Solution	Marks

Q.2 a. What is operator? Explain Conditional operator with example. **(5)**

Answer:

Ans: An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C language is rich in built-in operators and provides the following types of operators:

- **Unary Operators :** Operates only on one operand. e.g ++, ! , ~ etc
- **Binary Operator:** Operates on two operands. e.g +,-,*,/,%,<,>,>= etc
- **Tertiary or conditional operator :** operates on three operands.

2

Detailed Explanation of Conditional operator with example:

Conditional Operator **(?:)** is ternary operator (demands 3 operands), and is used in certain situations, replacing if-else condition phrases. Conditional operator's shape is:

Condition_phrase ? phrase1 : phrase2;

If *conditional_phrase* is true, *phrase1* is executed and whole phrase is assigned with value of this phrase1. If the conditional phrase is false, then*phrase2* is executed, and whole phrase is assigned with phrase2 value.

Example:

```
int a, b, c;
c = a > b ? a : b; // if a>b "execute" a, else b
```

Program:

```
#include <stdio.h>
main()
{
        int a, b, big;
        printf("Enter two numbers\n");
        scanf("%d%d",&a,&b);
        big = a > b ? a : b;            // if a>b "execute" a, else b
        printf("Biggest number is %d", big);
}
```

3

b. Explain various data types available in C with specific range. (5)

Answer:

Data types show that which type of input is to be stored into that variable.

C language supports a large number of data types:

1. **Basic type**
   a. Char
   b. int
   c. float
2. Derived data types
   a. Arrays
   b. Functions
   c. Structures
   d. Unions
   e. Pointers
3. User defined types
   a. Typedef
   b. Enumerated
4. Empty data type
   a. Void

Data Type	Bytes Used	Range of Values
char	1 byte	-128 to 127
int	2 byte	-32768 to 32767
float	4 byte	3.4 e -308 to 3.4 e +308
Unsigned char	1 byte	0 to 255
short int	1 byte	-128 to 127
Unsigned short int	1 byte	0 to 255
Unsigned int	2 bytes	0 to 65535
Long int	4 bytes	-2147483648 to 2147483647 or -2^{31} to $2^{31}-1$
Unsigned long int	4 bytes	0 to 4294967295 or 0 to $2^{32}-1$
Long double	10 bytes	3.4 e-4932 to 1.1 e+4932

c. Convert: $(5273)_8 = (?)_{10}$ **(6)**
$(4F2D)_{16} = (?)_2$

Answer:

Convert:

(I) $(5273)_8 = (?)_{10}$

Ans: $(5273)_8 = 5x8^3+2x8^2+7x8^1+3x8^0$

$= 5x512+2x64+7x8+3x1$

$= (2747)_{10}$ **3**

(II) $(4F2D)_{16} = (?)_2$

Ans: $(4F2D) =$ (0100) (1111) (0010) (1101)

4 F 2 D

$= (0100111100101101)_2$ **3**

Q.3 a. Explain syntax of printf() and scanf() function with example. (6)

Answer: **3+3**

printf() function: This is used to display the values of variables using standard output device e.g Monitor

The **syntax** of print() is:

printf("control string", v1,v2,v3.....,vn);

Where

v1,v2,v3,.....,vn are variables whose values are to be displayed on monitor.

Control string comprises of 3 things:

Characters that are to be printed on screen.
Escape sequence \n
Format specifiers (%d for char, %f for float, %c for char)

scanf() function : This is used to read the values of variable through keyboard.

The **syntax** of scanf() is:

scanf ("control string",& v1,&v2,&v3.....,&vn);

Where

v1,v2,v3,.....,vn are variables whose values are to be displayed on monitor.

Control string comprises of 3 things:

Escape sequence \n
Format specifiers (%d for char, %f for float, %c for char)

The symbol **&(ampersand)** represents the memory address where the value of the variable is to be stored.

Example:

```
#include <stdio.h>
main()
{
        int a, b,c;
        printf("Enter two numbers\n");
        scanf("%d%d",&a,&b);
        c=a+b;
        printf("Sum is %d", c);
}
```

b. Differentiate break and continue with example. (10)

Answer: **5+5**

The break statement will directly jump to the end of the current block of code and the continue statement will skip the rest of the code in the current loop block and will return to the evaluation part of the loop. Simply, continue statement is used to return to the start of a loop. The break statement is used to exit from a loop.

