1.

What will be the output of the following code?

```
void main ()
{ int i = 0 , a[3] ;
a[i] = i++;
printf (“%d",a[i]) ;
}
```

Ans: The output for the above code would be a garbage value. In the statement a[i] = i++; the value of the variable i would get assigned first to a[i] i.e. a[0] and then the value of i would get incremented by 1. Since a[i] i.e. a[1] has not been initialized, a[i] will have a garbage value.

2.

Why doesn't the following code give the desired result?

```
int x = 3000, y = 2000 ;
long int z = x * y ;
```

Ans: Here the multiplication is carried out between two ints x and y, and the result that would overflow would be truncated before being assigned to the variable z of type long int. However, to get the correct output, we should use an explicit cast to force long arithmetic as shown below:

```
long int z = ( long int ) x * y ;
```

Note that (long int)(x * y) would not give the desired effect.

3.

Why doesn't the following statement work?

```
char str[ ] = "Hello" ;
strcat ( str, '!' ) ;
```

Ans: The string function strcat( ) concatenates strings and not a character. The basic difference between a string and a character is that a string is a collection of characters, represented by an array of characters whereas a character is a single character. To make the above statement work writes the statement as shown below:

```
strcat ( str, "!" ) ;
```

4.

How do I know how many elements an array can hold?

Ans: The amount of memory an array can consume depends on the data type of an array. In DOS environment, the amount of memory an array can consume depends on the current memory model (i.e. Tiny, Small, Large, Huge, etc.). In general an array cannot consume more than 64 kb. Consider followingprogram, which shows the maximum number of elements an array of type int, float and char can have in case of Small memory model.

```
main( )
{
int i[32767] ;
float f[16383] ;
char s[65535] ;
}
```

5.

How do I write code that reads data at memory location specified by segment and offset?

Ans: Use peekb( ) function. This function returns byte(s) read from specific segment and offset locations in memory. The followingprogram illustrates use of this function. In this program from VDU memory we have read characters and its attributes of the first row. The information stored in file is then further read and displayed using peek( ) function.

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

main( )
{

char far *scr = 0xB8000000 ;
FILE *fp ;
int offset ;
char ch ;

if ( ( fp = fopen ( "scr.dat", "wb" ) ) == NULL )
{

printf ( "\nUnable to open file" ) ;
exit( ) ;

}

// reads and writes to file
for ( offset = 0 ; offset < 160 ; offset++ )
```

```
fprintf ( fp, "%c", peekb ( scr, offset ) ) ;
fclose ( fp ) ;

if ( ( fp = fopen ( "scr.dat", "rb" ) ) == NULL )
{

printf ( "\nUnable to open file" ) ;
exit( ) ;

}

// reads and writes to file
for ( offset = 0 ; offset < 160 ; offset++ )
{

fscanf ( fp, "%c", &ch ) ;
printf ( "%c", ch ) ;

}

fclose ( fp ) ;

}
```

6.
How do I compare character data stored at two different memory locations?

Ans: Sometimes in a program we require to compare memory ranges containing strings. In such a situation we can use functions like memcmp( ) or memicmp( ). The basic difference between two functions is that memcmp( ) does a case-sensitive comparison whereas memicmp( ) ignores case of characters. Followingprogram illustrates the use of both the functions.

```
#include <mem.h>

main( )
{
char *arr1 = "Kicit" ;
char *arr2 = "kicitNagpur" ;

int c ;

c = memcmp ( arr1, arr2, sizeof ( arr1 ) ) ;

if ( c == 0 )
printf ( "\nStrings arr1 and arr2 compared using memcmp are identical" ) ;
```

```
else
printf ( "\nStrings arr1 and arr2 compared using memcmp are not identical"
) ;

c = memicmp ( arr1, arr2, sizeof ( arr1 ) ) ;

if ( c == 0 )
printf ( "\nStrings arr1 and arr2 compared using memicmp are identical" )
;
else
printf ( "\nStrings arr1 and arr2 compared using memicmp are not
identical" ) ;
}
```

7.

Fixed-size objects are more appropriate as compared to variable size data objects. Using variable-size data objects saves very little space. Variable size data objects usually have some overhead. Manipulation of fixed-size data objects is usually faster and easier. Use fixed size when maximum size is clearly bounded and close to average. And use variable-size data objects when a few of the data items are bigger than the average size. For example,

```
char *num[10] = { "One", "Two", "Three", "Four",
"Five", "Six", "Seven", "Eight", "Nine", "Ten" } ;
```

Instead of using the above, use

```
char num[10][6] = { "One", "Two", "Three", "Four",
"Five", "Six", "Seven", "Eight", "Nine", "Ten" } ;
```

The first form uses variable-size data objects. It allocates 10 pointers, which are pointing to 10 string constants of variable size. Assuming each pointer is of 4 bytes, it requires 90 bytes. On the other hand, the second form uses fixed size data objects. It allocates 10 arrays of 6 characters each. It requires only 60 bytes of space. So, the variable-size in this case does not offer any advantage over fixed size.

8.

The Spawnl( ) function...

DOS is a single tasking operating system, thus only one program runs at a time. The Spawnl( ) function provides us with the capability of starting the execution of one program from within another program. The first program is called the parent process and the second program that gets called from within the first program is called a child process. Once the second program starts execution, the first is put on hold until the second program completes

execution. The first program is then restarted. The following program demonstrates use of spawnl( ) function.

```
/* Mult.c */

int main ( int argc, char* argv[ ] )
{
int a[3], i, ret ;
if ( argc < 3 || argc > 3 )
{
printf ( "Too many or Too few arguments..." ) ;
exit ( 0 ) ;
}

for ( i = 1 ; i < argc ; i++ )
a[i] = atoi ( argv[i] ) ;
ret = a[1] * a[2] ;
return ret ;
}

/* Spawn.c */
#include <process.h>
#include <stdio.h>

main( )
{
int val ;
val = spawnl ( P_WAIT, "C:\\Mult.exe", "3", "10",
"20", NULL ) ;
printf ( "\nReturned value is: %d", val ) ;
}
```

Here, there are two programs. The program 'Mult.exe' works as a child process whereas 'Spawn.exe' works as a parent process. On execution of 'Spawn.exe' it invokes 'Mult.exe' and passes the command-line arguments to it. 'Mult.exe' in turn on execution, calculates the product of 10 and 20 and returns the value to val in 'Spawn.exe'. In our call to spawnl( ) function, we have passed 6 parameters, P_WAIT as the mode of execution, path of '.exe' file to run as childprocess, total number of arguments to be passed to the child process, list of command line arguments and NULL. P_WAIT will cause our application to freeze execution until the child process has completed its execution. This parameter needs to be passed as the default parameter if you are working under DOS. under other operating systems that support multitasking, this parameter can be P_NOWAIT or P_OVERLAY. P_NOWAIT will cause the parentprocess to execute along with the child process, P_OVERLAY will load the child process on top of the parent process in the memory.

9.

Are the following two statements identical?

char str[6] = "Kicit" ;
char *str = "Kicit" ;
Ans: No! Arrays are not pointers. An array is a single, pre-allocated chunk of contiguous elements (all of the same type), fixed in size and location. A pointer on the other hand, is a reference to any data element (of a particular type) located anywhere. A pointer must be assigned to point to space allocated elsewhere, but it can be reassigned any time. The array declaration char str[6] ; requests that space for 6 characters be set aside, to be known by name str. In other words there is a location named str at which six characters are stored. The pointer declaration char *str ; on the other hand, requests a place that holds a pointer, to be known by the name str. This pointer can point almost anywhere to any char, to any contiguous array of chars, or nowhere.

10.

Is the following code fragment correct?

const int x = 10 ;
int arr[x] ;
Ans: No! Here, the variable x is first declared as an int so memory is reserved for it. Then it is qualified by a const qualifier. Hence, const qualified object is not a constant fully. It is an object with read only attribute, and in C, an object associated with memory cannot be used in array dimensions.

11.

How do I write code to retrieve current date and time from the system and display it as a string?

Ans: Use time( ) function to get current date and time and then ctime( ) function to display it as a string. This is shown in following code snippet.

```
#include <sys\types.h>

void main( )
{
time_t curtime ;
char ctm[50] ;

time ( &curtime ) ; //retrieves current time &
stores in curtime
printf ( "\nCurrent Date & Time: %s", ctime (
&curtime ) ) ;
}
```

12.

How do I change the type of cursor and hide a cursor?

Ans: We can change the cursor type by using function _setcursortype( ). This function can change the cursor type to solid cursor and can even hide a cursor. Following code shows how to change the cursor type and hide cursor.

```
#include <conio.h>
main( )
{
/* Hide cursor */
_setcursortype ( _NOCURSOR ) ;

/* Change cursor to a solid cursor */
_setcursortype ( _SOLIDCURSOR ) ;

/* Change back to the normal cursor */
_setcursortype ( _NORMALCURSOR ) ;
}
```

13.

How do I write code that would get error number and display error message if any standard error occurs?

Ans: Following code demonstrates this.

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

main( )
{
char *errmsg ;
FILE *fp ;
fp = fopen ( "C:\file.txt", "r" ) ;
if ( fp == NULL )
{
errmsg = strerror ( errno ) ;
printf ( "\n%s", errmsg ) ;
}
}
```

Here, we are trying to open 'file.txt' file. However, if the file does not exist, then it would cause

an error. As a result, a value (in this case 2) related to the error generated would get set in errno. errno is an external int variable declared in 'stdlib.h' and also in 'errno.h'. Next, we have called sterror( ) function which takes an error number and returns a pointer to standard error message related to the given error number.

14.

How do I write code to get the current drive as well as set the current drive?

Ans: The function getdisk( ) returns the drive number of current drive. The drive number 0 indicates 'A' as the current drive, 1 as 'B' and so on. The Setdisk( ) function sets the current drive. This function takes one argument which is an integer indicating the drive to be set. Following program demonstrates use of both the functions.

```
#include <dir.h>

main( )
{
int dno, maxdr ;

dno = getdisk( ) ;
printf ( "\nThe current drive is: %c\n", 65 + dno
) ;

maxdr = setdisk ( 3 ) ;
dno = getdisk( ) ;
printf ( "\nNow the current drive is: %c\n", 65 +
dno ) ;
}
```

15.

The functions memcmp( ) and memicmp( )

The functions memcmp( ) and memicmp( ) compares first n bytes of given two blocks of memory or strings. However, memcmp( ) performs comparison as unsigned chars whereas memicmp( ) performs comparison as chars but ignores case (i.e. upper or lower case). Both the functions return an integer value where 0 indicates that two memory buffers compared are identical. If the value returned is greater than 0 then it indicates that the first buffer is bigger than the second one. The value less than 0 indicate that the first buffer is less than the second buffer. The following code snippet demonstrates use of both

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

main( )
```

```
{
char str1[] = "This string contains some
characters" ;
char str2[] = "this string contains" ;
int result ;

result = memcmp ( str1, str2, strlen ( str2 ) ) ;
printf ( "\nResult after comapring buffer using
memcmp( )" ) ;
show ( result ) ;

result = memicmp ( str1, str2, strlen ( str2 ) ) ;
printf ( "\nResult after comapring buffer using
memicmp( )" ) ;
show ( result ) ;
}

show ( int r )
{
if ( r == 0 )
printf ( "\nThe buffer str1 and str2 hold
identical data" ) ;
if ( r > 0 )
printf ( "\nThe buffer str1 is bigger than buffer
str2" ) ;
if ( r < 0 )
printf ( "\nThe buffer str1 is less than buffer
str2" ) ;
}
```

16.

How do I write code to find an amount of free disk space available on current drive?

Ans: Use getdfree( ) function as shown in follow code.

