developed by C.A.R. Hoare in 1962. Given an array, one element is chosen and the others partitioned in two subsets - those less than the partition element and those greater than or equal to it. The same process is then applied recursively to the two subsets. When a subset has fewer than two elements, it doesn't need any sorting; this stops the recursion.

Q.1 Write a program to display all number from 1 to 100 those are power of 2 using **function recursion**

```
#include "stdio.h"
void powerof2(int,int);
main()
{
        powerof2(1,100);
}
void powerof2(int i,int j)
{
        if(i<=j)
        {
                if( (i&i-1)==0)
                        printf("%d ",i);
                i++;
                powerof2(i,j);
        }
}
```

Q.2 Calculate factorial of a number using Function Recursion

```
#include "stdio.h"
int fact(int);
main()
{
        int x,n;
        printf("Enter a no");
        scanf("%d",&n);
        x=fact(n);
        printf("%d",x);
}
int fact(int n)
{
        static int f=1;
        if(n>0)
        {
                f=f*n;
                fact(n-1);
        }
        return f;
}
```

Q.3 Write a program to find sum of digits of a number using **function recursion**

```
#include "stdio.h"
int sumofdigits(int);
main()
{
        int n,x;
        printf("enter any two numbers");
        scanf("%d",&n);
        x=sumofdigits(n);
        printf("%d",x);
}
int sumofdigits(int n)
{
        static int s=0;
        if(n>0)
        {
                s=s + n%10;
                n=n/10;
                sumofdigits(n);
       }
        return s;
}
```

Q.4 Write a program to reverse a number using **function recursion**

```
#include "stdio.h"
int reverse(int);
main()
{
        int n,x;
        printf("enter any two numbers");
        scanf("%d",&n);
        x=reverse(n);
        printf("%d",x);
}
int reverse(int n)
{
        static int s=0;
        if(n>0)
        {
                s=s*10 + n%10;
                n=n/10;
                 reverse(n);
      }
        return s;
}
```

Q.5 Check a number is prime or not using **Function recursion**

```
#include "stdio.h"
int prime(int);
main()
{
	int x,n;
	printf("Enter a no");
	scanf("%d",&n);
	x=prime(n);
	if(x==2)
		printf("Prime");
	else
		printf("Not");
}
int prime(int n)
{
	static int i=1,counter=0;
	if(i<=n)
	{
		if(n%i==0)
			counter++;
		i++;
		prime(n);
	}
	return counter;
}
```

Q 6 printing a number as a character string.

The recursive solution, in which printd first calls itself to
cope with any leading digits, then prints the trailing digit. Again, this version can fail on the largest negative number.

```
#include <stdio.h>
/* printd: print n in decimal */
void printd(int n)
{
if (n < 0) {
putchar(’-’);
n = -n;
}
if (n / 10)
printd(n / 10);
putchar(n % 10 + ’0’);
}
```

Q 7. Quicksort, a sorting algorithm to sort an array

We use the middle element of each subarray
for partitioning.

```
/* qsort: sort v[left]...v[right] into increasing order */
void qsort(int v[], int left, int right)
{
int i, last;
void swap(int v[], int i, int j);
if (left >= right) /* do nothing if array contains */
return; /* fewer than two elements */
swap(v, left, (left + right)/2); /* move partition elem */
last = left; /* to v[0] */
for (i = left + 1; i <= right; i++) /* partition */
if (v[i] < v[left])
swap(v, ++last, i);
swap(v, left, last); /* restore partition elem */
qsort(v, left, last-1);
qsort(v, last+1, right);
}
```

	We moved the swapping operation into a separate function swap because it occurs three times in qsort.
	/* swap: interchange v[i] and v[j] */
	void swap(int v[], int i, int j)
	{
	int temp;
	temp = v[i];
	v[i] = v[j];
	v[j] = temp;
	}

ASSIGNMENT

1	Comment the below while statement
	main()
	{
	    while(0==0)
	    {
	
	    }
	}
	a. It has syntax error as there is no statement within braces.
	b. It will run forever
	c. It compare 0 with 0 and since they are equal, it will exit the loop
	d. None

2	Which byte order is followed by Motorola Processors? a. Little-Endian b. Big-Endian c. Bi-Endian d. None

3	How many time while statement is executed
	main()
	{
	    int i=1;
	    while(i<=5)
	    {
	        printf("Hello,I am in loop");
	        i++;
	    }
	}
	a. 4 times
	b. 5 times
	c. 6 times
	d. infinite times

4	How many time display “yes boss”
	main()
	{
	    int i;
	    for(i=1;i<=30;i+5)
	    {
	        printf("Yes boss");
	    }
	}
	a. 5 times

	b. 6 times c. infinite times c. None

5	What will be the output?
	`main()`
	`{`
	`    int x=3,y=5;`
	`    x=x++ || ++y;`
	`    printf("%d %d",x,y);`
	`}`
	a. 5
	b. 6
	c. 1
	d. 0

6	What will be the output?
	`main()`
	`{`
	`    char x=-130;`
	`    char y=-5;`
	`    printf("%d ", x+y);`
	`}`
	a. -135
	b. 135
	c. -121
	d. 121

7	What will be the output? Which operator is used both as an operator and keyword? a. Right shifting operator b. cast operator c. sizeof operator d. Token pasting operator
8	Which modifier doubles range of a data types a. signed b. unsigned c. short d. long

9	Find the output
	`void main()`
	`{`
	`    int num=345,m,sum=0;`
	`    do`
	`    {`
	`        m=num%10;`
	`        num=num/10;`
	`        continue;`
	`        sum=sum*10+m;`
	`    }`
	`    while(num!=0);`
	`    printf("%d",sum);`
	`}`
	a. 3 4 5
	b. 5 4 3
	c. 0

	d. None of these

10	Find the output
	void main()
	{
	char ch=’r’;
	if(ch==’a’||ch=’h’)
	ch=’u’;
	printf(“%c”,ch);
	}
	a) r
	b) a
	c) u
	d) Compilation error

11	To execute all switch case statements a) Any one of the case statement match with switch condition b) First case must match with switch condition c) Default case must match with switch condition. d) None of these.
12	The major difference between strcpy() and strncpy() function is a. strcpy copy entire string but strncpy copy only specified number of characters b. strcpy copy nul character but strncpy does't copy nul character c. strcpy() takes two arguments and strncpy() takes three arguments d. None

13	What is the output
	main()
	{
	char x[5]="abc";
	char y[ ]="abc";
	printf("%d %d",sizeof(x),sizeof(y));
	}
	a. 5 5
	b. 4 4
	c. 5 4
	d. 3 3

14	Find the output
	main()
	{
	int i;
	for(i=1;i<=5;i++)
	{
	bbsr();
	}
	}
	int bbsr()
	{
	int x=1;
	static int y=1;
	printf("%d %d \n",x,y);
	x++;
	y++;
	}
	a. x & y both are incremented from 1 to 5
	b. x is incremented from 1 to 5 , but y is not incremented