Break	Continue
The **break** statement in C programming language has the following two usages: 1. When the **break** statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop. 2. It can be used to terminate a case in the **switch** statement (covered in the next chapter). If you are using nested loops (i.e., one loop inside another loop), the break statement will stop the execution of the innermost loop and start executing the next line of code after the block. ***Syntax:*** The syntax for a **break** statement in C is as follows:	The **continue** statement in C programming language works somewhat like the **break** statement. Instead of forcing termination, however, continue forces the next iteration of the loop to take place, skipping any code in between. For the **for** loop, **continue** statement causes the conditional test and increment portions of the loop to execute. For the **while** and **do...while** loops, **continue** statement causes the program control passes to the conditional tests. ***Syntax:*** The syntax for a **continue** statement in C is as follows: `continue;` ***Flow Diagram:***

Example:

```
#include <stdio.h>

int main ()
{
   /* local variable definition */
   int a = 10;

   /* while loop execution */
   while( a < 20 )
   {
      printf("value of a: %d\n",
a);
      a++;
      if( a > 15)
      {
         /* terminate the loop
using break statement */
         break;
      }
   }

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
```

Example:

```
#include <stdio.h>

int main ()
{
   /* local variable definition */
   int a = 10;

   /* do loop execution */
   do
   {
      if( a == 15)
      {
         /* skip the iteration */
         a = a + 1;
         continue;
      }
      printf("value of a: %d\n",
a);
      a++;

   }while( a < 20 );

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Q.4 a. What do you mean by recursion? Explain stack overhead in Recursion with example. (9)

Answer:

A program which is called itself is called recursion.

Ans. If you analyze the address of local variables of the recursive function, you will get two important results: the depth of recursion and the stack overheads in recursion. Since local variables of the function are pushed into the stack when the function calls another function, by knowing the address of the variable in repetitive recursive call, you will determine how much information is pushed into the stack. For example, the stack could grow from top to bottom, and the local variable j gets the address 100 in the stack in the first column. Suppose stack overheads are 16 bytes; in the next call j will have the address 84, in the call after that it will get the address 16. That is a difference of 16 bytes. The following program uses the same principle: the difference of the address in consecutive calls is the stack overhead.

3

Program

```
#include <stdio.h>
int fact(int n);
long old=0;  \\E
long current=0;     \\F
main()
{
   int k = 4,i;
   long diff;
   i =fact(k);
   printf("The value of i is %d\n",i);
   diff = old-current;
   printf("stack overheads are %16lu\n",diff);
}
int fact(int n)
}
  int j;
    static int m=0;
    if(m==0) old =(long) &j; \\A
    if(m==1) current =(long) &j;   \\B
    m++;            \\C
    printf("the address of j and m is %16lu %16lu\n",&j,&m);   \\D
    if(n<=0)
      return(1);
    else
       return(n*fact(n-1));
 }
```

Explanation

1. The program calculates factorials just as the previous program.
2. The variable to be analyzed is the local variable j, which is the automatic variable. It gets its location in the stack.
3. The static variable m is used to track the number of recursive calls. Note that the static variables are stored in memory locations known as data segments, and are not stored in stack. Global variables such as old and current are also stored in data segments.
4. The program usually has a three-segment text: first, storing program instructions or program code, then the data segment for storing global and static variables, and then the stack segment for storing automatic variables.
5. During the first call, m is 0 and the value of j is assigned to the global varable old. The value of m is incremented.
6. In the next call, m is 1 and the value of j is stored in current.
7. Note that the addresses of j are stored in long variables of type castings.
8. old and current store the address of j in consecutive calls, and the difference between them gives the stack overheads.
9. You can also check the address of j and check how the allocation is done in the stack and how the stack grows.

6

b. Write a program in C to find the transpose of a matrix. (7)

Answer:

This c program prints transpose of a matrix. It is obtained by interchanging rows and columns of a matrix. For example if a matrix is

1 2
3 4
5 6

then transpose of above matrix will be

1 3 5
2 4 6

When we transpose a matrix then the order of matrix changes, but for a square matrix order remains same.

```
#include <stdio.h>

int main()
{
  int m, n, c, d, matrix[10][10], transpose[10][10];

  printf("Enter the number of rows and columns of matrix ");
  scanf("%d%d",&m,&n);
  printf("Enter the elements of matrix \n");

  for( c = 0 ; c < m ; c++ )
  {
    for( d = 0 ; d < n ; d++ )
    {
      scanf("%d",&matrix[c][d]);
    }
  }

  for( c = 0 ; c < m ; c++ )
  {
    for( d = 0 ; d < n ; d++ )
    {
      transpose[d][c] = matrix[c][d];
    }
  }

  printf("Transpose of entered matrix :-\n");

  for( c = 0 ; c < n ; c++ )
  {
    for( d = 0 ; d < m ; d++ )
    {
      printf("%d\t",transpose[c][d]);
    }
    printf("\n");
  }