```
#include <stdio.h>
#include <stdlib.h>
#include <dir.h>
#include <dos.h>

main( )
{
int dr ; struct dfree disk ;
long freesp ;
```

```
dr = getdisk( ) ;
getdfree ( dr + 1 , &disk ) ;

if ( disk.df_sclus == 0xFFFF )
{
printf ( "\ngetdfree( ) function failed\n");
exit ( 1 ) ;
}

freesp = ( long ) disk.df_avail
* ( long ) disk.df_bsec
* ( long ) disk.df_sclus ;
printf ( "\nThe current drive %c: has %ld bytes
available as free space\n", 'A' + dr, freesp ) ;
}
```

17.

Use of array indices...
If we wish to store a character in a char variable ch and the character to be stored depends on the value of another variable say color (of type int), then the code would be as shown below:

```
switch ( color )
{
case 0 :
ch = 'R' ;
break ;
case 1 :
ch = 'G' ;
break ;
case 2 :
ch = 'B' ;
break ;
}
```

In place of switch-case we can make use of the value in color as an index for a character array. How to do this is shown in following code snippet.

```
char *str = "RGB' ;
char ch ;
int color ;
// code
ch = str[ color ] ;
```

18.

Function atexit( ) recevies parameter as the address of function of the type void fun (void). The function whose address is passed to atexit( ) gets called before the termination of program. If atexit( ) is called for more than one function then the functions are called in "first in last out" order. You can verify that from the output.

```
#include <stdio.h>
#include <stdlib.h>
void fun1( )
{
printf("Inside fun1\n");
}

void fun2( )
{
printf("Inside fun2\n");
}
main( )
{
atexit ( fun1 ) ;
/* some code */
atexit ( fun2 ) ;
printf ( "This is the last statement of
program?\n" );
}
```

19.

How do I write a user-defined function, which deletes each character in a string str1, which matches any character in string str2?

Ans: The function is as shown below:

```
Compress ( char str1[], char str2[] )
{
int i, j, k ;

for ( i = k = 0 ; str1[i] != '\0' ; i++ )
{
for ( j = 0 ; str2[j] != '\0' && str2[j] !=
str1[i] ; j++ )
;
if ( str2[j] == '\0' )
str1[k++] = str1[l] ;
```

```
}
str1[k] = '\0'
}
```

20.

How does free( ) know how many bytes to free?

Ans: The malloc( ) / free( ) implementation remembers the size of each block allocated and returned, so it is not necessary to remind it of the size when freeing.

21.

What is the use of randomize( ) and srand( ) function?

Ans: While generating random numbers in a program, sometimes we require to control the series of numbers that random number generator creates. The process of assigning the random number generators starting number is called seeding the generator. The randomize( ) and srand( ) functions are used to seed the random number generators. The randomize( ) function uses PC's clock to produce a random seed, whereas the srand( ) function allows us to specify the random number generator's starting value.

22.

How do I determine amount of memory currently available for allocating?

Ans: We can use function coreleft( ) to get the amount of memory available for allocation. However, this function does not give an exact amount of unused memory. If, we are using a small memory model, coreleft( ) returns the amount of unused memory between the top of the heap and stack. If we are using a larger model, this function returns the amount of memory between the highest allocated memory and the end of conventional memory. The function returns amount of memory in terms of bytes.

23.

How does a C program come to know about command line arguments?

Ans: When we execute our C program, operating system loads the program into memory. In case of DOS, it first loads 256 bytes into memory, called program segment prefix. This contains file table, environment segment, and command line information. When we compile the C program the compiler inserts additional code that parses the command, assigning it to the argv array, making the arguments easily accessible within our C program.

24.

When we open a file, how does functions like fread( )/fwrite( ), etc. get to know from where to read or to write the data?

Ans: When we open a file for read/write operation using function like fopen( ), it returns a pointer to the structure of type FILE. This structure stores the file pointer called position pointer, which keeps track of current location within the file. On opening file for read/write operation, the file pointer is set to the start of the file. Each time we read/write a character, the position pointer advances one character. If we read one line of text at a step from the file, then file pointer advances to the start of the next line. If the file is opened in append mode, the file pointer is placed at the very end of the file. Using fseek( )function we can set the file pointer to some other place within the file.

25.

The sizeof( ) function doesn't return the size of the block of memory pointed to by a pointer. Why?

Ans: The sizeof( ) operator does not know that malloc( ) has been used to allocate a pointer. sizeof( ) gives us the size of pointer itself. There is no handy way to find out the size of a block allocated by malloc( ).

26.

FP_SEG And FP_OFF…

Sometimes while working with far pointers we need to break a far address into its segment and offset. In such situations we can use FP_SEG and FP_OFF macros. Following program illustrates the use of these two macros.

```
#include <dos.h>

main( )
{
unsigned s, o ;
char far *ptr = "Hello!" ;

s = FP_SEG ( ptr ) ;
o = FP_OFF ( ptr ) ;
printf ( "\n%u %u", s, o ) ;
}
```

27.

How do I write a program to convert a string containing number in a hexadecimal form to its equivalent decimal?

Ans: The following program demonstrates this:

```
main( )
{
char str[] = "0AB" ;
int h, hex, i, n ;
n = 0 ; h = 1 ;
for ( i = 0 ; h == 1 ; i++ )
{
if ( str[i] >= '0' && str[i] <= '9' )
hex = str[i] - '0' ;
else
{
if ( str[i] >= 'a' && str[i] <= 'f' )
hex = str[i] - 'a' + 10 ;
else
if ( str[i] >= 'A' && str[i] <= 'F' )
hex = str[i] - 'A' + 10 ;
else
h = 0 ;
}
if ( h == 1 )
n = 16 * n + hex ;
}
printf ( "\nThe decimal equivalent of %s is %d",
str, n ) ;
}
```

The output of this program would be the decimal equivalent of 0AB is 171.

28.

How do I write code that reads the segment register settings?

Ans: We can use segread( ) function to read segment register settings. There are four segment registers—code segment, data segment, stack segment and extra segment. Sometimes when we use DOS and BIOS services in aprogram we need to know the segment register's value. In such a situation we can use segread( ) function. The following program illustrates the use of this function.

```
#include <dos.h>
main( )
{
struct SREGS s ;
segread ( &s ) ;
printf ( "\nCS: %X DS: %X SS: %X ES: %X",s.cs,
s.ds, s.ss, s.es ) ;
```

```
}
```

29.

What is environment and how do I get environment for a specific entry?

Ans: While working in DOS, it stores information in a memory region called environment. In this region we can place configuration settings such as command path, system prompt, etc. Sometimes in aprogram we need to access the information contained in environment. The function getenv( ) can be used when we want to access environment for a specific entry. Following program demonstrates the use of this function.

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

main( )
{
char *path = NULL ;

path = getenv ( "PATH" ) ;
if ( *path != NULL )
printf ( "\nPath: %s", path ) ;
else
printf ( "\nPath is not set" ) ;
}
```

30.

How do I display current date in the format given below?

Saturday October 12, 2002
Ans: Following program illustrates how we can display date in above given format.

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

main( )
{
struct tm *curtime ;
time_t dtime ;

char str[30] ;

time ( &dtime ) ;
curtime = localtime ( &dtime ) ;
strftime ( str, 30, "%A %B %d, %Y", curtime ) ;
```

```
printf ( "\n%s", str ) ;
}
```
Here we have called time( ) function which returns current time. This time is returned in terms of seconds, elapsed since 00:00:00 GMT, January 1, 1970. To extract the week day, day of month, etc. from this value we need to break down the value to a tm structure. This is done by thefunction localtime( ). Then we have called strftime( ) function to format the time and store it in a string str.

31.

If we have declared an array as global in one file and we are using it in another file then why doesn't the sizeof operator works on an extern array?

Ans: An extern array is of incomplete type as it does not contain the size. Hence we cannot use sizeof operator, as it cannot get the size of the array declared in another file. To resolve this use any of one the following two solutions:
1. In the same file declare one more variable that holds the size of array. For example,

array.c

```
int arr[5] ;
int arrsz = sizeof ( arr ) ;
```

myprog.c

```
extern int arr[] ;
extern int arrsz ;
```
2. Define a macro which can be used in an array declaration. For example,

myheader.h

```
#define SZ 5
```

array.c

```
#include "myheader.h"
int arr[SZ] ;
```

myprog.c

```
#include "myheader.h"
extern int arr[SZ] ;
```

32.

How do I write printf( ) so that the width of a field can be specified at runtime?

Ans: This is shown in following code snippet.

```
main( )
{
int w, no ;
printf ( "Enter number and the width for the
number field:" ) ;
scanf ( "%d%d", &no, &w ) ;
printf ( "%*d", w, no ) ;
}
```

Here, an '*' in the format specifier in printf( ) indicates that an int value from the argument list should be used for the field width.

33.

How to find the row and column dimension of a given 2-D array?

Ans: Whenever we initialize a 2-D array at the same place where it has been declared, it is not necessary to mention the row dimension of an array. The row and column dimensions of such an array can be determined programmatically as shown in followingprogram.

```
void main( )
{
int a[][3] = { 0, 1, 2,
9,-6, 8,
7, 5, 44,
23, 11,15 } ;

int c = sizeof ( a[0] ) / sizeof ( int ) ;
int r = ( sizeof ( a ) / sizeof ( int ) ) / c ;
int i, j ;

printf ( "\nRow: %d\nCol: %d\n", r, c ) ;
for ( i = 0 ; i < r ; i++ )
{
for ( j = 0 ; j < c ; j++ )
printf ( "%d ", a[i][j] ) ;
printf ( "\n" ) ;
}
}
```

34.

The access( ) function...

The access( ) function checks for the existence of a file and also determines whether it can be read, written to or executed. This function takes two arguments the filename and an integer indicating the access mode. The values 6, 4, 2, and 1 checks for read/write, read, write and execute permission of a given file, whereas value 0 checks whether the file exists or not. Followingprogram demonstrates how we can use access( ) function to check if a given file exists.

```
#include <io.h>

main( )
{
char fname[67] ;

printf ( "\nEnter name of file to open" ) ;
gets ( fname ) ;

if ( access ( fname, 0 ) != 0 )
{
printf ( "\nFile does not exist." ) ;
return ;
}
}
```

35.

How do I convert a floating-point number to a string?

Ans: Use function gcvt( ) to convert a floating-point number to a string. Following program demonstrates the use of this function.

```
#include <stdlib.h>

main( )
{
char str[25] ;
float no ;
int dg = 5 ; /* significant digits */

no = 14.3216 ;
gcvt ( no, dg, str ) ;
printf ( "String: %s\n", str ) ;
}
```

36.
What is a stack ?

Ans: The stack is a region of memory within which our programs temporarily store data as they execute. For example, when a program passes parameters to functions, C places the parameters on the stack. When the function completes, C removes the items from the stack. Similarly, when a function declares local variables, C stores the variable's values on the stack during the function's execution. Depending on the program's use of functions and parameters, the amount of stack space that a program requires will differ.

37.
Allocating memory for a 3-D array

```
#include "alloc.h"
#define MAXX 3
#define MAXY 4
#define MAXZ 5
main( )
{
int ***p, i, j, k ;
p = ( int *** ) malloc ( MAXX * sizeof ( int ** ) ) ;
for ( i = 0 ; i < MAXX ; i++ )
{
p[i] = ( int ** ) malloc ( MAXY * sizeof ( int * ) ) ;
for ( j = 0 ; j < MAXY ; j++ )
p[i][j] = ( int * ) malloc ( MAXZ * sizeof ( int ) ) ;
}
for ( k = 0 ; k < MAXZ ; k++ )
{
for ( i = 0 ; i < MAXX ; i++ )
{
for ( j = 0 ; j < MAXY ; j++ )
{
p[i][j][k] = i + j + k ;
printf ( "%d ", p[i][j][k] ) ;
}
printf ( "\n" ) ;
}
printf ( "\n\n" ) ;
}
}
```

Data Structures
How to distinguish between a binary tree and a tree?

Ans: A node in a tree can have any number of branches. While a binary tree is a tree

structure in which any node can have at most two branches. For binary trees we distinguish between the subtree on the left and subtree on the right, whereas for trees the order of the subtrees is irrelevant.
Consider the following figure...

This above figure shows two binary trees, but these binary trees are different. The first has an empty right subtree while the second has an empty left subtree. If the above are regarded as trees (not the binary trees), then they are same despite the fact that they are drawn differently. Also, an empty binary tree can exist, but there is no tree having zero nodes.

38.
How do I use the function ldexp( ) in a program?

Ans: The math function ldexp( ) is used while solving the complex mathematical equations. This function takes two arguments, a double value and an int respectively. The order in which ldexp( ) function performs calculations is (n * pow (2, exp)) where n is the double value and exp is the integer. The following program demonstrates the use of this function.

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

void main( )
{
double ans ;
double n = 4 ;

ans = ldexp ( n, 2 ) ;
printf ( "\nThe ldexp value is : %lf\n", ans ) ;
}
```

Here, ldexp( ) function would get expanded as (4 * 2 * 2), and the output would be the ldexp value is : 16.000000

39.
Can we get the mantissa and exponent form of a given number?

Ans: The function frexp( ) splits the given number into a mantissa and exponent form. The function takes two arguments, the number to be converted as a double value and an int to store the exponent form. The function returns the mantissa part as a double value. Following example demonstrates the use of this function.