<table>
<tr><td></td><td>c. x is not incremented from 1 to 5 , but y is incremented from 1 to 5
c. both x & y are not incremented</td></tr>
<tr><td>15</td><td>Find the Output<pre>
main()
{
        int i=1;
        while(i<=5)
        {
                int i=100;
                printf("%d ",i);
                i++;
        }
}
</pre>a. 100 100 100 100 …
b. 100,101,102,103,…
c. 1,2,3,4,5
d. 1,2,3,4,5, …</td></tr>
<tr><td>16</td><td>Find the output<pre>
main()
{
        lit(3);
}
int lit(int n)
{
        if(n>0)
        {
                printf("%d ",n);
                lit(--n);
                lit(--n);
        }
}
</pre>a. 3 2 1 1
b. 0 1 2 1
c. -1 0 -1 1
d. none</td></tr>
<tr><td>17</td><td>Which is the following statement is true
a) Copy of the variable is created when it is passed by reference
b) Copy of the variable is created when it is passed by value
c) Created both
d) Not created in both</td></tr>
<tr><td>18</td><td>Find the output<pre>
void main()
{
        int i=4,j=-3;
        mul(&i,j);
        printf("%d %d",i,j);
}
mul(int *a,int b)
{
        *a=*a**a;
        b=b*b;
}
</pre></td></tr>
</table>

	a) 4 3 b) 4 9 c) 16 -3 d) None of these
19	Efficient way of dividing x by 8 is a) x/8 b) x>>3 c) x<<3 d) None
20	Which operation done by Microprocessor for the expression. i++; a) Read b) Write c) Both read and write d) None
21	The relational operator == (equality) always returns a. 0 b. 1 c. 0 or 1 d. None
22	Bitwise operators are applicable only on a. integers b. integers and characters c. integers and floats d. integers,floats,and doubles
23	if(test( )) i++; a) i value is incremented if test function returns 0 b) i value is incremented if test function returns non-zero c) i value is incremented if test function returns nothing d) none

24	what will be the output of the program?
	`int f(int x)`
	`{`
	`  if(x <= 4)`
	`    return x;`
	`  return f(--x);`
	`}`
	`void main()`
	`{`
	`  printf("%d ", f(7));`
	`}`
	a.4567
	b.1 2 3 4
	c.4
	d. Runtime error

25	Which of the following is a limitation of array? a. Array index starts from 0. b. Array element can be accessed in pointer format c. Array does not supports negative indexing d. Array size can't be changed during run time.
26	Find output main()

```
{
        if((n&n-1)==0)
        {

        }
}
```
a.it check a number is prime number
b.it check square root of a number
c.it check number is power of 2
d.None

27
```
main()
{
        while(!printf("Hello"))
        {
                printf("Yes boss");
        }
}
```
a. It will print Hello only once
b. It will print Hello forever
c. It will printf Hello Yes boss for ever
d. No output

28 Which of the following is right expression
```
main()
{
        int x,y;

        x=5*9+3/2;

}
```
a. x=add(mul(5,9),div(3,2))
b. x=mul(5,9) + div(3,2)
c. x=mul(5,9) * sum(9,3) + div(3,2)
d. None

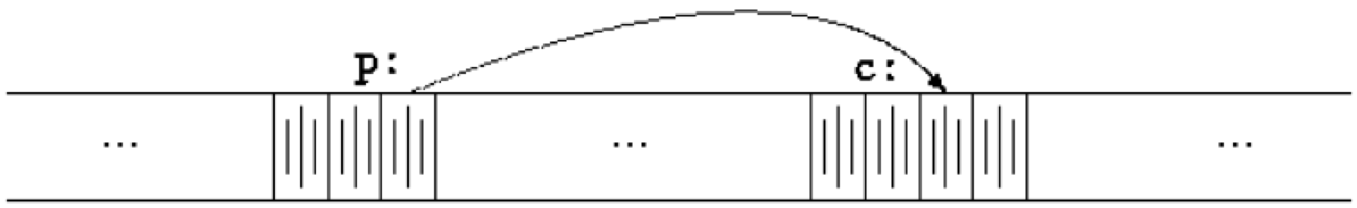
29 Find output
```
main()
{
        int x[5]={10,20,30,40,50};
        printf("%d" ,(x+2)[-1]);
}
```
a. 10
b. 20
c. 30
d. garbage

UNIT 4:
POINTERS (2 LECTURES)
STRUCTURE (4 LECTURES)
FILE HANDLING(1 LECTURES)

POINTERS

A pointer is a variable that contains the address of a variable. Pointers are much used in C, partly because they are sometimes the only way to express a computation, and partly because they usually lead to more compact and efficient code than can be obtained in other ways. Pointers and arrays are closely related; this chapter also explores this relationship and shows how to exploit it.

MEMORY ORGANIZATION
numbered or addressed memory cells that may be manipulated individually or in contiguous groups. One



common situation is that any byte can be a char, a pair of one-byte cells can be treated as a short integer, and four adjacent bytes form a long. A pointer is a group of cells (often two or four) that can hold an address. So if c is a char and p is a pointer that points to it, we could represent the situation this way:
The unary operator & gives the address of an object, so the statement
p = &c;
assigns the address of c to the variable p, and p is said to ‘‘point to’’ c. The & operator only applies to objects in memory: variables and array elements. It cannot be applied to expressions, constants, or register variables.

The unary operator * is the *indirection* or *dereferencing* operator; when applied to a pointer, it accesses the object the pointer points to. Suppose that x and y are integers and ip is a pointer to int. This artificial sequence shows how to declare a pointer and how to use & and *:

```
int x = 1, y = 2, z[10];
int *ip; /* ip is a pointer to int */
ip = &x; /* ip now points to x */
y = *ip; /* y is now 1 */
*ip = 0; /* x is now 0 */
ip = &z[0]; /* ip now points to z[0] */
```
The declaration of x, y, and z are what we’ve seen all along. The declaration of the pointer ip,
int *ip; is intended as a mnemonic; it says that the expression *ip is an int.

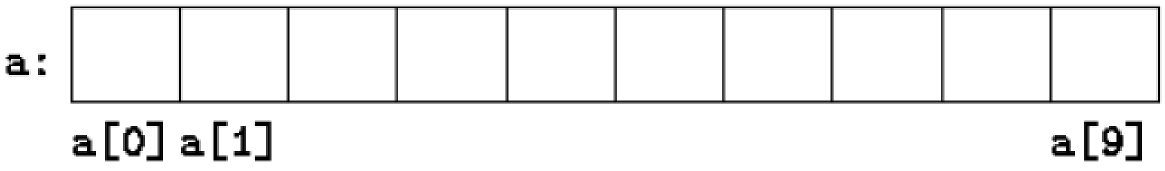
If ip points to the integer x, then *ip can occur in any context where x could, so
 *ip = *ip + 10; increments *ip by 10.
The unary operators * and & bind more tightly than arithmetic operators, so the assignment
 y = *ip + 1
takes whatever ip points at, adds 1, and assigns the result to y, while
 *ip += 1
increments what ip points to, as do
 ++*ip
and
 (*ip)++
The parentheses are necessary in this last example; without them, the expression would increment ip instead of what it points to, because unary operators like * and ++ associate right to left. Finally, since pointers are

variables, they can be used without dereferencing. For example, if iq is another pointer to int, iq = ip copies the contents of ip into iq, thus making iq point to whatever ip pointed to.

Pointers and Arrays:

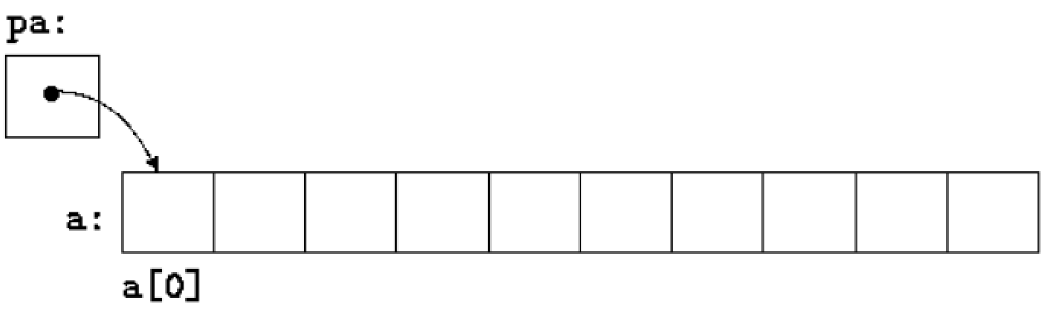
In C, there is a strong relationship between pointers and arrays, strong enough that pointers and arrays should be discussed simultaneously. Any operation that can be achieved by array subscripting can also be done with pointers. The pointer version will in general be faster but, at least to the uninitiated, somewhat harder to understand. The declaration

int a[10];