  return 0;
}
```

Q.5 a. Define structure. What is the difference between structure and Union? Create a structure STUDENT to keep the record of students and members of record should be accessed through pointers. (10)

2+4+4

Answer:

Ans. ***Structures*** are used when you want to process data of multiple data types but you still want to refer to the data as a single entity. Structures are similar to records in Cobal or Pascal. For example, you might want to process information on students in the categories of name and marks (grade percentages). Here you can declare the structure 'student' with the fields 'name' and 'marks', and you can assign them appropriate data types. These fields are called members of the structure. A member of the structure is referred to in the form of structurename.membername

Union is a composite type similar to structure. Even though it has members of different data types, it can hold data of only one member at a time.

Difference between Structure and union discussed as follows:

Structure	Union
1.The keyword **struct** is used to define a structure	1. The keyword **union** is used to define a union.
2. When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is greater than or equal to the sum of sizes of its members. The smaller members may end with unused slack bytes.	2. When a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of union is equal to the size of largest member.
3. Each member within a structure is assigned unique storage area of location.	3. Memory allocated is shared by individual members of union.
4. The address of each member will be in ascending order This indicates that memory for each member will start at different offset values.	4. The address is same for all the members of a union. This indicates that every member begins at the same offset value.
5 Altering the value of a member will not affect other members of the structure.	5. Altering the value of any of the member will alter other member values.
6. Individual member can be accessed at a time	6. Only one member can be accessed at a time.
7. Several members of a structure can initialize at once.	7. Only the first member of a union can be initialized.

Program

```
struct student
{
        char name[30];
        float marks;
};
main()
```

```
{
        struct student *student1;          //E
        struct student student2;           //F
char s1[30];
float f;
student1=&student2;
scanf("%d",name);
scanf("%s",&f);
*student1.name=s1;
*student2.marks=f;
printf("student name is %s", *student1.name);
printf("\n student marks is %f", *student1.marks);
}
```

Explanation:

1. Statement E indicates that student1 is the pointer to the structure.
2. Statement F defines the structure variable student2 so that memory is allocated to the structure.
3. Statement G assigns the address of the structure student2 to the pointer variable structure student1.refer to the structure using a pointer. This is because when
4. In the absence of statement G, you cannot refer to the structure using a pointer. This is because when you define the pointer to the structure, the memory allocation is done only for pointers; the memory is not allocated for structure. That is the reason you have to declare a variable of the structure type so that memory is allocated to the structure and the address of the variable is given to the point.
5. Statement J modifies a member of the structure using the * notation. The alternative notation is
6. student1-> name = f;
7. student1-> marks = s1;

b. Discuss the main operations performed on a file with example.

Answer: **6**

A file may be a *sequential file*, a *direct-access file*, or an *indexed sequential file*. A sequential file can be thought of as a linear sequence of components of the same type with no fixed maximum bound.

The major operations on the sequential files are:

Open operation: When a file is to be used, it is first required to be opened. The open operation requires two operands: the name of the file and the access mode telling whether the file is to be opened for reading or writing. If the access mode is "read," then the file must exist. If the access mode is "write," then if the file already exists, that file is emptied and the file position pointer is set to the start of the file. If the file does not exist then the operating system is requested to create a new empty file with a given name. The open operation requests the information about the locations and other properties of the file from the operating system. The operating system allocates the storage for this information and for buffers, and sets the file-position pointer to the first component of the file. The runtime library of C provides an `fopen(name,mode)` function for it. This function returns a pointer to the internal structure called FILE (you get the definition of this structure in `stdio.h`). This pointer is called a *file descriptor*; it is used by the C program to refer to the file for reading or writing purposes.

Read operation: This operation transfers the current file component to the designated program variable. The runtime library of C provides a function `fgetc(fp)`, where `fp` is a file descriptor, for `fscanf()`. `fscanf()` is similar to `scanf()` except that one extra parameter, `fp`, is required to be passed as the first parameter. The second and third parameters are the same as the first and second parameters of `scanf()`.

Write operation: This operation transfers the contents of the designated program variable to the new component created at the current position. The runtime library of C provides a function `fputc(c,fp)`, where `fp` is a file descriptor, and `c` is a character to be written in the file `fprintf()`. `fprintf()` is similar to `printf()` except that one extra parameter, `fp`, is required to be passed as the first parameter. The second and third parameters are the same as the first and second parameters of `printf()`.

Close operation: This operation notifies the operating system that the file can be detached from the program and that it can deallocate the internal storage used for the file. The file generally gets closed implicitly when the program terminates without explicit action by the programmer. But when the access mode is required to be changed it is required to be closed explicitly and reopened in the new mode. The runtime library of C provides an `fclose(fp)` function for it.

Q.6 a. Which one is better between Binary Search and Linear Search if elements are sorted. Differentiate with suitable example. (6)

Answer:

Binary Search is better than Linear Search. This can be illustrated by taking an suitable example which is discussed as follows:

Number of comparisons	
Linear Search (n)	Binary Search($\log_2 n$)
10	3
100	6
1,000	9
10,000	13
100,000	16

So for 100,000 items, binary search saves 999,984 comparisons compared to linear search. This is an analyzing improvement.

2+4

b. Explain bubble sort with a suitable program. (10)

Answer:

Bubble sorting is a simple sorting technique in which we arrange the elements of the list by forming pairs of adjacent elements. That means we form the pair of the i^{th} and $(i+1)^{th}$ element. If the order is ascending, we interchange the elements of the pair if the first element of the pair is greater than the second element. That means for every pair (list[i],list[i+1]) for i :=1 to (n−1) if list[i] > list[i+1], we need to interchange list[i] and list[i+1]. Carrying this out once will move the element with the highest value to the last or n^{th} position. Therefore, we repeat this process the next time with the elements from the first to *(n−1)th* positions. This will bring the highest value from among the remaining *(n−1)* values to the $(n-1)^{th}$ position. We repeat the process with the remaining (*n*−2) values and so on. Finally, we arrange the elements in ascending order. This requires to perform (*n*−1) passes. In the first pass we have (*n*−1) pairs, in the second pass we have (*n*−2) pairs, and in the last (or $(n-1)^{th}$) pass, we have only one pair. Therefore the order of the algorithm is $O(n^2)$.

Program