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

void main( )
{
```

```
double mantissa, number ;
int exponent ;

number = 8.0 ;
mantissa = frexp ( number, &exponent ) ;

printf ( "The number %lf is ", number ) ;
printf ( "%lf times two to the ", mantissa ) ;
printf ( "power of %d\n", exponent ) ;

return 0 ;
}
```

40.

How do I write code that executes certain function only at program termination?
Ans: Use atexit( ) function as shown in following program.

```
#include <stdlib.h>
main( )
{
int ch ;
void fun ( void ) ;
atexit ( fun ) ;
// code
}
void fun( void )
{
printf ( "\nTerminate program....." ) ;
getch( ) ;
}
```

41.

What are memory models?
Ans: The compiler uses a memory model to determine how much memory is allocated to the program. The PC divides memory into blocks called segments of size 64 KB. Usually, program uses one segment for code and a second segment for data. A memory model defines the number of segments the compiler can use for each. It is important to know which memory model can be used for a program. If we use wrong memory model, the program might not have enough memory to execute. The problem can be solved using larger memory model. However, larger the memory model, slower is your program execution. So we must choose the smallest memory model that satisfies our program needs. Most of the compilers support memory models like tiny, small, medium, compact, large and huge.

42.

How does C compiler store elements in a multi-dimensional array?

Ans: The compiler maps multi-dimensional arrays in two ways—Row major order and Column order. When the compiler places elements in columns of an array first then it is called column-major order. When the compiler places elements in rows of an array first then it is called row-major order. C compilers store multidimensional arrays in row-major order. For example, if there is a multi-dimensional array a[2][3], then according row-major order, the elements would get stored in memory following order:
a[0][0], a[0][1], a[0][2], a[1][0], a[1][1], a[1][2]

43.

If the result of an _expression has to be stored to one of two variables, depending on a condition, can we use conditional operators as shown below?

((i < 10) ? j : k) = l * 2 + p ;
Ans: No! The above statement is invalid. We cannot use the conditional operators in this fashion. The conditional operators like most operators, yields a value, and we cannot assign the value of an _expression to a value. However, we can use conditional operators as shown in following code snippet.

```
main( )
{
int i, j, k, l ;
i = 5 ; j = 10 ; k = 12, l = 1 ;
* ( ( i < 10 ) ? &j : &k ) = l * 2 + 14 ;
printf ( "i = %d j = %d k = %d l = %d", i, j, k, l ) ;
}
```

The output of the above program would be as given below:
i = 5 j = 16 k = 12 l = 1

44.

How can I find the day of the week of a given date?

Ans: The following code snippet shows how to get the day of week from the given date.

```
dayofweek ( int yy, int mm, int dd )
{
/*Monday = 1 and Sunday = 0 */
/* month number >= 1 and <= 12, yy > 1752 or so */
static int arr[ ] = { 0, 3, 2, 5, 0, 3, 5, 1, 4, 6, 2, 4 } ;
yy = yy - mm < 3 ;
return ( yy + yy / 4 - yy / 100 + yy / 400 + arr[ mm - 1] + dd ) % 7 ;
}
```

```
void main( )
{
printf ( "\n\n\nDay of week : %d ", dayofweek ( 2002, 5, 18 ) ) ;
}
```

45.

What's the difference between these two declarations?

```
struct str1 { ... } ;
typedef struct { ... } str2 ;
```

Ans : The first form declares a structure tag whereas the second declares a typedef. The main difference is that the second declaration is of a slightly more abstract type -- its users don't necessarily know that it is a structure, and the keyword struct is not used when declaring instances of it.

46.

How do I print the contents of environment variables?

Ans:. The following program shows how to achieve this:

```
main( int argc, char *argv[ ], char *env[ ] )
{
int i = 0 ;
clrscr( ) ;
while ( env[ i ] )
printf ( "\n%s", env[ i++ ] ) ;
}
```

main( ) has the third command line argument env, which is an array of pointers to the strings. Each pointer points to an environment variable from the list of environment variables.

47.48.

What would the second and the third printf( ) output the following program?

```
main( )
{
char *str[ ] = {
"Good Morning"
"Good Evening"
"Good Afternoon"
```

```
} ;
printf ( "\nFirst string = %s", str[0] ) ;
printf ( "\nSecond string = %s", str[1] ) ;
printf ( "\nThird string = %s", str[2] ) ;
}
```

Ans: For the above given program, we expect the output as Good Evening and Good Afternoon, for the second and third printf( ). However, the output would be as shown below.

```
First string = Good MorningGood EveningGood Afternoon
Second string = ( null )
Third string =
```

What is missing in the above given code snippet is a comma separator which should separate the strings Good Morning, Good Evening and Good Afternoon. On adding comma, we would get the output as shown below.

```
First string = Good Morning
Second string = Good Evening
Third string = Good Afternoon
```

49.

How do I use scanf( ) to read the date in the form 'dd-mm-yy' ?

Ans: There are two ways to read the date in the form of 'dd-mm-yy' one possible way is...

```
int dd, mm, yy ;
char ch ; /* for char '-' */
printf ( "\nEnter the date in the form of dd-mm-yy : " ) ;
scanf( "%d%c%d%c%d", &dd, &ch, &mm, &ch, &yy ) ;
```

And another best way is to use suppression character * as...

```
int dd, mm, yy ;
scanf( "%d%*c%d%*c%d", &dd, &mm, &yy ) ;
```

The suppression character * suppresses the input read from the standard input buffer for the assigned control character.

50.

How do I print a floating-point number with higher precision say 23.34568734 with only precision up to two decimal places?

Ans: This can be achieved through the use of suppression char '*' in the format string of printf( ) as shown in the following program.

```
main( )
```

```
{
int i = 2 ;
float f = 23.34568734 ;
printf ( "%.*f", i, f ) ;
}
```
The output of the above program would be 23.35.

51.
Are the expressions *ptr++ and ++*ptr same?

Ans: No. *ptr++ increments the pointer and not the value pointed by it, whereas ++*ptr increments the value being pointed to by ptr.

52.
strpbrk( )

The function strpbrk( ) takes two strings as parameters. It scans the first string, to find, the first occurrence of any character appearing in the second string. The function returns a pointer to the first occurrence of the character it found in the first string. The following program demonstrates the use of string function strpbrk( ).

```
#include <string.h>
main( )
{
char *str1 = "Hello!" ;
char *str2 = "Better" ;
char *p ;
p = strpbrk ( str1, str2 ) ;

if ( p )
printf ( "The first character found in str1 is %c", *p ) ;
else
printf ( "The character not found" ) ;
}
```
The output of the above program would be the first character found in str1 is e

div( )...

The function div( ) divides two integers and returns the quotient and remainder. This function takes two integer values as arguments; divides first integer with the second one and returns the answer of division of type div_t. The data type div_t is a structure that contains two long ints, namely quot and rem, which store quotient and remainder of division respectively. The following example shows the use of div( ) function.

```
#include <stdlib.h>
void main( )
{
div_t res ;

res = div ( 32, 5 ) ;
printf ( "\nThe quotient = %d and remainder = %d ", res.quot, res.rem ) ;
```

53.

Can we convert an unsigned long integer value to a string?

Ans: The function ultoa( ) can be used to convert an unsigned long integer value to a string. This function takes three arguments, first the value that is to be converted, second the base address of the buffer in which the converted number has to be stored (with a string terminating null character '\0') and the last argument specifies the base to be used in converting the value. Following example demonstrates the use of this function.

```
#include <stdlib.h>
void main( )
{
unsigned long ul = 3234567231L ;
char str[25] ;

ultoa ( ul, str, 10 ) ;
printf ( "str = %s unsigned long = %lu\n", str, ul ) ;
}
```

54.

ceil( ) and floor( )

The math function ceil( ) takes a double value as an argument. This function finds the smallest possible integer to which the given number can be rounded up. Similarly, floor( ) being a math function, takes a double value as an argument and returns the largest possible integer to which the given double value can be rounded down. The following program demonstrates the use of both the functions.

```
#include <math.h>
void main( )
{
double no = 1437.23167 ;
double down, up ;

down = floor ( no ) ;
up = ceil ( no ) ;
```

```
printf ( "The original number %7.5lf\n", no ) ;
printf ( "The number rounded down %7.5lf\n", down ) ;
printf ( "The number rounded up %7.5lf\n", up ) ;
}
```

The output of this program would be,
The original number 1437.23167
The number rounded down 1437.00000
The number rounded up 1438.00000

55.

How do I use function ecvt( ) in a program?

Ans: The function ecvt( ) converts a floating-point value to a null terminated string. This function takes four arguments, such as, the value to be converted to string, the number of digits to be converted to string, and two integer pointers. The two-integer pointer stores the position of the decimal point (relative to the string) and the sign of the number, respectively. If the value in a variable, used to store sign is 0, then the number is positive and, if it is non-zero, then the number is negative. The function returns a pointer to the string containing digits. Following program demonstrates the use of this function.

```
#include <stdlib.h>
main( )
{
char *str ;
double val ;
int dec, sign ;
int ndig = 4 ;

val = 22 ;
str = ecvt ( val, ndig, &dec, &sign ) ;
printf ( "string = %s dec = %d sign = %d\n", str, dec, sign ) ;

val = -345.67 ;
ndig = 8 ;
str = ecvt ( val, ndig, &dec, &sign ) ;
printf ( "string = %s dec = %d sign = %d\n", str, dec, sign ) ;

// number with a scientific notation
val = 3.546712e5 ;
ndig = 5 ;
str = ecvt ( val, ndig, &dec, &sign ) ;
printf ( "string = %s dec = %d sign = %d\n", str, dec, sign ) ;
}
```

The output of this program would be

string = 2200 dec = 2 sign = 0
string = 34567000 dec = 3 sign = 1
string = 35467 dec = 6 sign = 0

56.

How to run DIR command programmatically?

Ans: We can use the system( ) function to execute the DIR command along with its options. Following program shows how this can be achieved:

```
// mydir.c

main ( int argc, char *argv[ ] )
{
char str[30] ;

if ( argc < 2 )
exit ( 0 ) ;

sprintf ( str, "dir %s %s", argv[1], argv[2] ) ;
system ( str ) ;
}
```

If we run the executable file of this program at command prompt passing the command line arguments as follows:

```
> mydir abc.c /s
```

This will search the file 'abc.c' in the current directory.

57.

Suppose I have a structure having fields name, age, salary and have passed address of age to a function fun( ). How I can access the other member of the structure using the address of age?

Ans:

```
struct emp
{
char name[20] ;
int age ;
```

```
float salary ;
} ;
main( )
{
struct emp e ;
printf ( "\nEnter name: " ) ;
scanf ( "%s", e.name ) ;
printf ( "\nEnter age: " ) ;
scanf ( "%d", &e.age ) ;
printf ( "\nEnter salary: " ) ;
scanf ( "%f", &e.salary ) ;
fun ( &e.age ) ;
}
fun ( int *p )
{
struct emp *q ;
int offset ;
offset = ( char * ) ( & ( ( struct emp * ) 0 ) -> age ) - ( char * ) ( (
struct emp* ) 0 ) ;
q = ( struct emp * ) ( ( char * ) p - offset ) ;
printf ( "\nname: %s", q -> name ) ;
printf ( "\nage: %d", q -> age ) ;
printf ( "\nsalary: %f", q -> salary ) ;
}
```

58.

How to restrict the program's output to a specific screen region?

Ans: A C function window( ) can be used to restrict the screen output to a specific region. The window( ) function defines a text-mode window. The parameters passed to this function defines the upper-left and lower-right corner of the region within which you want the output. In the followingprogram , the string 'Hello!' gets printed within the specified region. To print the string we must use cprintf( ) function which prints directly on the text-mode window.