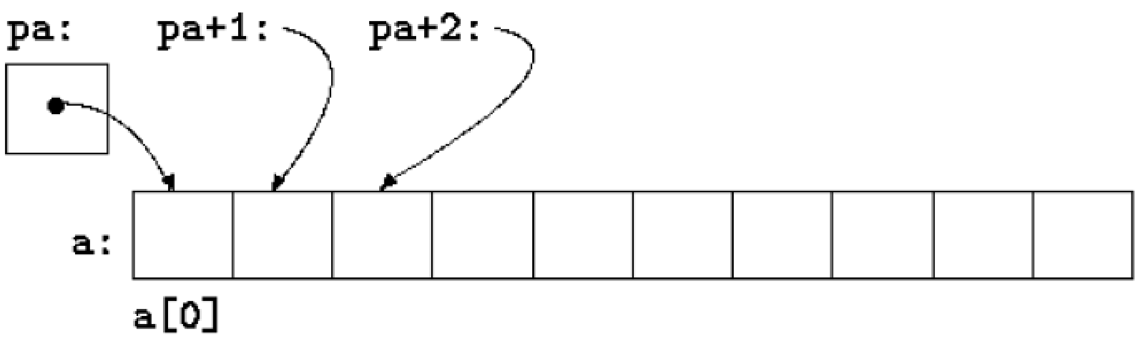


defines an array of size 10, that is, a block of 10 consecutive objects named a[0], a[1], ...,a[9].

The notation a[i] refers to the i-th element of the array. If pa is a pointer to an integer, declared as
int *pa;
then the assignment
pa = &a[0];
sets pa to point to element zero of a; that is, pa contains the address of a[0].
Now the assignment x = *pa will copy the contents of a[0] into x.



If pa points to a particular element of an array, then by definition pa+1 points to the next element, pa+i points i elements after pa, and pa-i points i elements before. Thus, if pa points to a[0], *(pa+1) refers to the contents of a[1], pa+i is the address of a[i], and *(pa+i) is the contents of a[i].



ASSIGNMENT

SLNO	QUESTIONS
1	Which concept is called “dereference”?
	Doing read or write operation in memory using the help of pointer is called dereference
2	What does maximum address by a pointer refer in TC and GCC compilers?
	In Turbo C pointer maximum address refers up to 1 MB of memory, whereas in GCC pointer maximum address refers up to 4 GB of memory.
3	What is null pointer?
	When a pointer refers initial address in memory or 0th address in memory it is called null pointer.
4	What is wild pointer?
	When a pointer refers unauthenticated address in memory it is called wild pointer.
5	What is dangling pointer?
	A reference that does not actually lead anywhere
6	What are the different reasons for segmentation fault in C?
	When segmentation fault occurs in a program, the program is simply terminated. There are different reasons for segmentation fault, such as:
7	What are near and far pointers in C?
	Far and near pointers are only introduced in Turbo C compiler. When the pointer refers to an address in the same segment it is called near pointer, but when it refers to an address in another
8	What are the applications of pointers in C?
	* Dynamic memory allocation
9	When does core dump occur in C ?
	A process dumps core when it is terminated by the operating system due to a fault in the program. The most typical reason for its occurrence is that the program has accessed an invalid
10	What is the difference between malloc and calloc functions?
	Malloc memory allocation is equivalent to a single-dimensional array but calloc memory allocation is equivalent to a doubledimensional array.
11	What is meant by static and dynamic memory allocation in C?
	Memory allocation in a program is only possible in load time or rum time of the program. The memory allocated at compile time is known as static allocation. That allocated at run time is
12	Why is pointer arithmetic required?
	Pointer arithmetic is required only to access a particular memory address.
13	Define in a one-line statement “memory leak”.
	At any moment if a pointer loses reference of a memory block in the heap it is called memory leak.

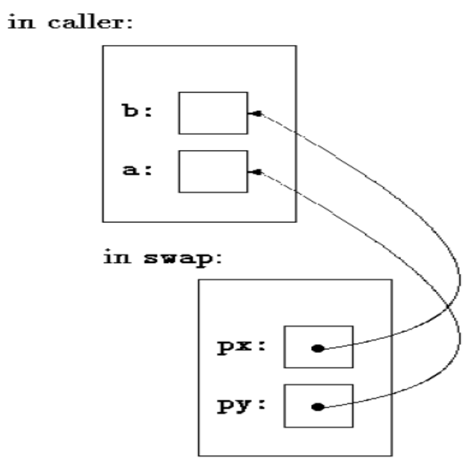
Pointers and Function Arguments:

Since C passes arguments to functions by value, there is no direct way for the called function to alter a variable in the calling function. For instance, a sorting routine might exchange two out-of-order arguments with a function called swap. It is not enough to write swap(a, b); where the swap function is defined as

```
void swap(int x, int y) /* WRONG */
{
int temp;
temp = x;
x = y;
y = temp;
}
```

Because of call by value, swap can't affect the arguments a and b in the routine that called it. The function above swaps *copies* of a and b.The way to obtain the desired effect is for the calling program to pass *pointers* to the values to be changed: swap(&a, &b);

Since the operator & produces the address of a variable, &a is a pointer to a. In swap itself, the parameters are declared as pointers, and the operands are accessed indirectly through them.



```
void swap(int *px, int *py) /* interchange *px and *py */
{
int temp;
temp = *px;
*px = *py;
*py = temp;
}
```

It is possible to pass part of an array to a function, by passing a pointer to the beginning of the subarray. For example, if a is an array, f(&a[2]) and f(a+2) both pass to the function f the address of the subarray that starts at a[2].

Within f, the parameter declaration can read

f(int arr[]) { ... }

or

f(int *arr) { ... }

So as far as f is concerned, the fact that the parameter refers to part of a larger array is of no consequence.

Q.1	Check a number is prime or not using pointer

	#include "stdio.h"
	int prime(int *);
	main()
	{
	    int x,n;
	    printf("Enter a no");
	    scanf("%d",&n);
	    x=prime(&n);
	    if(x==2)
	        printf("Prime");
	    else
	        printf("Not");
	}

```
int prime(int *p)
{
	int i=1,counter=0;
	while(i<= *p)
	{
		if(*p %i==0)
			counter++;
		i++;
	}
	return counter;
}
```

Q.2 Sum of all even digits in a given number using pointer

```
#include "stdio.h"
int SumofEvenDigits(int *);
main()
{
	int x,n;
	printf("Enter a no");
	scanf("%d",&n);
	x=SumofEvenDigits(&n);
	printf("%d",x);
}
int SumofEvenDigits(int *p)
{
	int sum=0,r;
	while(*p>0)
	{
		r=*p%10;
		if(r%2==0)
			sum=sum+r;
		*p=*p/10;
	}
	return sum;
}
```

Q.3 Swap between two numbers using pointer

```
#include "stdio.h"
void  swap(int *,int*);
main()
{
	int x,a,b;
	printf("Enter any two no");
	scanf("%d%d",&a,&b);
		printf("Before swap %d %d\n",a,b);
	swap(&a,&b);
		printf("After swap %d %d\n",a,b);
	printf("%d",x);
}
void swap(int *p,int *q)
{
	int temp;
	temp=*p;
	*p=*q;
	*q=temp;
}
```

Q.4 Find GCD of any two number using pointer

```
#include "stdio.h"
int gcd(int *,int*);
main()
{
        int x,a,b;
        printf("Enter any  two no");
        scanf("%d%d",&a,&b);
        x=gcd(&a,&b);
        printf("%d",x);
}
int gcd(int *p,int *q)
{
        int c;
        while((c=*p % *q)!=0)
        {
                *p=*q;
                *q=c;
        }
        return *q;
}
```

Q.5	Calculate factorial of a number using pointer

```
#include "stdio.h"
int fact(int *);
main()
{
        int x,n;
        printf("Enter a no");
        scanf("%d",&n);
        x=fact(&n);
        printf("%d",x);
}
int fact(int *p)
{
        int f=1;
        while(*p >0)
        {
                f=f* *p;
                (*p)--;
        }
        return f;
}
```

ASSIGNMENT& SOLUTION

q.1	what is the relation between arrays and pointers?
	array name itself is a pointer which refers the initial address of an array.
q.2	what is major difference between strcpy and strncpy?
	strcpy copies the entire string with null character, but strncpy copies only the specified number of characters and does not copy the null character.
q.3	what is the purpose of double-dimensional array?
	double-dimensional array is suitable to implement the non-linear data structure, such as graph and tree.
q.4	if base address of a double-dimensional array is given, how do we get the address of a particular element in an array?

