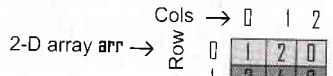
57. 120340

Explanation:

The initializers in the initialization list are bracketed to initialize individual rows. Since the number of initializers within the inner brackets is less than the row size, the last element of each row gets initialized to 0. The contents of the initialized array are as follows:



58. Compilation error (Size of type is unknown or zero)

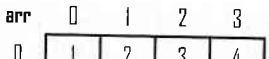
Explanation:

While declaring 2-D arrays, even if the initialization list is present, both the row size specifier and the column size specifier cannot be skipped. In the declaration statement int arr[][]={1,2,3,4}; there are four initializers, so the number of elements in the array will be at least four. There are three different ways to create an array of four elements:

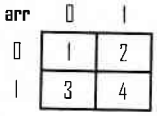
1. int arr[1][4]={1,2,3,4}, i.e. array having one row and four columns, or
2. int arr[4][1]={1,2,3,4}, i.e. array having four rows and one column, or
3. int arr[2][2]={1,2,3,4}, i.e. array having two rows and two columns

So, the compiler will not be able to determine the number of rows and columns in an array. Since arrays are stored in row major order, if the number of columns in a row of an array is specified, the compiler will be able to determine the number of rows and can create the array. Look at the following declarations and the arrays that get created:

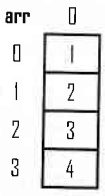
1. int arr[][4]={1,2,3,4}



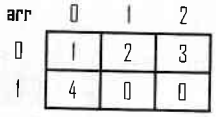
2. int arr[][2]={1,2,3,4}



3. int arr[][1]={1,2,3,4}



4. int arr[][3]={1,2,3,4} (Number of elements in the array will be greater than 4)



5. int arr[][5]={1,2,3,4} (Number of elements in the array will be greater than 4)

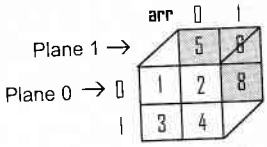
arr	0	1	2	3	4
0	1	2	3	4	0

Compilation error (Size of type is unknown or zero)

Explanation:

"While declaring n-D array, even if initialization list is present, it is mandatory to specify (n-1) fastest varying size specifiers so that compiler can uniquely determine the dimensions and create the array".

In case of 3-D arrays, even if the initialization list is present, it is mandatory to mention both the column size specifier and the row size specifier, as they vary faster as compared to plane size specifier as shown in the figure below. The plane size specifier can be skipped, if the initialization list is present, e.g. the declaration int arr[][2][2]={1,2,3,4,5,6,7,8}; is valid and the compiler will create an array that has two planes, each having two rows and two columns, as shown in the figure below:



Plane 0				Plane 1			
Row 0		Row 1		Row 0		Row 1	
Col 0	Col 1	Col 0	Col 1	Col 0	Col 1	Col 0	Col 1
a[0][0][0]	a[0][0][1]	a[0][1][0]	a[0][1][1]	a[1][0][0]	a[1][0][1]	a[1][1][0]	a[1][1][1]
1	2	3	4	5	6	7	8
21F8	21FA	21FC	21FE	2200	2202	2204	2206

In the declaration statement present in the given question, the plane size specifier is mentioned but the column size and the row size specifier are not mentioned. Hence, the compiler cannot uniquely determine the dimensions of the array. This leads to a compilation error.

1 2 3 4 0 0

Explanation:

Backward Reference: Refer to the explanations given in Answer numbers 58 and 59.

2367:21EA 2367:21EC
2367:21EE 2367:21F0

Explanation:

The printf statement prints the addresses of array elements. The printed addresses show that the elements of an array are stored in the memory using row major order of storage.

6 6 6

Explanation:

The declaration statement int arr[2][3]={1,2,3,4,5,6}; creates an array as shown in the figure below:

arr	0	1	2
0	1	2	3
1	4	5	6

The expression of form E1[E2][E3] (where one of the sub-expressions E1 or E2 is of array type or pointer type and the other sub-expressions are of integral type) gets converted to expression of form *(*(E1+E2)+E3). Hence, all the expressions arr[1][2], 1[arr][2], *(*(arr+1)+2) are equivalent and refer to the element in row 1 and column 2, i.e. 6.

63. Compilation error (Invalid indirection in function main)

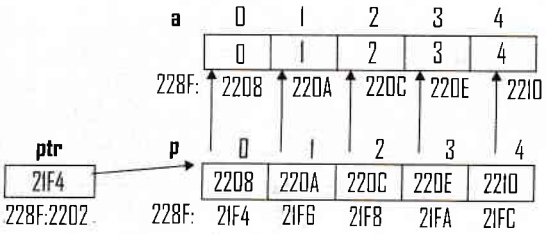
Explanation:

The expression 1[2][arr] gets converted to expression of form *(*(1+2)+arr). Application of dereference operator * on the expression of integer type, i.e. (1+2) is not valid and leads to a compilation error.

64. 0 228F:2202 228F:2208 228F:2208 228F:21F4 228F:2208

Explanation:

Suppose that the defined arrays and the pointer variable have been allocated memory as shown in the figure below. p and a being names of the arrays refer to the address of the first element of the array. Hence, the expressions p and a result in 228F:21F4 and 228F:2208, respectively. Both the expressions *p and *ptr refer to the value at memory address 228F:21F4 and result in 228F:2208. The expression &ptr refers to the address of variable ptr, i.e. 228F:2202. The expression **ptr, refers to value at memory address 228F:2208 and results in 0.



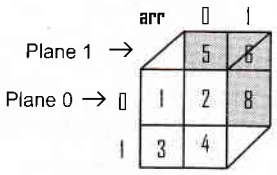
The printf statement prints the values of the evaluated expressions to produce the mentioned result.

As memory allocation is purely random the values of printed addresses may vary, if the code is executed on different machines or at different times.

65. 242F:21F8 242F:21F8 242F:21F8
242F:21F8 242F:2200 242F:2204
1 8

Explanation:

In an expression, if the number of subscripts used with an array name is less than the dimension of the array, then the expression refers to an address. Suppose that array a gets allocated at the memory location 21F8 and is stored in the memory as shown in figure below:



Plane 0				Plane 1			
Row 0		Row 1		Row 0		Row 1	
Col 0	Col 1	Col 0	Col 1	Col 0	Col 1	Col 0	Col 1
a[0][0][0]	a[0][0][1]	a[0][1][0]	a[0][1][1]	a[1][0][0]	a[1][0][1]	a[1][1][0]	a[1][1][1]
1	2	3	4	5	6	7	8
21F8	21FA	21FC	21FE	2200	2202	2204	2206

The expression:

1. a refers to the starting address of the first element of the array (plane 0), i.e. 242F:21F8.
2. a[0] refers to the starting address of plane 0, i.e. 242F:21F8.
3. a[0][0] refers to the address of plane 0 and row 0, i.e. 242F:21F8.
4. a[1] refers to the address of plane 1, i.e. 242F:2200.
5. a[1][1] refers to the address of plane 1 and row 1, i.e. 242F:2204.
6. a[0][0][0] refers to the value at plane 0, row 0 and column 0, i.e. 1.
7. a[1][1][1] refers to the value at plane 1, row 1 and column 1, i.e. 8.

66. Compilation error (Size of a and b is unknown in function main)

Explanation:

Declaring an object of type void is not allowed. Hence, the declaration statement void a,b; is erroneous.

67. Compilation error (Size of type is unknown or zero)

Explanation:

Pointer arithmetic is not allowed on void pointers. Hence, the statement *v_ptr++; is erroneous.

68. The value pointed by ptr is 2

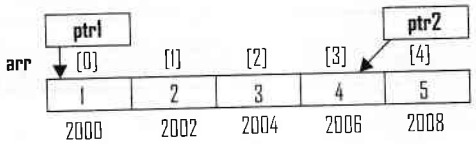
Explanation:

The pointer ptr is initialized with the address of the first element of the array arr. After incrementing it by 1, it points to the next element of the array, i.e. 2. The printf statement prints the value of the element pointed to by the pointer ptr.

69. The result of ptr2-ptr1 is 3

Explanation:

Suppose, the array arr gets allocated at the memory location 2000. ptr1 is initialized with the address of the first element of the array, i.e. 2000 and ptr2 is initialized with the address of arr[3], i.e. 2006. This is depicted in the figure below:



The expression ptr2-ptr1 will be computed as (ptr2-ptr1)/sizeof(int), i.e. (2006-2000)/2=3.

Backward Reference: Refer Section 4.4.1.4.3 for a description on pointer subtraction.

70. Compilation error (illegal use of pointers)

Explanation:

Application of multiplication operator on pointers is not allowed.

Answers to Multiple-choice Questions

71. a 72. b 73. d 74. c 75. a 76. b 77. d 78. a 79. b 80. a 81. d 82. a 83. c 84. d
85. b 86. a 87. a 88. b 89. a 90. b 91. b 92. a 93. c 94. b 95. a

Programming Exercises

Program 1 | Maximum-Minimum: Find the maximum and minimum element in a set of n elements

Algorithm:

Step 1: Start

Step 2: Assign the first array element to two different variables (i.e. max and min) that will hold the maximum and minimum value

Step 3: Loop through the remaining elements, starting from the second element. When a value larger than the present maximum value is found, it becomes the new maximum. Similarly, when a value smaller than the present minimum value is found, it becomes the new minimum

Step 4: After the termination of the loop, print the maximum and minimum values

Step 5: Stop

PE 4-1.c

```
1  //Maximum and minimum
2  #include<stdio.h>
3  main()
4  {
5  int elements[20], num, i, max, min;
6  printf("Enter the number of elements in the set (max. 20)\t");
7  scanf("%d",&num);
8  printf("Enter the elements:\n");
9  for(i=0;i<num;i++)
10     scanf("%d",&elements[i]);
11 max=min=elements[0]; //←Let max and min is the first item
12 for(i=1;i<num;i++)
13    if(elements[i]>max) //←if element[i]>max, then set max=element[i]
14       max=elements[i];
15    else if(elements[i]<min)    //←else if element[i]<min, then
16          min=elements[i];     //← set min=element[i]
17 printf("Maximum element in the set is %d\n",max);
18 printf("Minimum element in the set is %d\n",min);
19 }
```

Output window

```
Enter the number of elements in the set (max. 20)  5
Enter the elements:
12
-3
45
67
8
Maximum element in the set is 67
Minimum element in the set is -3
```

Program 2 | Find arithmetic mean, variance and standard deviation of n elements

Arithmetic mean is given as: $\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}$

Variance is given as: $\sigma_x = \frac{\sum_{i=1}^{n} (x_i - \bar{x})^2}{n}$

(Contd...)

Standard deviation is given as: $\sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n}}$

PE 4-2.c

```
//Arithmetic mean, variance and standard deviation
#include<stdio.h>
#include<math.h>
main()
{
float elements[20], sum=0.0, mean, var, sd;
int num, i;
printf("Enter the number of elements (max. 20)\t");
scanf("%d",&num);
printf("Enter the elements:\n");
for(i=0;i<num;i++)
    scanf("%f",&elements[i]);
for(i=0;i<num;i++)
    sum=sum+elements[i];
mean=sum/num;
sum=0.0;
for(i=0;i<num;i++)
    sum=sum+(elements[i]-mean)*(elements[i]-mean);
var=sum/num;
sd=sqrt(var);
printf("Arithmetic mean is %f\n",mean);
printf("Variance is %f\n",var);
printf("Standard deviation is %f\n",sd);
}
```

Output window

```
Enter the number of elements(max. 20)   6
Enter the elements:
2.1
2.9
2.3
2.4
1.8
2.5
Arithmetic mean is 2.333333
Variance is 0.115556
Standard deviation is 0.339935
```

Program 3 | Linear Search: Given a list of n elements and a key. Find whether the given key exists in the list or not. If it exists, print its position in the list

Algorithm:

Step 1: Start
Step 2: Read the elements present in the list and store them in an array
Step 3: Read the key to be searched in the list
Step 4: Loop to compare every element in the array with the key. When an equal value is found, print the location where the match has been found. If the loop finishes without finding a match, the search fails and print the message that key is not present in the list
Step 5: Stop

PE 4-3.c

```
//Linear Search
#include<stdio.h>
main()
{
int elements[20], num, i, key, found=0;
printf("Enter the number of elements (max. 20)\t");
scanf("%d",&num);
printf("Enter the elements:\n");
for(i=0;i<num;i++)
```

Output window

```
Enter the number of elements(max. 20)   6
Enter the elements:
12
10
5
-3
14
2
Enter the key that you want to search   -3
```

(Contd...)

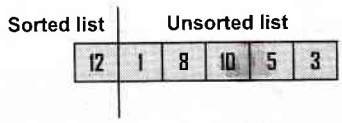
	PE 4-3.c	Output window
10	scanf("%d",&elements[i]); //←Read elements in the list	-3 exists at location no. 4
11	printf("Enter the key that you want to search\t");	
12	scanf("%d",&key); //←Read the key to be searched	
13	for(i=0;i<num;i++) //←Loop	
14	if(elements[i]==key) //←Comparison of element & key	
15	{ //←Key found	
16	printf("%d exists at location no. %d\n",key, i+1);	
17	found=1;	
18	}	
19	if(found==0) //←Key not found in the list	
20	printf("%d does not exist in the list",key);	
21	}	

Program 4 | Insertion Sort: Given list of n elements. Arrange them in an ascending order

Principle:

Insertion Sort works on the principle of sorting by insertion. Any given unsorted list can be divided into two lists such that one is sorted and the other is unsorted. For example, the given unsorted list

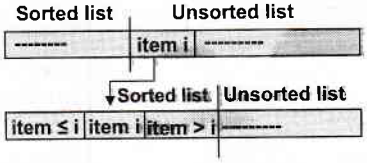
can be divided into two parts such that one list is sorted and other list is unsorted. The divided list is shown as:



Initially the sorted list consists of zero or one element, as the list containing zero or one element is always sorted and the unsorted list consists of the rest of the elements.

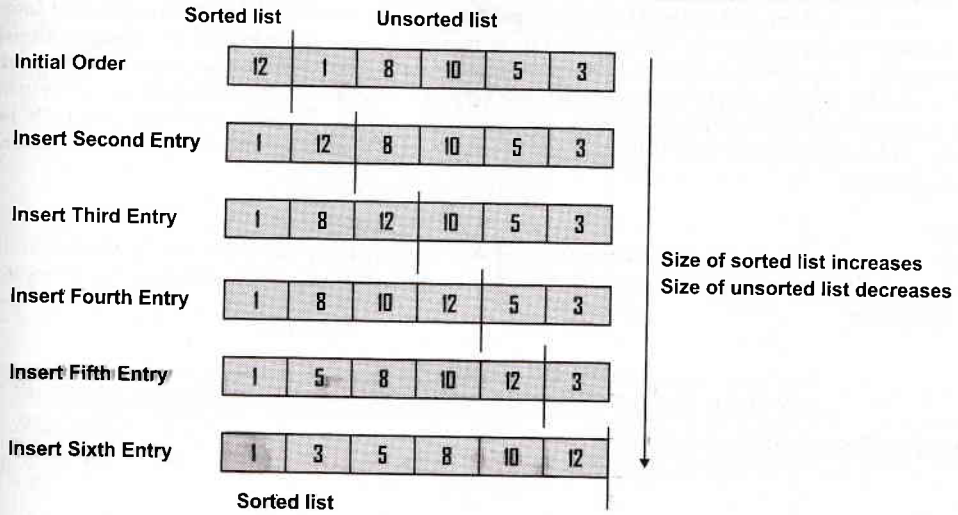
Insertion Sort sorts by removing one element from the unsorted list at a time and inserting it at a proper position in the sorted listed. To make room for the insertion, some of the elements in the sorted list need to be moved. Each iteration of Insertion Sort reduces the size of the unsorted list by one and increases the size of the sorted list by one. Ultimately, the unsorted list will vanish and the entire list will be sorted.

The general procedure of the Insertion Sort is shown in the figure below:



(Conta...)

Insertion Sort sorts the given list as shown below:



PE 4-4.c

```
1  //Insertion Sort
2  #include<stdio.h>
3  main()
4  {
5  int list[20], num, current, i, j;
6  printf("Enter the number of elements (max. 20)\t");
7  scanf("%d",&num);    //←Read the number of elements in the list.
8  printf("Enter the elements:\n");
9  for(i=0;i<num;i++)
10     scanf("%d",&list[i]);    //←Read the elements of the list
11 //←First element is sorted and the rest of the list is unsorted
12 for(i=1;i<num;i++)
13    if(list[i]<list[i-1])        //←Remove element from the unsorted
14    {                            //list and place it at proper position
15       current=list[i];          //in the sorted list
16       for(j=i-1;j>=0;j--)
17       {
18          list[j+1]=list[j];
19             if(j==0||list[j-1]<=current)
20                break;
21       }
22       list[j]=current;
23    }
24 printf("After sorting, elements are:\n");
25 for(i=0;i<num;i++)          //←Print sorted list
26    printf("%d\n",list[i]);
27 }
```

Output window

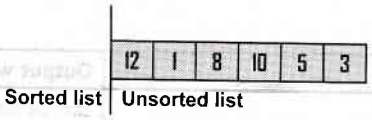
```
Enter the number of elements(max. 20)  6
Enter the elements:
12
10
5
-3
14
2
After sorting, elements are:
-3
2
5
10
12
14
```

Program 5 | Selection Sort: Given a list of n elements. Arrange them in an ascending order

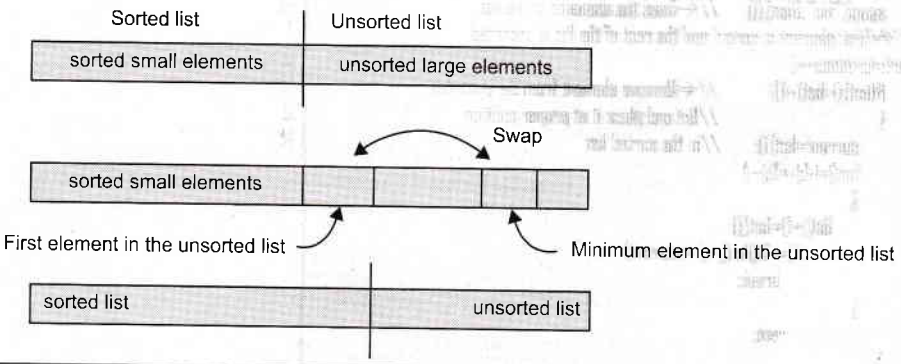
Insertion Sort has one major disadvantage. Insertion of an element removed from the unsorted list into the sorted list requires the elements in the sorted list to be moved to create space for the new element. Consider the insertion of sixth entry in the previous program. Insertion of 3 into the sorted list requires the movement of 5, 8, 10 and 12. These excessive movements become very expensive especially if the elements are very large such as records of employee's personal file or student transcripts. It would be far more efficient if an entry being moved could immediately be placed in its final position. Selection Sort accomplishes this goal and works on the following principle:

Principle:
Selection Sort works on the principle of sorting by selection. The given unsorted list is initially divided into two lists—the sorted list containing no element and the unsorted list containing all the elements. For example, the given unsorted list

can be divided into two parts as:

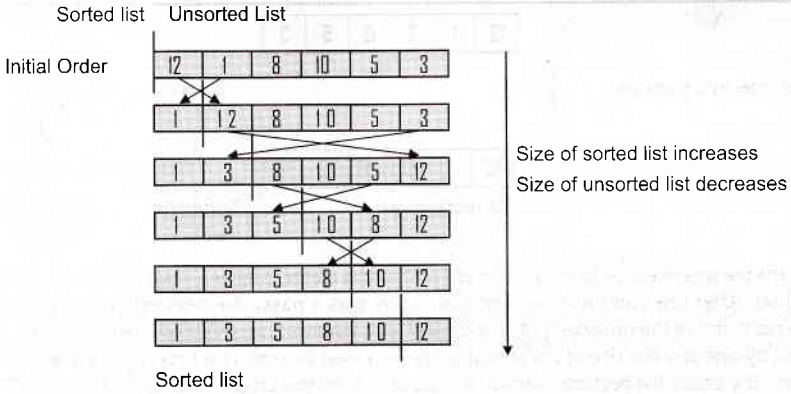


Selection Sort selects the minimum element from the unsorted list and exchanges it with the first element in the unsorted list. The selected element has moved to its final position; hence, the size of the sorted list is increased by one and the size of the unsorted list is decreased by one. This process of selecting the minimum element from the unsorted list, exchanging it with the first element in the unsorted list and then increasing the size of the sorted list and decreasing the size of the unsorted list by one is repeatedly followed till the entire list becomes sorted. The general procedure of Selection Sort is shown in the figure below:



(Contd...)

Selection Sort sorts the given list as shown below:



PE 4-5.c

```
1  //Selection Sort
2  #include<stdio.h>
3  main()
4  {
5  int list[20], num,min, temp, i, j;
6  printf("Enter the number of elements (max. 20)\t");
7  scanf("%d",&num);     //←Read number of elements in the list
8  printf("Enter the elements:\n");
9  for(i=0;i<num;i++)
10     scanf("%d",&list[i]);     //←Read the elements
11 for(i=0;i<num-1;i++)          //←Initially entire list is unsorted
12 {
13    min=i;
14    for(j=i+1;j<num;j++)     //←Select minimum element in the list
15       if(list[j]<list[min])
16          min=j;
17    { //←Place selected element at Ist position in the unsorted list
18    temp=list[min];
19    list[min]=list[i];
20    list[i]=temp;
21    }
22 }
23 printf("After sorting, elements are:\n");
24 for(i=0;i<num;i++)          //←Print sorted list
25    printf("%d\n",list[i]);
26 }
```

Output window

```
Enter the number of elements(max. 20)   6
Enter the elements:
12
10
5
-3
14
2
After sorting, elements are:
-3
2
5
10
12
14
```

Program 6 | Bubble Sort: Given a list of n elements. Arrange them in an ascending order

Principle:

Bubble Sort works on the following observation:

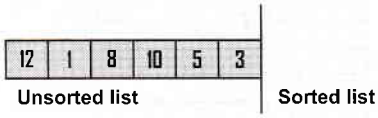
'Bubbles (or lighter elements) rise up in water and heavier elements sink'

The given unsorted list is initially divided into two lists—the sorted list containing no element and the unsorted list containing all the elements. For example, the given unsorted list

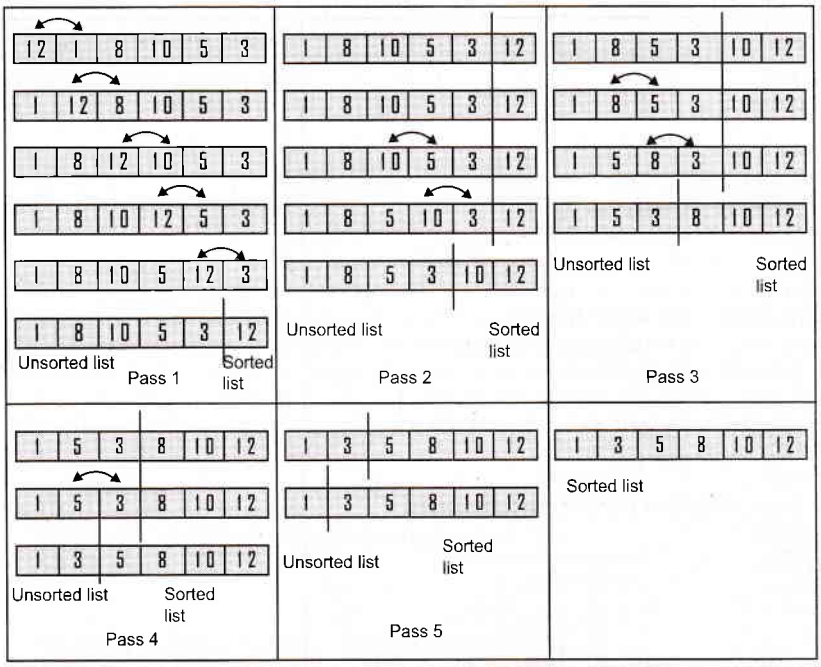
(Contd...)

Program 6 | Bubble Sort: Given a list of n elements. Arrange them in an ascending order *(Contd...)*

can be divided into two parts as:



Bubble Sort scans the unsorted list from left to right and swaps elements when a pair of adjacent elements is found to be out of order. After one complete iteration (also known as a **pass**), the heaviest element (i.e. the largest element) is at the right end of the unsorted list, but the earlier elements may still be out of order. The size of unsorted list is decreased by one and the size of the sorted list is increased by one. This process is repeated till the unsorted list vanishes and the entire list becomes sorted. Bubble Sort sorts the given list as shown below:



	PE 4-6.c	Output window
1	//Bubble Sort	Enter the number of elements(max. 20) 6
2	#include<stdio.h>	Enter the elements:
3	main()	12
4	{	10
5	int list[20], num,min, temp, i, j;	5
6	printf("Enter the number of elements (max. 20)\t");	-3
7	scanf("%d",&num); //←Read number of elements in the list	14
8	printf("Enter the elements:\n");	2
9	for(i=0;i<num;i++)	After sorting, elements are:
10	scanf("%d",&list[i]); //←Read the elements	-3
11	for(i=0;i<num-1;i++) //←Passes	2

(Contd...)

```
    for(j=0;j<num-1-i;j++)
        if(list[j]>list[j+1])    //←If elements are out of order, swap them
        {
            temp=list[j];
            list[j]=list[j+1];
            list[j+1]=temp;
        }
printf("After sorting, elements are:\n");
for(i=0;i<num;i++)            //←Print sorted list
    printf("%d\n",list[i]);
}
```

```
5
10
12
14
```

Program 7 | Given two sorted one-dimensional arrays A and B of size m and n, respectively. Merge them into a single-sorted array C that contains every element from arrays A and B in ascending order

PE 4-7.c

```
//Merge two Sorted arrays into one
#include<stdio.h>
main()
{
int A[20], B[20], C[40] ;
int i, j, l, h, m, n;
printf("Enter the number of elements in A (max. 20)\t");
scanf("%d",&m);              //←Read number of elements in A
printf("Enter the elements in sorted order:\n");
for(i=0;i<m;i++)
    scanf("%d",&A[i]);       //←Read the elements
printf("Enter the number of elements in B (max. 20)\t");
scanf("%d",&n);              //←Read number of elements in B
printf("Enter the elements in sorted order:\n");
for(i=0;i<n;i++)
    scanf("%d",&B[i]);       //←Read the elements
i=0; j=0; h=0;
while(i<m || j<n)
{
    if(A[i]<=B[j])
    {
        C[h]=A[i];
        i++;
    }
    else
    {
        C[h]=B[j];
        j++;
    }
    h++;
}
if(i==m)
    for(l=j;l<n;l++)
        C[h++]=B[l];
else if(j==n)
    for(l=i;l<m;l++)
        C[h++]=A[l];
```

Output window

```
Enter the number of elements in A (max. 20)    5
Enter the elements in sorted order:
1 3 5 7 9
Enter the number of elements in B (max. 20)    4
Enter the elements in sorted order:
2 4 6 8
After merging, elements are:
1 2 3 4 5 6 7 8 9
```

(Contd...)