```
#include <conio.h>
main( )
{
int i, j ;

window ( 20, 8, 60, 17 ) ;
for ( i = 0 ; i < 8 ; i++ )
for ( j = 0 ; j < 10 ; j++ )
cprintf ( "Hello!" ) ;
}
```

59.

Sometimes you need to prompt the user for a password. When the user types in the password, the characters the user enters should not appear on the screen. A standard library function getpass( ) can be used to perform such function. Maximum number of characters that can be entered as password is 8.

```
main( )
{
char *pwd ;

pwd = getpass ( "Enter Password" ) ;

if ( strcmp ( pwd, "orgcity" ) )
printf ( "\nPassword %s is incorrect", pwd ) ;
else
printf ( "\nCorrect Password" ) ;
}
```

60.

How to obtain the current drive through C ?

Ans: We can use the function _getdrive( ) to obtain the current drive. The _getdrive( ) function uses DOS function 0X19 to get the current drive number

```
#include <direct.h>
main( )
{
int disk ;
disk = _getdrive( ) + 'A' - 1 ;
printf ( "The current drive is: %c\n", disk ) ;
}
```

61.

How come the output for both the programs is different when the logic is same?

```
main( )
{
int i, j ;

for ( i = 1, j = 1 ; i <= 5, j <= 100 ; i++, j++ )
{
gotoxy ( 1, 1, ) ;
printf ( "%d %d", i, j ) ;
}
```

```
}

main( )
{
int i, j ;

for ( i =1, j = 1; j <= 100, i <= 5; i++, j++ )
{
gotoxy ( 1, 1 ) ;
printf ( "%d %d", i, j ) ;
}
}
```

Output -> 5 5
Even if logic of both the programs is same the output of the first program comes out to be 100, 100, but of the second program it is 5, 5. The comma operator plays a vital role inside the for loop. It always considers the value of the latest variable. So, at the time of testing the condition in for loop, the value of j will be considered in the firstprogram and value of i in the second.

62.
Can we get the x and y coordinate of the current cursor position ?

Ans : The function wherex( ) and wherey( ) returns the x-coordinate and y-coordinate of the current cursor position respectively. Both the functions return an integer value. The value returned by wherex( ) is the horizontal position of cursor and the value returned by wherey( ) is the vertical position of the cursor. Followingprogram shows how to use the wherex( ) and wherey( ) functions.

```
#include <conio.h>
main( )
{
printf ( "Just\n To\n Test\n Where\n the cursor\n goes" ) ;

printf ( "Current location is X: %d Y: %d\n", wherex( ), wherey( ) ) ;
}
```

63.
How do I programmatically delete lines in the text window?

Ans: While writing programs that perform screen-based I/O, you may want to-delete the current line's contents, moving one line up, all of the output that follows. In such cases a function called delline( ) can be used. Following code snippet illustrates the use of function delline( ).

```
#include <conio.h>
```

```
main( )
{
int i ;
clrscr( ) ;

for ( i = 0; i <= 23; i++ )
printf ( "Line %d\r\n", i ) ;

printf ( "Press a key to continue : " ) ;
getch( ) ;

gotoxy ( 2, 6 ) ;

for ( i = 6; i <= 12; i++ )
delline( ) ;

getch( ) ;
}
```

64.

How do I get the time elapsed between two function calls ?

Ans: The function difftime( ) finds the difference between two times. It calculates the elapsed time in seconds and returns the difference between two times as a double value.

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

main( )
{
int a[] = { 2, -34, 56, 78, 112, 33, -7, 11, 45, 29, 6 } ;
int s ;
time_t t1, t2 ; // time_t defines the value used for time function

s = sizeof ( a ) / 2 ;
t1 = time ( NULL ) ;
sel_sort ( a, s ) ; // sort array by selection sort
bub_sort ( a, s ) ; // sort array by bubble sort method
t2 = time ( NULL ) ;
printf ( "\nThe difference between two function calls is %f", difftime (
t2, t1 ) ) ;
}
```

In the above program we have called difftime( ) function that returns the time elapsed from t1

to t2.

65.

How do I use swab( ) in my program ?

Ans: The function swab( ) swaps the adjacent bytes of memory. It copies the bytes from source string to the target string, provided that the number of characters in the source string is even. While copying, it swaps the bytes which are then assigned to the target string.

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

main ( )
{
char *str1 = "hS eesll snsiasl not eh es as oher " ;
char *str2 ;
clrscr( ) ;
swab ( str1, str2, strlen ( str1 ) ) ;
printf ( "The target string is : %s\n", str2 ) ; // output -- She sells
snails on the sea shore
getch( ) ;
}
```

66.

Turbo C provides various command line compiler options which we can use through TCC. The compiler options include : displaying specific warning messages, generating 8087 hardware instructions, using a filename for generating assembly code, etc. Instead of compiler options being executed at command line we can use these compiler options in ourprogram . This can be achieved using #pragma options. We can use various flags with #pragma options to use the compiler options. All these flags are available in turbo C's online help.

67.

I have an array declared in file 'F1.C' as,

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

and used in the file 'F2.C' as,

```
extern int a[ ] ;
```

In the file F2.C, why sizeof doesn't work on the array a[ ]?

Ans: An extern array of unspecified size is an incomplete type. You cannot apply sizeof to it,

because sizeof operates during compile time and it is unable to learn the size of an array that is defined in another file. You have three ways to resolve this problem:
1. In file 'F1.C' define as,
int a[ ] = { 1, 2, 3, 4, 5, 6 } ;
int size_a = sizeof (a) ;
and in file F2.C declare as,
extern int a[ ] ;
extern int size_a ;

2. In file 'F1.H' define,

#define ARR_SIZ 6
In file F1.C declare as,
#include "F1.H"
int a[ARR_SIZ] ;
and in file F2.C declare as,
#include "F1.H"
extern int a[ARR_SIZ] ;
3. In file 'F1.C' define as,
int a[ ] = { 1, 2, 3, 4, 5, 6, -1 } ;

and in file 'F2.C' declare as,

extern int a[ ] ;
Here the element -1 is used as a sentinel value, so the code can understand the end without any explicit size.

68.

How to delete a line from text displayed on the screen?

Ans: Sometimes, specially when we are creating a text editor like program we may wish to allow user to delete a line. We can do so by using two functions namely clreol( ) and delline( ). The clreol( ) function deletes the line from the current cursor position to the end of line. The delline() function deletes the entire line at the current cursor position and moves up the following line. Following program shows how to use these functions.

```
#include <conio.h>

main( )
{
int i ;

for ( i = 1 ; i <= 20 ; i++ )
printf ( "This is Line %d\n", i ) ;
```

```
getch( ) ;
gotoxy ( 1, 7 ) ;
clreol( ) ;

getch( ) ;
gotoxy ( 1, 12 ) ;
delline( ) ;

getch( ) ;
}
```

69.

How do I programmatically insert lines in the text window?

Ans: We can insert a blank line in the text window using the insline( ) function. This function inserts line at current cursor position. While doing so, it shifts down the lines that are below the newly inserted line.

```
#include <conio.h>
void main( )
{
printf ( "The little snail was slowly moving up. She wanted\r\n" ) ;
printf ( "to reach the top of the tree. It was chilly\r\n" ) ;
printf ( "winter season. Most of the animals were resting in\r\n" ) ;
printf ( "their nests as there was a heavy snow fall.\r\n" ) ;
printf ( "\r\nPress any key to continue:" ) ;

gotoxy ( 10, 2 ) ;
getch( ) ;
insline( ) ;
getch( ) ;
}
```

70.

What will be the output of the following program?

```
main( )
{
unsigned int num ;
int i ;

printf ( "\nEnter any number" ) ;
scanf ( "%u", &num ) ;
```

```
for ( i = 0 ; i < 16 ; i++ )
printf ( "%d", ( num << i & 1 << 15 ) ? 1 : 0 ) ;
}
```

Ans: The output of this program is the binary equivalent of the given number. We have used bitwise operators to get the binary number.

71.
Graphics

Building Mouse Cursors...

In text mode the mouse cursor appears as a block, whereas in graphics mode it appears as an arrow. If we wish we can change the graphics cursor to any other shape the way Windows does. The mouse cursor in graphics mode occupies a 16 by 16 pixel box. By highlighting or dehighlighting some of the pixels in this box we can get the desired shape. For example, the following bit-pattern can be used to generate the cursor which looks like an hour-glass.

```
1111111111111111 0000000000000000
1000000000000001 0000000000000000
1111111111111111 0000000000000000
1000000000000001 0000000000000000
0100000000000010 1000000000000001
0010000000000100 1100000000000011
0000100000010000 1111000000001111
0000001001000000 1111110000111111
0000001001000000 1111110000111111
0000100000010000 1111000000001111
0010000000000100 1100000000000011
0100000000000010 1000000000000001
1000000000000001 0000000000000000
1111111111111111 0000000000000000
1000000000000001 0000000000000000
1111111111111111 0000000000000000
```

Mouse pointer bitmap Screen Mask the one's in the mouse pointer bitmap indicate that the pixel would be drawn whereas the zeros indicate that the pixel would stand erased. It is important to note that the mouse pointer bit pattern is 32 bytes long. However, while actually writing a program to change the pointer shape we need a 64 byte bit-map. This provision is made to ensure that when the cursor reaches a position on the screen where something is already written or drawn only that portion should get overwritten which is to be occupied by the mouse cursor. Of the 64 bytes the first 32 bytes contain a bit mask which is first ANDed with the screen image, and then the second 32 bytes bit mask is XORed with the screen image.

The following program changes the mouse cursor in graphics mode to resemble an hour glass.

```
# include "graphics.h"
# include "dos.h"

union REGS i, o ;
struct SREGS s ;

int cursor[32] =
{
/* Hour-glass screen mask */
0x0000, 0x0000, 0x0000, 0x0000,
0x8001, 0xc003, 0xf00f, 0xfc3f,
0xfc3f, 0xf00f, 0xc003, 0x8001,
0x0000, 0x0000, 0x0000, 0x0000,
/* The mouse pointer bitmap */
0xffff, 0x8001, 0xffff, 0x8001,
0x4002, 0x2004, 0x1008, 0x0240,
0x0240, 0x0810, 0x2004, 0x4002,
0x8001, 0xffff, 0x8001, 0xffff,
} ;
main( )
{
int gd = DETECT, gm ;
initgraph ( &gd, &gm, "c:\\tc\\bgi" ) ;
if ( initmouse( ) == -1 )
{
closegraph( ) ;
printf ( "\n Mouse not installed!" ) ;
exit( ) ;
}
gotoxy ( 10, 1 ) ; printf ( "Press any key to exit..." ) ;
changecursor ( cursor ) ; showmouseptr( ) ;
getch( ) ;
}

initmouse( )
{
i.x.ax = 0 ; int86 ( 0x33, &i, &o ) ;
return ( o.x.ax == 0 ? -1 : 0 ) ;
}
showmouseptr( )
{
i.x.ax = 1 ; int86 ( 0x33, &i, &o ) ;
}
changecursor ( int *shape )
{
i.x.ax = 9 ; /* service number */
```

```
i.x.bx = 0 ; /* actual cursor position from left */
i.x.cx = 0 ; /* actual cursor position from top */
i.x.dx = ( unsigned ) shape ; /* offset address of pointer image*/
segread ( &s ) ;
s.es = s.ds ; /* segment address of pointer */
int86x ( 0x33, &i, &i, &s ) ;
}
```

72.

Towers Of Hanoi

Suppose there are three pegs labeled A, B and C. Four disks are placed on peg A. The bottom-most disk is largest, and disks go on decreasing in size with the topmost disk being smallest. The objective of the game is to move the disks from peg A to peg C, using peg B as an auxiliary peg. The rules of the game are as follows:

Only one disk may be moved at a time, and it must be the top disk on one of the pegs. A larger disk should never be placed on the top of a smaller disk. Suppose we are to write a program to print out the sequence in which the disks should be moved such that all disks on peg A are finally transferred to peg C. Here it is...