	(note: if int*[5][5], &*[0][0] is given as 500, what will be the address of &*[3[4])
	to get the address of a particular element in a double dimensional array, the following formula is used. address = base address + (row number * column size + column number) * size of the array element.
q.5	what is the difference between character array and string?
	string is a character array terminated with null character '\0'. so, a string is a character array but a character array is not string.
q.6	what is the difference between char s[] and char *s?
	char s[] type can be dereference but char *s type can't be dereference
q.7	what are the limitation of an array in c
	array in c doesn't support negative indexing , and doesn't hold desimilar types of elements
q.8	if array size is not given , then how to access last element of an array
	sizeof(array)/sizeof(types)-1
q.9	what is the concept is called "array equivalent pointer"
	when array is used as the formal parameter of a function is called array equivalent pointer
q.10	what is the time complexity to insert or delete element in an array
	o(n)
q.11	what is the application of an array?
	application of an array is in implementing different data structures, such as stack, queue, tree, graph and searching, sorting.
q.12	what is the maximum dimension is allowed to an array
	it depends on memory size

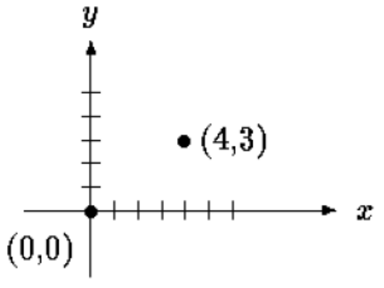
STRUCTURE & UNION:

A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling. (Structures are called ''records'' in some languages, notably Pascal.) Structures help to organize complicated data, particularly in large programs, because they permit a group of related variables to be treated as a unit instead of as separate entities. One traditional example of a structure is the payroll record: an employee is described by a set of attributes such as name, address, social security number, salary, etc. Some of these in turn could be structures: a name has several components, as does an address and even a salary. Another example, more typical for C, comes from graphics: a point is a pair of coordinate, a rectangle is a pair of points, and so on.

The main change made by the ANSI standard is to define structure assignment - structures may be copied and assigned to, passed to functions, and returned by functions. This has been supported by most compilers for many years, but the properties are now precisely defined. Automatic structures and arrays may now also be initialized.

Basics of Structures

Let us create a few structures suitable for graphics. The basic object is a point, which we will assume has an *x* coordinate and a *y* coordinate, both integers.



The two components can be placed in a structure declared like this:

```
struct point {
int x;
int y;
};
```

The keyword struct introduces a structure declaration, which is a list of declarations enclosed in braces. An optional name called a structure tag may follow the word struct (as with point here). The tag names this kind of structure, and can be used subsequently as a shorthand for the part of the declaration in braces. The variables named in a structure are called members.

A struct declaration defines a type. The right brace that terminates the list of members may be followed by a list of variables, just as for any basic type. That is,

```
struct { ... } x, y, z;
```

is syntactically analogous to

```
int x, y, z;
```

in the sense that each statement declares x, y and z to be variables of the named type and causes space to be set aside for them. A structure declaration that is not followed by a list of variables reserves no storage; it merely describes a template or shape of a structure. If the declaration is tagged, however, the tag can be used later in definitions of instances of the structure. For example, given the declaration of point above,

```
struct point pt;
```

defines a variable pt which is a structure of type struct point. A structure can be initialized by following its definition with a list of initializers, each a constant expression, for the members:

```
struct maxpt = { 320, 200 };
```

An automatic structure may also be initialized by assignment or by calling a function that returns a structure of the right type. A member of a particular structure is referred to in an expression by a construction of the form

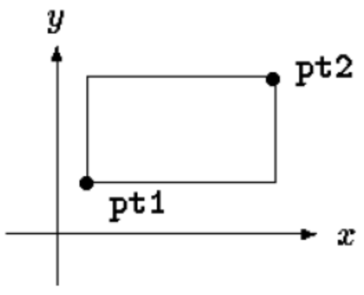
structure-name.member

The structure member operator ‘‘.’’ connects the structure name and the member name. To print the coordinates of the point pt, for instance,

```
printf("%d,%d", pt.x, pt.y);
```

or to compute the distance from the origin (0,0) to pt,

```
double dist, sqrt(double);
dist = sqrt((double)pt.x * pt.x + (double)pt.y * pt.y);
```



Structures can be nested. One representation of a rectangle is a pair of points that denote the diagonally opposite corners:

```
struct rect {
struct point pt1;
struct point pt2;
};
```

The rect structure contains two point structures. If we declare screen as

```
struct rect screen;
```

then screen.pt1.x refer to the *x* coordinate of the pt1 member of screen.

Structures and Functions:

The only legal operations on a structure are copying it or assigning to it as a unit, taking its address with &, and accessing its members. Copy and assignment include passing arguments to functions and returning values from functions as well. Structures may not be compared. A structure may be initialized by a list of constant member values; an automatic structure may also be initialized by an assignment.
Let us investigate structures by writing some functions to manipulate points and rectangles. There are at least three possible approaches: pass components separately, pass an entire structure, or pass a pointer to it. Each has its good points and bad points.

The first function, makepoint, will take two integers and return a point structure:

```
/* makepoint: make a point from x and y components */
struct point makepoint(int x, int y)
{
struct point temp;
temp.x = x;
temp.y = y;
return temp;
}
```

Notice that there is no conflict between the argument name and the member with the same name; indeed the re-use of the names stresses the relationship. makepoint can now be used to initialize any structure dynamically, or to provide structure arguments to a function:

```
struct rect screen;
struct point middle;
struct point makepoint(int, int);
screen.pt1 = makepoint(0,0);
screen.pt2 = makepoint(XMAX, YMAX);
middle = makepoint((screen.pt1.x + screen.pt2.x)/2,
(screen.pt1.y + screen.pt2.y)/2);
```

The next step is a set of functions to do arithmetic on points. For instance,

```
/* addpoints: add two points */
struct addpoint(struct point p1, struct point p2)
{
p1.x += p2.x;
p1.y += p2.y;
return p1;
}
```

Here both the arguments and the return value are structures. We incremented the components in p1 rather than using an explicit temporary variable to emphasize that structure parameters are passed by value like any others. As another example, the function ptinrect tests whether a point is inside a rectangle, where we have adopted the convention that a rectangle includes its left and bottom sides but not its top and right sides:

```
/* ptinrect: return 1 if p in r, 0 if not */
int ptinrect(struct point p, struct rect r)
{
return p.x >= r.pt1.x && p.x < r.pt2.x
&& p.y >= r.pt1.y && p.y < r.pt2.y;
}
```

This assumes that the rectangle is presented in a standard form where the pt1 coordinates are less than the pt2 coordinates.