	PE 4-7.c	Output window
38	printf("After merging, elements are:\n");	
39	for(i=0;i<m+n;i++) //←Print merged array C	
40	printf("%d ",C[i]);	
41	}	

Program 8 | Matrix addition: Add two matrices of order m × n

	PE 4-8.c	Output window
1	//Matrix Addition	Enter the order of matrices (max. 10 by 10) 3 3
2	#include<stdio.h>	Enter the elements of matrix-1:
3	main()	1 2 3
4	{	4 5 6
5	int mat1[10][10], mat2[10][10], resultant[10][10];	7 8 9
6	int m, n, row, col;	Enter the elements of matrix-2:
7	printf("Enter the order of matrices (max. 10 by 10)\t");	2 3 4
8	scanf("%d %d",&m, &n);	1 2 3
9	printf("Enter the elements of matrix-1:\n");	1 1 0
10	for(row=0;row<m;row++)	The result of matrix addition is :
11	{	3 5 7
12	for(col=0;col<n;col++)	5 7 9
13	scanf("%d",&mat1[row][col]);	8 9 9
14	}	
15	printf("Enter the elements of matrix-2:\n");	
16	for(row=0;row<m;row++)	
17	{	
18	for(col=0;col<n;col++)	
19	scanf("%d",&mat2[row][col]);	
20	}	
21	for(row=0;row<m;row++)	
22	for(col=0;col<n;col++)	
23	resultant[row][col]=mat1[row][col]+mat2[row][col];	
24	printf("The result of matrix addition is:\n");	
25	for(row=0;row<m;row++)	
26	{	
27	for(col=0;col<n;col++)	
28	printf("%d ",resultant[row][col]);	
29	printf("\n");	
30	}	
31	}	

Program 9 | Matrix multiplication: Multiply two matrices

Given two matrices A and B

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} B_{11} & B_{12} & B_{13} \\ B_{21} & B_{22} & B_{12} \end{bmatrix}$$

(Contd...)

The result of the matrix multiplication is given as:

$$C_{2\times3} = A_{2\times2}\, B_{2\times3} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} & A_{11}B_{13} + A_{12}B_{23} \\ A_{21}B_{31} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} & A_{21}B_{13} + A_{22}B_{23} \end{bmatrix}$$

PE 4-9.c

```
/* Matrix Multiplication */
#include<stdio.h>
#include<stdlib.h>
main()
{
int mat1[10][10], mat2[10][10], resultant[10][10]={0};
int m1, n1, m2, n2, i, j, k;
printf("Enter the order of matrix-1 (max. 10 by 10)\t");
scanf("%d %d",&m1, &n1);
printf("Enter the elements of matrix-1:\n");
for(i=0;i<m1;i++)
{
   for(j=0;j<n1;j++)
      scanf("%d",&mat1[i][j]);
}
printf("Enter the order of matrix-2 (max. 10 by 10)\t");
scanf("%d %d",&m2, &n2);
printf("Enter the elements of matrix-2:\n");
for(i=0;i<m2;i++)
{
   for(j=0;j<n2;j++)
      scanf("%d",&mat2[i][j]);
}
if(n1!=m2)
{
   printf("Matrices are not compatible for multiplication\n");
   exit(0);
}
else
{
   for(i=0;i<m1;i++)
      for(j=0;j<n2;j++)
         for(k=0;k<n1;k++)
            resultant[i][j]=resultant[i][j]+mat1[i][k]*mat2[k][j];
}
printf("The result of matrix multiplication is:\n");
for(i=0;i<m1;i++)
{
   for(j=0;j<n2;j++)
      printf("%d ",resultant[i][j]);
   printf("\n");
}
}
```

Output window

```
Enter the order of matrix-1 (max. 10 by 10)    2 3
Enter the elements of matrix-1:
1 2 3
4 5 6
Enter the order of matrix-2 (max. 10 by 10)    3 3
Enter the elements of matrix-2:
2 3 4
1 2 3
1 1 0
The result of matrix multiplication is :
7 10 10
19 28 31
```

Program 10 | Find the sum of principal diagonal elements of a square matrix

The set of elements extending from the upper-left-most corner to the lower-right-most corner in a square matrix are known as **principal diagonal elements**. An element A_{ij} of a square matrix is principle diagonal element if and only if i=j.

PE 4-10.c

```
1  //Sum of principle diagonal elements
2  #include<stdio.h>
3  main()
4  {
5  int matrix[10][10];
6  int order, sum=0, i, j;
7  printf("Enter the order of the square matrix(max. 10)\t");
8  scanf("%d",&order);
9  printf("Enter the elements of matrix:\n");
10 for(i=0;i<order;i++)
11 {
12     for(j=0;j<order;j++)
13         scanf("%d",&matrix[i][j]);
14 }
15 for(i=0;i<order;i++)
16     sum=sum+matrix[i][i];
17 printf("Sum of elements of principal diagonal is %d",sum);
18 }
```

Output window

```
Enter the order of the square matrix (max. 10)    3
Enter the elements of matrix:
1 2 3
4 5 6
7 8 9
Sum of elements of principal diagonal is 15
```

Program 11 | Matrix transpose: Find the transpose of a given matrix

The **transpose** of the matrix A is another matrix A^T, which can be found by any one of the following actions:

1. Writing the rows of A as the columns of A^T
2. Writing the columns of A as the rows of A^T
3. Reflect A about its main diagonal to obtain A^T(only possible in case of square matrix).

The transpose of an m×n matrix A with elements A_{ij} is an n×m matrix $A^T=A_{ji}$, $1\le i\le n$ and $1\le j\le m$.

PE 4-11.c

```
1  //Matrix Transpose
2  #include<stdio.h>
3  main()
4  {
5  int matrix[10][10], matrix_transpose[10][10];
6  int  m, n, i, j;
7  printf("Enter the order of the matrix(max. 10 by 10)\t");
8  scanf("%d %d",&m, &n);
9  printf("Enter the elements of the matrix:\n");
10 for(i=0;i<m;i++)
11 {
12     for(j=0;j<n;j++)
13         scanf("%d",&matrix[i][j]);
14 }
15 for(i=0;i<n;i++)
```

Output window

```
Enter the order of matrix (max. 10 by 10)    2 4
Enter the elements of matrix:
1 2 3 4
5 6 7 8
Transpose of the matrix is:
1 5
2 6
3 7
4 8
```

(Contd...)

```
    for(j=0;j<m;j++)
        matrix_transpose[i][j]=matrix[j][i];
printf("Transpose of the matrix is:\n");
for(i=0;i<n;i++)
{
    for(j=0;j<m;j++)
        printf("%d ",matrix_transpose[i][j]);
    printf("\n");
}
}
```

Program 12 | Check whether a given square matrix is symmetric or not

A square matrix A is **symmetric** if $A=A^T$ (i.e. the matrix is equal to its transpose).

PE 4-12.c

```
//Symmetric matrix
#include<stdio.h>
main()
{
int matrix[10][10], matrix_transpose[10][10], unequal=0;
int i,j,order;
printf("Enter the order of the square matrix(max. 10 by 10)\t");
scanf("%d",&order);
printf("Enter the elements of the matrix:\n");
for(i=0;i<order;i++)
{
    for(j=0;j<order;j++)
        scanf("%d",&matrix[i][j]);
}
for(i=0;i<order;i++)
    for(j=0;j<order;j++)
        matrix_transpose[i][j]=matrix[j][i];
for(i=0;i<order;i++)
{
    for(j=0;j<order;j++)
        if(matrix[i][j]!=matrix_transpose[i][j])
        {
            unequal=1;
            break;
        }
    if(unequal==1)
        break;
}
if(unequal==0)
    printf("The matrix is symmetric\n");
else
    printf("The matrix is not symmetric\n");
}
```

Output window

```
Enter the order of the square matrix (max. 10 by 10)    3
Enter the elements of matrix:
1 2 3
2 1 4
3 4 1
The matrix is symmetric
```

(Contd...)

Program 13 | Upper triangular matrix: Extract the upper triangular matrix from a square matrix

A square matrix in which all the elements below the main (i.e. principal) diagonal are zero is known as **upper triangular matrix** and a square matrix in which all the elements above the main diagonal are zero is known as **lower triangular matrix**.

Upper triangular matrix can be extracted from a square matrix by extracting the elements of principle diagonal and the elements that lie above it.

PE 4-13.c

```
1  //Extraction of Upper Triangular matrix
2  #include<stdio.h>
3  main()
4  {
5  int matrix[10][10], ut_matrix[10][10], unequal=0;
6  int i,j,order;
7  printf("Enter the order of the square matrix(max. 10 by 10)\t");
8  scanf("%d",&order);
9  printf("Enter the elements of the matrix:\n");
10 for(i=0;i<order;i++)
11 {
12     for(j=0;j<order;j++)
13         scanf("%d",&matrix[i][j]);
14 }
15 for(i=0;i<order;i++)
16     for(j=0;j<order;j++)
17         if(i<=j)
18             ut_matrix[i][j]=matrix[i][j];
19         else
20             ut_matrix[i][j]=0;
21 printf("Upper Triangular matrix is:\n");
22 for(i=0;i<order;i++)
23 {
24     for(j=0;j<order;j++)
25         printf("%d ",ut_matrix[i][j]);
26     printf("\n");
27 }
28 }
```

Output window

```
Enter the order of the square matrix (max. 10 by 10)  3
Enter the elements of the matrix:
1 2 3
2 1 4
3 4 1
Upper Triangular matrix is:
1 2 3
0 1 4
0 0 1
```

Program 14 | Strictly upper triangular matrix: Check whether a given matrix is strictly upper triangular or not

An upper triangular matrix is **strictly upper triangular** if the elements of the principal diagonal are zero.

PE 4-14.c

```
1  //Strictly Upper Triangular matrix
2  #include<stdio.h>
3  main()
```

Output window

```
Enter the order of the square matrix (max. 10 by 10)  3
Enter the elements of matrix:
0 2 3
```

(Contd...)

```
{
int matrix[10][10], notzero=0;
int i,j,order;
printf("Enter the order of the square matrix(max. 10 by 10)\t");
scanf("%d",&order);
printf("Enter the elements of the matrix:\n");
for(i=0;i<order;i++)
{
    for(j=0;j<order;j++)
        scanf("%d",&matrix[i][j]);
}
for(i=0;i<order;i++)
{
    for(j=0;j<order;j++)
        if(i>=j)
            if(matrix[i][j]!=0)
            {
                notzero=1;
                break;
            }
    if(notzero==1)
        break;
}
if(notzero==1)
    printf("The given matrix is not strictly upper triangular\n");
else
    printf("The given matrix is strictly upper triangular\n");
}
```

```
0 0 4
0 0 0
The given matrix is strictly upper triangular
```

Output window (second execution)

```
Enter the order of the square matrix (max. 10 by 10)  3
Enter the elements of matrix:
6 2 3
0 0 4
2 0 0
The given matrix is not strictly upper triangular
```

Program 15 | Matrix Inverse: Find the inverse of a 3 × 3 matrix

The **inverse** of a square matrix A, is a matrix A^{-1} such that $AA^{-1}=I$, where I is the identity matrix. The matrix A has an inverse if and only if the determinant of A (written as |A|) is not equal to zero. A matrix whose inverse exists is known as **invertible matrix**

Finding the inverse of a matrix using Gauss–Jordan elimination method:

Step 1: Start
Step 2: Read the elements of the matrix A whose inverse is to be found
Step 3: Check whether its determinant is zero or not. If it is zero, print that inverse does not exist and stop, else proceed to Step 4
Step 4: Form the augmented matrix B. It is formed by augmenting the matrix A with an identity matrix of the same dimensions. If the matrix A is of order m × n, the augmented matrix B = [A | I] is of order m × 2n. It consists of two parts: the first part corresponds to A and the second part corresponds to I
Step 5: Apply elementary row operations on the augmented matrix B so that its first part reduces to identity matrix. By performing these row operations, the inverse of the matrix A appears in the second part
Step 6: The matrix augmentation can be undone to retrieve the inverse of the matrix
Step 7: Print the inverse of the matrix
Step 8: Stop

(Contd...)

Example:

$$A=\begin{bmatrix}2&3&1\\1&1&2\\2&3&4\end{bmatrix}\quad I=\begin{bmatrix}1&0&0\\0&1&0\\0&0&1\end{bmatrix}\quad B=\begin{bmatrix}2&3&1&1&0&0\\1&1&2&0&1&0\\2&3&4&0&0&1\end{bmatrix}$$

Step 1: $R_1 \rightarrow R_1/B[0][0]$

$$B=\begin{bmatrix}1&3/2&1/2&1/2&0&0\\1&1&2&0&1&0\\2&3&4&0&0&1\end{bmatrix}$$

Step 2: $R_2 \rightarrow R_2-B[1][0]*R_1$

$$B=\begin{bmatrix}1&3/2&1/2&1/2&0&0\\0&-1/2&3/2&-1/2&1&0\\2&3&4&0&0&1\end{bmatrix}$$

Step 3: $R_3 \rightarrow R_3-B[2][0]*R_1$

$$B=\begin{bmatrix}1&3/2&1/2&1/2&0&0\\0&-1/2&3/2&-1/2&1&0\\0&0&3&-1&0&1\end{bmatrix}$$

Step 4: $R_2 \rightarrow R_2/B[1][1]$

$$B=\begin{bmatrix}1&3/2&1/2&1/2&0&0\\0&1&-3&1&-2&0\\0&0&3&-1&0&1\end{bmatrix}$$

Step 5: $R_1 \rightarrow R_1-B[0][1]*R_2$

$$B=\begin{bmatrix}1&0&5&-1&3&0\\0&1&-3&1&-2&0\\0&0&3&-1&0&1\end{bmatrix}$$

Step 6: $R_3 \rightarrow R_3-B[2][0]*R_2$

$$B=\begin{bmatrix}1&0&5&-1&3&0\\0&1&-3&1&-2&0\\0&0&3&-1&0&1\end{bmatrix}$$

Step 7: $R_3 \rightarrow R_3/B[2][2]$

$$B=\begin{bmatrix}1&0&5&-1&3&0\\0&1&-3&1&-2&0\\0&0&1&-1/3&0&1/3\end{bmatrix}$$

Step 8: $R_1 \rightarrow R_1-B[0][2]*R_3$

$$B=\begin{bmatrix}1&0&0&2/3&3&-5/3\\0&1&-3&1&-2&0\\0&0&1&-1/3&0&1/3\end{bmatrix}$$

(Contd...)

Step 9: $R_2 \rightarrow R_2 - B[1][2]*R_3$

$$B = \begin{bmatrix} 1 & 0 & 0 & 2/3 & 3 & -5/3 \\ 0 & 1 & 0 & 0 & -2 & 1 \\ 0 & 0 & 1 & -1/3 & 0 & 1/3 \end{bmatrix}$$

Step 10: Undo the matrix augmentation and print inverse

$$A^{-1} = \begin{bmatrix} 2/3 & 3 & -5/3 \\ 0 & -2 & 1 \\ -1/3 & 0 & 1/3 \end{bmatrix}$$

PE 4-15.c

```
//Inverse of a matrix
#include<stdio.h>
main()
{
float matrix[3][3], aug_matrix[3][6];
float identity[3][3]={1,0,0,0,1,0,0,0,1};    //←3×3 identity matrix
float c,r, sub, det;
int i,j,order=3,k, row, col;
printf("Enter the elements of 3 by 3 matrix:\n");
for(i=0;i<order;i++)
{
    for(j=0;j<order;j++)
        scanf("%f",&matrix[i][j]);    //←Read the elements of the matrix
}
//←Calculate the determinant of the matrix
det=matrix[0][0]*(matrix[1][1]*matrix[2][2]-matrix[2][1]*matrix[1][2]) -
matrix[0][1]*(matrix[1][0]*matrix[2][2]-matrix[2][0]*matrix[1][2]) +
matrix[0][2]*(matrix[1][0]*matrix[2][1]-matrix[2][0]*matrix[1][1]);
if(det!=0)    //←if determinant is not zero inverse can be found
{
    for(i=0;i<order;i++)    //←augmenting the matrix with identity matrix
        for(j=0;j<order;j++)
        {
            aug_matrix[i][j]=matrix[i][j];
            aug_matrix[i][j+3]=identity[i][j];
        }
//←Elementary row operations
    for(i=0;i<order;i++)
        for(j=0;j<order;j++)
        {
            if(i==j)
            {
//←Implementing Steps 1, 4 and 7 described in the example above
                c=aug_matrix[i][i];
                for(k=0;k<6;k++)
                    aug_matrix[i][k]=aug_matrix[i][k]/c;
//←Implementing Steps 2,3,5,6,8 and 9 described in the example above
                for(row=0;row<order;row++)
```

Output window

```
Enter the elements of 3 by 3 matrix:
2 3 1
1 1 2
2 3 4
Inverse of the matrix is:
 0.67    3.00   -1.67
 0.00   -2.00    1.00
-0.33    0.00    0.33
```

(Contd...)

(Contd...)

	PE 4-15.c	Output window
39	{	
40	sub=aug_matrix[row][j];	
41	for(col=0;col<6;col++)	
42	if(row!=i)	
43	{	
44	aug_matrix[row][col]-=sub*aug_matrix[i][col];	
45	}	
46	}	
47	}	
48	}	
49	printf("Inverse of the matrix is:\n");	
50	for(i=0;i<order;i++) //←Printing the inverse of the matrix	
51	{	
52	for(j=0;j<order;j++)	
53	printf("%5.2f ",aug_matrix[i][j+3]);	
54	printf("\n");	
55	}	
56	}	
57	else //←if determinant is zero, print that inverse does not exist	
58	printf("Inverse does not exist");	
59	}	

Test Yourself

1. Fill in the blanks in each of the following:
   a. An array is used for the storage of _____ data.
   b. The array index in C language starts with _____.
   c. The elements of an array are always stored in _____ memory locations.
   d. The size specifier in an array declaration must be a compile time expression of _____ type.
   e. The elements of an array can be accessed by using _____ operator.
   f. The object pointed to by a pointer can be indirectly accessed by using _____ operator.
   g. The expression equivalent to the expression `arr[5][4]`, where `arr` is an integer array is _____.
   h. The biggest advantage of arrays is their _____ capabilities.
   i. In an expression, if the number of subscripts used with the array is less than the dimensions of the array, the expression always refers to a/an _____.
   j. The comparison of two null pointers always results in _____.

2. State whether each of the following is true or false. If false, explain why.
   a. In an array declaration, the number of initializers in the initialization list should be less than or at most equal to the value of size specifier.
   b. The index of an array must be a positive integer greater than zero.
   c. A pointer variable can be initialized with a constant value zero.
   d. A pointer to any type of object can be assigned to a pointer of type `void*` without explicit type casting.
   e. A `void` pointer can be assigned to a pointer variable without explicit type casting.
   f. The name of the array refers to the base address of the complete array.
   g. The size of an array cannot be changed at the run time.
   h. If the size specifier is not mentioned in an array declaration, the size of the array is automatically initialized to a single element.
   i. Multi-dimensional arrays in C are stored in the memory using column major order of storage.
   j. The declaration statement `int* a[10];` declares `a` as a pointer to an integer array of `10` elements.

3. Programming exercises:
   a. Write a C program to find the sum of all the elements of an array.
   b. An array consists of integers. Write a C program to count the number of elements less than, greater than and equal to zero.
   c. Write a C program to check whether a given matrix is skew-symmetric or not.
   d. Write a C program to extract lower-triangular matrix from a square matrix.
   e. Write a C program that returns the position of the largest element in an array.
   f. In a class there are twenty students and each student undergoes five courses. Write a C program to find out the average marks secured by each student and the overall average of the class.

5

FUNCTIONS

Learning Objectives

In this chapter, you will learn about:

- Functions
- Advantages of using functions
- Classification of functions as user-defined functions and library functions
- User-defined functions
- How to declare, define and call functions
- Way of increasing flexibility of functions
- Different ways of supplying inputs to a function
- return statement
- How to provide default inputs to a function
- Recursion and its use to solve problems
- Classification of recursion
- How recursion works
- Tower of Hanoi problem
- Function type and pointers to functions
- Array of function pointers
- Passing arrays and functions to functions
- Commonly used library functions
- Variable argument functions

5.1 Introduction

In the previous chapters, you have seen how to declare identifiers (Chapter 1), how to write expressions (Chapter 2) and how to write statements (Chapter 3). In this chapter, I will tell you how to group these components in a function so that these components can be reused in a program. I will describe the advantages of using functions, how to declare, define and call them. You will be familiarized with the methods of increasing flexibility of a function and different ways of passing inputs to a function. Finally, we will have a discussion about the advanced topics like pointers to functions, arrays of function pointers and passing functions to a function.

5.2 Functions

Most of the computer programs that solve real-world problems are much bigger and complex than the programs presented in the first few chapters. The existing software engineering practices used to develop such complicated programs work on the following principles:

1. **Top-down design, modularization, stepwise refinement and bottom-up development:** According to this principle, a complex problem should be modularized (i.e. divided) into sub-problems that are simpler, manageable and easier to solve as compared to the original problem. If the divided sub-problems are still complex and cannot be easily solved, they are further divided into sub-problems. Each level of division provides a refinement and simplicity to the problem. This process of modularization is carried out till the sub-problems are simple enough and can be easily solved. The solutions for these simple problems are then developed and merged to provide a solution for the overall complex problem. This approach of problem solving is also known as **'divide-and-conquer strategy.'** This strategy is practically followed in real life whereby a senior officer responsible for the execution of a work divides the work among his subordinates. The subordinate officers may further divide the assigned work among their subordinates, get the work done and report back to their senior officer. This hierarchical division of work is shown in Figure 5.1.

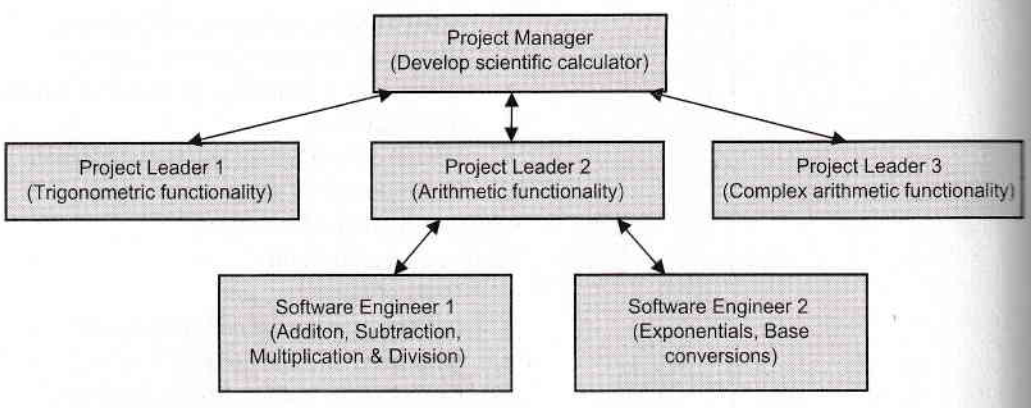


Figure 5.1 | Hierarchical division of work

Thus, in this approach of solution development, a solution to the given problem is thought of at an abstract level. This abstract solution is divided into modules, and each level of division refines the solution by adding details to the divided modules. The process of division is carried out till the divided modules are well defined and simple enough to be generated (i.e. coded). The functionality of each module is kept in a separate function. These functions are relatively independent of each other and interact with each other to provide a solution to the overall problem.

2. **'Don't reinvent the wheel.'** Another important software engineering principle states that **'Don't reinvent the wheel.'** This means that the functionality that has already been developed should be reused instead of being developed again. Functions help a lot in realizing this principle. The commonly required functionality is developed and kept in standard libraries for the use in the form of library functions. In the previous chapters, we have used the input and output functionality by using scanf and printf library functions. The C standard library provides a rich set of functionality for performing the common mathematical calculations, string and character manipulations, input/output and other useful operations.

The above two software engineering principles give a hint about the importance and the need of functions. Several other advantages of modularizing a program into functions include:

1. Reduction in code redundancy
2. Enabling code reuse
3. Better readability
4. Information hiding
5. Improved debugging and testing
6. Improved maintainability

As already described in Chapter 1, a C program is made up of functions. Functions interact with each other to accomplish a particular task. They are classified according to the following criteria:

1. Based upon who develops the function
2. Based upon the number of arguments a function accepts

5.3 Classification of Functions

5.3.1 Based Upon who Develops the Function

Based upon who develops the function, functions are classified as:

1. User-defined functions
2. Library functions

5.3.1.1 User-defined functions

User-defined functions are the functions that are defined (i.e. developed) by the user at the time of writing a program. The user develops the functionality by writing the body of the function. These functions are sometimes referred to as **programmer-defined functions.** Program 5-1 illustrates the use of user-defined functions add, sub and println.