```
main( )
{
int n = 4 ;
move ( n, 'A', 'B', 'C' ) ;
}

move ( n, sp, ap, ep )
int n ;
char sp, ap, ep ;
{
if ( n == 1 )
printf ( "\n Move from %c to %c ", sp, ep ) ;
else
{
move ( n - 1, sp, ep, ap ) ;
move ( 1, sp, ' ', ep ) ;
move ( n - 1, ap, sp, ep ) ;
}
}
```

And here is the output...

```
Move from A to B
Move from A to C
Move from B to C
Move from A to B
```

```
Move from C to A
Move from C to B
Move from A to B
Move from A to C
Move from B to C
Move from B to A
Move from C to A
Move from B to C
Move from A to B
Move from A to C
Move from B to C
```

This problem is the famous Towers of Hanoi problem, wherein three pegs are to be employed for transferring the disks with the given criteria. Here's how we go about it. We have three pegs: the starting peg, sp, the auxiliary peg ap, and the ending peg, ep, where the disks must finally be. First, using the ending peg as an auxiliary or supporting peg, we transfer all but the last disk to ap. Next the last disk is moved from sp to ep. Now, using sp as the supporting peg, all the disks are moved from ap to ep. 'A', B and C denote the three pegs. The recursive function move( ) is called with different combinations of these pegs as starting, auxiliary and ending pegs.

73.

What would be the output of following program?

```
struct syntax
{
int i ;
float g ;
char c ;
}
main( )
{
printf ( "I won't give you any error" ) ;
}
```

Ans: The above program would get compiled successfully and on execution it would print the message given in printf(). What strikes in the above code snippet is the structure syntax which is declared but not terminated with the statement terminator, the semicolon. The compiler would not give any error message for it, as it assumes that main( ) function have a return type of struct syntax and hence would successfully compile and execute the program.

74.

How to get the memory size ?

Ans: Consider the following program

```
#include <stdio.h>
#include <bios.h>
main( )
{
int memsize;
memsize = biosmemory( ) ;
printf ( "RAM size = %dK\n",memsize ) ;
return 0 ;
}
```

The function biosmemory uses BIOS interrupt 0x12 to return the size of memory.

75.

Float Format
How does C compiler stores float values ?

Ans: In C, the float values are stored in a mantissa and exponent form. While writing a number we specify the exponent part in the form of base 10. But, in case of C compiler, the exponent for floats is stored in the form of base 2. Obviously, because, computer stores the numbers in binary form. The C compiler follows an IEEE standard to store a float. The IEEE format expresses a floating-point number in a binary form known as `normalized' form. Normalization involves adjusting the exponent so that the "binary point" (the binary analog of the decimal point) in the mantissa always lies to the right of most significant nonzero digit. In binary representation, this means that the most significant digit of the mantissa is always a 1. This property of the normalized representation is exploited by the IEEE format when storing the mantissa. Let us consider an example of generating the normalized form of a floating point number. Suppose we want to represent the decimal number 5.375. Its binary equivalent can be obtained as shown below:

```
2 | 5
.375 x 2 = 0.750 0
|------
.750 x 2 = 1.500 1
2 | 2 1
.500 x 2 = 1.000 1
|------
2 | 1 0
|------
| 0 1
```

Writing remainders in reverse writing whole parts in the same order we get 101 order in which they are obtained we get 011 thus the binary equivalent of 5.375 would be 101.011. The normalized form of this binary number is obtained by adjusting the exponent until the decimal point is to the right of most significant 1. In this case the result is 1.01011 x 22. The IEEE

format for floating point storage uses a sign bit, a mantissa and an exponent for representing the power of 2. The sign bit denotes the sign of the number: a 0 represents a positive value and a 1 denotes a negative value. The mantissa is represented in binary. Converting the floating-point number to its normalized form results in a mantissa whose most significant digit is always 1. The IEEE format takes advantage of this by not storing this bit at all. The exponent is an integer stored in unsigned binary format after adding a positive integer bias. This ensures that the stored exponent is always positive. The value of the bias is 127 for floats and 1023 for doubles. Thus, 1.01011 x 22 is represented as shown below:

--- -------------- --
| 0 | 100 0000 1 | 010 1100 0000 0000 0000 0000 |
--- -------------- --

sign bit exponent- mantissa stored in normalized form obtained after adding a bias 127 to exponent 2

Data Structures

Which is the best sorting method?

Ans: There is no sorting method that is universally superior to all others. The programmer must carefully examine the problem and the desired results before deciding the particular sorting method. Some of the sorting methods are given below:
Bubble sort : When a file containing records is to be sorted then Bubble sort is the best sorting method when sorting by address is used.

Bsort : It can be recommended if the input to the file is known to be nearly sorted.

Meansort : It can be recommended only for input known to be very nearly sorted.

Quick Sort : In the virtual memory environment, where pages of data are constantly being swapped back and forth between external and internal storage. In practical situations, quick sort is often the fastest available because of its low overhead and its average behavior.

Heap sort : Generally used for sorting of complete binary tree. Simple insertion sort and straight selection sort : Both are more efficient than bubble sort. Selection sort is recommended for small files when records are large and for reverse situation insertion sort is recommended. The heap sort and quick sort are both more efficient than insertion or selection for large number of data.

Shell sort : It is recommended for moderately sized files of several hundred elements.

Radix sort : It is reasonably efficient if the number of digits in the keys is not too large.

76.

Calculating Wasted Bytes On Disk

When a file gets stored on the disk, at a time DOS allocates one cluster for it. A cluster is nothing but a group of sectors. However, since all file sizes cannot be expected to be a multiple of 512 bytes, when a file gets stored often part of the cluster remains unoccupied. This space goes waste unless the file size grows to occupy these wasted bytes. The following program finds out how much space is wasted for all files in all the directories of the current drive.

```
#include <dir.h>
#include <dos.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
unsigned bytes_per_cluster ;
unsigned long wasted_bytes ;
unsigned long num_files = 0 ;
main( )
{
int ptr = 0, flag = 0, first = 0 ;
struct ffblk f[50] ;
struct dfree free ;
/* get cluster information and calculate bytes per cluster */
getdfree ( 0, &free ) ;
bytes_per_cluster = free.df_bsec * free.df_sclus ;
chdir ( "\\" ) ;
/* check out files in root directory first */
cal_waste( ) ;
/* loop until all directories scanned */
while ( ptr != -1 )
{
/* should I do a findfirst or a findnext? */
if ( first == 0 )
flag = findfirst ( "*.*", &f[ptr], FA_DIREC ) ;
else
flag = findnext ( &f[ptr] ) ;
while ( flag == 0 )
{
/* make sure its a directory and skip over . & .. entries */
if ( f[ptr].ff_attrib == FA_DIREC && f[ptr].ff_name[0] != '.' )
{
flag = chdir ( f[ptr].ff_name ) ; /* try changing directories */
if ( flag == 0 ) /* did change dir work? */
{
cal_waste( ) ;
first = 0 ; /* set for findfirst on next pass */
break ;
}
```

```
}
flag = findnext ( &f[ptr] ) ; /* search for more dirs */
}
if ( flag != 0 || ptr == 49 ) /* didn't find any more dirs */
{
ptr-- ;
chdir ( ".." ) ; /* go back one level */
first = 1 ; /* set to findnext on next pass */
}
else
ptr++ ;
}
printf ( "There are %lu bytes wasted in %lu files.\n", wasted_bytes,
num_files ) ;
}
cal_waste( )
{
int flag = 0 ;
long full_cluster ;
struct ffblk ff ;
/* look for all file types */
flag = findfirst ( "*.*", &ff, FA_RDONLY | FA_HIDDEN | FA_SYSTEM | FA_ARCH
) ;
while ( flag == 0 )
{
num_files++ ;
full_cluster = ff.ff_fsize / bytes_per_cluster * bytes_per_cluster ;
wasted_bytes += bytes_per_cluster - ( ff.ff_fsize - full_cluster ) ;
flag = findnext ( &ff ) ;
}
}
```

Data Structures

Polish Notation

The method of writing all operators either before their operation, or after them, is called Polish notation, in honor of its discoverer, the Polish mathematician Jan Lukasiewicz. When the operators are written before their operands, it is called the prefix form. When the operators come after their operands. It is called the postfix form, or, sometimes reverse Polish form or suffix form. In this context, it is customary to use the coined phrase infix form to denote the usual custom of writing binary operators between their operands. For example, the expression A + B becomes +AB in prefix form and AB+ in postfix form. In the expression A + B x C, the multiplication is done first, so we convert it first, obtaining first A + (BCx) and then ABCx+ in postfix form. The prefix form of this expression is +A x BC. The prefix and postfix forms are not related by taking mirror images or other such simple transformation. Also all

parentheses have been omitted in the Polish forms.

77.

The Longjmp And Setjmp

The C programming language does not let you nest functions. You cannot write a function definition inside another function definition, as in:

```
int fun1( )
{
int fun2() /* such nesting of functions is not allowed */
{
.....
}
}
```

Because of this restriction it is not possible to hide function names inside a hierarchy. As a result all the functions that you declare within a program are visible to each other. This of course is not a major drawback since one can limit visibility by grouping functions within separate C source files that belong to different logical units of the program. C does, however, suffer in another way because of this design decision. It provides no easy way to transfer control out of a function except by returning to the expression that called the function. For the vast majority of function calls, that is a desirable limitation. You want the discipline of nested function calls and returns to help you understand flow of control through a program. Nevertheless, on some occasions that discipline is too restrictive. The program is sometimes easier to write, and to understand, if you can jump out of one or more function invocations at a single stroke. You want to bypass the normal function returns and transfer control to somewhere in an earlier function invocation.

For example, you may want to return to execute some code for error recovery no matter where an error is detected in your application. The setjmp and the longjmp functions provide the tools to accomplish this. The setjmp function saves the "state" or the "context" of the process and the longjmp uses the saved context to revert to a previous point in the program. What is the context of the process? In general, the context of a process refers to information that enables you to reconstruct exactly the way the process is at a particular point in its flow of execution. In C program the relevant information includes quantities such as values of SP, SS, FLAGS, CS, IP, BP, DI, ES, SI and DS registers.

To save this information Turbo C uses the following structure, which is defined, in the header file 'setjmp.h'.

```
typedef struct
{
unsigned j_sp ;
unsigned j_ss ;
unsigned j_flag ;
unsigned j_cs ;
```

```
unsigned j_ip ;
unsigned j_bp ;
unsigned j_di ;
unsigned j_es ;
unsigned j_si ;
unsigned j_ds ;
} jmp_buf[1] ;
```

This is a system-dependent data type because different systems might require different amounts of information to capture the context of a process. In Turbo C, jmp_buf is simply an array of ten 2-byte integers. To understand the mechanics of setjmp and longjmp, look at the following code
fragment.