Arrays of Structures:

Consider writing a program to count the occurrences of each C keyword. We need an array of character strings to hold the names, and an array of integers for the counts. One possibility is to use two parallel arrays, keyword and keycount, as in

```
char *keyword[NKEYS];
int keycount[NKEYS];
```

But the very fact that the arrays are parallel suggests a different organization, an array of structures. Each keyword is a pair:

```
char *word;
int cout;
```

and there is an array of pairs. The structure declaration

```
struct key {
char *word;
int count;
} keytab[NKEYS];
```

declares a structure type key, defines an array keytab of structures of this type, and sets aside storage for them. Each element of the array is a structure. This could also be written

```
struct key {
char *word;
int count;
};
struct key keytab[NKEYS];
```

Unions

A *union* is a variable that may hold (at different times) objects of different types and sizes, with the compiler keeping track of size and alignment requirements. Unions provide a way to manipulate different kinds of data in a single area of storage, without embedding any machine-dependent information in the program. They are analogous to variant records in pascal. As an example such as might be found in a compiler symbol table manager, suppose that a constant may be an int, a float, or a character pointer. The value of a particular constant must be stored in a variable of the proper type, yet it is most convenient for table management if the value occupies the same amount of storage and is stored in the same place regardless of its type. This is the purpose of a union - a single variable that can legitimately hold any of one of several types. The syntax is based on structures:

```
union u_tag {
int ival;
float fval;
char *sval;
} u;
```

The variable u will be large enough to hold the largest of the three types; the specific size is implementation-dependent. Any of these types may be assigned to u and then used in expressions, so long as the usage is consistent: the type retrieved must be the type most recently stored. It is the programmer's responsibility to keep track of which type is currently stored in a union; the results are implementation-dependent if something is stored as one type and extracted as another. Syntactically, members of a union are accessed as

union-name.member

or

union-pointer->member

just as for structures. If the variable utype is used to keep track of the current type stored in u, then one might see code such as

```
if (utype == INT)
printf("%d\n", u.ival);
if (utype == FLOAT)
printf("%f\n", u.fval);
if (utype == STRING)
printf("%s\n", u.sval);
else
printf("bad type %d in utype\n", utype);
```

Unions may occur within structures and arrays, and vice versa. The notation for accessing a member of a union in a structure (or vice versa) is identical to that for nested structures. For example, in the structure array defined by

```
struct {
char *name;
int flags;
int utype;
union {
int ival;
float fval;
char *sval;
} u;
} symtab[NSYM];
```

SLNO	**QUESTION**
1	What is the self-referential structure?
	When a structure is nested within the same structure and nested structure variable is a pointer is known as self-referential structure

2	What is slack byte of the structure?
	Every structure allocates memory in form of block, So when the data member of the structure doesn't properly accommodate in the block ,few bytes of memory is lost in that block, which is knows as slack byte or undefined byte of the structure.
3	What is the difference between structure and union
	Structure: 1. Each data members of a structure begins at different location 2. Size of structure is size of all data members Union: 1. Each data members of a structure begins at same location.
4	What two structure variable can't be compared?
	Two structure variable can't be compared , because of slack byte.
5	What do you mean by "active data member " of a union
	The first data member of a union is known as the active data member , which should be longest data member of the union.
6	What is application of structure?
	The application of structure is ☐ To create memory link ☐ Data encapsulation
7	What is the use of bit-field?
	Bit-field is created using a structure. It is used to create the protocol header and developing different fonts
8	What is the application of union?
	Application of union is : to create a share memory , so locking mechanism will be implemented
9	How to compare between two different structure?
	Only using byte-order comparison
10	What is application of self-referential structure?
	Self referential structure is used only for memory link.

LABORATORY EXPERIMENT

Passing structure array to function

1. Array of Structure can be passed **to function as a Parameter**.
2. Function can also return Structure as **return type**.

Note: When reference (i.e ampersand) is not specified in main , so this passing is simple pass by value. If you are passing by reference then no need of return type

/* student DEFINITION and Declaration */

```
struct student{
long int regno;
char name[20];
char class[10];
char branch[10];
};

typedef struct student cse;

/*operation in structure*/

void record_add(cse[],int);

void record_view(cse[]);

int main()
{

cse s[3];
int i=0, ch;

while(1)
{
printf("\n1: Add");

printf("\n2: view");

printf("\n3: Quit");

printf("\n Enter your choice");

scanf("%d",&ch);

switch(ch)
{

case 1:
   /* pass by value */
    record_add(s,3);

    break;

case 2:
  /* call by value */
    record_view(s);

    break;

case 3:
    return 0;

default:
    printf("\n Invalid choice");

}
}
return 0;
}

void record_add(cse std[],int n)
{
int i:
printf("\n enter student details");
for(i=0;i<n;i++)
{
printf("\n Regno: ");scanf("%ld",&std[i].regno);

printf("\n Name: ");scanf("%s",std[i].name);

printf("\n Class: "); scanf("%s",std[i].class);

printf("\n Branch: ");scanf("%s",std[i].branch);
}

}

void record_view(cse std[])
{
   int i:
   printf("\n enter the serial number");
   scanf("%d".&i):
   printf("\n %dth student details ",i);

   printf("\n Regno: %ld",std[i-1].regno);

   printf("\n Name: %s",std[i-1].name);

   printf("\n Class: %s",std[i-1].class);

   printf("\n Branch: %s",std[i-1].branch);


}
```

OUTPUT*****

```
1: Add
2: view
3: Quit
 Enter your choice
1

 enter student details
 Regno: 1510000010

 Name: RAJESH

 Class: 1stSem

 Branch: CSE

 Regno: 1510000012
 Name: KABITA

 Class: 1stSem

 Branch: ECE

 Regno: 1510000013

 Name: RAHUL

 Class: 1stSem

 Branch: ME

1: Add
2: view
3: Quit
 Enter your choice2

 enter the serial number1

 1th student details
 Regno: 1510000010
 Name: RAJESH
 Class: 1stSem
 Branch: CSE
1: Add
2: view
3: Quit
 Enter your choice2

 enter the serial number2

 2th student details
 Regno: 1510000012
 Name: KABITA
 Class: 1stSem
 Branch: ECE
1: Add
2: view
3: Quit
 Enter your choice
```

FILE HANDLING

The examples so far have all read the standard input and written the standard output, which are automatically defined for a program by the local operating system. The next step is to write a program that accesses a file that is *not* already connected to the program. One program that illustrates the need for such operations is cat, which concatenates a set of named files into the standard output. cat is used for printing files on the screen, and as a general-purpose input collector for programs that do not have the capability of accessing files by name. For example, the command cat x.c y.c prints the contents of the files x.c and y.c (and nothing else) on the standard output. The question is how to arrange for the named files to be read - that is, how to connect the external names that a user thinks of to the statements that read the data.
The rules are simple. Before it can be read or written, a file has to be *opened* by the library function fopen. fopen takes an external name like x.c or y.c, does some housekeeping and negotiation with the operating system (details of which needn't concern us), and returns a pointer to be used in subsequent reads or writes of the file. This pointer, called the *file pointer*, points to a structure that contains information about the file, such as the location of a buffer, the current character position in the buffer, whether the file is being read or written, and whether errors or end of file have occurred. Users don't need to know the details, because the definitions obtained from <stdio.h> include a structure declaration called FILE. The only declaration needed for a file pointer is exemplified by

```
    FILE *fp;
    FILE *fopen(char *name, char *mode);
```

This says that fp is a pointer to a FILE, and fopen returns a pointer to a FILE. Notice that FILE is a type name, like int, not a structure tag; it is defined with a typedef.
The call to fopen in a program is

```
    fp = fopen(name, mode);
```

The first argument of fopen is a character string containing the name of the file. The second argument is the *mode*, also a character string, which indicates how one intends to use the file. Allowable modes include read ("r"), write ("w"), and append ("a"). Some systems distinguish between text and binary files; for the latter, a "b" must be appended to the mode string.
If a file that does not exist is opened for writing or appending, it is created if possible. Opening an existing file for writing causes the old contents to be discarded, while opening for appending preserves them. Trying to read a file that does not exist is an error, and there may be other causes of error as well, like trying to read a file when you don't have permission. If there is any error, fopen will return NULL.