Line	Prog 5-1.c	Output window
1	//User defined functions	Enter the values 4 3
2	#include<stdio.h>	----------
3	//Function declarations or function prototypes	Result of addition is 7
4	println();	Result of subtraction is 1
5	int add(int, int);	**Remarks:**
6	int sub(int x, int y);	• println, add and sub are user-defined functions
7	//main function, the master function	• In line numbers 4, 5 and 6, user-defined functions are declared
8	main()	• In line numbers 23 to 34, they are defined
9	{	• In line numbers 15, 16 and 17, they are called
10	int a,b,sum, diff;	• Line numbers 23, 27 and 31 consist of headers of the functions println, add and sub
11	printf("Enter the values\t");	• The variables declared in the function headers or function declarations are known as parameters
12	scanf("%d %d",&a, &b);	• In line numbers 6, x and y are the parameter names
13	//Function invocations	• In line numbers 27 and 31, a and b are the parameter names
14	//Asking the workers to do work	• The parameters declared inside the function headers are similar to the variables declared inside the body of the function
15	sum=add(a,b);	• **main is also a user-defined function**
16	diff=sub(a,b);	
17	println();	
18	//Master presents the results returned by workers	
19	printf("Result of addition is %d\n",sum);	
20	printf("Result of subtraction is %d\n",diff);	
21	}	
22	//Function definitions	
23	println()	
24	{	
25	printf("----------\n");	
26	}	
27	int add(int a, int b)	
28	{	
29	return a+b;	
30	}	
31	int sub(int a, int b)	
32	{	
33	return a-b;	
34	}	

Program 5-1 | A program that illustrates the use of user-defined functions

As you have seen in the code snippet in Program 5-1, there are three aspects of working with user-defined functions:

1. Function declaration, also known as function prototype
2. Function definition
3. Function use, also known as function call or function invocation

5.3.1.1.1 Function Declaration

All identifiers (except labels) need to be declared before they are used. As function names are also identifiers, this is true for functions as well. All the functions need to be declared

defined[†] before they are used (i.e. called[‡]). The general form of a **function declaration** is:

```
[return_type] function_name([parameter_list or parameter_type_list]);
```

The important points about the function declaration are as follows:

1. The terms enclosed within the square brackets are optional and might not be present in a function declaration statement. The terms shown in **bold** are the mandatory parts of a function declaration.
2. The function declaration consists of the name of the function along with its return type and parameter list or parameter-type list enclosed within parentheses. Function declaration is also known as **function prototype**. For example, in Program 5-1 the declaration of function `add` in line number 5 consists of **parameter-type list**, and the declaration of function `sub` in line number 6 consists of **parameter list**.
3. Function names are identifiers. All syntactic rules discussed in Section 1.5.1 for writing identifier names are applicable for writing the function names as well. The name of a function is also termed as **function designator**.
4. The specification of the return type is optional. If specified, the return type of a function can be any type (e.g. `char`, `int`, `float`, `int*`, `int**`, `void`, etc.) except array type and function type.[§] For example, in Program 5-1 the return type of the function `println` is not specified and the return type of functions `add` and `sub` is `int`.
5. The syntactic rules for writing a parameter-type list and parameter list in a function declaration are as follows:
   a. The **parameter-type list** is a comma-separated list of parameter types. The parameter type can be any type (e.g. `char`, `int`, `float`, `int*`, `int**`, `void`, etc.) except function type. If only a parameter-type list is mentioned, the function declaration is said to have **abstract parameter declaration**.
   b. A parameter name can optionally follow each parameter type. A parameter name should be a valid variable name. If parameter names follow parameter types in a parameter-type list, it becomes a **parameter list**. If the function declaration consists of a parameter list, it is said to have **complete parameter declaration**. For example, in Program 5-1 (in line number 5) function `add` has abstract parameter declaration and (in line number 6) function `sub` has complete parameter declaration.
   c. Using a combination of complete parameter declaration and abstract parameter declaration (i.e. naming some of the parameters and leaving the rest of them unnamed) is also allowed. For example, the following declarations of function `add` are also allowed:

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

   d. No two parameter names appearing in the parameter list can be the same.
   e. The shorthand declaration of parameters in the parameter list is not allowed.
6. Function declaration is a statement, so it must be terminated with a semicolon.

[†] Refer Section 5.3.1.1.2 for a description on function definitions.
[‡] Refer Section 5.3.1.1.3 for a description on function calls.
[§] Refer Section 5.3.1.1.8 for a description on the function type.

7. A function need not be declared, if it is defined before it is called.

The following function declarations are valid:

```
1. add();                  //←Return type and parameter list are not present
2. int add(int,int);       //←int is the return type and int, int is the parameter-type list
3. int* add(int,float);    //←int* is the return type and int, float is the parameter-type list
4. int add(int a, int b);  //←Parameter list contains the names of parameters, i.e. a and b
5. int add(int, int b);    //←Combination of abstract and complete parameter declaration
```

The following function declarations are not valid:

```
1. int add(int a, float a);  //←Both the parameter names are the same
2. int add&sub(int, int);    //←Name of the function is not valid as it contains the special
                                character &
3. int add(int a,b);         //←Shorthand declaration of parameters is not allowed
4. int add(int a, int b)     //←The declaration is not terminated with a semicolon
```

Function prototypes (i.e. function declarations) are important and their necessity can be seen from two different perspectives:

1. **User perspective** — It tells the user how to use a pre-defined or library function.[¶] It tells the user the number of parameters along with their types that a function expects and its return type. This is necessary and sufficient information for a user to use a function. For example, consider the following function prototype:

   ```
   int add(int,int);
   ```

   It tells the user that function add expects two integers and returns the result as an integer. With all this information, the user will be able to use the function add. Function prototype does not provide any information about how the functionality is implemented by the function. We have been able to use the printf function in the previous chapters because we know its prototype. The prototype of the printf function is available in the header file stdio.h. We do not know anything about how printing functionality is implemented by the printf function.

2. **Compiler perspective** — It allows the compiler to perform **type checking**. By type checking the compiler ensures that while making a function call, the user provides the correct number and the correct type of arguments. If the number of arguments is not the same as the number of parameters or if their types are not compatible with the types of parameters provided in the function declaration, the compiler issues an error message.

If some of the parameters are provided with default arguments,[††] the number of arguments in a function call can be lesser than the number of parameters.

[¶] Refer Section 5.3.1.2 for a description on library functions.

[††] Refer Section 5.3.1.1.5 for a description on default arguments.

5.3.1.1.2 Function Definition

Function definition, also known as **function implementation**, means composing a function. Every function definition consists of two parts:

1. Header of the function
2. Body of the function

Thus, defining a function involves composing its header and the body.

5.3.1.1.2.1 Header of a Function

The general form of **header of a function** is:

```
[return_type] function_name([parameter_list])
```

The important points about the function header are as follows:

1. The terms enclosed within the square brackets are optional and might not be present in a function header. The terms shown in bold are the mandatory part of a function header.
2. Unlike function declaration, the header of a function can only have complete parameter declaration. It cannot have abstract parameter declaration or a combination of abstract and complete parameter declaration. The variables declared in the parameter list will receive the data sent by the calling function.[‡‡] They serve as the inputs to the function.
3. No two parameter names appearing in the parameter list can be the same.
4. The shorthand declaration of parameters in the parameter list is not allowed.
5. The return type and the number and the types of parameters in the function header should exactly match the corresponding return type and the number and types of parameters in the function declaration, if it is present. For example, look at the function declarations in line numbers 4, 5 and 6 and function headers in line numbers 23, 27 and 31 in Program 5-1.
6. It is not mandatory to have the same names for the parameters in the function declaration and function definition. For example, in Program 5-1, the names of parameters in the declaration of function `sub` in line number 6 are `x` and `y` while the names of parameters in the header of the function `sub` in line number 31 are `a` and `b`.
7. The header of a function is not terminated with a semicolon.

5.3.1.1.2.2 Body of a Function

The **body of a function** consists of a set of statements enclosed within braces. The body of a function can have non-executable statements and executable statements. The non-executable statements can only come before the executable statements. The non-executable statements declare the local variables in the function and the executable statements determine its functionality, i.e. what the function does. A function can optionally have special executable statement known as the `return` statement.[§§] The `return` statement is used to return the result of the computations done in the called function and/or to return the program control back to the calling function.

Backward Reference: Executable and non-executable statements (Chapter 3).

[‡‡] Refer Section 5.3.1.1.3 for a description on calling functions and called functions.

[§§] Refer Section 5.3.1.1.3.3.1 for a description on the `return` statement.

Forward Reference: Local variables (Chapter 7).

5.3.1.1.3 Function Invocation/Call/Use

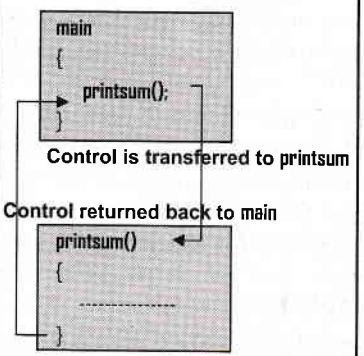
The call to a function can be well described along with the discussion on the classification of functions. Depending upon their inputs (i.e. parameters) and outputs, functions are classified as:

1. Functions with no input–output
2. Functions with inputs and no output
3. Function with inputs and one output
4. Function with inputs and outputs

5.3.1.1.3.1 Function with No Input–Output

A **function with no input–output** does not accept any input and does not return any result. Since no input is to be given to the function, the parameter list of such functions is empty. Even if the parameter list is empty, the function header must have the empty set of parentheses or with the keyword void.[¶¶] These functions have limited functionality and are not flexible (i.e. they cannot be used in a variety of circumstances). Due to their limited functionality they have limited utility too. Consider the snippet in Program 5-2.

	Trace Col. 2	Prog 5-2.c
1		//Function with no input-output
2		#include<stdio.h>
3		//Function declaration
4		printsum();
5		//main function, the master function
6	1	main()
7		{
8	2	printsum();
9	6	}
10		//Definition of function printsum
11	3	printsum()
12		{
13	4	printf("Sum of 2 and 3 is %d",2+3);
14	5	}



Output window

Sum of 2 and 3 is 5

Warnings (2):

- Function should return a value in function main
- Function should return a value in function printsum

Remarks:

- Ignore the warnings for the time being
- printsum is a function with no input–output
- The order of execution of the statements is depicted in the trace column (i.e. column 2)

Program 5-2 | A program that uses a function with no input–output

The important points about functions with no input–output are as follows:

1. In Program 5-2, the function printsum has no input and does not return any result.
2. The function printsum has been invoked, i.e. called in line number 8. A function with no inputs can be **called** by writing a **function designator** (i.e. name of the function) fol-

[¶¶] Refer Section 5.3.1.1.3.1.1 for a description on void functions.

lowed by a **function call operator,** i.e. ( ). The function designator followed by the function call operator is known as a **function call**.

3. The function that calls a function (i.e. which contains a function call) is known as a **calling function**, and the function that has been called is known as a **called function**. In the given code, main is the calling function and printsum is the called function.
4. A function call terminated with a semicolon is known as a **function call statement**.
5. After the execution of the function call statement, the program control is transferred to the called function. The execution of the calling function is suspended and the called function starts execution. For example, in Program 5-2, after the execution of the function call statement in line number 8, the program control transfers to line number 11. The order of execution of statements in a program can be checked by tracing the program. The program trace is depicted in column 2. Note the position of trace arrows 2 and 3 in Program 5-2.
6. After the execution of the called function (with no output) is complete, the program control returns to the calling function, and the calling function resumes its execution. In Program 5-2, this is depicted by trace steps 5 and 6 in column 2.

1. The **execution of C program** always begins with the function main. Function main need not be explicitly called.
2. Tracing is a debugging technique in which the statements of a program are executed one by one. Non-executable statements are not executed. Hence, during the tracing, the program control does not stop at non-executable statements. Thus, for non-executable statements trace arrows are not shown. The shortcut key for tracing in Borland TC 3.0 and 4.5 is F7. The shortcut key for tracing in MS-Visual C++ 6.0 is F11. Keep on pressing these keys to trace the program.

5.3.1.1.3.1.1 void Functions

Program 5-2 on compilation gives a warning message 'Function should return a value.' We have been ignoring this warning since Chapter 1 but now it is the time to know the reason behind this warning and how to remove it.

Every function in C language is supposed to return an integer value. If the return type of a function is not specified, it is assumed to be int by default. Thus, in Program 5-2, the return type of the functions main and printsum is assumed to be int. As no return statement is used within the body of these functions to return the expected integer value, the compiler gives the warning message 'Function should return a value.'

Removal of warning message

If a function does not return any value, then the return type of the function should be specified as void (means nothing). Functions whose return type is void are known as **void functions.** Reconsider the code snippet mentioned in Program 5-2 with void mentioned as the return type of the functions main and printsum. The modified form of the code listed in Program 5-2 is mentioned in Program 5-3.

Line	Prog 5-3.c	Output window
1	//Function with no input-output	Sum of 2 and 3 is 5
2	#include<stdio.h>	**Remarks:**
3	//Function declaration	• As the functions printsum and main do not return any value, void is specified as their return type
4	void printsum();	• The program now on compilation does not give any warning message
5	//main function, the master function	• **Some compilers (e.g. Borland TC 4.5) do not allow void to be specified as return type of the function main. They enforce the return type of the function main to be int**
6	void main()	**What to do?**
7	{	• If Borland TC 4.5 is used, either leave the return type of function main unspecified or specify it as int and place return 0; as the last statement of function main. 0 is an arbitrary value. Any integer value can be used instead of 0
8	printsum();	
9	}	
10	//Function definition	
11	void printsum()	
12	{	
13	printf("Sum of 2 and 3 is %d", 2+3);	
14	}	

Program 5-3 | A program that uses a void function

The important points about void functions are as follows:

1. A void function does not return any value. Either no return statement should be present inside the body of a void function or if it is present, it should be of the form return;. The return statement of the form[+++] return expression; cannot be used inside the body of a void function. When a return statement of the form return; is placed inside the body of a void function, its execution terminates the execution of the void function and returns the program control back to the calling function. The code snippet in Program 5-4 illustrates this fact.

Line	Trace	Prog 5-4.c	Output window
1		//Return statement inside void function	This is a void function
2		#include<stdio.h>	This is a statement before return statement
3		//Function declaration	**Remarks:**
4		void printsum();	• After the function call in line number 8 gets executed, the program control transfers to line number 10
5		//Function definitions	• This is depicted by trace arrows 2 and 3
6	1	void main()	• Execution of the function main is suspended and the printsum function starts execution
7		{	• After the execution of the return statement in line number 14, the program control returns back to the main function
8	2	printsum();	
9	7	}	
10	3	void printsum()	
11		{	
12	4	printf("This is a void function\n");	
13	5	printf("This is a statement before return statement\n");	
14	6	return;	

(Contd...)

[+++] Refer Section 5.3.1.1.3.3.1 for a description on various forms of return statement.

```
5
6
7
        printf("This is a statement after return statement\n");
        printf("Unreachable code\n");
    }
```

- This is depicted by trace arrows 6 and 7
- The execution of the function printsum is terminated and the main function resumes its execution
- printf statements in line numbers 15 and 16 remain unreachable

Program 5-4 | A program that illustrates the use of return statement inside void function

2. A void function call expression evaluates to void. Hence, such expressions cannot be placed on the right side of an assignment operator. For example, the expression a=printsum() is erroneous if printsum is a void function. The code snippet in Program 5-5 illustrates this fact.

Line	Prog 5-5.c	Output window
1	//void function call expression cannot be assigned to a variable	Compilation error "Not an allowed type in function main"
2	#include<stdio.h>	**Remarks:**
3	void printsum(void);	• The return type of the function printsum is void
4	void main(void)	• An expression of type void cannot be assigned to a variable
5	{	• Hence, the expression a=printsum() in line number 7 is erroneous
6	int a;	
7	a=printsum();	
8	printf("The value of a is %d",a);	
9	}	
10	void printsum(void)	
11	{	
12	printf("Sum of 2 and 3 is %d",2+3);	
13	}	

Program 5-5 | A program that illustrates the void function call expression, which cannot be assigned to a variable

Also, the keyword void is sometimes placed within parentheses in the function header to signify that the function does not have any input. This is depicted in the code snippet in Program 5-5.

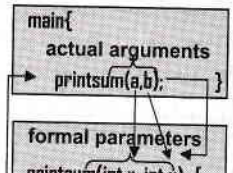
5.3.1.1.3.2 Function with Inputs and No Output

The function printsum developed in Program 5-2 is rigid. Each invocation of the function printsum prints the sum of 2 and 3. It cannot be used to print the sum of different values. The reason behind this rigidity of the printsum function is the lack of inputs to it. A function can be made flexible by adding inputs to it. The modified flexible form of the code listed in Program 5-2 is mentioned in Program 5-6.

The observable points about the code snippet given in Program 5-6 are as follows:

1. The printsum function developed in Program 5-6 is flexible as compared to the printsum function developed in Program 5-2. It can now be used to print the sum of any two integer values.
2. This flexibility is due to the added inputs. The printsum function now accepts two inputs of the integer type.

	Trace	Prog 5-6.c	(Column 4)	Output window
1		//Function with inputs and no output		Enter values of a & b 4 6
2		#include<stdio.h>		Sum of 4 and 6 is 10
3		//Function declaration		Enter values of a & b again 7 2
4		void printsum(int, int);		Sum of 7 and 2 is 9
5		//Function definitions		**Remarks:**
6	1	void main()		• Function printsum accepts two arguments, i.e. inputs
7		{		• In line number 11, a and b are known as **actual arguments**
8		int a,b;		• In line number 16, x and y are known as **formal parameters**
9	2	printf("Enter values of a & b\t");		• The parameters declared in the function header are like other local variables declared inside the body of a function
10	3	scanf("%d %d",&a,&b);		• After execution of the function call in line number 11, the values of a and b are copied into the variables x and y and the control is transferred to the function printsum
11	4	printsum(a,b);		
12	8	printf("Enter values of a & b again\t");		
13	9	scanf("%d %d",&a,&b);		
14	10	printsum(a,b);		
15	14	}		
16	5,11	void printsum(int x, int y)		
17		{		
18	6,12	printf("Sum of %d and %d is %d\n",x,y,x+y);		
19	7,13	}		



Program 5-6 | A program that uses a function with inputs

3. A function with inputs can be called in a similar way as a function without input is called, i.e. by using a function call operator. Inputs to a function are given by providing comma-separated expressions within the parentheses of the function call operator. For example, the printsum function defined in Program 5-6 can be called in the following ways:

```
printsum(2,3);          //←Inputs are constants 2 and 3
printsum(a,b);          //←Inputs are variables a and b
printsum(a+2,b-3);      //←Inputs are expressions a+2 and b-3
```

4. The expressions that appear within the parentheses of a function call are known as **actual arguments**, and the variables declared in the parameter list in the function header are known as **formal parameters**. For example, in Program 5-6, a and b are actual arguments of printsum function and x and y are the formal parameters.
5. The commas separating the actual arguments in a function call are not comma operators. If commas separating arguments in a function call are considered to be comma operators, then no function could have more than one argument. The commas appearing between the arguments in a function call are just separators.
6. The below-mentioned steps are followed when a function with inputs is invoked:
   a. The actual argument expressions are evaluated.
   b. The program control is transferred to the called function and the result of the evaluation of the actual argument expressions are assigned to the formal parameters on one-to-one basis as shown in column 4 of Program 5-6.

c. The execution of the calling function is suspended and the called function starts the execution.

7. When the execution of the called function (with no output) is complete, the program control returns to the calling function, and the calling function resumes its execution.

Consider the code snippet in Program 5-7, which has a more generalized form of printsum function defined in Program 5-6. The developed printsum function can now print the output in decimal, octal or hexadecimal number system according to the user's requirement.

Line	Prog 5-7.c	Output window
1	//Further generalization of printsum	Enter the values of a & b 2 10
2	#include<stdio.h>	Enter base of output(D, O or H) H
3	//Function printsum accepts three inputs	Sum of 2 and 10 in hexadecimal is C
4	void printsum(int, int, char);	**Remarks:**
5	//Function definition	• The flexibility of the function printsum is increased by providing an additional input, i.e. base
6	void main()	• If flushall➲ function is not used before the use of the scanf function, the scanf function might not prompt the user to enter a character
7	{	• The function flushall is used to flush, i.e. empty the streams➲ so that the scanf function prompts the user to enter a character
8	int a,b;	
9	char base;	
10	printf("Enter the values of a & b\t");	
11	scanf("%d %d",&a,&b);	
12	printf("Enter base of output(D, O or H)\t");	
13	flushall();	
14	scanf("%c",&base);	
15	printsum(a,b,base);	
16	}	
17	void printsum(int x, int y, char base)	
18	{	
19	if(base=='d'||base=='D')	
20	printf("Sum of %d and %d in decimal is %d",x,y,x+y);	
21	else if(base=='o'||base=='O')	
22	printf("Sum of %d and %d in octal is %o",x,y,x+y);	
23	else if(base=='h'||base=='H')	
24	printf("Sum of %d and %d in hexadecimal is %X",x,y,x+y);	
25	}	

Program 5-7 | A program that uses a more generalized form of the printsum function developed in Program 5-6

➲ **Forward Reference:** flushall() and streams (refer Question number 15 and its answer, Chapter 6).

5.3.1.1.3.3 Function with Inputs and One Output

The function printsum developed in Programs 5-6 and 5-7 receives inputs but does not return any value, rather it prints the result of the computation. However, the printing of the result of the computation by the called function is not always desired. The result of the computation

may be required in the calling function for further processing. The best software engineering practices suggest the following:

1. The developed functions should be kept as general as possible so that they can be used in different situations.
2. Functions should generally be coded without involving any direct I/O operation (i.e. direct use of I/O functions like printf, scanf, getch, etc.). A function should receive inputs in the form of arguments and return the result of computations instead of directly printing it.
3. A function should behave like a 'black box' that receives inputs, and outputs the desired value.

The result of the computations performed inside the called function is returned to the calling function by using the return statement.

5.3.1.1.3.3.1 return Statement

The **return statement** is used to return the result of the computations performed in the called function and/or to transfer the program control back to the calling function. There are two forms of the return statement:

1. return;
2. return expression;

The important points about the return statement are as follows:

1. **First form of the return statement, i.e. return;:**
   a. This form of the return statement is used when a function does not return any value (i.e. inside void functions).
   b. It cannot be used inside a function whose return type is not void.
   c. It terminates the execution of the called function and transfers the program control back to the calling function without returning any value.
2. **Second form of the return statement, i.e. return expression;:**
   a. This form of the return statement returns the function's result along with the program control back to the calling function.
   b. It cannot be placed inside the body of a void function and can only appear inside the body of a function whose return type is not void.
   c. The expression following the keyword return in the return statement is known as the **return expression**.
   d. The return expression can be an arbitrarily complex expression and can even have function calls. For example, in the statement return n*fact(n-1);, the return expression consists of a call to the function fact.
   e. The return expression is evaluated and the result of evaluation of the return expression is returned to the calling function along with the program control.
   f. If the return type of a function and the type of the result of evaluation of a return expression is not the same, the result of evaluation of the return expression is implicitly type casted to the return type of the function, if they are compatible. If they are incompatible, there will be a compilation error. Consider Program 5-8 that makes use of a function to compute the area of a circle.

Line	Prog 5-8.c
1	//Area of a circ
2	#include<stdio.h
3	//Function decl
4	circle_area(int)
5	//Function defi
6	void main()
7	{
8	int radius;
9	float area;
10	printf("Ente
11	scanf("%d"
12	area=circle
13	printf("Area
14	}
15	circle_area(int
16	{
17	return 3.14
18	}

Program 5-8 | A
is

The observable
i. The area
ii. The value
float value
int (as int i
return exp
above, th
type cast
expressio
an intege
and then
ment, i.e.
iii. The preci
tion circle

Line	Prog 5-9.c
1	//Area of a c
2	#include<std
3	//Function d
4	float circle_a
5	//Function d
6	void main()
7	{
8	int radius

Line	Prog 5-8.c
1	//Area of a circle
2	#include<stdio.h>
3	//Function declaration
4	circle_area(int);
5	//Function definitions
6	void main()
7	{
8	int radius;
9	float area;
10	printf("Enter the radius of circle\t");
11	scanf("%d",&radius);
12	area=circle_area(radius);
13	printf("Area of circle is %f\n",area);
14	}
15	circle_area(int radius)
16	{
17	return 3.1428*radius*radius;
18	}

Output window

```
Enter the radius of circle    2
Area of circle is 12.000000
```

Remarks:

- The area of circle that gets printed is 12.000000 instead of the actual value of 12.571200
- This happened because the return type of function circle_area is not mentioned. If the return type of a function is not mentioned, it is assumed to be int by default

Program 5-8 | A program illustrating that the specification of the return type is mandatory if the return type is other than int

The observable points about the code snippet in Program 5-8 are as follows:

i. The area of the circle printed is 12.000000 instead of the actual value 12.571200.

ii. The value of the area actually computed inside the function circle_area is 12.571200 (i.e. a float value) but since the return type of the function is not mentioned, it is assumed to be int (as int is the default return type of a function). The type of result of evaluation of the return expression is not the same as the return type of the function. Thus, as mentioned above, the result of evaluation of the return expression 3.1428*radius*radius, i.e. 12.571200 is type casted (i.e. demoted) to an integer value 12 before being returned. Hence, in the expression area=circle_area(radius), the sub-expression circle_area(radius) evaluates to 12. Since an integer value, i.e. 12 is assigned to a float variable area, it is firstly promoted to 12.000000 and then assigned. This value of area is then printed by the printf function in the next statement, i.e. line number 13.

iii. The precise value of an area can be obtained by specifying the return type of the function circle_area as float. This is shown in the code snippet given in Program 5-9.

Line	Prog 5-9.c
1	//Area of a circle
2	#include<stdio.h>
3	//Function declaration
4	float circle_area(int);
5	//Function definitions
6	void main()
7	{
8	int radius;

Output window

```
Enter the radius of circle    2
Area of circle is 12.571200
```

Remarks:

- float is specified as the return type of the function circle_area
- The value of area that gets printed is 12.571200 instead of 12.000000 (as printed in Program 5-8)

(Contd...)