```
#include "setjmp.h"
jmp_buf buf ;
main( )
{
if ( setjmp ( buf ) == 0 )
process( ) ;
else
handle_error( ) ; /* executed when longjmp is called */
}
process( )
{
int flag = 0 ;
/* some processing is done here */
/* if an error occurs during processing flag is set up */
if ( flag )
longjmp ( buf, 1 ) ;
}
```

Upon entry to setjmp the stack contains the address of the buffer buf and the address of the if statement in the main function, to which setjmp will return. The setjmp function copies this return address as well as the current values of registers, SP, SS, FLAGS, BP, DI, ES, SI and DS, into the buffer buf. Then setjmp returns with a zero. In this case, the if statement is satisfied and the process( ) function is called. If something goes wrong in process( ) (indicated by the flag variable), we call longjmp with two arguments: the first is the buffer that contains the context to which we will return. When the stack reverts back to this saved state, and the return statement in longjmp is executed, it will be as if we were returning from the call to setjmp, which originally saved the buffer buf. The second argument to longjmp specifies the return value to be used during this return. It should be other than zero so that in the if statement we can tell whether the return is induced by a longjmp.

The setjmp/longjmp combination enables you to jump unconditionally from one C function to another without using the conventional return statements. Essentially, setjmp marks the destination of the jump and longjmp is a non-local goto that executes the jump.
Data Structures

Comparison Trees...

The comparison trees also called decision tree or search tree of an algorithm, is obtained by tracing through the actions of the algorithm, representing each comparison of keys by a vertex of the tree (which we draw as a circle). Inside the circle we put the index of the key against which we are comparing the target key. Branches (lines) drawn down from the circle represent the possible outcomes of the comparison and are labeled accordingly. When the algorithm terminates, we put either F (for failure) or the location where the target is found at the end of the appropriate branch, which we call a leaf, and draw as a square. Leaves are also sometimes called end vertices or external vertices of the tree. The remaining vertices are called the internal vertices of the tree. The comparison tree for sequential search is especially simple.

78.

Suppose we have a floating-point number with higher precision say 12.126487687 and we wish it to be printed with only precision up to two decimal places. How can I do this?

Ans. This can achieved through the use of suppression char '*' in the format string of printf( ) which is shown in the following program.

```
main( )
{
int p = 2 ;
float n = 12.126487687 ;
printf ( "%.*f",p, n ) ;
}
```

79.

Spawning

All programs that we execute from DOS prompt can be thought of as children of COMMAND.COM. Thus, the program that we execute is a child process, whereas COMMAND.COM running in memory is its parent. The process of a parent process giving birth to a child process is known as 'spawning'. If the spawned program so desires, it may in turn spawn children of its own, which then execute and return control to their parent. Who is the parent of COMMAND.COM? COMMAND.COM itself. We can trace the ancestors of our program using the field Parent Process ID (PID) present at offset 0x16 in the Program Segment Prefix (PSP). To trace this ancestry our program should first locate its PSP, extract the parent process ID from it and then use this to find PSP of the parent. This process can be repeated till we reach COMMAND.COM (process ID of COMMAND.COM is its own PSP), the father of all processes. Here is a program which achieves this...

```
/* SPAWN.C */
#include "dos.h"
```

```
unsigned oldpsp, newpsp, far *eb_seg, i ;
char far *eb_ptr ;

main( )
{
oldpsp = _psp ;

while ( 1 )
{
printf ( "\n" ) ;
printname ( oldpsp ) ;
printf ( " spawned by " ) ;

newpsp = * ( ( unsigned far * ) MK_FP ( oldpsp, 0x16 ) ) ;

if ( * ( ( unsigned * ) MK_FP ( newpsp, 0x16 ) ) == newpsp )
break ;
else
oldpsp = newpsp ;

printname ( newpsp ) ;
}

printf ( "%-20s (%04X)", "COMMAND.COM", newpsp ) ;
}

printname ( unsigned lpsp )
{
char drive[5], dir[68], name[13], ext[5] ;

eb_seg = ( unsigned far * ) MK_FP ( lpsp, 0x2C ) ;
eb_ptr = MK_FP ( *eb_seg, 0 ) ;

i = 0 ;
while ( 1 )
{
if ( eb_ptr[i] == 0 )
{
if ( eb_ptr[i + 1] == 0 && eb_ptr[i + 2] == 1 )
{
i += 4 ;
break ;
}
}
i++ ;
```

```
}

fnsplit ( eb_ptr + i, drive, dir, name, ext ) ;
strcat ( name, ext ) ;
printf ( "%-20s (%04X)", name, oldpsp ) ;
}
```

On running the program from within TC the output obtained is shown below. SPWAN.EXE (58A9) spawned by TC.EXE (0672) TC.EXE (0672) spawned by COMMAND.COM (05B8). The program simply copies its own process ID in the variable oldpsp and then uses it to extract its own filename from its environment block. This is done by the function printname( ). The value in oldpsp is then used to retrieve the parent's PID in newpsp. From there the program loops reporting the values of oldpsp, newpsp and the corresponding file names until the program reaches COMMAND.COM.

The printname( ) function first locates the environment block of the program and then extracts the file name from the environment block. The fnsplit( ) function has been used to eliminate the path present prior to the file name. Do not run the program from command line since it would give you only one level of ancestry.

Data Structures

Choosing the data structures to be used for information retrieval. For problems of information retrieval, consider the size, number, and location of the records along with the type and structure of the keys while choosing the data structures to be used. For small records, high-speed internal memory will be used, and binary search trees will likely prove adequate. For information retrieval from disk files, methods employing multiway branching, such as trees, B-trees , and hash tables, will usually be superior. Tries are particularly suited to applications where the keys are structured as a sequence of symbols and where the set of keys is relatively dense in the set of all possible keys. For other applications, methods that treat the key as a single unit will often prove superior. B-trees, together with various generalization and extensions, can be usefully applied to many problems concerned with external information retrieval.

80.

Variably Dimensioned Arrays

While dealing with Scientific or Engineering problems one is often required to make use of multi-dimensioned array. However, when it comes to passing multidimensional arrays to a function C is found wanting. This is because the C compiler wants to know the size of all but the first dimension of any array passed to a function. For instance, we can define a function compute (int n, float x[]), but not compute (int n, x[][]).

Thus, C can deal with variably dimensioned 1-D arrays, but when an array has more than one dimension, the C compiler has to know the size of the last dimensions expressed as a constant. This problem has long been recognized, and some of the solutions that are often used are:

Declare the arrays in the functions to be big enough to tackle all possible situations. This can lead to a wastage of lot of precious memory in most cases. Another solution is to construct multiple-dimension array as an array of pointers. For example, a matrix (2-D array) of floats can be declared as a 1-D array of float pointers, with each element pointing to an array of floats. The problem with this method is that the calling function has to define all arrays in this fashion. This means that any other computations done on the arrays must take this special structure into account.

Another easy solution, though seldom used, exists. This is based on the following method:

Pass the array to the function as though it is a pointer to an array of floats (or the appropriate data type), no matter how many dimensions the array actually has, along with the dimensions of the array. Reference individual array elements as offsets from this pointer.
Write your algorithm so that array elements are accessed in storage order. The following program for multiplying two matrices illustrates this
procedure.

```
# define M 3
# define N 2
# define P 4

float a[M][N], b[N][P], c[M][P] ;
void mulmat ( int, int, int, float*, float*, float* ) ;

main( )
{
int i, j ;
for ( i = 0 ; i < M ; i++ )
for ( j = 0 ; j < N ; j++ )
a[i][j] = i + j ;

for ( i = 0 ; i < N ; i++ )
for ( j = 0 ; j < P ; j++ )
b[i][j] = i + j ;

mulmat ( M, N, P, a, b, c ) ;
for ( i = 0 ; i < M ; i++ )
{
printf ( "\n" ) ;
for ( j = 0 ; j < P ; j++ )
printf ( "%f\t", c[i][j] ) ;
}
}
```

```
void mulmat ( int m, int n, int p, float *a, float *b, float *c )
{
float *ptrtob, *ptrtoc ;
int i, j, k, nc ;

/* set all elements of matrix c to 0 */
for ( i = 0 ; i < m * p ; i++ )
*( c + i ) = 0 ;

for ( i = 0 ; i < m ; i++ )
{
ptrtob = b ;
for ( k = 0 ; k < n ; k++ )
{
ptrtoc = c ;

for ( j = 0 ; j < p ; j++ )
*ptrtoc++ += *a * *ptrtob++ ;
a++ ;
}
c += p ;
}
}
```

We know that C stores array elements in a row-major order. Hence to ensure that the elements are accessed in the storage order the above program uses a variation of the normal matrix-multiplication procedure. The pseudo code for this is given below:

```
for i = 1 to m
for j = 1 to p
c[i][j] = 0
end
for k = 1 to n
for j = 1 to p
c[i][j] = c[i][j] + a[i][k] * b[k][j]
end
end
end
```

81.

Why is it not possible to scan strings from keyboard in case of array of pointers to string?

Ans: When an array is declared, dimension of array should be specified so that compiler can allocate memory for the array. When array of pointers to strings is declared its elements would contain garbage addresses. These addresses would be passed to scanf( ). So strings can be received but they would get stored at unkown locations. This is unsafe.

82.
Bit Arrays

If in a program a variable is to take only two values 1 and 0, we really need only a single bit to store it. Similarly, if a variable is to take values from 0 to 3, then two bits are sufficient to store these values. And if a variable is to take values from 0 through 7, then three bits will be enough, and so on. Why waste an entire integer when one or two or three bits will do? Because there aren't any one bit or two bit or three bit data types available in C. However, when there are several variables whose maximum values are small enough to pack into a singlememory location, we can use `bit fields' to store several values in a single integer. Bit fields are discussed in most standard C texts. They are usually used when we want to store assorted information which can be accommodated in 1, 2, 3 bits etc.

For example, the following data about an employee can be easily stored using bit fields.

male or female
single, married, divorced or widowed
have one of the eight different hobbies
can choose from any of the fifteen different schemes proposed by the company to pursue his/her hobby.

This means we need one bit to store gender, two to store marital status, three for hobby, and four for scheme (with one value used for those who are not desirous of availing any of the schemes). We need ten bits altogether, which means we can pack all this information into a single integer, since an integer is 16 bits long.

At times we may need to store several True or False statuses. In such cases instead of using bit fields using an array of bits would be more sensible. On this array we may be required to perform the following operations:

Set a bit (make it 1).
Clear a bit (make it 0).
Test the status of a bit in the array.
Reach the appropriate bit slot in the array.
Generate a bit mask for setting and clearing a bit.
We can implement these operations using macros given below:

```
#define CHARSIZE 8
#define MASK ( y ) ( 1 << y % CHARSIZE )
#define BITSLOT ( y ) ( y / CHARSIZE )
#define SET ( x, y ) ( x[BITSLOT( y )] |= MASK( y ) )
#define CLEAR ( x, y ) ( x[BITSLOT( y )] &= ~MASK( y ) )
#define TEST ( x, y ) ( x[BITSLOT( y )] & MASK( y ) )
#define NUMSLOTS ( n ) ( ( n + CHARSIZE - 1) / CHARSIZE )
```

Using these macros we can declare an array of 50 bits be saying,

```
char arr[NUMSLOTS(50)] ;
```
To set the 20th bit we can say,

```
SET(arr, 20 ) ;
```
And if we are to test the status of 40th bit we may say,
```
if ( TEST ( arr, 40 ) )
```
Using bit arrays often results into saving a lot of precious memory. For example, the following program which implements the Sieve of Eratosthenes for generating prime numbers smaller than 100 requires only 13 bytes. Had we implemented the same logic using anarray of integers we would have required an array of 100 integers, that is 200 bytes.

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

#define MAX 100

main( )
{
char arr[NUMSLOTS( MAX )] ;
int i, j ;

memset ( arr, 0, NUMSLOTS( MAX ) ) ;
for ( i = 2 ; i < MAX ; i++ )
{
if ( !TEST ( arr, i ) )
{
printf ( "\n%d", i ) ;
for ( j = i + i ; j < MAX ; j += i )
SET ( arr, j ) ;
}
}
}
```

83.

Information Hiding in C

Though C language doesn't fully support encapsulation as C++ does, there is a simple technique through which we can implement encapsulation in C. The technique that achieves this is modular programming in C. Modular programming requires a little extra work from the programmer, but pays for itself during maintenance. To understand this technique let us take the example of the popular stack data structure. There are many methods of implementing a stack (array, linked list, etc.). Information hiding teaches that users should be able to push and pop the stack's elements without knowing about the stack's implementation. A benefit of this sort of information hiding is that users don't have to change their code even if the implementation details change.