The next thing needed is a way to read or write the file once it is open. getc returns the next character from a file; it needs the file pointer to tell it which file.

```
    int getc(FILE *fp)
```

getc returns the next character from the stream referred to by fp; it returns EOF for end of file or error.
putc is an output function:

```
    int putc(int c, FILE *fp)
```

putc writes the character c to the file fp and returns the character written, or EOF if an error occurs. Like getchar and putchar, getc and putc may be macros instead of functions.

When a C program is started, the operating system environment is responsible for opening three files and providing pointers for them. These files are the standard input, the standard output, and the standard error; the corresponding file pointers are called stdin, stdout, and stderr, and are declared in <stdio.h>. Normally stdin is connected to the keyboard and stdout and stderr are connected to the screen, but stdin and stdout may be redirected to files or pipes.

getchar and putchar can be defined in terms of getc, putc, stdin, and stdout as follows:

```
    #define getchar() getc(stdin)
    #define putchar(c) putc((c), stdout)
```

For formatted input or output of files, the functions fscanf and fprintf may be used. These are identical to scanf and printf, except that the first argument is a file pointer that specifies the file to be read or written; the format string is the second argument.

```
    int fscanf(FILE *fp, char *format, ...)
    int fprintf(FILE *fp, char *format, ...)
```

With these preliminaries out of the way, we are now in a position to write the program cat to concatenate files. The design is one that has been found convenient for many programs. If there are command-line arguments, they are interpreted as filenames, and processed in order. If there are no arguments, the standard input is processed.

```
#include <stdio.h>
/* cat: concatenate files, version 1 */
main(int argc, char *argv[])
{
FILE *fp;
void filecopy(FILE *, FILE *)
if (argc == 1) /* no args; copy standard input */
filecopy(stdin, stdout);
else
while(--argc > 0)
if ((fp = fopen(*++argv, "r")) == NULL) {
printf("cat: can’t open %s\n, *argv);
return 1;
} else {
filecopy(fp, stdout);
fclose(fp);
}
return 0;
}
/* filecopy: copy file ifp to file ofp */
void filecopy(FILE *ifp, FILE *ofp)
{
int c;
while ((c = getc(ifp)) != EOF)
putc(c, ofp);
}
```

The file pointers stdin and stdout are objects of type FILE *. They are constants, however, *not* variables, so it is not possible to assign to them. The function

```
int fclose(FILE *fp)
```

is the inverse of fopen, it breaks the connection between the file pointer and the external name that was established by fopen, freeing the file pointer for another file. Since most operating systems have some limit on the number of files that a program may have open simultaneously, it’s a good idea to free the file pointers when they are no longer needed, as we did in cat. There is also another reason for fclose on an output file - it flushes the buffer in which putc is collecting output. fclose is called automatically for each open file when a program terminates normally. (You can close stdin and stdout if they are not needed. They can also be reassigned by the library function freopen.)

Error Handling - Stderr and Exit

The treatment of errors in cat is not ideal. The trouble is that if one of the files can’t be accessed for some reason, the diagnostic is printed at the end of the concatenated output. That might be acceptable if the output is going to a screen, but not if it’s going into a file or into another program via a pipeline.

To handle this situation better, a second output stream, called stderr, is assigned to a program in the same way that stdin and stdout are. Output written on stderr normally appears on the screen even if the standard output is redirected.

Let us revise cat to write its error messages on the standard error.

```
#include <stdio.h>
/* cat: concatenate files, version 2 */
main(int argc, char *argv[])
{
FILE *fp;
void filecopy(FILE *, FILE *);
char *prog = argv[0]; /* program name for errors */
if (argc == 1 ) /* no args; copy standard input */
filecopy(stdin, stdout);
else
while (--argc > 0)
if ((fp = fopen(*++argv, "r")) == NULL) {
fprintf(stderr, "%s: can’t open %s\n",
prog, *argv);
exit(1);
} else {
filecopy(fp, stdout);
fclose(fp);
}
if (ferror(stdout)) {
fprintf(stderr, "%s: error writing stdout\n", prog);
exit(2);
}
exit(0);
}
```

The program signals errors in two ways. First, the diagnostic output produced by fprintf goes to stderr, so it finds its way to the screen instead of disappearing down a pipeline or into an output file. We included the program name, from argv[0], in the message, so if this program is used with others, the source of an error is identified.

SLNO	QUESTIONS
1	What is the difference between EOF and End Of File?
	EOF is a macro , whose expansion values is -1, where as End Of File is the ASCII value of CTRL+Z character (26) , which represents end of file character.
2	What is the difference between “w” and “w+”
3	“w” and “w+” both are used as the file opening modes. In “w” mode program only does write operations but in “w+” mode program does both read and write operation.
4	What are the different types of files
5	Linux operating system supports seven different types of file, such Regular files Directory file FIFO file Character special file Block special file
6	What are the different file opening modes
	There are three different file opening modes such as, Read mode Write mode
7	What is the difference between fseek and lseek
	fseek and lseek , both are used to transfer the control to any position of a file, where as the fseek is a standard library function and lseek is a system call.
8	What is the difference between file and directory
	File is name reference to an inode number in disk, where it contains data. And directory also the name reference of inode number in disk , where it contains file and directory.
9	What is file system?
	File system is the data structure used by the operating system to organize files and directory.
10	What are the active file pointer in C
	In linux file system supports two different active file pointer in C, such as *__IO __ptr_read and* __IO__ptr_write
11	What is the maximum size of a file
	The size of the file depends on the file system used by the operating system

LABORATORY EXPERIMENT

File handling in doing student details operation

```
******SOURCE CODE***
#include<stdio.h>
/* student details */

struct student{
long int regno;
char name[20];
char class[10];
char branch[10];
};

typedef struct student cse;
/*list of global files that can be access form anywhere in program*/
FILE *fp;
char path[100]="C:\\Users\\sony\\Desktop\\novel_computer_virus\\";
/*operation in structure*/

void record_add(cse[],int);

void record_view();

int main()
{
cse s[2];
int i=0, ch;

while(1)
{
printf("\n1: Add");

printf("\n2: view");

printf("\n3: Quit");

printf("\n Enter your choice");

scanf("%d",&ch);

switch(ch)
{

case 1:
  /* pass by value */
   record_add(s,2);

   break;

case 2:
 /* call by value */
   record_view();

   break;
case 3:
   return 0;

default:
   printf("\n Invalid choice");

}
}

return 0;
}

void record_add(cse std[],int n)
{
int i;
char temp[100];
strcpy(temp,path);
strcat(temp,"HIT_Student.dat");

printf("\n enter student details");
for(i=0;i<n;i++)
{
fp=fopen(temp,"ab+");

printf("\n Regno: ");scanf("%ld",&std[i].regno);

printf("\n Name: ");scanf("%s",std[i].name);

printf("\n Class: "); scanf("%s",std[i].class);

printf("\n Branch: ");scanf("%s",std[i].branch);

fseek(fp,0,SEEK_END);
fwrite(&std[i],sizeof(std[i]),1,fp);
fclose(fp);

printf("The record is sucessfully saved\n");
}

}

void record_view()
{
   int i=0;
   char temp[100];

   cse std;


   strcpy(temp,path);
   strcat(temp,"HIT_Student.dat");

   fp=fopen(temp,"rb");

   while(fread(&std,sizeof(std),1,fp)==1)
   {
  i++;
   printf("\n %dth student details ",i);

   printf("\n Regno: %ld",std.regno);

   printf("\n Name: %s",std.name);
```

```
    printf("\n Class: %s",std.class);

    printf("\n Branch: %s",std.branch);

}
fclose(fp);
          }
```

UNIT 5
BASIC ALGORITHMS (6 LECTURES)

Searching (Linear and Binary), Basic Sorting Algorithms (Bubble sort)

Write and explain linear search procedure or algorithm with a suitable example.

Linear search technique is also known as sequential search technique. The linear search is a method of searching an element in a list in sequence. In this method, the array is searched for the required element from the beginning of the list/array or from the last element to first element of array and continues until the item is found or the entire list/array has been searched.