Line	Prog 5-9.c	Output window
9	float area;	
10	printf("Enter the radius of circle\t");	
11	scanf("%d",&radius);	
12	area=circle_area(radius);	
13	printf("Area of circle is %f\n",area);	
14	}	
15	float circle_area(int radius)	
16	{	
17	return 3.1428*radius*radius;	
18	}	

Program 5-9 | A program that illustrates the effect of specification of the return type of a function

3. There is no constraint on the number of return statements that can be placed inside the body of a function. Although, a number of return statements can be placed inside the body of a function, only one of them that appears first in the logical flow of control gets executed. With the execution of this return statement, the program control returns to the calling function and the rest of the statements that appear after this return statement remain unreachable.
4. **'A function can return only one value.'** It is not possible to return more than one value by writing multiple return statements as mentioned in point 3 above or by writing return value1, value2, . . . valueN;. In this statement value1, value2, . . . valueN is the return expression, which is evaluated first and then its outcome is returned. The return expression consists of comma operators. The comma operator guarantees left-to-right evaluation and returns the result of the rightmost sub-expression. Hence, the expression value1, value2, . . . valueN evaluates to valueN and this value is returned. Program 5-10 illustrates this fact.

Line	Prog 5-10.c	Output window
1	//Attempt to return more than one value	Sum is 8
2	#include<stdio.h>	Difference is 8
3	//Function declaration	**Remarks:**
4	int sum_diff(int,int);	• In line number 16, an attempt is made to return values of sum and diff
5	//Function definitions	• However, the return statement can return only one value
6	void main()	• The return statement in line number 16 returns the value of diff, i.e. the value of the rightmost return sub-expression
7	{	
8	int a=10, b=2;	
9	printf("Sum is %d\n",sum_diff(a,b));	
10	printf("Difference is %d\n",sum_diff(a,b));	
11	}	
12	int sum_diff(int a,int b)	
13	{	
14	int sum=a+b;	
15	int diff=a-b;	
16	return sum,diff;	
17	}	

Program 5-10 | A program illustrating that the return statement cannot return more than one value

As we have seen, it is not possible to return more than one value (without making the use of structures➲) by making use of the return statement. However, it is possible to indirectly return more than one value to the calling function. This indirect method of returning more than one value to the calling function is discussed in the next section.

Forward Reference: Structures (Chapter 9).

5.3.1.1.3.4 Function with Inputs and Outputs

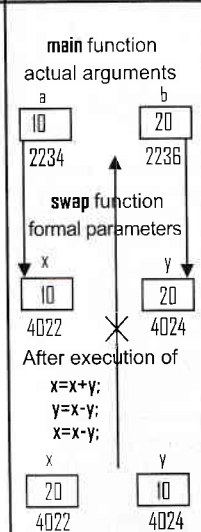
More than one value can be indirectly returned to the calling function by making the use of pointers. In fact, the pointers can also be used to pass arguments to a function. Depending upon whether the values or addresses (i.e. pointers) are passed as arguments to a function, the argument passing methods in C language are classified as:

1. Pass by value
2. Pass by address

5.3.1.1.3.4.1 Passing Arguments by Value

The method of passing arguments by value is also known as **call by value.** In this method, the values of actual arguments are copied to the formal parameters of the function. If the arguments are passed by value, the changes made in the values of formal parameters inside the called function are not reflected back to the calling function. The code snippet listed in Program 5-11 illustrates this concept.

Line	Prog 5-11.c
1	//Use of pass by value in swap function
2	#include<stdio.h>
3	//Function declaration
4	void swap(int,int);
5	//Function definitions
6	void main()
7	{
8	int a=10,b=20;
9	printf("Before swap values are %d %d\n",a,b);
10	swap(a,b);
11	printf("After swap values are %d %d\n",a,b);
12	}
13	void swap(int x, int y)
14	{
15	x=x+y;
16	y=x-y;
17	x=x-y;
18	printf("In swap function values are %d %d\n",x,y);
19	}



Output window

Before swap values are 10 20
In swap function values are 20 10
After swap values are 10 20

Remarks:

- On the execution of the function call, i.e. swap(a,b);, the values of actual arguments a and b are copied into the formal parameters x and y
- Formal parameters are allocated at separate memory locations
- A change made in the formal parameters is independent of the actual arguments
- On returning from the called function, the formal parameters are destroyed and the access to the actual arguments gives values that are unchanged

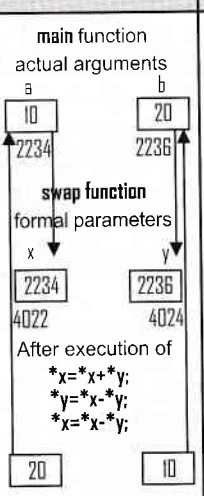
Program 5-11 | A program that illustrates pass by value

Analogy: The reason why the changes made in the formal parameters in the called function are not reflected back to the calling function can be understood by looking at this analogy. The main function, i.e. the master function wants to get some changes done in a file from its subordinate worker, i.e. the swap function. The main function got the file (i.e. actual arguments) Xeroxed and has handed over the Xeroxed copy of the file (i.e. formal parameters) to the swap function for changes. The swap function has made changes in the Xeroxed copy and has returned the file back to the main function. On getting the control back, the main function is still referring to the original file and finds that no changes have been made in it. The changes have been made in the Xeroxed copy, so how can the main function find changes in the original file?

5.3.1.1.3.4.2 Passing Arguments by Address/Reference

The method of passing arguments by address or reference is also known as **call by address** or **call by reference**. In this method, the addresses of the actual arguments are passed to the formal parameters of the function. If the arguments are passed by reference, the changes made in the values pointed to by the formal parameters in the called function are reflected back in the calling function. The code snippet listed in Program 5-12 illustrates this concept.

Line	Prog 5-12.c
1	//Use of pass by reference in swap function
2	#include<stdio.h>
3	//Function declaration
4	int swap(int*,int*);
5	//Function definitions
6	void main()
7	{
8	int a=10,b=20;
9	printf("Before swap values are %d %d\n",a,b);
10	swap(&a,&b);
11	printf("After swap values are %d %d\n",a,b);
12	}
13	int swap(int *x, int *y)
14	{
15	*x=*x+*y;
16	*y=*x-*y;
17	*x=*x-*y;
18	printf("In swap function values are %d %d\n",*x,*y);
19	}



Output window

Before swap values are 10 20
In swap function values are 20 10
After swap values are 20 10

Remarks:

- Addresses of the actual arguments are passed instead of their values
- Changes made in the called function are actually done in the memory locations of the actual arguments
- On returning from the called function, the formal parameters are destroyed but since the changes were made at the memory locations of the actual arguments, they can still be found there

Program 5-12 | A program that illustrates pass by reference

Analogy: The reason why the changes made in the called function are reflected back to the calling function can be understood by looking at this analogy. The main function, i.e. the master function wants to get some changes done in a file from its subordinate worker, i.e. the swap function. The main function has kept the file (i.e. actual arguments) in a file cabinet (i.e. memory). The main function tells the swap function the changes to be made and the location of the file in

the cabinet (i.e. the memory address). The swap function opens up the file cabinet, locates the file, makes changes in it, places it back at the same position in the cabinet and reports to the main function that the work has been done. On getting the control back, the main function opens up the file cabinet, looks at the file and finds the changes made in it.

5.3.1.1.3.4.3 Returning More Than One Value Indirectly

Consider the code listed in Program 5-10, where we tried to return more than one value by making the use of the return statement and failed. I will now illustrate how to return more than one value to the calling function indirectly by making the use of a call by reference. In the code snippet listed in Program 5-13, the called function indirectly returns more than one value to the calling function.

Line	Prog 5-13.c	Output window
1	//Indirectly returning more than one value	Sum is 12 Difference is 8 **Remarks:** • Mixed method of passing arguments is used • Two arguments, i.e. a and b are passed by value • Other two arguments, i.e. sum and diff are passed by reference • The results of the computations made in the called function are stored in the memory locations of the actual arguments (i.e. sum and diff) by making the use of passed addresses • Actually, sum_diff function does not return any value
2	#include<stdio.h>	
3	//Function declaration	
4	void sum_diff(int,int,int*,int*);	
5	//Function definitions	
6	void main()	
7	{	
8	int a=10, b=2;	
9	int sum, diff;	
10	sum_diff(a,b,&sum,&diff);	
11	printf("Sum is %d\n",sum);	
12	printf("Difference is %d\n",diff);	
13	}	
14	void sum_diff(int a,int b, int*sum, int*diff)	
15	{	
16	*sum=a+b;	
17	*diff=a-b;	
18	}	

Program 5-13 | A program that illustrates the method to indirectly return more than one value by making the use of pass by reference

5.3.1.1.4 Passing Arrays to Functions

Like simple variables, arrays can also be passed to functions. There are two ways to pass arrays to functions:

1. Passing individual elements of an array one by one
2. Passing an entire array at a time

Passing individual elements of an array one by one is similar to passing basic variables. The individual elements of an array can be passed either by value or by reference. However, this way of passing an array is not preferred due to the following reasons:

1. If the number of elements in an array is large, passing the entire array will take a large number of function calls, as one element is passed with each function call. As the function calls are time consuming, this method of passing an array to a function will deteriorate the performance of a program.

2. When the individual elements of an array are passed to the function one by one, the complete array will never be available to the called function for processing at a time. The called function will always have a piecemeal array.

The code segments listed in Program 5-14 illustrate the passing of array elements one by one.

Line	Prog 5-14a.c	Prog 5-14b.c	Output window
1	//Individual elements of array passed	//Individual elements of array passed	Enter the no. of elements 5
2	//by value	//by reference	Enter elements of array
3	#include<stdio.h>	#include<stdio.h>	2
4	int sum_array(int,int);	int sum_array(int*,int);	4
5	void main()	void main()	5
6	{	{	7
7	int arr[10], nele, lc, sum=0;	int arr[10], nele, lc, sum=0;	1
8	printf("Enter the no. of elements\t");	printf("Enter the no. of elements\t");	Sum is 19
9	scanf("%d",&nele);	scanf("%d",&nele);	**Remarks:**
10	printf("Enter elements of array\n");	printf("Enter elements of array\n");	• Iteration is used to pass the elements of the array one by one
11	for(lc=0;lc<nele;lc++)	for(lc=0;lc<nele;lc++)	• Number of iterations required to pass n elements of an array to a function is n
12	scanf("%d",&arr[lc]);	scanf("%d",&arr[lc]);	
13	for(lc=0;lc<nele;lc++)	for(lc=0;lc<nele;lc++)	
14	{	{	
15	sum=sum_array(arr[lc],sum);	sum=sum_array(&arr[lc],sum);	
16	}	}	
17	printf("Sum is %d",sum);	printf("Sum is %d",sum);	
18	}	}	
19	int sum_array(int element, int sum)	int sum_array(int* element, int sum)	
20	{	{	
21	return sum+element;	return sum+*element;	
22	}	}	

Program 5-14 | A program that illustrates the passing of an array element by element

Passing entire array at a time is a preferred way of passing arrays to functions. The entire array is always passed by reference.

The following sections describe the passing of one-dimensional and multi-dimensional arrays to functions.

5.3.1.1.4.1 Passing One-dimensional Arrays to Functions

The syntactic rules to pass one-dimensional arrays to a function are as follows:

1. The actual argument in the function call should only be the name of the array without any subscript.
2. The corresponding formal parameter in the function definition must be of array type or pointer type (i.e. pointer to the first element of the array). If a formal parameter is of array type, it will be implicitly converted to pointer type.
3. The corresponding parameter type in the function declaration should be of array type or pointer type.

The code snippet mentioned in Program 5-15 illustrates the different ways of passing a one-dimensional array to a function.

Line	Prog 5-15a.c (Column 2)	Prog 5-15b.c (Column 3)	Output window
1	//Passing 1-D array	//Passing 1-D array	Enter the no. of elements 5
2	#include<stdio.h>	#include<stdio.h>	Enter elements of array
3	void find_max_min(int[],int);	void find_max_min(int*,int);	2
4	void main()	void main()	4
5	{	{	5
6	int arr[10], nele, lc, sum=0;	int arr[10], nele, lc, sum=0;	7
7	printf("Enter the no. of elements\t");	printf("Enter the no. of elements\t");	1
8	scanf("%d",&nele);	scanf("%d",&nele);	Max is 7
9	printf("Enter elements of array\n");	printf("Enter elements of array\n");	Min is 1
10	for(lc=0;lc<nele;lc++)	for(lc=0;lc<nele;lc++)	**Remarks:**
11	scanf("%d",&arr[lc]);	scanf("%d",&arr[lc]);	• Passing the entire array at a time is an efficient way of passing a number of values to a function
12	find_max_min(arr, nele);	find_max_min(arr, nele);	• In column 2, in line number 16, the declared formal parameter arr is of array type
13	printf("Max is %d\n",arr[0]);	printf("Max is %d\n",arr[0]);	• It will be implicitly converted to pointer type
14	printf("Min is %d\n",arr[1]);	printf("Min is %d\n",arr[1]);	• Hence the declaration of arr made in line number 16 in column 2 will be converted to the declaration of arr made in line number 16 in column 3
15	}	}	• The two declarations of arr are equivalent
16	void find_max_min(int arr[], int nele)	void find_max_min(int* arr, int nele)	
17	{	{	
18	int lc, max=arr[0], min=arr[0];	int lc, max=arr[0], min=arr[0];	
19	for(lc=1;lc<nele;lc++)	for(lc=1;lc<nele;lc++)	
20	if(arr[lc]>max)	if(arr[lc]>max)	
21	max=arr[lc];	max=arr[lc];	
22	else if(arr[lc]<min)	else if(arr[lc]<min)	
23	min=arr[lc];	min=arr[lc];	
24	arr[0]=max; arr[1]=min;	arr[0]=max; arr[1]=min;	
25	}	}	

Program 5-15 | A program that illustrates the method of passing a one-dimensional array to a function

5.3.1.1.4.2 Passing Two-dimensional Arrays to Functions

The syntactic rules to pass two-dimensional arrays to a function are as follows:

1. The actual argument in the function call should be the name of an array.
2. The corresponding formal parameter in the function definition must be of array type or pointer type (i.e. pointer to the first element of the array).
    a. If the formal parameter is of array type, it is mandatory to specify the column specifier. In general, in case of n-D arrays, if the formal parameter is of array type, it is mandatory to specify (n-1) fastest varying specifiers.
    b. If the formal parameter is of pointer type, it must be a pointer to an element of the two-dimensional array (i.e. one-dimensional array having the number of columns same as the number of columns specified for the two-dimensional array). In general, for n-D arrays, if the formal parameter is of pointer type, it must be a pointer to (n-1)-D array having the size specifications same as the (n-1) fastest varying size specifications for the n-D array.

3. The corresponding parameter type in the function declaration should be a matching array type or pointer type.

The code snippet in Program 5-16 illustrates the passing of a two-dimensional array to a function.

Line	Prog 5-16.c
1	//Passing 2-D array
2	#include<stdio.h>
3	void largest_ele(int[][10],int*,int*);
4	void main()
5	{
6	int arr[10][10];
7	int rows, cols, rc, cc;
8	printf("Enter no. of rows in array(<10)\t");
9	scanf("%d",&rows);
10	printf("Enter no. of cols in array(<10)\t");
11	scanf("%d",&cols);
12	printf("Enter elements of array:\n");
13	for(rc=0;rc<rows;rc++)
14	for(cc=0;cc<cols;cc++)
15	scanf("%d",&arr[rc][cc]);
16	largest_ele(arr,&rows,&cols);
17	printf("Largest element is %d\n",arr[rows][cols]);
18	printf("Located in row no. %d\n",rows);
19	printf("Located in column no. %d\n",cols);
20	}
21	void largest_ele(int arr[][10],int *rows, int *cols)
22	{
23	int row=0, col=0, rc=0, cc=0, max=arr[0][0];
24	for(rc=0; rc<*rows;rc++)
25	for(cc=0;cc<*cols;cc++)
26	if(arr[rc][cc]>max)
27	{
28	max=arr[rc][cc];
29	row=rc; col=cc;
30	}
31	*rows=row; *cols=col;
32	}

Output window

```
Enter no. of rows in array(<10)    3
Enter no. of cols in array(<10)    3
Enter elements of array:
8 4 6
7 9 3
2 1 5
Largest element is 9
Located in row no. 1
Located in column no. 1
```

Remarks:

- In line number 21, the declared formal parameter is of array type
- It will be implicitly converted to pointer type
- The equivalent declaration is int(*)[10], i.e. pointer to one-dimensional array of 10 integers
- It is assumed that the row and column number starts with 0

Program 5-16 | A program to illustrate the method of passing of a two-dimensional array to a function

5.3.1.1.5 Default Arguments

In Section 5.3.1.1.3.2, we have seen how functions can be made flexible by adding inputs to them. Each input adds some flexibility to the function and makes the function more general. However, some inputs are the same in majority of the cases and have special values only in rare circumstances. For example, in Program 5-7, the common base input to the function printsum is '0', i.e. decimal number system. In rare circumstances, the user wants the output to be in an octal number system or a hexadecimal number system. These general functions are sometimes unwieldy as the values are to be supplied for each argument.

The C language frees the programmer from this difficulty by providing the concept of **default arguments**. A **default argument** is a value that is an appropriate argument value for a parameter in majority of the cases. Consider the code snippet in Program 5-17 that makes the use of default argument for base input in printsum function discussed in Program 5-7.

Line	Prog 5-17.c	Output window
1	//Default arguments	Use of default arguments:
2	#include<stdio.h>	General conditions:
3	//Function declaration	Sum of 5 and 6 in decimal is 11
4	void printsum(int, int, char base='D');	Sum of 3 and 4 in decimal is 7
5	//Function definition	Rare conditions:
6	void main()	Sum of 6 and 9 in hexadecimal is F
7	{	Sum of 6 and 9 in octal is 17
8	printf("Use of default arguments:\n");	**Remarks:**
9	printf("General conditions:\n");	• In line number 4, the parameter base is initialized with the value 'D'
10	printsum(5,6);	• This initialization makes 'D' as default argument for the parameter base
11	printsum(3,4);	• A function that provides a default argument for a parameter can be invoked with or without an argument for this parameter
12	printf("Rare conditions:\n");	• In line numbers 10 and 11, the function printsum is invoked without specifying an argument for the parameter base
13	printsum(6,9,'H');	• In line numbers 13 and 14, arguments 'H' and 'O', respectively, are specified as arguments for the parameter base. These values override the default argument value 'D'
14	printsum(6,9,'O');	• Borland Turbo C 3.0 IDE does not support the use of default arguments
15	}	
16	void printsum(int x, int y, char base)	
17	{	
18	if(base=='d'||base=='D')	
19	printf("Sum of %d and %d in decimal is %d\n",x,y,x+y);	
20	else if(base=='o'||base=='O')	
21	printf("Sum of %d and %d in octal is %o\n",x,y,x+y);	
22	else if(base=='h'||base=='H')	
23	printf("Sum of %d and %d in hexadecimal is %X\n",x,y,x+y);	
24	}	

Program 5-17 | A program that illustrates the use of default arguments

The important points about the default arguments are as follows:

1. The arguments can be made default by using initialization syntax within the parameter list during the function declaration. For example, in line number 4 in Program 5-17, the parameter base has been made default by initializing it with 'D'.
2. A function that provides a default argument for a parameter can be invoked with or without an argument for this parameter.
3. However, if an argument is provided, it overrides the default argument value.
4. A function declaration can specify default arguments for all or for a subset of parameters. If the default arguments are specified only for a subset of parameters, then these parameters should be kept on the trailing side. The code snippet in Program 5-18 illustrates this fact.

Line	Prog 5-18.c	Output window
1	//Default arguments for a subset of parameters	Compilation errors "Default value missing following parameter b", "Too few parameters in call to 'add(int, int, int)' in function main" **Remark:** • In line number 4, the default argument for the parameter b cannot be specified unless and until the default argument for parameter c is specified **What to do?** • Either specify the default argument for the parameter c or remove the default argument value for the parameter b
2	#include<stdio.h>	
3	//Function declaration	
4	int add(int a, int b=12, int c);	
5	//Function definitions	
6	void main()	
7	{	
8	add(10,12);	
9	}	
10	int add(int a, int b, int c)	
11	{	
12	printf("The result after addition is %d\n",a+b+c);	
13	}	

Program 5-18 | A program illustrating that the specification of default arguments for a subset of parameters

The code snippet in Program 5-19 is the rectified version of the code listed in Program 5-18.

Line	Prog 5-19.c	Output window
1	//Default arguments can be specified for parameters that lie on the	The result after addition is 30 The result after addition is 19 **Remarks:** • In line number 4, the default arguments are specified for two trailing parameters b and c • Since, no default argument is specified for the parameter a, at least one argument is required to invoke the function add • In line number 8, the argument value overrides the default argument value for the parameter b
2	//trailing side of the parameter list	
3	#include<stdio.h>	
4	int add(int a, int b=12, int c=8);	
5	void main()	
6	{	
7	add(10);	
8	add(10,1);	
9	}	
10	int add(int a, int b, int c)	
11	{	
12	printf("The result after addition is %d\n",a+b+c);	
13	}	

Program 5-19 | A program illustrating that the default arguments can be specified for the parameters that lie on the trailing side of the parameter list

5. The default argument should not be specified in the function definition. If the default argument is provided in the parameter list of function definition as well, there will be 'Default argument value redeclared error.' The code snippet in Program 5-20 illustrates this fact.
6. It is not mandatory to have a default argument as a constant expression. Any expression can be used as the default argument. When the default argument is an expression, the expression is evaluated when the function is called. The code snippet in Program 5-21 illustrates this fact.

Line	Prog 5-20.c	Output window
1	//Redeclaration of default arguments	Compilation error "Default argument value redeclared"
2	#include<stdio.h>	**Remark:**
3	//Function declaration along with the specification of the default	• The default arguments are specified in the function declaration, they should not be re-specified in the header of the function definition
4	//arguments	**What to do?**
5	int add(int a=12, int b=8);	• Remove default argument values from the header of the function definition
6	void main()	
7	{	
8	add();	
9	add(10);	
10	add(10,12);	
11	}	
12	//Function definition with re-specification of the default arguments	
13	int add(int a=12, int b=8)	
14	{	
15	printf("The result after addition is %d\n",a+b);	
	}	

Program 5-20 | A program illustrating that the default arguments should not be re-declared in the header of the function definition

Line	Prog 5-21.c	Output window
1	//Use of an expression as default argument	The result after addition is 14
2	#include<stdio.h>	The result after addition is 12
3	//Function declarations	The result after addition is 22
4	int sub(int,int);	**Remarks:**
5	int add(int a=12,int b=sub(3,1));	• In line number 5, the default argument for the parameter b is an expression sub(3,1)
6	//Function definitions	• Carefully note the order of declaration of function sub and function add
7	void main()	• Before specifying sub as the default argument, it should be either declared or defined
8	{	• Change the order of declaration of function sub and function add, i.e. interchange the contents of line numbers 4 and 5 and observe the result of compilation
9	add();	
10	add(10);	
11	add(10,12);	
12	}	
13	int add(int a, int b)	
14	{	
15	printf("The result after addition is %d\n",a+b);	
16	}	
17	int sub(int a, int b)	
18	{	
19	return a-b;	
20	}	

Program 5-21 | A program that illustrates the use of an expression as the default argument

5.3.1.1.6 Command Line Arguments

We have seen that arguments are given to the functions to increase their flexibility. Since main is also a function, can we give arguments to the function main? The answer to this question is YES! The main function can also accept arguments. The arguments to a called function are supplied from the calling function. However, main is the first function that gets invoked at the program startup. Therefore,

how are arguments supplied to the function main? The arguments to the function main are supplied from **command line** and thus, have a special name known as **command line arguments.**

Forward Reference: Command line arguments (Chapter 6).

5.3.1.1.7 Recursion

Recursion is a powerful programming technique that can be used to solve the problems that can be expressed in terms of similar problems of smaller size. For example, consider a problem to find the factorial of a number n. The problem of finding the factorial of n can be expressed in terms of a similar problem of smaller size as n!=n×(n-1)!. Recursion provides an elegant way of solving such problems.

In recursive programming, a function calls itself. A function that calls itself is known as a **recursive function,** and the phenomenon is known as **recursion**. Recursion is classified according to the following criteria:

1. Whether the function calls itself directly (i.e. **direct recursion**) or indirectly (i.e. **indirect recursion**).
2. Whether there is any pending operation on return from a recursive call. If the recursive call is the last operation of a function, the recursion is known as **tail recursion**.
3. Pattern of recursive calls. According to the pattern of recursive calls, recursion is classified as:
    a. Linear recursion
    b. Binary recursion
    c. n-ary recursion

5.3.1.1.7.1 Direct and Indirect Recursion

A function is **directly recursive** if it calls itself, i.e. the function body contains an explicit call to itself. **Indirect recursion** occurs when a function calls another function, which in turn calls another function, eventually resulting in the original function being called again. The functions involved in indirect recursion are known as **mutually recursive functions**. Figure 5.2 illustrates direct and indirect recursion.

Direct recursion		Indirect recursion	
A()	//←Direct recursive function	A()	//←Mutually recursive function A
{		{	
----------	//←Statements	----------	//←Statements
----------		B();	//←Function A calls function B
A();	//←Call to itself	----------	
----------		}	
----------		B()	//←Mutually recursive function B
}		{	
		----------	//←Statements
		A();	//←Function B calls function A

		}	

Figure 5.2 | Direct and indirect recursion

Direct recursive functions are simpler and more elegant as compared to indirectly recursive functions and are most commonly used. The code snippet in Program 5-22 illustrates the use of recursion to find the factorial of a number.