```
#include <stdio.h>

#define MAX 10

void swap(int *x,int *y)

{

int temp;

temp = *x;

*x = *y;
```

```
 *y = temp;
 }
 void bsort(int list[], int n)
 {
 int i,j;
 for(i=0;i<(n-1);i++)
 for(j=0;j<(n-(i+1));j++)
 if(list[j] > list[j+1])
 swap(&list[j],&list[j+1]);
 }
 void readlist(int list[],int n)
 {
 int i;
 printf("Enter the elements\n");
 for(i=0;i<n;i++)
 scanf("%d",&list[i]);
 }
 void printlist(int list[],int n)
 {
 int i;
 printf("The elements of the list are: \n");
 for(i=0;i<n;i++)
 printf("%d\t",list[i]);
 }
```

```
void main()
{
int list[MAX], n;
printf("Enter the number of elements in the list max = 10\n");
scanf("%d",&n);
readlist(list,n);
printf("The list before sorting is:\n");
printlist(list,n);
bsort(list,n);
printf("The list after sorting is:\n");
printlist(list,n);
}
```

Q.7 a. What is Queue? How it is differentiated from stack? (5)

Answer: **2+3**

Queue is a list of elements with insertions permitted at one end—called the rear, and deletions permitted from the other end—called the front. This means that the removal of elements from a queue is possible in the same order in which the insertion of elements is made into the queue. Thus, a queue data structure exhibits the *FIFO* (*first in first out*) property. `insert` and `delete` are the operations that are provided for insertion of elements into the queue and the removal of elements from the queue, respectively.

QUEUE	STACK
queue is a list of elements with insertions permitted at one end—called the rear, and deletions permitted from the other end—called the front. This means that the removal of elements from a queue is possible in the same order in which the insertion of elements is made into the	**stack** is simply a list of elements with insertions and deletions permitted at one end—called the stack top. That means that it is possible to remove elements from a stack in reverse order from the insertion of elements into the stack. Thus, a stack data structure

queue. Thus, a queue data structure exhibits the *FIFO* (*first in first out*) property. `insert` and `delete` are the operations that are provided for insertion of elements into the queue and the removal of elements from the queue, respectively.	exhibits the LIFO (last in first out) property. Push and pop are the operations that are provided for insertion of an element into the stack and the removal of an element from the stack, respectively.

b. How can array be implemented using Stack? Give the suitable example. (6)

c. Write a program to reverse a linked list. (5)

Q.8 a. What do you mean by Binary Search Tree? Write a program to count the number of nodes in BST.

Answer: Refer Pages 357, 364-366 from Text Book-I

b. Discuss how Binary Trees can be traversed.

Answer: Refer Pages 354-355 from Text Book-I

Q.9 a. What is depth-first traversal and breadth-first traversal?

Answer: Refer Pages 396-399 from Text Book-I

b. What is minimum-cost spanning tree? How minimum cost can be calculated? Explain with example.

Answer: Refer Pages 627-631 from Text Book-I

TEXT BOOK

C & Data Structures, P.S. Deshpande and O.G. Kakde, Dreamtech Press, 2005.