Algorithm:

```
Step 1: set-up a flag to indicate “element not found”
Step 2: Take the first element in the list
Step 3: If the element in the list is equal to the desired element
            Set flag to “element found”
            Display the message “element found in the list”
            Go to step 6
Step 4: If it is not the end of list,
            Take the next element in the list
            Go to step 3
Step 5: If the flag is “element not found”
        Display the message “element not found”
Step 6: End of the Algorithm
```

Binary search

Formulate recursive algorithm for binary search with its timing analysis. Binary search is quicker than the linear search. However, it cannot be applied on unsorted data structure. The binary search is based on the approach divide-and-conquer. The binary search starts by testing the data in the middle element of the array. This determines target is whether in the first half or second half. If target is in first half, we do not need to check the second half and if it is in second half no need to check in first half. Similarly we repeat this process until we find target in the list or not found from the list. Here we need 3 variables to identify first, last and middle elements.

To implement binary search method, the elements must be in sorted order. Search is performed as follows:

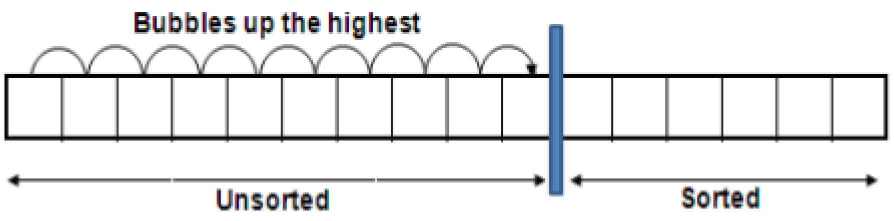
- The key is compared with item in the middle position of an array
- If the key matches with item, return it and stop
- If the key is less than mid positioned item, then the item to be found must be in first half of array, otherwise it must be in second half of array.
- Repeat the procedure for lower (or upper half) of array until the element is found.

Recursive Algorithm:

```
Binary_Search(a,key,lb,ub)
Begin
 Step 1: [initialization] lb=0 ub=n-1;
Step 2: [search for the item] Repeat through step 4 while lower bound(lb) is less than upper bound.
Step 3: [obtain the index of middle value] mid = (lb+ub)/2
Step 4: [compare to search for item]
            if(key < a[mid]) then
              ub=mid-1
          otherwise if( key > a[mid]) then
             lb=mid+1;
        otherwise if(key==a[mid]) Write "match found"
            return (mid)
      return Binary_Search(a,key,lb,ub)
Step 5: [unsuccessful search] Write "match not found" S
Step 6: [end of algorithm]
```

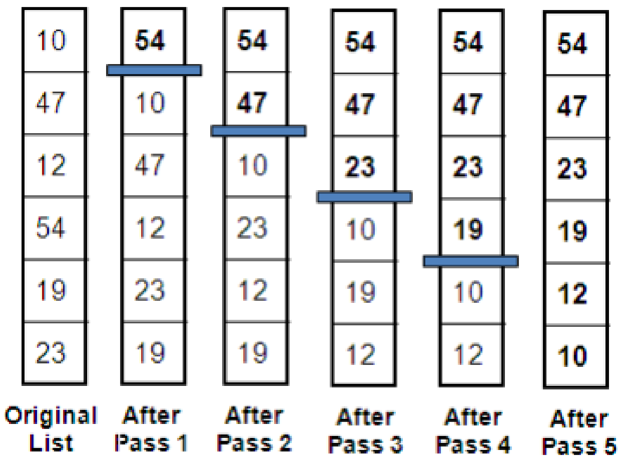
Explain the algorithm for bubble sort and give a suitable example.

In bubble sort method the list is divided into two sub-lists sorted and unsorted. The smallest element is bubbled from unsorted sub-list. After moving the smallest element the imaginary wall moves one element ahead. The bubble sort was originally written to bubble up the highest element in the list. But there is no difference whether highest / lowest element is bubbled. This method is easy to understand but time consuming. In this type, two successive elements are compared and swapping is done. Thus, step-by-step entire array elements are checked. Given a list of 'n' elements the bubble sort requires up to n-1 passes to sort the data.



Algorithm for Bubble Sort: Bubble_Sort (A [ ] , N)

```
Step 1 : Repeat For P = 1 to N – 1 Begin
Step 2 : Repeat For J = 1 to N – P Begin
Step 3 : If ( A [ J ] < A [ J – 1 ] ) Swap ( A [ J ] , A [ J – 1 ] )
            End For
        End For
Step 4 : Exit
```



Example: A list of unsorted elements are: 10 47 12 54 19 23 (Bubble up for highest value shown here) as show in above figure.

Q.1	Write a program for linear search
	```
	#include "stdio.h"
	int linearsearch(int [],int,int);
	main()
	{
	        int x[8]={10,20,30,40,50,60,70,80};
	        int i,n,status;
	        printf("enter number to search");
	        scanf("%d",&n);
	        status=binarysearch(x,8,n);
	        if(status==1)
	                printf("found");
	        else
	                printf("Not found");
	}
	int binarysearch(int x[],int size,int n)
	{
	        int i;
	        for(i=0;i<size;i++)
	        {
	                if(n==x[i])
	                        return 1;

	        }
	        return 0;
	}
	```
Q.2	Write a program for binary Search
	```
	#include "stdio.h"
	int binarysearch(int [],int,int,int);
	```

```
main()
{
            int x[8]={10,20,30,40,50,60,70,80};
            int i,n,status;
            printf("enter number to search");
            scanf("%d",&n);
            status=binarysearch(x,0,7,n);
            if(status==1)
                        printf("found");
            else
                        printf("Not found");
}
int binarysearch(int x[],int low,int up,int n)
{
            int mid;
            while(low<=up)
            {
                        mid=(low+up)/2;
                        if(n==x[mid])
                                    return 1;
                        else
                        if(n < x[mid])
                                    up=mid-1;
                        else
                                    low=mid+1;
            }
            return 0;
}
```

Q.3 Write a program for bubble sort

```
#include “stdio.h”
Void bubble_sort(int [],int);
main()
{
      int a[5]={4,9,40,2,25};
      int i;
      bubble_sort(a,5);
      for(i=0;i<5;i++)
            printf("%d ",a[i]);
}

void bubble_sort(int a[5],int size)
{
      int i,j,temp;
      for(i=0;i<s;i++)
      {
            for(j=0;j<s-i;j++)
            {
                  if(a[j]>a[j+1])
                  {
                        temp=a[j];
                        a[j]=a[j+1];
                        a[j+1]=temp;
                  }
            }
      }
}
```

Q.4 Write a program for linear sort

```
#include “stdio.h”
void linear_sort(int [],int);
main()
{
    int a[5]={4,9,40,2,25};
    int i;
    bubble_sort(a,5);
    for(i=0;i<5;i++)
        printf("%d ",a[i]);
}

void linear_sort(int a[5],int size)
{
    int i,j,temp;
    for(i=0;i<s;i++)
    {
        for(j=i+1;j<s;j++)
        {
            if(a[i]>a[j])
            {
                temp=a[i];
                a[i]=a[j];
                a[j]=temp;
            }
        }
    }
}
```

Q.5 Write a program to insert an element in appropriate position of sorted array

```
#include "stdio.h"
main()
{
    int x[8];
    int i,j,n;
    printf("Enter the elements for an array in sorted order");
    for(i=0;i<8;i++)
    {
        scanf("%d",&x[i]);
    }
    printf("Enter the element to insert");
    scanf("%d",&n);
    i=0;
    while(i<8 && n>x[i])
    {
        i++;
    }
    for(j=7;j>i;j--)
        x[j]=x[j-1];
    x[j]=n;
    for(i=0;i<8;i++)
        printf("%d ",x[i]);
}
```

SURPRISE TEST

Question and Answer

1. ***What are data type qualifiers?***
   const and volatile both are used for data type qualifiers. Const makes a variable read-only type. But volatile used to modify constant value using pointer during compiler optimization.
2. ***What is the difference between call by value and call by reference?***
   In case of call by value , the actual argument value can't be changed , but in case of call by address the actual argument value can be changed.
3. ***Can modulus operator be applied to float variables?***
   No
4. ***What is the importance of main( ) in C language?***
   Execution of a c program begins from main function and main function pass control to other function for execution.
5. ***What do you mean by structured programming approach?***

6. ***What is the use of sizeof operator?***

   sizeof is an operator which shows how much memory is reserved by different variable or object.