Line	Trace	Prog 5-22.c	Output window
1		//Recursion to find the factorial of a number	Enter the number 3 Factorial of 3 is 6 **Remarks:** • The body of the function fact contains call to itself • Thus, fact is a directly recursive function • Though recursion is very powerful and highly expressive, it is hard to visualize • Trace the program and carefully observe the execution of function calls • Trace arrows in column 2 depicts the order of execution of statements
2		#include<stdio.h>	
3		//Function declaration	
4		int fact(int);	
5		//Function definitions	
6	1	void main()	
7		{	
8		int no, factorial;	
9	2	printf("Enter the number\t");	
10	3	scanf("%d",&no);	
11	4	factorial=fact(no);	
12	15	printf("Factorial of %d is %d", no, factorial);	
13	16	}	
14		//Definition of directly recursive function fact	
15	5,8,11	int fact(int no)	
16		{	
17	6,9,12	if(no==1)	
18	13	return 1;	
19	6,12	else	
20	7,10	return no*fact(no-1);	
21	14	}	

Program 5-22 | A program that makes the use of a recursive function to find the factorial of a number

The important points about how to develop recursive functions are as follows:

1. Thinking recursively is the first step to solve a problem using recursion.
2. Every recursive solution consists of two cases:
   a. **Base case:** Base case is the smallest instance of problem, which can be easily solved and there is no need to further express the problem in terms of itself, i.e. in this case no recursive call is given and the recursion terminates. Base case forms the **terminating condition** of the recursion. There may be more than one base case in a recursive solution. Without the base case, the recursion will never terminate and will be known as **infinite recursion.** For example, no==1 is the base case of the recursive function fact listed in Program 5-22.
   b. **Recursive case:** In a recursive case, the problem is defined in terms of itself, while reducing the problem size. For example, when fact(n) is expressed as n×fact(n-1), the size of the problem is reduced from n to n-1.
3. Express the solution in the form of base cases and recursive cases. For example, the factorial problem can be expressed as:

$$\text{fact(no)} = \begin{cases} 1 & \text{when no} = 1 \\ \text{no} \times \text{fact(no} - 1) & \text{when no} > 1 \end{cases}$$

Relation of the above form is known as **recurrence relation.**

b. Incorporate the pending operation into the auxiliary parameter in such a way that the non-tail recursive function no longer has a pending operation. For example, the pending operation of multiplication is incorporated into the auxiliary parameter result as no*result.

Consider another application of recursion in finding the terms of a Fibonacci series. In the Fibonacci series, every value is the sum of previous two values. The first two values of the Fibonacci series are 0 and 1. The values 0 1 1 2 3 5 8 13 21 ... form the Fibonacci series. The recurrence relation for finding any term in Fibonacci series is:

$$fib(n) = \begin{cases} 0 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ fib(n-1) + fib(n-2) & \text{for } n > 2 \end{cases}$$

Program 5-24a lists the code that uses a non-tail recursive function fib_norm to find a Fibonacci term. The conversion of a non-tail recursive function to a tail recursive function is done in Program 5-24b.

Line	Prog 5-24a.c	Prog 5-24b.c	Output window
1	//Non-tail recursive Fibonacci function	//Tail recursive Fibonacci function	Enter term no. 4
2	#include<stdio.h>	#include<stdio.h>	Fibonacci term is 2
3	//Function declaration	//Function declaration	**Remarks:**
4	int fib_norm(int);	int fib_tail(int,int,int);	• fib_norm is a non-tail recursive function as the last operation to be performed in function fib_norm is addition instead of being a recursive call
5	void main()	void main()	• fib_norm has two base cases, i.e. when n=1 and when n=2
6	{	{	• fib_tail is the corresponding tail recursive version
7	int n, term;	int n, term;	• Two auxiliary parameters, i.e. next and result are used
8	printf("Enter term no.\t");	printf("Enter term no.\t");	• The pending addition operation in fib_norm is incorporated in the auxiliary parameter of fib_tail as next+result
9	scanf("%d",&n);	scanf("%d",&n);	
10	term=fib_norm(n);	term=fib_tail(n,1,0);	
11	printf("Fibonacci term is %d",term);	printf("Fibonacci term is %d",term);	
12	}	}	
13	//Non-tail recursive function fib_norm	//Tail recursive version of fib_norm	
14	int fib_norm(int n)	int fib_tail(int n,int next, int result)	
15	{	{	
16			
17	if(n==1) return 0;	if(n==1) return result;	
18	if(n==2) return 1;	return fib_tail(n-1, next+result, next);	
19	return fib_norm(n-1)+fib_norm(n-2);	}	
20	}		

Program 5-24 | Non-tail recursive and tail recursive functions to find a Fibonacci term

4. Tail recursive functions can be easily transformed into iterative functions to improve the efficiency of a program.

5.3.1.7.3 Pattern of Recursive Calls

Based upon the number of recursive calls within a function, the recursion is classified as:

1. Linear recursion
2. Binary recursion
3. n-ary recursion

5.3.1.1.7.3.1 Linear Recursion

The simplest form of recursion is **linear recursion**. A linearly recursive function makes only one recursive call. The function fact discussed in Program 5-22 is a linearly recursive function, as there is only one recursive call within its body. The next section describes how recursion works and how function calls form a linear structure.

5.3.1.1.7.3.1.1 How Recursion Works

Consider the code listed in Program 5-22. Figure 5.3 shows how recursion works to compute the value of factorial of 4.

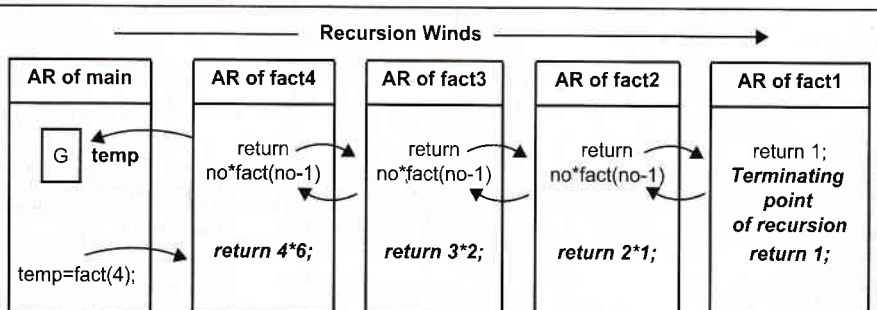


Figure 5.3 | Winding and unwinding of linear recursion

i G (in the above figure) signifies garbage value of local variable temp and **AR** stands for activation record.

The function main gives a call to the function fact with 4 as an argument. Execution of this call creates an activation record for the function fact. The activation record of the function main is packed, placed on the run-time stack, and the activation record of the function fact becomes live. The value of no in the live activation record is 4. Since no≠1 in the current activation, the statement return no*fact(no-1); gets executed. The return expression itself contains a call to the function fact with 3 as an argument. The execution of this function call packs the current activation record of fact, places it onto the run-time stack and creates a new activation record with the value of no as 3 and makes it live. The same process is repeated till the activation record with the value of no as 1 gets created. This part of recursion in which a number of activation records are created and piled up on the run-time stack is known as **winding of recursion**. During the winding of recursion, new activation records keep on getting created. As each activation record requires some

memory space, the memory requirement of a program increases during the winding of recursion. If there is no spare memory space for creating the new activation records, the recursion terminates abnormally.

When memory space is available, the winding of recursion terminates when the terminating condition of recursion is reached. In the code snippet listed in Program 5-22, the recursion terminates when the value of `n` becomes 1. From this point onwards, the recursion starts **unwinding**. During the unwinding process, the called activation returns a value to its calling activation. After returning the value, the activation record of the called activation is destroyed and the memory occupied by it is freed. As shown in Figure 5.3, the last activation returns 1 to the second last activation, which in turn returns 2 to the third last activation and so on. In this way the first activation of the function `fact` returns 24 to the function `main`.

The term **activation** means execution of a function. If a function is executing, it is said to be **active**. In a C program, multiple functions can be active at the same time. For example, suppose function `main` calls a function `fun1`, which in turn calls another function `fun2`. While the function `fun2` is executing, the functions `main`, `fun1` and `fun2` are all **active**. When the function `fun2` completes its execution and returns the program control to the function `fun1`, only the functions `main` and `fun1` remain active and the function `fun2` becomes **inactive**.

Activation of each function requires a separate **activation record**. An activation record refers to the chunk of memory, which holds the following:

1. **Dynamic link:** It points to the activation record of the caller.
2. **Saved state:** It refers to the contents of the program counter and registers when the function is called. It is used to restore the context of the caller function when the program control returns.
3. **Parameters:** They refer to the memory space required by the parameters declared within the header of the function.
4. **Local variables:** They refer to the memory space required by the automatic local variables.
5. **Temporary storage:** It refers to the storage used for evaluating the expressions.

Dynamic link
Saved state
Parameters
Local variable
Temporary storage

An activation record is automatically created when a function starts the execution and is automatically destroyed when a function returns the control to its caller. The activation records for all of the active functions are stored in the region of memory called the **stack**.

Forward Reference: Automatic and local variables (Chapter 7).

5.3.1.7.3.2 Binary Recursion

A **binary recursive function** calls itself twice. The `fib_norm` function listed in Program 5-24a is a binary recursive function. In the binary recursion, the tree of recursive calls is a **binary** tree. Figure 5.4 depicts the tree of recursive calls for `fib_norm(3)`.

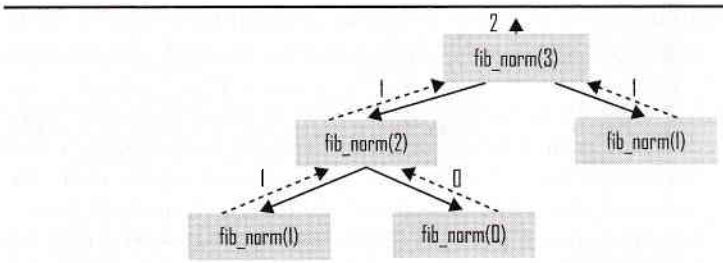


Figure 5.4 | Tree of recursive calls to the function fib_norm

Binary recursion is used in solving some of the important computing problems like:

1. Tower of Hanoi problem
2. Sorting by merge sort
3. Searching by binary search
4. Fibonacci series generation, etc.

Binary tree is a non-linear data structure in which every node of a tree can have at most two children. The tree shown in Figure 5.4 is a binary tree.

5.3.1.1.7.3.2.1 Tower of Hanoi Problem

Tower of Hanoi is one of the classical problems of computer science. The problem states that:

1. There are three stands (Stands 1, 2 and 3) on which a set of disks, each with a different diameter, are placed.
2. Initially, the disks are stacked on Stand 1, in order of size, with the largest disk at the bottom.

The initial structure of Tower of Hanoi with three disks is shown in Figure 5.5.

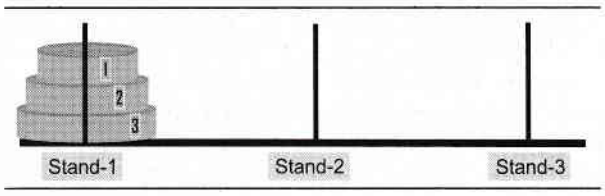


Figure 5.5 | Tower of Hanoi with three disks

The **'Tower of Hanoi problem'** is to find a sequence of disk moves so that all the disks are moved from Stand-1 to Stand-3, adhering to the following rules:

1. Move only one disk at a time.
2. A larger disk cannot be placed on top of a smaller disk.
3. All disks except the one being moved should be on a stand.

'Tower of Hanoi' is tough and computationally expensive. However, the expressive power of recursion can be used to easily formulate a solution to this problem. The general strategy for solving the Tower of Hanoi problem with n disks is shown in Figure 5.6.

1. Move the topmost n-1 disks from Stand-1 to Stand-2.
2. Move the largest disk from Stand-1 to Stand-3.
3. Move n-1 disks from Stand-2 to Stand-3.
4. Final structure.

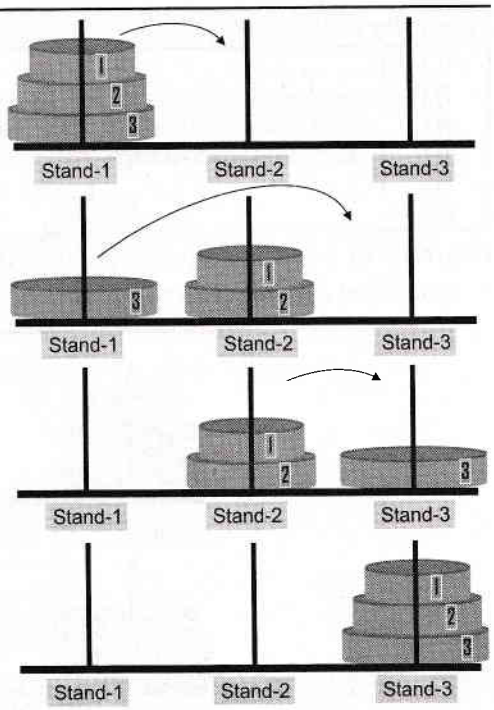


Figure 5.6 | General strategy to solve the Tower of Hanoi problem with three disks

The movement of n-1 disks forms the recursive case of a recursive solution to move n disks. The base case of a solution involves the movement of only one disk. The recurrence relation for solving the Tower of Hanoi problem can be written as:

$$\text{TowerOfHanoi(disks)} = \begin{cases} \text{move the disk} & \text{if disks} = 1 \\ \text{TowerOfHanoi(disks} - 1) & \text{if disks} > 1 \end{cases}$$

The code snippet listed in Program 5-25 solves the Tower of Hanoi problem.

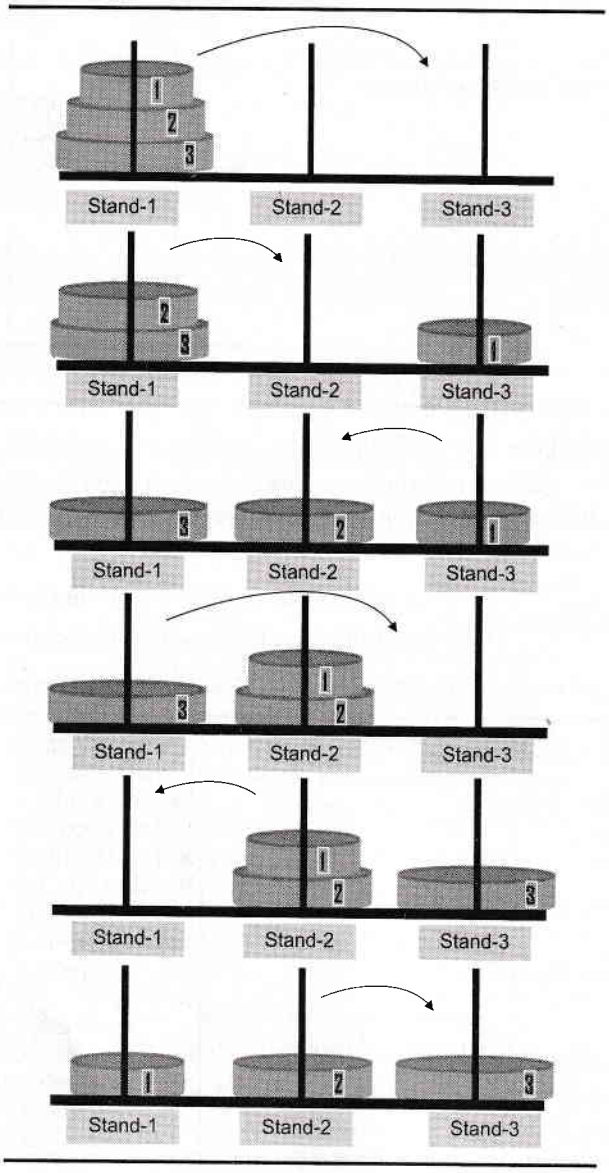
Line	Prog 5-25.c	Output window
	#include<stdio.h>	Follow these moves:
2	//Function declaration	Move disk 1 from 1 to 3
3	void move(int,int,int,int);	Move disk 2 from 1 to 2
4	//Function definitions	Move disk 1 from 3 to 2
5	void main()	Move disk 3 from 1 to 3
6	{	Move disk 1 from 2 to 1
7	int disks=3;	Move disk 2 from 2 to 3
8	printf("Follow these moves:\n");	Move disk 1 from 1 to 3
9	move(disks,1,3,2);	**Remarks:**
10	}	• Line number 15 codes step 1 of the general solution shown in Figure 5.6
11	void move(int count,int start,int finish,int temp)	• Line number 16 is the base case and codes step 2 of the general solution shown in Figure 5.6
12	{	
13	if(count>0)	

(Contd...)

Line	Prog 5-25.c	Output window
14	{	• Line number 17 codes the step 3 of the general solution shown in Figure 5.6
15	move(count-1,start,temp,finish);	• How disks will be actually moved can be seen by tracing the program and keeping track of argument values to the recursive calls
16	printf("Move disk %d from %d to %d\n",count,start,finish);	
17	move(count-1,temp,finish,start);	
18	}	
19	}	

Program 5-25 | A program to solve the Tower of Hanoi problem

The actual disk movements are shown in Figure 5.7.



(*Contd...*)

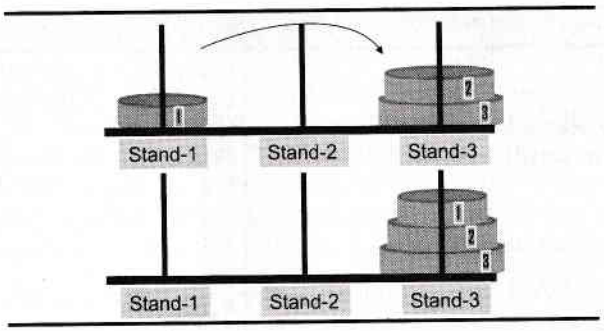
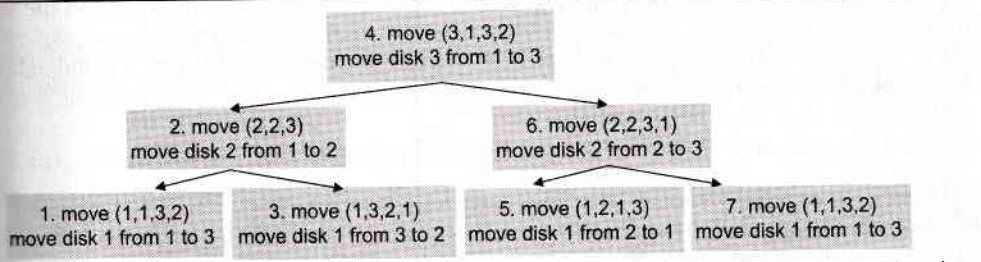


Figure 5.7 | Actual disk movements in solution to the Tower of Hanoi problem with three disks

Binary tree of recursive calls to the `move` function is shown in Figure 5.8.



**Disk movements can be determined by taking in-order traversal of the tree. Disk movements are numbered.

Figure 5.8 | Tree of recursive calls to the function `move`

5.3.1.1.7.3.3 n-ary Recursion

The most general form of recursion is **n-ary recursion** where n is not a constant but some parameter of function. **n-ary recursive** functions are used in generating permutations. The permutations of integers 1, 2 and 3 are as follows:

1	*2*	*3*
1	*3*	*2*
2	*1*	*3*
2	*3*	*1*
3	*1*	*2*
3	*2*	*1*

The code snippet listed in Program 5-26 uses n-ary recursion to print the permutations of integers 1, 2 and 3.

Line	Prog 5-26.c
1	//n-ary recursion
2	#include<stdio.h>
3	//Definition of n-ary recursive function
4	permute(int array[], int parray[],int L,int N)
5	{
6	int i,j;
7	//Base case: Processing the permutations
8	if(L>N)
9	{
10	for(i=1;i<=N;i++)
11	printf("%d ",parray[i]);
12	printf("\n");
13	}
14	//Recursive Case: Number of case depends upon the parameter
15	//value N. Number of time recursive calls are given is variable.
16	else
17	{
18	for(i=1;i<=N;i++)
19	{
20	if(array[i]==0)
21	{
22	parray[L]=i;
23	array[i]=1;
24	permute(array,parray,L+1,N);
25	array[i]=0;
26	}
27	}
28	}
29	}
30	main()
31	{
32	int array[10]={0}, parray[10],n;
33	printf("Generating permutations of 1 to n\n");
34	printf("Enter the value of n(<10)\t");
35	scanf("%d",&n);
36	permute(array,parray,1,n);
37	}

Output window

```
Generating permutations of 1 to n
Enter the value of n(<10)    3
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

Remarks:

- permute is an n-ary recursive function
- The number of time recursive calls are given to permute depends upon the value of the parameter n
- Since the parameter n is a variable, the number of recursive calls in the activation of permute varies
- Trace the program to understand it clearly

Program 5-26 | A program that illustrates the use of recursion to print permutations

5.3.1.1.8 Pointers to Functions

Like recursion, pointers to functions provide an extremely interesting, efficient and elegant programming technique. The following concepts allow the creation of a pointer to a function:

1. Like variables, a compiled code upon execution gets some space in the main memory. Thus, a function in the program code is placed at some memory location in the Code Segment.

2. Functions like all other identifiers (except labels) do have a type. **Function type** is one of the derived data types. It consists of return type of the function and types of its parameters. For example, type of a function mult that accepts one integer and one float argument and returns a float value is float(int,float). The construction of a function type from its return type and parameter types is called **'function type derivation'**.
3. It is possible to create a pointer to any type (even void type). Hence, the creation of a pointer to a function type is also possible. A pointer to a function, commonly known as **function pointer,** is a variable that points to the starting address of the function.

Unfortunately, pointers to functions are less frequently used because of their complicated syntax. The following aspects of function pointers must be mastered so that they can be used in a correct way:

1. Declaration of a function pointer
2. Assigning or initializing a function pointer
3. Calling a function using a function pointer

Backward Reference: For a description on Code Segment (CS) refer Answer number 28 (Chapter 4).

5.3.1.1.8.1 Declaration of a Function Pointer

Consider the function fact developed in Program 5-22, which accepts an integer and returns an integer value. The type of function fact is int(int). A pointer to the function type int(int) is declared as:

```
int (*ptr)(int);
```

In the above declaration,[✍] ptr is a pointer to a function that accepts an integer and returns an integer value.

While reading C declaration, remember that [] and () bind more tightly than *. Hence, in declaration statement int* ptr(int);, the identifier ptr is bound to () instead of * and is read as: **ptr is a function that accepts an integer and returns an integer pointer**. The () can be used to bind ptr with *. In declaration statement int(*ptr)(int);, () is used to bind ptr with *. Hence, this declaration is read as: **ptr is a pointer to a function that accepts an integer and returns an integer value.**

Table 5.1 mentions some of the functions developed in this chapter, their types and pointers to functions of that type.

Table 5.1 | Pointers to function types

Sl.No	Function name(s)	Program number	Function type	Pointer to function type
1.	println	5-1	int()	int(*)()
2.	add, sub	5-1	int(int,int)	int(*)(int,int)
3.	printsum, main	5-5	void(void)	void(*)(void)
4.	printsum	5-6	void(int,int)	void(*)(int,int)
5.	printsum	5-7	void(int,int,char)	void(*)(int,int,char)

(Contd...)

6.	circle_area	5-9	float(int)	float(*)(int)
7.	swap	5-12	int(int*,int*)	int(*)(int*,int*)
8.	sum_diff	5-13	void(int,int,int*,int*)	void(*)(int,int,int*,int*)
9.	find_max_min	5-15a	int(int[],int)	int(*)(int[],int)
10.	find_max_min	5-15b	int(int*,int)	int(*)(int*,int)
11	largest_ele	5-16	void(int[][10],int*,int*)	void(*)(int[][10],int*,int*)
12.	f_calling_fs	5-30	void(int,int,int(*)(int,int))	void(*)(int,int,int(*)(int,int))

5.3.1.1.8.2 Assigning or Initializing a Function Pointer

A pointer to a function of type T can be assigned or initialized with the address of a function of type T or with a pointer of the same type. To assign or initialize a function pointer with the address of a function, just place the function designator (i.e. the name of the function) of a suitable and known function on the right side of the assignment operator. In the following statements, the address of the function sub is assigned to the function pointer str:

```
int sub(int,int);
int (*str)(int,int);
str=sub;
```

In the following statements, the function pointer atr is initialized with the address of the function add:

```
int add(int,int);
int(*atr)(int,int)=add;
```

The important points about the function pointer assignment or function pointer initialization are as follows:

1. At the time of function pointer assignment or initialization, the function designator must be known, i.e. declared or defined.
2. The function designator implicitly refers to the starting address of the function. However, the function designator can optionally be preceded by the address-of operator (&) to signify the address of function. The following two statements are equivalent:

```
int (*atr)(int, int)=add;
int (*atr)(int,int)=&add;
```

5.3.1.1.8.3 Calling a Function Using Function Pointer

A function pointer can be used to call a function in any of the following two ways:

1. By explicitly dereferencing it using the dereference operator, i.e. *
2. By using its name instead of the function's name

Program 5-27 illustrates the method of calling a function using the function pointers.

Line	Trace	Prog 5-27.c	Output window
1		//Calling functions using function pointers	Calling functions using function pointers:
2		#include<stdio.h>	The result of addition is 22
3		int add(int a,int b);	The result of addition is 5
4	1	main()	**Remarks:**
5		{	• Type of function add is int(int,int)
6		//Assigning address by using function designator only	• ptr1 and ptr2 are pointers to a function of type int(int,int)
7	2	int (*ptr1)(int,int)=add;	• ptr1 is assigned an address of the function add by using the function designator only
8		//Assigning address by using address-of operator	• ptr2 is assigned an address of the function add by using address-of operator and the function designator
9	3	int (*ptr2)(int,int)=&add;	• ptr1 and ptr2 both point to the function add
10	4	printf("Calling functions using function pointers:\n");	• In line number 12, ptr1 is dereferenced and is used to call the function add
11		//Calling function by dereferencing function pointer	• In line number 14, ptr2 is used to call the function add without dereferencing it
12	5	(*ptr1)(10,12);	• Trace the program and note the trace arrow numbering
13		//Calling by using function pointer name	
14	9	ptr2(2,3);	
15	13	}	
16	6,10	int add(int a, int b)	
17		{	
18	7,11	printf("The result of addition is %d\n",a+b);	
19	8,12	}	

Program 5-27 | A program that illustrates the method of calling function using function pointers

5.3.1.1.9 Array of Function Pointers

Like arrays of pointers to other types, it is possible to create array of pointers to function type (i.e. array of function pointers). The following declaration statement declares arr as an array of pointers to functions that accept two integers and returns an integer:

```
int (* arr[4])(int,int);
```

The important points about the above declaration and the array of function pointers are as follows:

1. arr is an array of function pointers. Each pointer takes 2 bytes or 4 bytes in the memory depending upon the compiler and the working environment used. Hence, the total memory space allocated to arr will be 8 bytes or 16 bytes. The code snippet in Program 5-28 illustrates this fact.

Line	Prog 5-28.c	Output window
1	//Size of array of function pointers	Memory allocated to arr is 8 bytes
2	#include<stdio.h>	**Remarks:**
3	main()	• Turbo C 3.0 gives the above-mentioned result. If Turbo C 4.5 is used, the result will be 16 bytes
4	{	• The name of an array does not decompose to a pointer type if it is an operand of sizeof operator
5	int (*arr[4])(int,int);	• sizeof operator gives the memory allocated to the complete array
6	printf("Memory allocated to arr is %d bytes",sizeof(arr));	
7	}	

Program 5-28 | A program that finds the size of an array of function pointers

2. Like other arrays, arrays of function pointers can also be initialized by providing an initialization list. The initializers in the initialization list should be function designators of the known functions (i.e. declared or defined) of appropriate type. All the initializing functions should have the same type.
3. The array of function pointers can be used to call functions in a generalized way. The code snippet in Program 5-29 illustrates the initialization of an array of function pointers and the method to call functions in a generalized way.

Line	Prog 5-29.c
1	//Array of function pointers
2	#include<stdio.h>
3	//Function declarations
4	int add(int,int);
5	int sub(int,int);
6	int mult(int,int);
7	int div(int,int);
8	//Function definitions
9	main()
10	{
11	//Array of function pointers initialized with initialization list
12	int (*arr[4])(int,int)={add,sub,mult,div};
13	int lc;
14	printf("Calling functions using iteration:\n");
15	//Functions called in a generalized way by using loop
16	for(lc=0;lc<4;lc++)
17	arr[lc](6,3);
18	}
19	int add(int a,int b)
20	{
21	printf("Result of addition of %d and %d is %d\n",a,b,a+b);
22	}
23	int sub(int a,int b)
24	{
25	printf("Result of subtraction of %d and %d is %d\n",a,b,a-b);
26	}
27	int mult(int a,int b)
28	{
29	printf("Result of multiplication of %d and %d is %d\n",a,b,a*b);
30	}
31	int div(int a,int b)
32	{
33	printf("Result of division of %d and %d is %d\n",a,b,a/b);
34	}

Output window

```
Calling functions using iteration:
Result of addition of 6 and 3 is 9
Result of subtraction of 6 and 3 is 3
Result of multiplication of 6 and 3 is 18
Result of division of 6 and 3 is 2
```

Remarks:

- All functions add, sub, mult and div have the same type, i.e. int(int,int)
- These functions accept two integers and return an integer
- arr is an array of 4 function pointers of type int(*)(int,int)
- arr is initialized with an initialization list
- All the initializers are of the same type
- It can also be initialized as int(*arr[4])(int,int)={&add,&sub,&mult,&div}

Program 5-29 | A program that illustrates the use of array of function pointers

5.3.1.10 Passing Function to a Function as an Argument

A function can accept arguments of pointer type. We have seen the application of pointers to pass arrays as arguments to the functions. Pointers can also be used to pass functions to a function. The code snippet in Program 5-30 illustrates the use of pointers to pass functions to a function.

Line	Trace	Prog 5-30.c	Output window
1		//Passing function to a function as an argument	Passing functions to a function:
2		#include<stdio.h>	Result of addition of 10 and 20 is 30
3		//Function declarations	Result of subtraction of 10 and 20 is -10
4		int add(int,int);	**Remarks:**
5		int sub(int,int);	• The third argument to the function f_calling_fs is a function pointer of type int(*)(int,int)
6		//Declaration of function whose third parameter is a function ptr	• In line number 13, the address of the function add is passed to the function f_calling_fs
7		void f_calling_fs(int,int,int(*)(int,int));	• In line number 18, the passed argument is used to call the function. Since the address of the function add has been passed, the call fun(a,b) is equivalent to add(a,b)
8		//Function definition	• Similarly, in line number 14, the function sub is passed to f_calling_fs and on the second execution of line number 18, it is called
9	1	void main()	• Trace the program and note the trace arrow numbering
10		{	
11	2	printf("Passing functions to a function:\n");	
12		// Third argument in the following function calls is a function designator	
13	3	f_calling_fs(10,20,add);	
14	6	f_calling_fs(10,20,sub);	
15	7	}	
16	4,11	void f_calling_fs(int a, int b, int (*fun)(int,int))	
17		{	
18	5,12	fun(a,b);	
19	8,15	}	
20	13	int add(int a,int b)	
21		{	
22	9	printf("Result of addition of %d and %d is %d\n",a,b,a+b);	
23	10	return 0;	
24		}	
25	14	int sub(int a,int b)	
26		{	
27	16	printf("Result of subtraction of %d and %d is %d\n",a,b,a-b);	
28	17	return 0;	
29		}	

Program 5-30 | A program that illustrates the passing of functions to a function

5.3.1.2 Library Functions

Library functions or pre-defined functions are the functions whose functionality has already been developed by someone and are available to the user for use. For example, printf and scanf are library functions. There are two aspects of working with library functions:

1. Declaration of library functions
2. Use of library functions

5.3.1.2.1 Declaration of Library Functions/Role of Header Files

We have seen that the user-defined functions need to be declared before they are called. This is true for library functions as well. A library function needs to be declared before it is called. The

declarations of library functions are available in their respective header files. To make these declarations accessible in a program file, the corresponding header files are included. For example, the prototype, i.e. the declaration of printf function is available in the header file stdio.h. That is why stdio.h is included before calling the printf function. If the header file containing the declaration of the library function is not included before its use, there will be a compilation error 'Prototype missing.'

5.3.1.2.2 Use of Library Functions

Library functions are used in the same way as user-defined functions, i.e. by using a function call operator. The role and usage of some of the common library functions are listed below.

5.3.1.2.2.1 Library of Mathematical Functions

The mathematical library defines some of the common mathematical functions. The declarations of these mathematical functions are available in the header file **math.h**. Table 5.2 lists the commonly used mathematical functions available in the math library.

Table 5.2 | Mathematical functions available in math library

S.No	Function	Function declaration and use	Role
		Trigonometric functions	
1.	acos	double acos(double x);	Returns arc cosine of x in radians
2.	asin	double asin(double x);	Returns arc sine of x in radians
3.	atan	double atan(double x);	Returns arc tangent of x in radians
4.	atan2	double atan2(double y, double x);	Returns the arc tangent in radians of y/x based on the signs of both values to determine the correct quadrant
5.	cos	double cos(double x);	Returns the cosine of a radian angle x
6.	cosh	double cosh(double x);	Returns hyperbolic cosine of x
7.	sin	double sin(double x);	Returns the sine of a radian angle x
8.	sinh	double sinh(double x);	Returns hyperbolic sine of x
9.	tan	double tan(double x);	Returns the tangent of a radian angle x
10.	tanh	double tanh(double x);	Returns hyperbolic tangent of x
		Exponential, logarithmic and power functions	
11.	exp	double exp(double x)	Returns the value of e raised to the x^{th} power
12.	frexp	double frexp(double x, int *exponent);	frexp splits a double number x into mantissa and exponent. Given x, frexp calculates the mantissa m and exponent n such that $x = m*2^n$
13,	ldexp	double ldexp(double x, int exponent);	Returns x multiplied by 2 raised to the power of exponent, i.e. returns $x*2^n$
14.	log	double log(double x);	Returns the natural logarithm (*base e*) of x
15.	log10	double log10(double x);	Returns the common logarithm (*base 10*) of x
16.	pow	double pow(double x, double y);	Returns x raised to the power of y
17.	sqrt	double sqrt(double x);	Returns the square root of x

(Contd...)

Other mathematical functions			
	ceil	double ceil(double x);	Returns the smallest integer value greater than or equal to x
	fabs	double fabs(double x);	Returns the absolute value of x (a negative value becomes positive, positive value remains unchanged)
	floor	double floor(double x);	Returns the largest integer value less than or equal to x
	fmod	double fmod(double x, double y);	Calculates x modulo y, i.e. returns the remainder of x divided by y

The return type of every math library function is double.

5.3.1.2.2.2 Library of Standard Input/Output Functions

The functionality of standard input and output operations is provided by this library. The declarations of these functions are available in the header file stdio.h. stdio is an acronym for standard input output. The common standard input/output functions are printf, scanf, gets, puts, getchar, putchar, etc.

Forward Reference: gets, puts and other input–output functions (Chapter 6).

5.3.1.2.2.3 Library of String Processing Functions

This library consists of functions that are used for string processing. The common string library functions are strcpy, strrev, strcat, strcmp, strcmpi, etc. The declarations of these functions are available in the header file string.h. The role and working of string library functions will be discussed in Chapter 6 after the discussion on character arrays.

Forward Reference: String processing functions (Chapter 6).

5.3.2 Based upon the Number of Arguments a Function Accepts

Based upon the number of arguments a function accepts, functions are classified as follows:

1. Fixed argument functions
2. Variable argument functions

5.3.2.1 Fixed Argument Functions

A function that accepts a fixed number of arguments is called a **fixed argument function**. If the fixed argument function does not specify any default argument, invoking a fixed argument function with a lesser number of arguments than expected leads to a compilation error. A fixed argument function cannot even be invoked by supplying more number of arguments than expected. For example, pow function listed in Table 5.2 expects two arguments of type double. The following invocations of pow function are erroneous:

```
pow();          //←Lesser number of arguments supplied than expected
pow(2.0);       //←Lesser number of arguments supplied than expected
pow(2.0,1.5,1.0); //←More number of arguments supplied than expected
```

5.3.2.2 Variable Argument Functions

A function that accepts a variable number of arguments is called a **variable argument function**. For example, printf is a variable argument function, which can accept one or more arguments. The type of first argument must be char* and there is no constraint about the type of rest of the arguments. The following calls to printf function are valid:

```
printf("Hello");          //←Only one argument of type char*
printf("%d",2);           //←Two arguments. The type of the first argument is char* and the
                          //    second is int
printf("%s %s","Hi","!!"); //←Three arguments, all of type char*
```

A function that accepts a variable number of arguments can be created by using the macros va_start, va_arg, va_end available in the header file stdarg.h. The piece of code in Program 5-31 illustrates the development of a variable argument function.

Line	Prog 5-31.c	Output window
1	//Variable argument functions	The result of addition of 3 numbers is 39
2	//File stdarg.h is to be included for using va_list, va_start, etc.	The result of addition of 5 numbers is 150
3	#include<stdarg.h>	
4	#include<stdio.h>	
5	//Ellipses (i.e. three dots) are used to declare variable argument function	
6	int sum(int no_of_arguments, ...);	
7	//Function definitions	
8	main()	
9	{	
10	int result;	
11	//Function sum invoked with 4 arguments	
12	result=sum(3,12,13,14);	
13	printf("The result of addition of 3 numbers is %d\n",result);	
14	//Function sum invoked with 6 arguments	
15	result=sum(5,10,20,30,40,50);	
16	printf("The result of addition of 5 numbers is %d\n",result);	
17	}	
18	//Definition of a variable argument function	
19	int sum(int no_of_arguments,...)	
20	{	
21	int arg,i=0,total=0;	
22	va_list ptr;	
23	va_start(ptr,no_of_arguments);	
24	arg=va_arg(ptr,int);	
25	while(i++<no_of_arguments)	
26	{	
27	total+=arg;	
28	arg=va_arg(ptr,int);	
29	}	
30	va_end(ptr);	
31	return total;	
32	}	

Program 5-31 | A program that makes use of variable argument functions

The important points about the code listed in Program 5-31 and the variable argument functions are as follows:

1. Since the function sum is 'a fixed number of argument followed by a variable number of argument' function, it is declared as int sum(int no_of_arguments,...);. **Ellipses** (...) are used while declaring a variable argument function.
2. **Role of ellipses:** The number of arguments that can be passed to a variable argument function is not fixed. Hence, while declaring a variable argument function, it is not possible to list the types of all the arguments that might be passed to the function during the function call. The solution to this problem is provided by ellipses. While declaring a variable argument function, ellipses (...) are used in the parameter list. **The presence of ellipses (...) tells the compiler that when the function is called, zero or more arguments may follow and that the type of the arguments is not known.** Ellipses (...) used in the declaration of the variable argument function **suspend the type checking.**
   The prototype/declaration of printf function is int printf(const char*,...);. The prototype says that there can be one or more arguments in the printf function call. The type of first argument would be const char* and the latter arguments can be of any type. Due to ellipses (...) the following uses of printf function are valid:
   1. printf("Hello Readers");
   2. printf("%d %d",2,3);
   3. printf("%d %s %c",2,"Hi",'I');
3. The variable argument functions are developed with the help of macros va_start, va_arg and va_end, declared in the header file stdarg.h. Therefore, the header file stdarg.h is included so that the macros can be used.
4. The header file stdarg.h also declares a type va_list that holds the information needed by the macros va_arg and va_end. A variable ptr of type va_list is declared in the function sum.
5. The macro va_start takes two parameters ptr and lastfix. The type of the first parameter ptr is va_list and lastfix is the last fixed parameter supplied to the variable argument function. The last fixed parameter supplied to the variable argument function sum is no_of_arguments and is of type int. The macro va_start sets ptr to point to the first of the variable arguments being passed to the function.
6. The macro va_arg is used to return the arguments in the variable list. The first time va_arg is used, it returns the first argument in the list. Each successive time va_arg is used, it returns the next argument in the list. The macro va_arg returns the values of type given to it as its second argument (for example, int in the code listed in Program 5-31).
7. The macro va_end should be called after va_arg has read all the arguments. If the macro va_end is not used, the program may show strange and undefined behavior.
8. The order in which the macros va_start, va_arg and va_end should be called is:
   a. va_start must be called before the first call to va_arg or va_end.
   b. va_end should only be called after va_arg has read all the arguments.

Variable argument functions actually have a fixed number of arguments followed by a variable number of arguments.
There should be only three dots, i.e. (...) in **ellipses**. Usage of more than three dots in ellipses leads to a compilation error.
The syntax of **using macros** is similar to the syntax of using functions.

Forward Reference: Macros (Chapter 8).

5.4 Summary

1. Functions help in modularizing a program into smaller simple parts.
2. Functions are classified based upon: (a) who develops the function and (b) the parameter and the return type of the function.
3. Based upon who developed the function, they are categorized as: (a) user-defined functions and (b) library functions.
4. Based upon the parameter and the return type of the function, they are categorized as (a) functions with no input and no output, (b) functions with inputs but no output, (c) functions with inputs and a single output and (d) functions with inputs and multiple outputs.
5. User-defined functions are defined by the user at the time of writing a program and are also known as programmer-defined functions.
6. There are three aspects of working with user-defined functions: (a) function declaration, (b) function definition and (c) function call.
7. Function definition, also known as function implementation means composing a function. Every function definition consists of two parts: (a) header of the function and (b) body of the function.
8. A function with no input–output does not accept any input and does not return any result.
9. The execution of a C program always begins with the function `main`. It need not to be called explicitly.
10. Functions whose return type is `void` are known as `void` functions. `void` functions do not return any value.
11. While calling a function, the expressions that appear within the parentheses of a function call are known as actual arguments, and the variables declared in the parameter list in the header of function definition are known as formal parameters.
12. The `return` statement is to return the result of computations done in the called function and/or the program control back to the calling function.
13. There are two forms of `return` statement: (a) `return;` and (b) `return expression;`.
14. Depending upon whether values or addresses are passed as arguments to a function, the argument passing methods in C language are classified as: (a) pass by value and (b) pass by reference/address.
15. If arguments are passed by value, the changes made in the values of formal parameters inside the called function are not reflected back to the calling function.
16. If the arguments are passed by reference/address, the changes made in the values pointed to by the formal parameters in the called function are reflected back to the calling function.
17. A function can return only one value by using the `return` statement but it can indirectly return more than one value using the concept of pass by reference/address.
18. When an array is passed as an argument to a function, it implicitly gets converted to a pointer type.

19. The arguments can be made default by using an initialization syntax within the parameter list during the function declaration.
20. The default argument should not be specified in the function definition.
21. Function calling itself is called recursive function and the process is known as recursion.
22. Recursive functions may be: (a) direct recursive/indirect recursive and (b) tail recursive/non-tail recursive.
23. There are three patterns of recursive calls: (a) linear, (b) binary and (c) n-ary.
24. Like recursion, pointers to functions provide an extremely interesting, efficient and elegant programming technique.
25. A pointer to a function, commonly known as the function pointer, is a variable that points to the address of a function.
26. Library functions or pre-defined functions are the functions whose functionality has already been developed by someone and is available to the user for use.
27. The arguments to the function `main` are supplied at the command line and thus have a special name known as command line arguments.

Exercise Questions

Conceptual Questions and Answers

1. *What is a function? What are the advantages of using functions?*

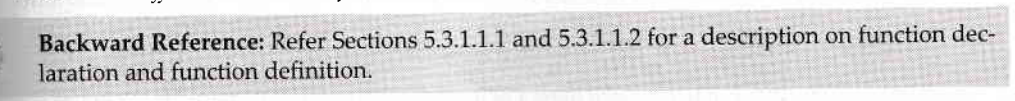
   A function is a group of statements that performs a specific task and is relatively independent of the remaining code. Functions are used to organize programs into smaller and independent units. Several advantages of modularizing the program into functions include:

   1. Reduction in code redundancy
   2. Enabling code reuse
   3. Better readability
   4. Information hiding
   5. Improved maintainability

2. *Do functions have a type like other identifiers? If yes, how is it derived?*

   Yes, functions do have a type like all other identifiers except labels. Function type is one of the derived types and consists of return type of the function and the types of its parameters. For example, the type of a function `mult` that accepts one integer and one `float` parameter and returns a `float` value is `float(int,float)`. The construction of a function type from its return type and parameter types is called 'function type derivation'.

3. *What are the differences between a function declaration and a function definition?*



Backward Reference: Refer Sections 5.3.1.1.1 and 5.3.1.1.2 for a description on function declaration and function definition.

The major differences between a function declaration and a function definition are as follows:

a. A function can only be defined once but can be declared many times.
b. A function can be declared within the body of some other function but cannot be defined within the body of some other function.

c. A function definition can also serve as a function declaration but the vice versa is not true. The function definition serves as a function declaration if it is present before the function call.

d. The function definition can be changed without changing the function declaration but if the function declaration is changed, it becomes necessary to change the function definition.

e. For using (i.e. calling) a function, it is sufficient and necessary to know the function declaration without knowing anything about how it is defined.

4. *What is meant by prototyping a function? Why is a function prototype necessary?*

The function declaration is also called a function prototype. Hence, function prototyping means declaring a function.

Backward Reference: Refer Section 5.3.1.1.1 for a description on function prototype and its necessity.

5. *'C is a strongly typed language'. What does that mean?*

'C is a strongly typed language' means that the arguments of every function call are type checked during the compilation. If the compiler detects a type mismatch between the type of an argument and the type of corresponding parameter, an implicit-type conversion is applied if possible. If it is not possible to apply implicit-type conversion, the compiler issues an error message. That is why functions cannot be called until they are declared or defined. The declaration or definition of function is necessary for the compiler to perform the type checking on the arguments of the function call against the function parameter list.

6. *Is it mandatory to specify the same name for the parameters in the declaration and definition of a function?*

No, it is not mandatory to have the same name for the parameters in the function declaration and the function definition. In fact, it is not even compulsory to write names of the parameters in the function declaration.

Backward Reference: Refer Section 5.3.1.1.1 for a description on abstract parameter declaration and complete parameter declaration.

7. *I want to write a function* add *that should add the contents of two integer variables and return their sum. I have made the following declaration for the function:*

```
int add(int v1,v2);
```

The compiler is not accepting it and is showing an error. Why?

Backward Reference: Refer Section 5.3.1.1.1 (Point 5) for a description on syntactic rules for writing the parameter list and the parameter type list.

The compiler shows an error due to erroneous parameter list. The shorthand declaration of parameters in the parameter list is not allowed and leads to the compilation error. The rectified declaration for the function can be written as `int add(int v1,int v2);`.

8. *What are user-defined functions and library or pre-defined functions? Is* main *a library function or a user-defined function?*

Backward Reference: Refer Sections 5.3.1.1 and 5.3.1.2 for a description on user-defined functions and library functions.

User-defined functions are defined by the user at the time of writing a program. Library functions are the functions whose functionality has already been developed by someone and are available to the user for use.

main is a user-defined function because the functionality to the main function is always added by the user by writing its body.

9. *Why do we include header file(s) in our programs? What is their role?*

Backward Reference: Refer Section 5.3.1.2.1 for a description on the role of header files.

10. *What is meant by the terms actual arguments and formal parameters?*

Backward Reference: Refer Section 5.3.1.1.3.2 (Point 4) for a description on actual arguments and formal parameters.

11. *What are the different ways of passing arguments to a function?*

Backward Reference: Refer Sections 5.3.1.1.3.4.1 and 5.3.1.1.3.4.2 for a description on pass by value and pass by reference.

12. *Does C actually have a pass by reference?*

No, the C language actually does not have a pass by reference. The C language always passes the argument by value. The call by reference is artificially simulated by passing addresses by value. In a call by reference, the l-values given as actual arguments are copied into the parameters of pointer type.

13. *How are arrays passed to the functions?*

Arrays are always passed by reference. The word array here means the entire array and not the individual array elements.

Backward Reference: Refer Section 5.3.1.1.4 for a description on passing arrays to functions.

14. *What are the various forms of* return *statement? What is the specific use of each form?*

Backward Reference: Refer Section 5.3.1.1.3.3.1 for a description on return statement.

15. *It is said that 'Function can only return one value'. Can't I return more than one value by writing* return value1, value2, value3;?

Backward Reference: Refer Section 5.3.1.1.3.3.1 (Point 4) to answer this question.

16. *Can a function have more than one* return *statement within its body?*

Yes, a function can have more than one return statement within its body. There is no constraint about the number of return statements that can be placed within a function's body. For example, the following piece of code is valid:

```
int funct()
{
    return 1;                          //←Control returns from this point
    printf ("This can never be executed"); //← This point onwards, code is unreachable
    return 2;
    return 3;
    printf("There are multiple return statements");
}
```

Although, a number of return statements can be placed inside the body of a function, only one of them that appears first in the logical flow of control gets executed. With the execution of this return statement, the program control returns to the calling function and the rest of the statements that appear after this return statement remain unreachable.

On compiling the mentioned code, there will be no error, but the compiler issues a warning message 'Unreachable code in function funct'. This warning is due to the fact that the program control returns to the calling function with the execution of the first return statement and can never reach the latter part of the code.

17. *I have developed the following piece of code to compute the area of a circle. It outputs* 78.000000 *instead of the actual value of the area of circle, i.e.* 78.537498. *Why?*

```
circle_area(int);
main()
{
    int rad=5.5;
    float area;
    area=circle_area(rad);
    printf("The area of circle is %f",area);
}
circle_area(int rad)
{
    float area;
    area=3.1415*rad*rad;
    return area;
}
```

Backward Reference: Refer Section 5.3.1.1.3.3.1 (Point 2 (f)) to answer this question.

18. *In the programs that I have written till now, I got a warning message 'Function should return a value'. What does this mean?*

This warning message comes if the return type of a given function is not void, and in the body of the function return statement has not been used to return any value. For example, consider the following piece of code:

```
main()
{
    printf("Warning message");
}
```

The mentioned code on compilation gives a warning message: 'Function should return a value'. There can be three different ways to remove this warning:

```
main()
{
    printf("Warning message");
    return 0;
}
```
Way I

```
void main()
{
    printf("Warning message");
}
```
Way II

```
#pragma warn -rvl
main()
{
    printf("Warning message");
}
```
Way III

A compilation of the above-mentioned codes will not generate a warning message because:

1. Way I returns an integer value.
2. Way II mentions the return type as void.
3. Way III configures the compiler using pragma directive in such a way that it does not generate 'Function should return a value' warning message.

Forward Reference: pragma directive (Chapter 8).

19. *What is the difference between a warning and an error?*

Warning is only an indicator that something may go wrong but an error is a notification that some mistake (i.e. syntactic violation) has occurred. The compiler can be configured to turn off the display of warning messages but it cannot be stopped from displaying error messages.

20. *Can a return type of a function be an array type or a function type?*

No. The return type of a function shall be void or an object type other than an array type and a function type. Arrays and functions are returned to the calling function in the same way as they are passed to the called function, i.e. by the means of pointers. For example, consider the following code snippets:
In Code I (a), the return type of the function fun_returning_array is an array type. The given code on compilation gives an error. Code 1 (b) shows the rectified version of Code I (a) whereby an array arr is returned by the means of a pointer, i.e. the address of the first element of the array. In Code II (a), the function area_of_square is passed to the function fun by the means of a pointer. In Code II (b), the function fun returns function area_of_square by the means of a pointer.