Consider the following scenario:

To be able to appreciate the benefits of modular programming and thereby information hiding, would first show a traditional implementation of the stack data structure using pointers and a linked list of structures. The main( )function calls the push( ) and pop( ) functions.

```
#include <alloc.h>
typedef int element ;
void initialize_stack ( struct node ** ) ;
void push ( struct node **, element ) ;
element pop ( struct node * ) ;
int isempty ( struct node * ) ;
struct node
{
element data ;
struct node *next ;
} ;
void main( )
{
struct node *top ;
element num ;
initialize_stack ( &top ) ;
push ( &top, 10 ) ;
push ( &top, 20 ) ;
push ( &top, 30 ) ;
if ( isempty ( top ) )
printf ( "\nStack is empty" ) ;
else
{
num = pop ( top ) ;
printf ( "\n Popped %d", num ) ;
}
}
void initialize_stack ( struct node **p )
{
*p = NULL ;
}
void push ( struct node **p, element n )
{
struct node *r ;
r = ( struct node *) malloc ( sizeof ( struct node ) ) ;
r -> data = n ;
if ( *p == NULL )
r -> next = NULL ;
else
r -> next = *p ;
*p = r ;
}
```

```
element pop ( struct node *p )
{
element n ;
struct node *r ;
n = p -> data ;
r = p ;
p = p -> next ;
free ( r ) ;
return ( n ) ;
}
int isempty ( struct node *p )
{
if ( p == NULL )
return ( -1 ) ;
else
return ( 0 ) ;
}
```

Notice how the specific implementation of the data structure is strewn throughout main( ). main( ) must see the definition of the structure node to use the push( ), pop( ), and other stack functions. Thus the implementation is not hidden, but is mixed with the abstract operations.

Data Structures

Radix Sort

This sorting technique is based on the values of the actual digits in the positional representations of the numbers being sorted. Using the decimal base, for example, where the radix is 10, the numbers can be partitioned into ten groups on the sorter. For example, to sort a collection of numbers where each number is a four-digit number, then, All the numbers are first sorted according to the the digit at unit's place.
In the second pass, the numbers are sorted according to the digit at tenth place. In the third pass, the numbers are sorted according to the digit at hundredth place. In the forth and last pass, the numbers are sorted according to the digit at thousandth place.
During each pass, each number is taken in the order in which it appears in partitions from unit's place onwards. When these actions have been performed for each digit, starting with the least significant and ending with most significant, the numbers are sorted. This sorting method is called the radix sort.

Let us take another example. Suppose we have a list of names. To sort these names using radix sort method we will have to classify them into 26 groups The list is first sorted on the first letter of each name, i.e. the names are arranged in 26 classes, where the first class consists of those names that begin with alphabet 'A', the second class consists of those names that begin with alphabet 'B' and so on.
During the second pass each class is alphabetized according to the second letter of the name, and so on.

Exception Handling in C

Consider the following program:

```
#include <math.h>
void main( )
{
float i ;
i = pow ( -2, 3 ) ;
printf ( "%f", i ) ;
}

int matherr ( struct exception *a )
{
if ( a -> type == DOMAIN )
{
if ( !strcmp ( a -> name, "pow" ) )
{
a -> retval = pow ( - ( a -> arg1 ), a -> arg2 ) ;
return 1 ;
}
}
return 0 ;
}
```

If we pass a negative value in pow( ) function a run time error occurs. If we wish to get the proper output even after passing a negative value in the pow( ) function we must handle the run time error. For this, we can define a function matherr( ) which is declared in the 'math.h' file. In this function we can detect the run-time error and write our code to correct the error. The elements of the exception structure receives the function name and arguments of the function causing the exception.

Data Structures

AVL Trees

For ideal searching in a binary search tree, the heights of the left and right sub-trees of any node should be equal. But, due to random insertions and deletions performed on a binary search tree, it often turns out to be far from ideal. A close approximation to an ideal binary search tree is achievable if it can be ensured that the difference between the heights of the left and the right sub trees of any node in the tree is at most one. A binary search tree in which the difference of heights of the right and left sub-trees of any node is less than or equal to one is known as an AVL tree. AVL tree is also called as Balanced Tree. The name "AVL Tree" is derived from the names of its inventors who are Adelson-Veilskii and Landi. A node in an AVL tree have a new field to store the "balance factor" of a node which denotes the difference of height between the left and the right sub-trees of the tree rooted at that node. And it can assume one of the
three possible values {-1,0,1}.

85.

Unique combinations for a given number

How do I write a program which can generate all possible combinations of numbers from 1 to one less than the given number ?

```
main( )
{
long steps, fval, bstp, cnt1 ;
int num, unit, box[2][13], cnt2, cnt3, cnt4 ;
printf ( "Enter Number " ) ;
scanf ( "%d", &num ) ;
num = num < 1 ? 1 : num > 12 ? 12 : num ;
for ( steps = 1, cnt1 = 2 ; cnt1 <= num ; steps *= cnt1++ ) ;
for ( cnt1 = 1 ; cnt1 <= steps ; cnt1++ )
{
for ( cnt2 = 1 ; cnt2 <= num ; cnt2++ )
box[0][cnt2] = cnt2 ;
for ( fval = steps, bstp = cnt1, cnt2 = 1 ; cnt2 <= num ; cnt2++ )
{
if ( bstp == 0 )
{
cnt4=num ;
while ( box[0][cnt4] == 0 )
cnt4-- ;
}
else
{
fval /= num - cnt2 + 1 ;
unit = ( bstp + fval - 1 ) / fval ;
bstp %= fval ;
for ( cnt4 = 0, cnt3 = 1 ; cnt3 <= unit ; cnt3++ )
while ( box[0][++cnt4] == 0 ) ;
}
box[1][cnt2] = box[0][cnt4] ;
box[0][cnt4] = 0 ;
}
printf ( "\nSeq.No.%ld:", cnt1 ) ;
for ( cnt2 = 1 ; cnt2 <= num ; cnt2++ )
printf ( " %d", box[1][cnt2] ) ;
}
}
```

This program computes the total number of steps. But instead of entering into the loop of the first and last combination to be generated it uses a loop of 1 to number of combinations. For example, in case of input being 5 the number of possible combinations would be factorial 5, i.e. 120. The program suffers

from the limitation that it cannot generate combinations for input beyond 12 since a long int cannot handle the resulting combinations.

Data Structures

Hashing...

Hashing or hash addressing is a searching technique. Usually, search of an element is carried out via a sequence of comparisons. Hashing differs from this as it is independent of the number of elements n in the collection of data. Here, the address or location of an element is obtained by computing some arithmetic function. Hashing is usually used in file management. The general idea is of using the key to determine the address of a record. For this, a function fun( ) is applied to each key, called the hash function. Some of the popular hash functions are: 'Division' method, 'Midsquare' method, and 'Folding' method. Two records cannot occupy the same position. Such a situation is called a hash collision or a hash clash. There are two basic methods of dealing with a hash clash. The first technique, called rehashing, involves using secondary hash function on the hash key of the item. The rehash function is applied successively until an empty position is found where the item can be inserted. If the hash position of the item is found to be occupied during a search, the rehash function is again used to locate the item. The second technique, called chaining, builds a linked list of all items whose keys hash to the same values. During search, this short linked list is traversed sequentially for the desired key. This technique involves adding an extra link field to each table position.

86.

The following program demonstrates how to get input from the user in graphics mode, echoed in the current colors and font size and font style.

```
#define ON 1
#define OFF 0
#include <graphics.h>
main( )
{
char nameString[80], ageString[80] ;
int age, gd = DETECT, gm ;
initgraph ( &gd, &gm, "c:\\tc\\bgi" ) ;
setbkcolor ( BLUE ) ;
setcolor ( YELLOW ) ;
settextstyle ( GOTHIC_FONT, HORIZ_DIR, 0 ) ;
moveto ( 0, 0 ) ;
outtext ( "Enter your name: " ) ;
getGrString ( nameString ) ;
moveto ( 0, gety( ) + textheight ( "A" ) ) ;
outtext ( "Name: " ) ;
outtext ( nameString ) ;
moveto ( 0, gety( ) + textheight ( "A" ) ) ;
```

```
outtext ( "Press key to exit! " ) ;
getch( ) ;
closegraph( ) ;
restorecrtmode( ) ;
}
getGrString ( char *inputString )
{
int stringIndex = 0, oldColor ;
char ch, outString[2] ;
/* xVal will store the screen position for each char */
int xVal[255] ;
outString[1] = 0 ;
xVal[0] = getx( ) ;
do
{
cursor ( ON ) ;
ch = getch( ) ;
cursor ( OFF ) ;
if ( ch == 0 ) /* avoid dealing with all special keys */
getch( ) ;
else
{
if ( ch == 8 ) /* backspace */
{
oldColor = getcolor( ) ;
--stringIndex ;
if ( stringIndex < 0 )
stringIndex = 0 ;
/* move to ( old horz position, current vert position ) */
moveto ( xVal[stringIndex], gety( ) ) ;
setcolor ( getbkcolor( ) ) ;
outString[0] = inputString[stringIndex] ;
outtext ( outString ) ;
moveto ( xVal [stringIndex], gety( ) ) ;
setcolor ( oldColor ) ;
}
else
{
inputString[stringIndex] = ch ;
outString[0] = ch ;
outtext ( outString ) ;
++stringIndex ;
xVal[stringIndex] = getx( ) ;
}
}
} while ( ch != 13 && ch != 10 ) ;
```

```
inputString[stringIndex] = 0 ;
}
cursor ( int on )
{
int curX, oldColor ;
/* we'll use an underscore as a cursor */
char uBarStr[2] = { '_', 0 } ;
if ( !on )
{
oldColor = getcolor( ) ;
setcolor ( getbkcolor( ) ) ;
}
/* save horizontal position before drawing cursor */
curX = getx( ) ;
outtext ( uBarStr ) ;
moveto ( curX, gety( ) ) ;
/* if we changed the color to erase cursor, change it back */
if ( !on )
setcolor ( oldColor ) ;
}
```

The function getGrString( ) echoes graphically the user input and stores it in a buffer, and the function cursor( ) handles the cursor position.

System Utility

What is garbage collection?

Ans: Suppose some memory space becomes reusable because a node is released from a linked list. Hence, we want the space to be available for future use. One way to bring this about is to immediately reinsert the space into the free-storage list. However, this method may be too time-consuming for the operating system. The operating system may periodically collect all the deleted space onto the free-storage list. The technique that does this collection is called Garbage Collection. Garbage Collection usually takes place in two steps: First the Garbage Collector runs through all lists, tagging whose cells are currently in use, and then it runs through the memory, collecting all untagged space onto the free-storage list. The Garbage Collection may take place when there is only some minimum amount of space or no space at all left in the free-storage list, or when the CPU is idle and has time to do the collection. Generally speaking, the Garbage Collection is invisible to the programmer.

87.

How do I get the time elapsed between two function calls ?