MISCELLANEOUS

STORAGE CLASS, HEADER FILES, FILE INCLUSION

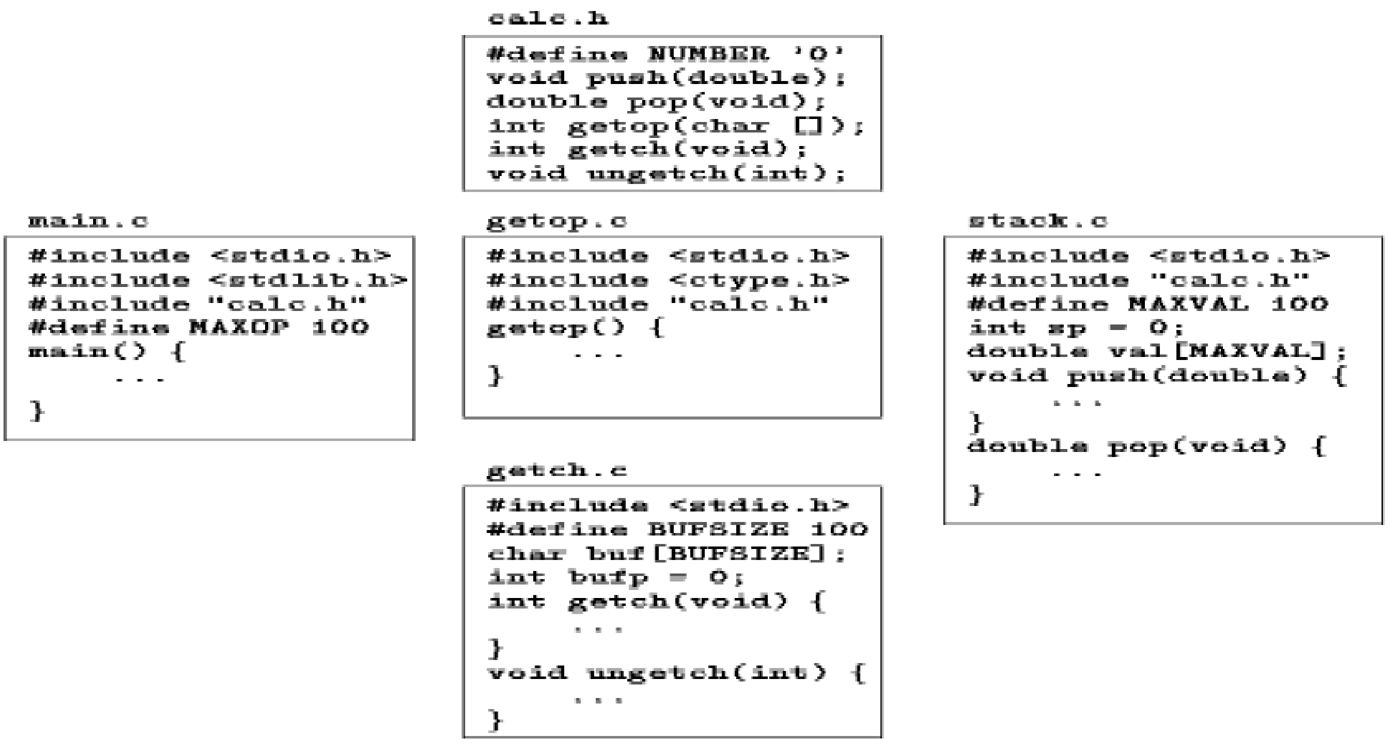
Storage Classes

SLNO	QUESTIONS	SLNO	QUESTIONS
1	What is storage class?	4	Why pointer doesn't refer a register variable?
	Storage class is a data structure used by every C compiler which decides the scope,life ,default initial value and storage of a variable and functions.		Registers are used as the memory element of the microprocessor. They have no address, So pointer doesn't refers to a register variable.
2	What is the difference between program scoping	5	What is the use static storage class in C?
2	What is the difference between program scoping and file scoping		Static is used in two different places, such as
	Program scoping: If a variable or function is defined in one file, and that can be accessed in other file of a C program, is known as program scoping. File scoping: If a variable or function is defined in one file, and that can't be accessed in other file of a C program, but only can be access in same file, is known as file scoping.		Static is used in two different places, such as To avoid the reinitialisation : when static is used as the local variable File scoping: when static is used as global variable.
3	What is difference between writing static as local and global variable	6	What are the different scope rules in C? Different scope rules in C is
	Static is used in two different places, such as Static is used in two different places, such as To avoid the reinitialisation : when static is used as the local variable File scoping: when static is used as global		Program scoping File scoping Function scoping Block scoping

Header files:

Let is now consider dividing the calculator program into several source files, as it might be is each of the components were substantially bigger. The main function would go in one file, which we will call main.c; push, pop, and their variables go into a second file, stack.c; getop goes into a third, getop.c. Finally, getch and ungetch go into a fourth file, getch.c; we separate them from the others because they would come from a separately-compiled library in a realistic program. There is one more thing to worry about - the definitions and declarations shared among files. As much as possible, we want to centralize this, so that there is only one copy to get and keep right as the program evolves. Accordingly, we will place this common material in a header file, calc.h, which will be included as necessary. (The #include line is described in Section

The resulting program then looks like this:



calc.h

```
#define NUMBER '0'
void push(double);
double pop(void);
int getop(char []);
int getch(void);
void ungetch(int);
```

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include "calc.h"
#define MAXOP 100
main() {
    ...
}
```

getop.c

```
#include <stdio.h>
#include <ctype.h>
#include "calc.h"
getop() {
    ...
}
```

stack.c

```
#include <stdio.h>
#include "calc.h"
#define MAXVAL 100
int sp = 0;
double val[MAXVAL];
void push(double) {
    ...
}
double pop(void) {
    ...
}
```

getch.c

```
#include <stdio.h>
#define BUFSIZE 100
char buf[BUFSIZE];
int bufp = 0;
int getch(void) {
    ...
}
void ungetch(int) {
    ...
}
```

Compile and load a C program that resides on multiple source files

The mechanics of how to compile and load a C program that resides on multiple source files vary from one system to the next. On the UNIX system, for example, the cc command mentioned in Chapter 1 does the job. Suppose that the three functions are stored in three files called main.c, getline.c, and strindex.c. Then the command

```
cc main.c getline.c strindex.c
```

compiles the three files, placing the resulting object code in files main.o, getline.o, and strindex.o, then loads them all into an executable file called a.out. If there is an error, say in main.c, the file can be recompiled by itself and the result loaded with the previous object files, with the command

```
cc main.c getline.o strindex.o
```

The cc command uses the ‘‘.c’’ versus ‘‘.o’’ naming convention to distinguish source files from object files.

File inclusion:

File inclusion makes it easy to handle collections of #defines and declarations (among other things). Any source line of the form

```
#include "filename"
or
#include <filename>
```

is replaced by the contents of the file *filename*. If the *filename* is quoted, searching for the file typically begins where the source program was found; if it is not found there, or if the name is enclosed in < and >, searching follows an implementation-defined rule to find the file. An included file may itself contain #include lines. There are often several #include lines at the beginning of a source file, to include common #define statements and extern declarations, or to access the function prototype declarations for library functions from headers like <stdio.h>. (Strictly speaking, these need not be files; the details of how headers are accessed are implementation-dependent.)