Ans: The function difftime( ) finds the difference between two times. It calculates the elapsed time in seconds and returns the difference between two times as a double value.

```
#include <time.h>
```

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

main( )
{
int a[] = { 2, -34, 56, 78, 112, 33, -7, 11, 45, 29, 6 } ;
int s ;
time_t t1, t2 ; // time_t defines the value used for time function

s = sizeof ( a ) / 2 ;
t1 = time ( NULL ) ;
sel_sort ( a, s ) ; // sort array by selection sort
bub_sort ( a, s ) ; // sort array by bubble sort method
t2 = time ( NULL ) ;
printf ( "\nThe difference between two function calls is %f", difftime (
t2, t1 ) ) ;
}
```

In the above program we have called difftime( ) function that returns the time elapsed from t1 to t2.

88.

General

```
main( )
{
char *s ;
s = fun ( 128, 2 ) ;
printf ( "\n%s", s ) ;
}
fun ( unsigned int num, int base )
{
static char buff[33] ;
char *ptr ;
ptr = &buff [ sizeof ( buff ) - 1 ] ;
*ptr = '\0' ;
do
{
*--ptr = "0123456789abcdef"[ num % base ] ;
num /= base ;
} while ( num != 0 ) ;
return ptr ;
}
```

The above program would convert the number 128 to the base 2. You can convert a number to a hexadecimal or octal form by passing the number and the base, to the function fun( ).

Data Structures

What is a priority queue?

Ans: As we know in a stack, the latest element is deleted and in a queue the oldest element is deleted. It may be required to delete an element with the highest priority in the given set of values and not only the oldest or the newest one. A data structure that supports efficient insertions of a new element and deletions of elements with the highest priority is known as priority queue. There are two types of priority queues: an ascending priority queue is a collection of items into which items can be inserted arbitrarily and from which only the smallest item can be removed. A descending order priority queue is similar but allows only the largest item to be deleted.

89.

What is the difference between const char *p, char const *p, and char* const p ?

'const char *p' and 'char const *p' are the same, i.e. p points to a constant character. On the other hand, 'char* const p' means p is a constant pointer pointing to a character which means we cannot change the pointer p but we can change the character which p is pointing to.

90.

What's the difference between a null pointer, a NULL macro, the ASCII NUL character and a null string?

Ans: A null pointer is a pointer which doesn't point anywhere. A NULL macro is used to represent the null pointer in source code. It has a value 0 associated with it. The ASCII NUL character has all its bits as 0 but doesn't have any relationship with the null pointer. The null string is just another name for an empty string "".

System Utility

Sparse Matrix...

A sparse matrix is one where most of its elements are zero. There is no precise definition as to know whether a matrix is sparsed or not, but it is a concept which we all can recognize intuitively. The natural method of representing matrices in memory as two-dimensional arrays may not be suitable for sparse matrices. That is one may save space by storing only those entries which may be nonzero. If this is done, then the matrix may be thought of as an ordered list of non-zero elements only. Information about a non-zero element has three parts:
an integer representing its row,
an integer representing its column and
the data associated with this element.

That is, each element of a matrix is uniquely characterized by its row and column position, say i, j. We might store that matrix as a list of 3-tuples of the form (i, j, data), as shown below,

Although the non-zero elements may be stored in the array in any order, keeping them ordered in some fashion may be advantageous for further processing. Note that above array is arranged in increasing order of the row number of non-zero elements. Moreover, for elements in the same row number, the array is arranged in order of increasing column number.

91.

Pointers

What does the error "Null Pointer Assignment" mean and what causes this error?

Ans: The Null Pointer Assignment error is generated only in small and medium memory models. This error occurs in programs which attempt to change the bottom of the data segment. In Borland's C or C++ compilers, Borland places four zero bytes at the bottom of the data segment, followed by the Borland copyright notice "Borland C++ - Copyright 1991 Borland Intl.". In the small and medium memory models, a null pointer points to DS:0000. Thus assigning a value to the memory referenced by this pointer will overwrite the first zero byte in the data segment. At program termination, the four zeros and the copyright banner are checked. If either has been modified, then the Null Pointer Assignment error is generated. Note that the pointer may not truly be null, but may be a wild pointer that references these key areas in the data segment.

Data Structures

How to build an expression trees ?

Ans: An expression tree is a binary tree which is built from simple operands and operators of an (arithmetic or logical) expression by placing simple operands as the leaves of a binary tree and the operators as the interior nodes. If an operator is binary , then it has two nonempty subtrees, that are its left and right operands (either simple operands or sub expressions). If an operator is unary, then only one of its subtrees is nonempty, the one on the left or right according as the operator is written on the right or left of its operand. We traditionally write some unary operators to the left of their operands, such as "-" (unary negation) or the standard functions like log( ), sin( ) etc. Others are written on the right, such as the factorial function ()!. If the operator is written on the left, then in the expression tree we take its left subtree as empty. If it appears on the right, then its right subtree will be empty. An example of an expression tree is shown below for the expression (-a < b) or (c + d) .

92.

Can we get the remainder of a floating point division ?

Ans : Yes. Although the % operator fails to work on float numbers we can still get the remainder of floating point division by using a function fmod( ). The fmod( ) function divides the two float numbers passed to it as parameters and returns the remainder as a floating-point value. Following program shows fmod( ) function at work.

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

```
main( )
{
printf ( "%f", fmod ( 5.15, 3.0 ) ) ;
}
```

The above code snippet would give the output as 2.150000.

93.

How to extract the integer part and a fractional part of a floating point number?

Ans: C function modf( ) can be used to get the integer and fractional part of a floating point.

```
#include "math.h"

main( )
{
double val, i, f ;
val = 5.15 ;
f = modf ( val, &i ) ;
printf ( "\nFor the value %f integer part = %f and fractional part = %f",
val, i, f ) ;
}
```

The output of the above program will be:

For the value 5.150000 integer part = 5.000000 and fractional part =
0.150000

94.

How do I define a pointer to a function which returns a char pointer?

Ans:

```
char * ( *p )( ) ;
```

or

```
typedef char * ( * ptrtofun )( ) ;
ptrtofun p ;
```

Here is a sample program which uses this definition.

```
main( )
{
typedef char * ( * ptrtofun ) ( ) ;
char * fun( ) ;
ptrtofun fptr ;
char *cptr ;
fptr = fun ;
```

```
cptr = (*fptr) ( ) ;
printf ( "\nReturned string is \"%s\"", cptr ) ;
}
char * fun( )
{
static char s[ ] = "Hello!" ;
printf ( "\n%s", s ) ;
return s ;
}
```

95.

What's wrong with the following declaration: char* ptr1, ptr2 ; get errors when I try to use ptr2 as a pointer.

Ans: char * applies only to ptr1 and not to ptr2. Hence ptr1 is getting declared as a char pointer, whereas, ptr2 is being declared merely as a char. This can be rectified in two ways :

```
char *ptr1, *ptr2 ;
typedef char* CHARPTR ; CHARPTR ptr1, ptr2 ;
```

96.

How to use scanf( ) to read the date in the form of dd-mm-yy?

Ans: To read the date in the form of dd-mm-yy one possible way is,

```
int dd, mm, yy ;
char ch ; /* for char '-' */
printf ( "\nEnter the date in the form of dd-mm-yy : " ) ;
scanf( "%d%c%d%c%d", &dd, &ch, &mm, &ch, &yy ) ;
```

Another way is to use suppression character * is as follows:

```
int dd, mm, yy ;
scanf( "%d%*c%d%*c%d", &dd, &mm, &yy ) ;
```

The suppression character '*' suppresses the input read from the standard input buffer for the assigned control character.

97.

Why the output of sizeof ('a') is 2 and not 1 ?

Ans: Character constants in C are of type int, hence sizeof ('a') is equivalent to sizeof (int), i.e. 2. Hence the output comes out to be 2 bytes.

98.

Can we use scanf( ) function to scan a multiple words string through keyboard?

Ans: Yes. Although we usually use scanf( ) function to receive a single word string and gets( ) to

receive a multi-word string from keyboard we can also use scanf( ) function for scanning a multi-word string from keyboard. Following program shows how to achieve this.

```
main( )
{
char buff[15] ;
scanf ( "%[^\n]s", buff ) ;
puts ( buff ) ;
}
```

In the scanf( ) function we can specify the delimiter in brackets after the ^ character. We have specified '\n' as the delimiter. Hence scanf( ) terminates only when the user hits Enter key.

99.

How to set the system date through a C program ?

Ans: We can set the system date using the setdate( ) function as shown in the following program. The function assigns the current time to a
structure date.

```
#include "stdio.h"
#include "dos.h"

main( )
{
struct date new_date ;

new_date.da_mon = 10 ;
new_date.da_day = 14 ;
new_date.da_year = 1993 ;

setdate ( &new_date ) ;
}
```

100.

How can I write a general-purpose swap without using templates?

Ans: Given below is the program which uses the stringizing preprocessor directive ## for building a general purpose swap macro which can swap two integers, two floats, two chars, etc.

```
#define swap( a, b, t ) ( g ## t = ( a ), ( a ) = ( b ), ( b ) = g ## t )
int gint;
char gchar;
float gfloat ;
main( )
```

```
{
int a = 10, b = 20 ;
char ch1 = 'a' , ch2 = 'b' ;
float f1 = 1.12, f2 = 3.14 ;
swap ( a, b, int ) ;
printf ( "\na = %d b = %d", a, b ) ;
swap ( ch1, ch2, char ) ;
printf ( "\nch1 = %c ch2 = %c", ch1, ch2 ) ;
swap ( f1, f2, float ) ;
printf ( "\nf1 = %4.2f f2 = %4.2f", f1, f2 ) ;
}
swap ( a, b, int ) would expand to,
( gint = ( a ), ( a ) = ( b ), ( b ) = gint )
```

101.

What is a heap ?

Ans : Heap is a chunk of memory. When in a program memory is allocated dynamically, the C run-time library gets the memory from a collection of unused memory called the heap. The heap resides in a program's data segment. Therefore, the amount of heap space available to the program is fixed, and can vary from one program to another.

102.

How to obtain a path of the given file?

Ans: The function searchpath( ) searches for the specified file in the subdirectories of the current path. Following program shows how to make use of the searchpath( ) function.

```
#include "dir.h"

void main ( int argc, char *argv[] )
{
char *path ;
if ( path = searchpath ( argv[ 1 ] ) )
printf ( "Pathname : %s\n", path ) ;
else
printf ( "File not found\n" ) ;
}
```

103.

Can we get the process identification number of the current program?

Ans: Yes! The macro getpid( ) gives us the process identification number of the program currently running. The process id. uniquely identifies a program. Under DOS, the getpid( ) returns the Program

Segment Prefix as the process id. Following program illustrates the use of this macro.

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

void main( )
{
printf ( "The process identification number of this program is %X\n",
getpid( ) ) ;
}
```

104.

How do I write a function that takes variable number of arguments?

Ans: The following program demonstrates this.

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

void main( )
{
int i = 10 ;
float f = 2.5 ;
char *str = "Hello!" ;
vfpf ( "%d %f %s\n", i, f, str ) ;
vfpf ( "%s %s", str, "Hi!" ) ;
}

void vfpf ( char *fmt, ... )
{
va_list argptr ;
va_start ( argptr, fmt ) ;
vfprintf ( stdout, fmt, argptr ) ;
va_end ( argptr ) ;
}
```

Here, the function vfpf( ) has called vfprintf( ) that take variable argument lists. va_list is an array that holds information required for the macros va_start and va_end. The macros va_start and va_end provide a portable way to access the variable argument lists. va_start would set up a pointer argptr to point to the first of the variable arguments being passed to the function. The macro va_end helps the called function to perform a normal return.

105.
Can we change the system date to some other date?

Ans: Yes, We can! The function stime( ) sets the system date to the specified date. It also sets the system time. The time and date is measured in seconds from the 00:00:00 GMT, January 1, 1970. The following program shows how to use this function.

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

void main( )
{
time_t tm ;
int d ;

tm = time ( NULL ) ;

printf ( "The System Date : %s", ctime ( &tm ) ) ;
printf ( "\nHow many days ahead you want to set the date : " ) ;
scanf ( "%d", &d ) ;

tm += ( 24L * d ) * 60L * 60L ;

stime ( &tm ) ;
printf ( "\nNow the new date is : %s", ctime ( &tm ) ) ;
}
```

In this program we have used function ctime( ) in addition to function stime( ). The ctime( ) function converts time value to a 26-character long string that contains date and time.

106.

How to use function strdup( ) in a program?

Ans : The string function strdup( ) copies the given string to a new location. The function uses malloc( ) function to allocate space required for the duplicated string. It takes one argument a pointer to the string to be duplicated. The total number of characters present in the given string plus one bytes get allocated for the new string. As this function uses malloc( ) to allocate memory, it is the programmer's responsibility to deallocate the memory using free( ).

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

void main( )
{
char *str1, *str2 = "double";

str1 = strdup ( str2 ) ;
```

```
printf ( "%s\n", str1 ) ;
free ( str1 ) ;
}
```

107.

On including a file twice I get errors reporting redefinition of function.
How can I avoid duplicate inclusion?

Ans: Redefinition errors can be avoided by using the following macro definition. Include this definition in the header file.

```
#if !defined filename_h
#define filename_h
/* function definitions */
#endif
```

Replace filename_h with the actual header file name. For example, if name of file to be included is 'goto.h' then replace filename_h with 'goto_h'.

108.
How to write a swap( ) function which swaps the values of the variables using bitwise operators.

Ans: Here is the swap( ) function.

```
swap ( int *x, int *y )
{
*x ^= *y ;
*y ^= *x ;
*x ^= *y ;
}
```

The swap( ) function uses the bitwise XOR operator and does not require any temporary variable for swapping.