```
int[3] fun_returning_array();
main()
{
    int *ptr;
    ptr=fun_returning_array();
    printf("%d %d %d",ptr[0],ptr[1],ptr[2]);
}
int[3] fun_returning_array()
{
    int arr[3]={1,2,3};
    return arr;
}
```
Code I (a) (Return type is array type)

```
int* fun_returning_array();
main()
{
    int *ptr;
    ptr=fun_returning_array();
    printf("%d %d %d",ptr[0],ptr[1],ptr[2]);
}
int* fun_returning_array()
{
    int arr[3]={1,2,3};
    return arr;
}
```
Code I (b) (Return type is a pointer)

(*Contd...*)

```
int area_of_square(int side)
{
    return side*side;
}
int fun(int (*fun_name)(int))
{
//The parameter fun_name contains the address of function
//area_of_square
    return fun_name(2);
//The function area_of_square is called with argment 2. The
//result returned by the function area_of_square is returned
//by the function fun
}
main()
{
//The function fun is called with the name of function
//area_of_square as an argument
    printf("Area is %d",fun(area_of_square));
}
```

Code II (a)(Function passed to a function)

```
int area_of_square(int side)
{
    return side*side;
}
int(*fun())(int)
{
//Function fun accepts no argument. It returns a pointer to
//a function that accepts an integer and returns an integer
    return area_of_square;
}
main()
{
//The function fun is called without any argument. It returns a
//pointer to the function area_of_square. The returned
//pointer is used to call the function area_square with
//argument 2
    printf("Area is %d",fun()(2));
}
```

Code II (b)(Function returned by the means of a pointer)

21. *What are activation records?*

Backward Reference: Refer Section 5.3.1.1.7.3.1.1 for a description on activation records.

22. *What is recursion? What are the advantages and disadvantages of recursion over iteration?*

Backward Reference: Refer Section 5.3.1.1.7.1 for a description on direct and indirect recursion.

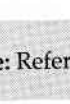
The merits and demerits of recursion over iteration are listed below:

Iteration	Recursion
1. Performance wise, iteration is superior as compared to recursion	1. Performance of recursion is poor as compared to iteration. Recursion involves calling the same function again and again. The execution of a function call is time consuming as the entire state of a calling function needs to be saved before the control is passed to the called function. Therefore, precious computing time is wasted in book-keeping tasks
2. Memory requirement of an iterative function is less as compared to that of a recursive function	2. Recursion involves function calls. Each function call requires creation of an activation record, which takes some memory. The memory required by an activation record is directly proportional to the number of local variables declared within the recursive function.

(Contd...

3. Infinite iteration will not terminate 4. Iteration is diffucult to express in some cases	The total memory required by the recursion is equal to the memory taken by all the activation records that exist at some particular instance 3. Infinite recursion will automatically terminate when there is no memory space left for the creation of the activation records 4. One of the major advantages of the recursion is the **ease of expression**. The tasks that are expressible in terms of themselves can be easily coded by using recursive functions. For example, the computation of the factorial of a number. The factorial of a number is equal to the number multiplied by the factorial of the number minus one, i.e. fact(n)=n*fact(n–1)

23. *What is tail recursion?*



Backward Reference: Refer Section 5.3.1.1.7.2 for a description on tail recursion.

24. *How do the declaration statements* int *func(int);, int(*func)(int); and int(*func())(int); *differ from each other?*

While reading C declarations, remember that [] and () bind more tightly than *. In the declaration statement, int *func(int); the identifier name func is bound to () instead of * and it is read as: **func is a function that accepts an integer and returns a pointer to an integer**. In the declaration statement, int (*func)(int); () is used to bound func to *. Hence, the declaration is read as: **func is a pointer to a function that accepts an integer and returns an integer**. In the declaration statement int(*func())(int); the identifier name func is bound to inner () instead of * and is read as: **func is a function that accepts no argument and returns a pointer to a function that accepts an integer and returns an integer.**

25. *If there is a type mismatch between the type of argument and the type of corresponding parameter, will the compiler apply implicit-type conversion? Is the same applicable if there is a mismatch between the type of value returned and the return type of a function?*

Yes, if the type of the argument and the corresponding parameter do not match or if the type of value returned does not match the return type of the function, an implicit-type conversion is applied, if possible. If it is not possible to apply an implicit-type conversion, the complier issues an error message.

26. *I have encountered the following piece of code:*

```
int add(int v1,int v2=10);
main()
{
    int result;
    result=add(5);
}
int add(int v1,int v2)
{
    return v1+v2;
}
```

There are two parameter names in the parameter list of the function add. *I have read that if the number of arguments is incorrect or the types of arguments are not compatible with the types of parameters, the compile time error is issued. In the call to function* add *only one argument is given instead of two, but still the code is executing. Why is the compiler not showing an error?*

The compiler does not show an error because v2 is a default argument. A function that provides a default argument for a parameter can be invoked with or without an argument for that parameter. If an argument is not provided, the default argument value is used, but if it is provided it overrides the default argument value.

27. *What are variable argument functions? How are they created?*

Backward Reference: Refer Section 5.3.2.2 for a description on variable argument functions.

28. *A variable argument function can have a variable number of arguments, so it is not possible to list the type and number of all the arguments that might be passed to a function. Therefore, how can I make declaration for a variable argument function, and how is type checking done for a variable argument function?*

Backward Reference: Refer to the role of ellipses mentioned in Section 5.3.2.2 to answer this question.

29. *Are the following declarations equivalent?*
    1. void funct();
    2. void funct(parameter_list,...);
    3. void funct(...);

No, the specified declarations are not equivalent. In declaration 1, funct is declared as a function that accepts no arguments. In declaration 2, funct is declared as a function that at least accepts the arguments of the specific type mentioned in the parameter list. In declaration 3, funct is declared as a function that can take zero or more arguments.

30. *Why does the following piece of code not compile successfully?*

```
test_function()
{
    printf("Control is now in test function");
    return;
}
main()
{
    printf("There is a simple call to a test function");
    test_function();
    printf("Control returns to main after executing test function");
}
```

The code does not compile successfully because the return type of the function test_function is not specified. By default, it would be considered as int. In the body of the function test_function, the first form of the return statement (i.e. return;) is used, but it can only be used if the return type of the function is void. That is why the compiler shows an error. There are two ways of removing this error:

1. Specify the return type of the function test_function as void.
2. Use the second form of return statement (i.e. return expression;) in the body of the function test_function. Write return 0; instead of return;.

31. *I have written the following piece of code:*

```
inc_value(int a)
{
5+return a;
}
main()
{
    int a=10,c;
    c=inc_value(a);
    printf("The incremented value of a returned is %d",c);
}
```

Why is the following piece of code not working?

The code is not working because it is not valid to write 5+return a;. return is a statement and cannot be used as an operand of an operator. Only the expressions can form operands of an operator. Instead of 5+return a; it should have been return 5+a; or return a+5;.

32. *What will the output of the following piece of code be?*

```
main()
{
    printf("%d",sizeof(printf("Hello Readers!!")));
}
```

The output of the code **WILL NOT** be Hello Readers!!2. The given piece of code on execution outputs 2. Remember that the operand of sizeof operator is not evaluated. Thus, when the expression printf("Hello Readers!!") is given to it as an operand, it is not evaluated, and the operator operates on its return type, i.e. int. Thus, the output comes out to be 2. Consider another example:

```
main()
{
    int a=2;
    printf("%d ",sizeof(a+=2));
    printf("%d ",a);
}
```

The mentioned piece of code on execution outputs 2 2 instead of 2 4 because the expression a+=2 is not evaluated.

33. *What will the output of the following piece of code be?*

```
main()
{
    printf("goto statement trying to transfer control to other function");
    goto target_pt;
}
other_funct()
{
    target_pt:
    printf("The target label is present in other function");
}
```

The mentioned piece of code on compilation gives an error 'Undefined label target_pt in function main'. The goto statement can only transfer the control from one point to another within the same function. It cannot take the control from one function to another.

34. *Both the function call statement and the* goto *statement can be used to transfer the control from one point to another. Then, why does the* goto *statement cannot be used to transfer control from one function to another?*

The goto statement cannot be used to take the control from one function to another. Transferring the control from one function to another is not as simple as transferring control within the same function. If a control is to be transferred from one function (i.e. calling function) to another (i.e. called function), the following two additional tasks along with some other activities are to be performed:

1. **Saving all the computations performed in the calling function prior to the function call:** All the computations performed in the calling function prior to the function call need to be saved so that they need not be carried out again upon returning from the called function. To save all the computations performed, all the local variables declared within the calling function are saved before executing the function call. The stored values of the local variables are restored after returning from the called function.
2. **Saving the point of function call:** The point from where the function call is given is saved so that the control can return to the same point after executing the called function. The point of the function call can be saved by taking the dump of content of **registers**, specifically IP register. Instruction Pointer (IP) register is a 16-bit register that points to the memory location of the next statement to be executed. When the control returns from the called function, the content of the Instruction Pointer register is restored so that the statement next to the statement containing the function call gets executed.

Execution of these additional tasks requires some time and that is why function calls are time consuming. Transferring the control within the same function just requires the manipulation of content of the Instruction Pointer and does not require the above tasks to be carried out. Since the goto statement just manipulates the content of the Instruction Pointer and does not carry out the above-mentioned tasks, it cannot be used to transfer the control from one function to another.

35. *Inputs are given to the functions by means of arguments.* main *is also a function. Therefore, can we give inputs to the function* main *by supplying arguments?*

Yes, inputs to the function main can be given by making use of **command line arguments.**

Forward Reference: Command line arguments (Chapter 6).

Code Snippets

Determine the output of the following code snippets. Assume that the inclusion of the required standard header files has been made and there is no prototyping error due to them. Prototypes of user-defined functions are explicitly mentioned, if required.

36.
```
main()
{
    int a;
    a=printf("Hello")+printf("Readers!!");
    printf("\n%d characters printed",a);
}
```

37.
```
main()
{
    int a=10,b=20,c;
    c=add(a,b);
    printf("The result after addition is %d",c);
}
int add(int a, int b)
{
    return a+b;
}
```

38.
```
main()
{
    int add(int,int),a,b;
    a= b=10;
    printf("The result of addition is %d",add(a,b));
}
int add(int a,int b)
{
    return a+b;
}
```

39.
```
int add(int,int);
main()
{
    int a=10,b=10,c;
    c=add(a,b);
    printf("The result after addition is %d",c);
}
int add(int a, int b)
{
    return a+b;
}
```

40.
```
main()
{
    int add(int,int),a,b,c;
    a=10;b=20;
    c=add(a,b);
    printf("The result of addition is %d",add(a,b));
}
int add(int a, b)
{
    return a+b;
}
```

41.
```
main()
{
    int add(int,int),a,b,c;
    a=10; b=20;
    c=add(a,b);
    printf("The result of addition is %d",c);
    int add(int a,int b)
```

```
    {
        return a+b;
    }
}
```

42.
```
void fun(int a)
{
    printf("The value of a inside fun is %d\n",a);
}
main()
{
    int a=10,b;
    b=fun(a);
    printf("The value of b after call to fun is %d",b);
}
```

43.
```
fun(int a)
{
    printf("The value of a inside fun is %d",a);
}
main()
{
    int a=10,b;
    b=fun(a);
    printf("\nThe value of b after call to fun is %d",b);
}
```

44.
```
fun(int a)
{
    printf("The value of a inside fun is %d\n",a);
     a+2;
}
main()
{
    int a=10,b;
    b=fun(a);
    printf("The value of b after call to fun is %d",b);
}
```

45.
```
int add(int,int);
main()
{
    int a=10,b=20,c;
    c=add(a,b);
    printf("The result after addition is %d",c);
}
int add(int a, int b)
{
    a+b;
}
```

46.
```
int add(int,int);
main()
```

```
{
    int a=10,b=20,c;
    c=add(a,b);
    printf("The result after addition is %d",c);
}
int add(int a, int b)
{
    a+b;
    return;
}
47. int add(int a,int b)
{
    return a+b;
}
main()
{
    int c;
    c=add(10);
    printf("The result after addition is %d",c);
}
48. int add(int a,int b=12)
{
    return a+b;
}
main()
{
    int c;
    c=add(10);
    printf("The result after addition is %d",c);
}
49. int add(int a,int b=12)
{
    return a+b;
}
main()
{
    int c;
    c=add(10,20);
    printf("The result after addition is %d",c);
}
50. int add(int a=12,int b)
{
    return a+b;
}
main()
{
    int c;
    c=add(10,20);
    printf("The result after addition is %d",c);
}
```

51.
```
int swap(int a,int b)
{
    a^=b^=a^=b;
    printf("The values of a and b in swap are %d %d\n",a,b);
}
main()
{
    int a=10,b=20;
    printf("This is illustration of pass by value\n");
    printf("The values of a and b before swap are %d %d\n",a,b);
    swap(a,b);
    printf("The values of a and b after swap are %d %d\n",a,b);
}
```

52.
```
int swap(int *a,int *b)
{
    *a^=*b^=*a^=*b;
    printf("The values of a and b in swap are %d %d\n",*a,*b);
}
main()
{
    int a=10,b=20;
    printf("This is illustration of pass by reference or address\n");
    printf("The values of a and b before swap are %d %d\n",a,b);
    swap(&a,&b);
    printf("The values of a and b after swap are %d %d\n",a,b);
}
```

53.
```
int sum_diff(int a,int b)
{
    int sum=a+b;
    int diff=a-b;
    return sum,diff;
}
main()
{
    int a=20,b=10;
    printf("Sum is %d and Difference is %d\n",sum_diff(a,b),sum_diff(a,b));
}
```

54.
```
int sum_diff(int a,int b)
{
    int sum=a+b;
    int diff=a-b;
    return sum,-return diff;
}
main()
{
    int a=20,b=10;
    printf("Sum is %d and Difference is %d\n",sum_diff(a,b),sum_diff(a,b));
}
```

55.
```
int sum_diff(int a,int b)
{
    int sum=a+b;
    int diff=a-b;
    return sum;
    return diff;
}
main()
{
    int a=20,b=10;
    printf("Sum is %d and Difference is %d\n",sum_diff(a,b),sum_diff(a,b));
}
```

56.
```
sum_diff(int a,int b,int *sum,int *diff)
{
    *sum=a+b;
    *diff=a-b;
}
main()
{
    int a=20,b=10,sum,diff;
    sum_diff(a,b,&sum,&diff);
    printf("Sum is %d and Difference is %d\n",sum,diff);
}
```

57.
```
fun1()
{
    return printf("Control is in Function1\n");
}
fun2()
{
    return printf("Control is in Function2\n");
}
main()
{
    printf("%d %d",fun1(),fun2());
}
```

58.
```
fun1()
{
    return printf("Control is in Function1\n");
}
fun2()
{
    return printf("Control is in Function2\n");
}
main()
{
    printf("%d",fun1()+fun2());
}
```

59.
```
int fact(int no)
{
    if(no==1)
        return 1;
    else
        return no*fact(no-1);
}
main()
{
    int temp;
    temp=fact(4);
    printf("The value of factorial of 4 is %d", temp);
}
```

60.
```
main()
{
    printf("Infinite Recursion\n");
    main();
}
```

61.
```
check_ptr(int [2][3]);
main()
{
    int arr[2][3]={1,2,3,4,5,6};
    printf("Size of arr in function main is %d\n",sizeof(arr));
    check_ptr(arr);
}
check_ptr(int arr[2][3])
{
    printf("Size of arr in function check is %d",sizeof(arr));
}
```

62.
```
int add(int a,int b)
{
    return a+b;
}
main()
{
    int (*ptr)(int,int);
    ptr=add;
    printf("The result of addition is %d\n",ptr(2,3));
    printf("The result of addition is %d",(*ptr)(2,3));
}
```

63.
```
int add(int a,int b)
{
    return a+b;
}
int sub(int a,int b)
{
    return a-b;
}
int mul(int a,int b)
```

```
{
    return a*b;
}
int div(int a,int b)
{
    return a/b;
}
main()
{
    int (*ptr[4])(int,int)={add,sub,mul,div};
    int i;
    for(i=0;i<4;i++)
        printf("The result of called function %d is %d\n",i+1,ptr[i](10,5));
}
```

```
int add(int,int);
int sub(int,int);
fun(int(*)(int,int));
main()
{
    printf("%d\n",fun(add));
    printf("%d",fun(sub));
}
fun(int (*a)(int,int))
{
    return a(2,3);
}
int add(int a,int b)
{
    return a+b;
}
int sub(int a,int b)
{
    return a-b;
}
```

```
int add(int a,int b){return a+b;}
int sub(int a,int b){return a-b;}
int mult(int a,int b){return a*b;}
int div(int a,int b){return a/b;}
int (*f_returning_fps(int))(int,int);
main()
{
    int i=1, j=3, res1,res2;
    res1=f_returning_fps(i)(15,5);
    printf("Result of operation1 is %d\n",res1);
    res2=f_returning_fps(j)(15,5);
    printf("Result of operation2 is %d\n",res2);
}
int(*f_returning_fps(int a))(int,int)
```

```
{
    int (*arr[4])(int,int)={add,sub,mult,div};
    return arr[a];
}
```

Multiple-choice Questions

66. A function can return
    a. No value
    b. Only one value
    c. Two values
    d. As many values as the user likes

67. By default, the return type of a function is
    a. char
    b. int
    c. float
    d. void

68. A function can be
    a. Defined within another function
    b. Declared within another function
    c. Both defined as well as declared within another function
    d. None of these

69. Which of the following can be a possible return type of a function?
    a. Array type
    b. Function type
    c. Pointer type
    d. All of these

70. Which of the following is not a valid parameter type for a function?
    a. Array type
    b. Function type
    c. Pointer type
    d. None of these

71. A function that calls itself within its own body is called
    a. Mutually recursive
    b. Indirect recursive
    c. Direct recursive
    d. None of these

72. The changes made in the parameters in the called function are reflected to the calling function. The probable method of argument passing is:
    a. Pass by value
    b. Pass by reference
    c. Any of pass by value or pass by reference
    d. None of these

73. The method used to pass an array to a function is
    a. Value
    b. Reference
    c. Cannot be passed to functions
    d. None of these

74. Which of the following is a definite advantage of recursion over iteration?
    a. Better execution speed
    b. Saving in memory space
    c. Ease of expression
    d. None of these

75. The declaration statement int *ptr(int,int); declares ptr to be a
    a. Pointer to a function that accepts two integers and returns an integer
    b. A function that accepts two integers and returns a pointer to an integer
    c. Pointer to an array of two integers
    d. None of these

76. The execution of a program
   a. Always starts with main function
   b. Starts with the function that is defined first
   c. Can start from any function
   d. None of these

77. The type of a function depends upon
   a. Its return type
   b. Types of its parameters
   c. Its return type and types of its parameters
   d. None of these

78. The values given to a function at the time of making the function call are called
   a. Actual arguments
   b. Formal arguments
   c. Formal parameters
   d. None of these

79. The statement that is used to terminate the execution of a function is
   a. break statement
   b. return statement
   c. continue statement
   d. exit function call statement

80. main is a
   a. User-defined function
   b. Library function
   c. Pre-defined function
   d. None of these

81. In the C statement, a=f1(1,2)+f2(2,3)/f3(3,4);, the order in which functions f1, f2 and f3 are called is
   a. f1, f2, f3
   b. f2, f3, f1
   c. f3, f2, f1
   d. Random order

82. In the C statement, a=f1(1,2),f2(2,3),f3(3,4);, the order in which functions f1, f2 and f3 are called is
   a. f1, f2, f3
   b. f2, f3, f1
   c. f3, f2, f1
   d. Random order

83. In the C statement, printf("%d %d %d",f1(1,2),f2(2,3),f3(3,4));, the order in which functions f1, f2 and f3 are called is
   a. f1, f2, f3
   b. f3, f2, f1
   c. Random order
   d. The order is unspecified and is compiler dependent

84. The number of times Infinite recursion is printed by the following C program is

```
main()
{
   printf("Infinite recursion\n");
   main();
}
```

   a. Infinite number of times
   b. 32767 times
   c. Till the run-time stack does not overflow
   d. 65535 times

85. Which of the following is a variable argument function?
   a. printf
   b. puts
   c. gets
   d. strcpy

Outputs and Explanations to Code Snippets

36. HelloReaders!!
14 characters printed

Explanation:

The printf function call is a valid expression. The printf function returns an integer value equal to the number of characters it prints. Hence, printf("Hello") prints Hello and returns 5. Similarly, printf("Readers!! prints Readers!! and returns 9. The values returned by the printf functions are summed up and the final value is assigned to the integer variable a. The value of a is printed by the next printf statement.

37. Compilation error "Call to undefined function 'add' in function main()"

Explanation:

A function needs to be defined or declared before it is called. In the given piece of code, function add is neither defined nor declared before it is called. Hence, the compiler will not be able to perform type-checking and therefore issues an error message.

38. The result of addition is 20

Explanation:

Backward Reference: Refer the explanation given in Answer number 3.

It is valid to declare a function within the body of some other function. The function add is declared within the body of the function main before its call. Upon invocation, function add returns the result of the addition of the values of a and b, i.e. 20. The returned result is printed by the printf function.

39. The result of addition is 20

Explanation:

The only constraint about the place of declaration of a function is that it should be before its call. The declaration can be either in the local scope➲ or in the global scope.➲ In the given piece of code, the function add has been declared in the global scope.

Forward Reference: Local scope and Global scope (Chapter 7).

40. Compilation error

Explanation:

Shorthand declaration of the parameters in the parameter list is not allowed and this leads to the compilation error. The rectified declaration of the parameter list is as follows:

```
int add(int a, int b)
{.....}
```

41. Compilation error

Explanation:

Backward Reference: Refer the explanation given in Answer number 3.

A function can be declared but cannot be defined within the body of some other function. In the given piece of code, function add is defined within the body of the function main. This is not valid and leads to the compilation error.

42. Compilation error

Explanation:

Backward Reference: Refer Section 5.3.1.1.3.1.1 (Point 2).

The return type of the function fun is void. It will not return any value. If it does not return any value, how can the returned value be assigned to b? Hence, writing b=fun(a); is erroneous and leads to the compilation error.

43. The value of a inside fun is 10
The value of b after call to fun is 31

Explanation:

The return type of the function fun is not specified and by default will be considered as int. The function fun is expected to return an integer value but no return statement is used inside its body to return a value. **If no return statement is used inside the body of a function to return a value, then by default it returns the content of the accumulator register (AX). The content of the accumulator register is the result of the last computation.** The printf function prints a string and returns a value equal to the number of characters it prints. Therfore, after the execution of printf function, the content of the accumulator register will be the value returned by the printf function, i.e. 31. The content of the accumulator register will be returned by the function fun, will be assigned to the variable b and will be printed later.
Try changing the number of characters in the string given to the function printf in the function fun and observe the values of b.

The content of the accumulator register can be observed by **tracing** the program and looking at its content in the **register window**. In Borland TC 3.0, register window can be opened by going to the Window menu and invoking the Register option. In Borland TC 4.5, register window can be opened by going to the View menu and invoking the Register option.

44. The value of a inside fun is 10
The value of b after call to fun is 12

Explanation:

Backward Reference: Refer the explanation given in Answer number 43.

The last computation performed in the function fun is a+2. After the execution of this computation, content of the accumulator would be 12. As no return statement is used in the function fun, it returns the content of the accumulator register, i.e. 12.

45. The result after addition is 30

Explanation:

Since no return statement is present, the result of the last computation that is present in the accumulator register (i.e. result of a+b) is returned.

46. Compilation error

Explanation:

The first form of the return statement (i.e. return;) can only be used if the return type of the function is void. In the given code, the return type of the function add is int, so the second form of the return statement, i.e. return expression; should have been used instead of return;.

47. Compilation error "Too few parameter in call to add(int,int) in function main"

Explanation:

Function add is a fixed argument function and expects two arguments. As it is called with only one argument, i.e. 10, there is a mismatch in the number of arguments and the number of parameters. Therefore, the compiler issues an error message.

48. The result after addition is 22

Explanation:

There will be no compilation error as in Question number 47. If a function provides a default argument for a parameter, then it can be invoked with or without an argument for that parameter. In the given piece of code, the default argument (i.e. 12) is provided for the parameter b. Hence it is not mandatory to provide an argument for the parameter b.

49. The result after addition is 30

Explanation:

If an argument corresponding to the parameter with the default argument is provided in a function call, it overrides the value of the corresponding default argument. In the given piece of code, function add is called with two arguments, i.e. 10 and 20. The value 20 overrides the default argument value. Hence, the value of b in the function add will be 20. Thus, the value returned by the function add will be 30 and it gets printed by the printf function.

50. Compilation error "Default value missing following parameter a"

Explanation:

A function declaration can specify default arguments for all or for a subset of parameters. If default arguments are specified only for the subset of parameters, then they should be specified for the parameters that lie on the trailing side. Hence, it is not possible to specify the default argument for the parameter a unless and until the default argument for the parameter b is specified.

51. This is illustration of pass by value
The values of a and b before swap are 10 20
The values of a and b in swap are 20 10
The values of a and b after swap are 10 20

Explanation:

Since the values of a and b are passed by value, the changes made in the values of the parameters inside the called function are not reflected to the calling function.

52. This is illustration of pass by reference or address
The values of a and b before swap are 10 20
The values of a and b in swap are 20 10
The values of a and b after swap are 20 10

Explanation:

Since the values of a and b are passed by reference, the changes made in the values pointed to by the parameters inside the called function are reflected to the calling function.

53. Sum is 10 and Difference is 10

Explanation:

A function can return only one value. It seems that return sum,diff; returns the value of both sum and diff. However, it is not true. In the statement return sum,diff;, the return expression sum,diff is evaluated first and then its outcome is returned. The comma operator involved in the expression guarantees left to right evaluation and returns the result of the rightmost sub-expression. Therefore, the return expression sum,diff evaluates to the result of the evaluation of diff. Hence, both the calls to function sum_diff, returns the value of diff, i.e. 10. That is why the output comes out to be Sum is 10 and Difference is 10.

54. Compilation error "Expression syntax in function main"

Explanation:

return is a statement and not an expression. It cannot be used as an operand of any operator. Writing return sum, return diff; is not valid as return statement is an operand of comma operator. It should be either return sum; return diff; or return sum, diff;.

55. Sum is 30 and Difference is 30

Explanation:

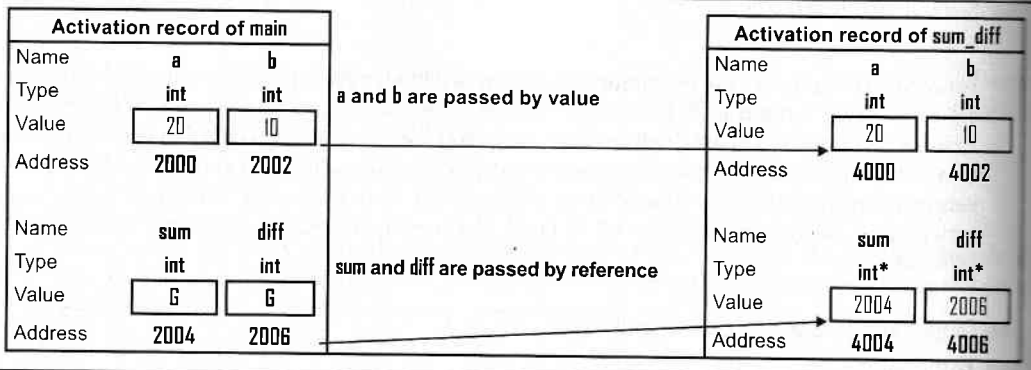
A function can have more than one return statement within its body. If more than one return statement is present inside the body of a function, only the return statement that appears first in the logical flow of control gets executed. In the given piece of code, the statement return sum; appears first in the logical flow of control. Therefore, it gets executed and the control along with the value of sum is returned to the calling function, i.e. main. The statement return diff; will never be executed and forms an unreachable part of the code. Hence, both the calls to the function sum_diff, return the value of sum, i.e. 30. That is why the output comes out to be Sum is 30 and Difference is 30.

56. Sum is 30 and Difference is 10

Explanation:

By making the use of the return statement, a function can return only one value. However, it is possible to indirectly get more than one result from a function either by using global variables➲ or pass by reference. In the given piece of code, pass by reference is used to indirectly get two outputs from the function sum_diff.

Suppose, the variables a, b, sum and diff that are local➲ to the function main are allocated at the memory locations 2000, 2002, 2004 and 2006, respectively. The parameters declared in the header of the function sum_diff are local to the function sum_diff and are allocated at separate memory locations, say 4000, 4002, 4004 and 4006 respectively. Note that the type of the variables sum and diff in the function main is int while the type of variables sum and diff in the function sum_diff is int*. The variables a and b are passed by value while the variables sum and diff are passed by reference. The passed values and the execution of statements are shown in the following figure:



G (in the above figure) means garbage.

The variables in the statement *sum=a+b; refer to the local variables of the function sum_diff. This statement places the result of addition of a and b, i.e. 30 at the memory location 2004, i.e. in the sum variable of the function main. Similarly, *diff=a-b, places the difference of a and b, i.e. 10 at the memory location 2006, i.e. in the diff variable of the function main. In this way, the function sum_diff has indirectly returned two values to the calling function, i.e. main. Thus, reference to the variables sum and diff in the function main after the execution of the function sum_diff gives 30 and 10, respectively, instead of garbage values.

Forward Reference: Local variables and global variables (Chapter 7).

57. Control is in Function2
Control is in Function1
24 24

Explanation:

The comma operator guarantees left-to-right evaluation, but the commas separating the arguments in a function call are not comma operators. If the commas separating the arguments in a function call are considered as comma operators, then no function could have more than one argument. Hence, arguments are not guaranteed to be evaluated from left to right. The order of evaluation of arguments in a function call is unspecified and is compiler dependent. In Borland TC 3.0 and TC 4.5, the evaluation takes place from right to left.

58. Control is in Function1
Control is in Funciton2
48

Explanation:

The expression fun1()+fun2() gets evaluated first and the result of its evaluation is printed. The operands of + operator are evaluated from left to right. Hence, the function fun1 is called first and then the function fun2 is called.

59. The value of factorial of 4 is 24

Explanation:

Backward Reference: Refer Section 5.3.1.1.7.3.1.1 for the answer.

60. Infinite Recursion
Infinite Recursion
Infinite Recursion
Infinite Recursion ...

Caution:

Keeps on printing 'Infinite Recursion' till the run-time stack does not overflow.

Explanation:

The given piece of code, if executed using Turbo C 3.0, keeps on printing 'Infinite Recursion' till the run-time stack does not overflow. The run-time stack overflows when a large number of activation records are stacked up and there is no memory space left for creating and stacking new activation records. Once the run-time stack overflow occurs, the program will terminate. That is why it is said that **'Infinite recursion will automatically terminate but infinite iteration will not'**. Note that in Turbo C 4.5, it is not allowed to call the function main from within the function main.

61. Size of arr in function main is 12
Size of arr in function check is 2 (In Borland Turbo C 4.5 the output will be 4)

Explanation:

arr declared inside the body of the function main is a two-dimensional array of integers having two rows and three columns. The parameter arr declared in the header of the function check_ptr as int arr[2][3] implicitly gets converted to int (*arr)[3], i.e. pointer to an integer array of size 3. That is why, the size occupied by arr in the function main is 12, and in the function check_ptr is 2 (as a pointer takes two bytes in Borland TC 3.0 irrespective of the data to which it points).

62. The result of addition is 5
The result of addition is 5

Explanation:

The declaration statement int(*ptr)(int,int); declares ptr as a pointer to a function that accepts two integers and returns an integer. The assignment statement ptr=add; assigns the starting address of the function add to the pointer ptr. The function add can be invoked by the means of pointer by either writing ptr(2,3); or (*ptr)(2,3);, where 2 and 3 are the values of the arguments to the function add.

63. The result of called function 1 is 15
The result of called function 2 is 5
The result of called function 3 is 50
The result of called function 4 is 2

Explanation:

The declaration statement int (*ptr[4])(int,int)={add,sub,mul,div}; declares ptr as an array of pointers to functions that accepts two integers and returns an integer. It also initializes the array locations with the starting addresses of the functions add, sub, mul and div. These functions are called in the loop by writing p[i](10,5), where 10 and 5 are the arguments to the functions. The functions called for the values of i: 0, 1, 2 and 3 are add, sub, mul and div, respectively. The values returned by these functions are then printed.

64. 5
-1

Explanation:

The declaration fun(int(*)(int,int); declares fun as a function that accepts a pointer to a function that accepts two integers and returns an integer. The return type of fun is not specified and by default would be int. In the function main, fun is called with add as an argument. This means that the starting address of the function add is passed as an argument to the parameter a of the function fun.

Within the body of the function fun, the expression a(2,3), calls the function pointed to by a with 2 and 3 as the arguments. Since a at present points to the function add, the function add is called with the arguments 2 and 3. The value returned by the function add, i.e. 5 is returned by the function fun. Therefore, 5 gets printed. In the next printf statement, the function fun is called with sub as the argument. The starting address of the function sub is passed as an argument to the parameter a of the function fun. The expression a(2,3), calls the function pointed to by a with 2 and 3 as the arguments. Since a now points to the function sub, the function sub is called with the arguments 2 and 3. The value returned by the function sub, i.e. -1 is returned by the function fun and is printed in the function main. Thus, the output.

65. Result of operation1 is 10
Result of operation2 is 3

Explanation:

f_returning_fps is a function that takes an integer and returns a pointer to a function that takes two integers and returns an integer. When the function f_returning_fps is invoked with argument values i=1 and j=3, it returns pointers to the functions sub and div, respectively. The returned pointers are used to invoke the respective functions with argument values 15 and 5. The invoked functions sub and div return integer values 10 and 3, respectively. These returned values are assigned to the variables res1 and res2 and are printed by the printf function.

Answers to Multiple-choice Questions

66. b 67. b. 68. b 69. c 70. b 71. c 72. b 73. b 74. c 75. b 76. a 77. c. 78. a 79. b
80. a 81. b 82. a 83. d 84. c 85. a

Programming Exercises

Program 1 | Devise a C function that checks whether a given number is prime or not and illustrate its use

PE 5-1.c

```
1  //Function to check whether a given number is prime or not
2  #include<stdio.h>
3  int prime(int no);    //←Function declaration
4  main()
5  {
6  int num;
7  printf("Enter the number to be checked:\t");
8   scanf("%d", &num);
9  if(prime(num)==0)
10     printf("Number is not prime\n");
11 else
12     printf("Number is prime\n");
13 }
14 int prime(int no)    //←Function definition
15 {
16     int i;
17     for(i=2;i<no;i++)
18         if(no%i==0)     //←Is number divisible by any number from 2 to n-1
19             return 0;   //←if yes, number is not prime, return 0
20     return 1;           //←if no, number is prime, return 1
21 }
```

Output window

```
Enter the number to be checked:  10
Number is prime
```

Output window (second execution)

```
Enter the number to be checked:  15
Number is not prime
```

Program 2 | Devise a C function that sums all the elements of an array. Illustrate its use

PE 5-2.c

```
1  //Function that sums all the elements of an array
2  #include<stdio.h>
3  int sumall(int array[], int num);    //←Function declaration
4  main()
5  {
6  int num, i, result, elements[20];
7  printf("Enter the number of elements in the array (max. 20):\t");
8   scanf("%d", &num);
9  printf("Enter the elements:\n");
10 for(i=0;i<num;i++)
11     scanf("%d",&elements[i]);
12 result=sumall(elements, num);
13 printf("The sum of all the elements of the array is %d",result);
14 }
15 int sumall(int array[], int num)    //←Function definition
16 {
17     int i,sum=0;
18     for(i=0;i<num;i++)
19         sum=sum+array[i];
20     return sum;
21 }
```

Output window

```
Enter the number of elements in the array (max. 20)    5
Enter the elements:
10 2 4 7 11
The sum of all the elements of the array is 34
```

Program 3 | Devise a C function that checks whether two matrices can be multiplied or not. If yes, multiply them. Illustrate the use of the developed function

PE 5-3.c

```
1  //Matrix Multiplication with the help of functions
2  #include<stdio.h>
3  #include<stdlib.h>
4  int mat_multiply(int mx1[][10], int m1, int n1, int mx2[][10], int m2, int n2, int mx3[][10]);
5  main()
6  {
7  int mx1[10][10], mx2[10][10], mx3[10][10]={0};
8  int m1, n1, m2, n2, i, j, indicator;
9  printf("Enter the order of matrix-1 (max. 10 by 10)\t");
10 scanf("%d %d",&m1, &n1);
11 printf("Enter the elements of matrix-1:\n");
12 for(i=0;i<m1;i++)
13 {
14     for(j=0;j<n1;j++)
15         scanf("%d",&mx1[i][j]);
16 }
17 printf("Enter the order of matrix-2 (max. 10 by 10)\t");
18 scanf("%d %d",&m2, &n2);
19 printf("Enter the elements of matrix-2:\n");
20 for(i=0;i<m2;i++)
21 {
22     for(j=0;j<n2;j++)
23         scanf("%d",&mx2[i][j]);
24 }
```

Output window

```
Enter the order of matrix-1 (max. 10 by 10)    2 3
Enter the elements of matrix-1:
1 2 3
4 5 6
Enter the order of matrix-2 (max. 10 by 10)    3 2
Enter  the elements of matrix-2:
1 2
3 4
5 6
The result of matrix multiplication is:
22 28
49 64
```

Output window (second execution)

```
Enter the order of matrix-1 (max. 10 by 10)    2 3
Enter the elements of matrix-1:
1 2 3
4 5 6
Enter the order of matrix-2 (max. 10 by 10)    2 2
Enter  the elements of matrix-2:
1 2
3 4
Matrices are not compatible for multiplication
```

(Contd...)

Line	PE 5-3.c	Output window

```
25  indicator=mat_multiply(mx1, m1, n1, mx2, m2, n2, mx3);
26  if(indicator==0)
27      printf("Matrices are not compatible for multiplication\n");
28  else
29  {
30  printf("The result of matrix multiplication is:\n");
31  for(i=0;i<m1;i++)
32  {
33      for(j=0;j<n2;j++)
34          printf("%d ",mx3[i][j]);
35      printf("\n");
36  }
37  }
38  }
39  int mat_multiply(int mx1[][10], int m1, int n1, int mx2[][10], int m2, int n2, int mx3[][10])
40  {
41      int i, j, k;
42      if(n1!=m2)
43          return 0;
44      else
45      {   for(i=0;i<m1;i++)
46              for(j=0;j<n2;j++)
47                  for(k=0;k<n1;k++)
48                      mx3[i][j]=mx3[i][j]+mx1[i][k]*mx2[k][j];
49          return 1;
50      }
51  }
```

Program 4 | Merge Sort: Given a list of n elements, arrange them in an ascending order using Merge Sort

Divide-and-conquer is an algorithm design strategy. It works as follows:

1. It checks whether the given instance of problem P is small or not. The given instance is said to be small if it can be easily solved.
2. If the given instance is small, solve it and return the solution. Else, follow the next step.
3. Divide the given instance of problem into smaller sub-problems P_1, P_2, P_3,....P_n.
4. Solve the smaller sub-problems recursively by applying divide-and-conquer strategy.
5. Combine the solutions for sub-problems P_1, P_2, P_3,....P_n into a solution for P.

Merge Sort is a sorting algorithm that is based on divide-and-conquer strategy. Merge sort works as follows:

1. The size of the given list is determined.
2. If it is 0 or 1 (i.e. it is a small problem), then the list is already sorted. Otherwise, for the lists of the size greater than 1, follow the next step.
3. The unsorted list is divided into two halves of approximately equal size (i.e. division of problem P into P_1 and P_2).
4. The divided sub-lists are recursively sorted by applying Merge Sort.
5. The sorted sub-lists are merged back into one sorted list.

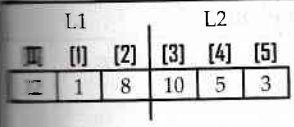
For example, Merge sort sorts the given unsorted list L as follows:

L

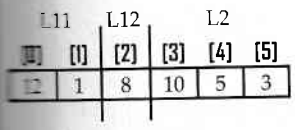
[0]	[1]	[2]	[3]	[4]	[5]
12	1	8	10	5	3

//←The list L is divided at midpoint into two halves L1 and L2

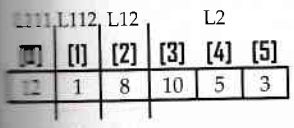
(Contd...



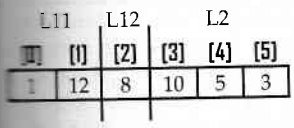
//←The list L1 is further divided at midpoint into two halves L11 and L12



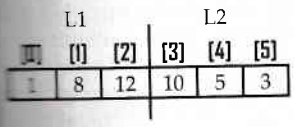
//←The list L11 is further divided at midpoint into two halves L111 and L112



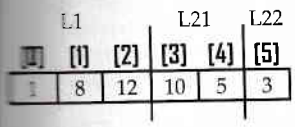
//←The lists L111 and L112 are of size 1 and are already sorted. They are merged to form the sorted list L11



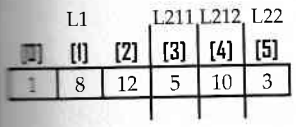
//←List L12 is of size 1 and is already sorted. The list L11 is also sorted. The sorted lists L11 and L12 are merged to form the sorted list L1



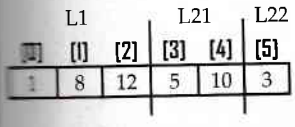
//←The list L2 is divided at midpoint into two halves L21 and L22



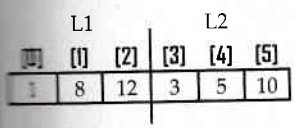
//←The list L21 is further divided at midpoint into two halves L211 and L212



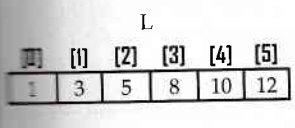
//←The lists L211 and L212 are of size 1 and are already sorted. They are merged to form the sorted list L21



//←List L22 is of size 1 and is already sorted. The list L21 is also sorted. The sorted lists L21 and L22 are merged to form the sorted list L2



//←Both the lists L1 and L2 are sorted. They are merged to form the sorted list L



(Contd...)

Line	PE 5-4.c
1	//Merge Sort
2	#include<stdio.h>
3	int mergesort(int list[], int high, int low);
4	int merge(int num[], int low, int mid, int high);
5	main()
6	{
7	int list[20], num, i;
8	printf("Enter the number of elements (max. 20)\t");
9	scanf("%d",&num);
10	printf("Enter the elements:\n");
11	for(i=0;i<num;i++)
12	scanf("%d",&list[i]);
13	mergesort(list, 0, num-1);
14	printf("After sorting, elements are:\n");
15	for(i=0;i<num;i++)
16	printf("%d\n",list[i]);
17	}
18	int mergesort(int list[], int low, int high)
19	{
20	int mid;
21	if(low<high)
22	{
23	mid=(low+high)/2;
24	mergesort(list, low, mid);
25	mergesort(list, mid+1, high);
26	merge(list, low, mid, high);
27	}
28	}
29	int merge(int list[], int low, int mid, int high)
30	{
31	int temp[20], k;
32	int h=low, i=low, j=mid+1;
33	while((h<=mid) && (j<=high))
34	{
35	if(list[h]<=list[j])
36	{
37	temp[i]=list[h];
38	h=h+1;
39	}
40	else
41	{
42	temp[i]=list[j];
43	j=j+1;
44	}
45	i=i+1;
46	}
47	if(h>mid)
48	for(k=j;k<=high;k++)
49	{
50	temp[i]=list[k];

Output window

```
Enter the number of elements(max. 20)  6
Enter the elements:
12
10
5
-3
14
2
After sorting, elements are:
-3
2
5
10
12
14
```

(Contd...)

```
            i++;
        }
    else
        for(k=h;k<=mid;k++)
        {
            temp[i]=list[k];
            i++;
        }
    for(k=low;k<=high;k++)
        list[k]=temp[k];
    return 0;
}
```

Program 5 | Quick Sort: Given a list of n elements, arrange them in ascending order using Quick sort

Quick Sort is another efficient sorting algorithm that is based on the divide-and-conquer strategy. In Merge Sort, the list was divided at its midpoint into sub-lists that were independently sorted and later merged. In Quick Sort, the division into two sub-lists is made so that the sorted sub-lists do not need to be merged later. This can be accomplished by picking up an element in the list known as the **pivot element**. The elements of the list are rearranged, so that all the elements that are less than the pivot element come towards the left of the pivot element and all the elements greater than the pivot element come after it (i.e. towards its right). This rearrangement is known as **partitioning**. After partitioning, the pivot element is at its final position. The sub-list of lesser elements (i.e. towards the left of pivot element) and greater elements (i.e. towards the right of pivot element) are recursively sorted by using Quick Sort.

Partitioning:

C.A.R. Hoare, the developer of the Quick Sort algorithm, used the following approach to partition a list:

1. Consider the first element of the list as the pivot element.
2. Rearrange the elements of the list so that the pivot element is moved to its final position. This rearrangement can be done as follows:

a. Suppose the given list is:

[0]	[1]	[2]	[3]	[4]	[5]
12	1	8	10	5	3

b. At the end of the list, append an element that is greater than all the elements present in the list.

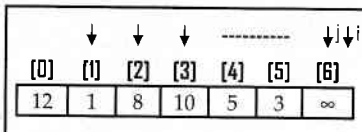
[0]	[1]	[2]	[3]	[4]	[5]	[6]
12	1	8	10	5	3	∞

c. The first element of the unsorted list is the pivot element. Take two pointers, say i and j. The pointer i points to the pivot element and the pointer j points to the appended largest element.

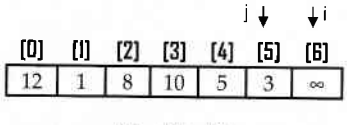
↓i						↓j
[0]	[1]	[2]	[3]	[4]	[5]	[6]
12	1	8	10	5	3	∞

d. Increment the pointer i, till a value greater than the pivot element is encountered. Decrement the pointer j, till a value smaller than the pivot element is encountered. If the pointer i is towards the left of pointer j (i.e. i<j), swap the values pointed to by them else swap the value pointed to by the pointer j with the pivot element. After this process, the pivot element will be at its final position.

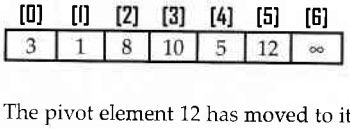
(Contd...)



//←The pointer i is moved till element greater than the pivot element is encountered. Since there is no element greater than the pivot element the pointer i will stop at the appended largest element. If ∞ would have not been appended, the pointer i would have strayed into garbage field.



//←The pointer j points to the element lesser than the pivot element.



//←Since pointer j is towards the le of pointer i, swap the pivot element with the element pointed to by j. The pivot element comes to its final position.

The pivot element 12 has moved to its final position. It divides the list into two sub-lists. One containing the elements lesser than the pivot element and one containing elements greater than the pivot element (empty in this case). This clearly indicates that the divided sub-list may have a significantly different size. The divided sub-lists are recursively sorted by using Quick Sort.

Line | **PE 5-5.c**

```
1  //Quick Sort
2  #include<stdio.h>
3  int quicksort(int list[], int high, int low);
4  int partition(int num[], int low, int high);
5  int swap(int list[], int i, int j);
6  main()
7  {
8      int list[21], num, i;
9      printf("Enter the number of elements (max. 20)\t");
10     scanf("%d",&num);
11     printf("Enter the elements:\n");
12     for(i=0;i<num;i++)
13         scanf("%d",&list[i]);
14     list[num]=10000;
15     quicksort(list, 0, num-1);
16     printf("After sorting, elements are:\n");
17         for(i=0;i<num;i++)
18             printf("%d\n",list[i]);
19 }
20 int quicksort(int list[], int low, int high)
21 {
22     int pos;
23     if(low<high)
24     {
25         pos=partition(list, low, high+1);
26         quicksort(list, low, pos-1);
27         quicksort(list, pos+1, high);
28     }
29 }
30 int partition(int list[], int low, int high)
```

Output window

```
Enter the number of elements(max. 20)  5
Enter the elements:
12
10
5
-3
14
2
After sorting, elements are:
-3
2
5
10
12
14
```

Remark:

- In the given code it is assumed that the elements entered in the array will be less than 10000

(Contd...

```
{
    int v=list[low], i=low, j=high;
    do
    {
        do
        {
            i++;
        }while(list[i]<v);
        do
        {
            j--;
        }while(list[j]>v);
        if(i<j)
            swap(list, i, j);
    }while(i<j);
    list[low]=list[j];
    list[j]=v;
    return j;
}
int swap(int list[], int i, int j)
{
    int temp;
    temp=list[i];
    list[i]=list[j];
    list[j]=temp;
    return 0;
}
```

Program 6 | Binary search: Given a list of n elements arranged in ascending order and a key, find whether the given key exists in the list or not. If it exists, print its position in the list

Binary search is an efficient searching algorithm based on the divide-and-conquer strategy. It is based on the assumption that the elements of the list are arranged in an ascending order. Similar to the linear search, it works by comparing the key with the elements of the list, but with a difference in the pattern of making comparisons.

In the binary search, initially the key is compared with the element present at the middle position of the list. If both are equal, the key is found and the search is finished. If the key is less than the middle element, search the key in the list present towards the left of the middle element. If the key is greater than the middle element, search the key in the list present towards the right of the middle element.

Line	PE 5-6.c	Output window
1	//Binary Search	Enter the number of elements(max. 20) 6
2	#include<stdio.h>	Enter the elements in ascending order:
3	int binarysearch(int list[], int low, int high, int key);	10
4	main()	15
5	{	32
6	int list[20], num, i, key, low, high, index;	48
7	printf("Enter the number of elements (max. 20)\t");	92
8	scanf("%d",&num);	128
9	printf("Enter the elements in ascending order:\n");	Enter the key that you want to search 48
10	for(i=0;i<num;i++)	48 exists at location no. 4

(Contd...)

```
11      scanf("%d",&list[i]);          //←Read elements in the list
12    printf("Enter the key that you want to search\t");
13    scanf("%d",&key);              //←Read the key to be searched
14    index=binarysearch(list, 0, num-1, key);
15    if(index==-1)
16        printf("%d does not exist in the list",key);
17    else
18        printf("%d exists at location no. %d\n",key, index+1);
19    }
20    int binarysearch(int list[], int low, int high, int key)
21    {
22        int  mid;
23        if(low==high)     //←if low==high, there is only one element
24        {
25            if(list[low]==key)     //←if that element is equal to key
26                return low;        //←return its index
27            else                   //←else key is not present in the list
28                return -1;         //←return -1 as it is  not a valid index value
29        }
30        else
31        {
32            mid=(low+high)/2;    //←middle position is found
33            if(list[mid]==key)     //←if element at middle position=key
34                return mid;        //←return the index of middle location
35            else if(list[mid]>key) //←if key<middle element, search left portion of the list
36                    return binarysearch(list, low, mid-1, key);
37                else               //←search the right portion of the list
38                    return binarysearch(list, mid+1, high, key);
39        }
40    }
```

Output window (second execution)

```
Enter the number of elements(max. 20)  5
Enter the elements in ascending order
10
15
32
48
92
128
Enter the key that you want to search  50
50 does not exist in the list
```

Test Yourself

1. Fill in the blanks in each of the following:
   a. _____help in modularizing a program into smaller simple parts.
   b. The execution of a C program always begins with function _____.
   c. The expressions that appear within the parentheses of a function call are known as _____.
   d. The two ways of passing arguments to a function are _____ and _____.
   e. The variables declared in the parameter declaration list in the function header are known as _____.
   f. The first argument to the printf function should be of _____ type.
   g. The return type of each math library function is _____ .
   h. The return type of a function cannot be _____.
   i. _____ is a special case of recursion in which the last operation of a function is a recursive call.
   j. By default, the return type of a function is _____.
   k. Execution of each function requires a separate _____.
   l. The activation records for all of the active functions are stored in the region of memory called _____.
   m. The part of recursion in which a number of activation records are created and piled up is known as _____.

2. State whether each of the following is true or false. If false, explain why.
   a. C is a strongly typed language.
   b. main is a library-defined function.
   c. There can be only one return statement within a function body.
   d. printf is an example of a variable argument function.
   e. The function designator implicitly refers to the starting address of the function.
   f. The return statement is used to terminate the execution of a program.
   g. A function can be defined within the body of another function, and the function defined within another function is known as nested function.
   h. Directly recursive functions are also known as mutually recursive functions.
   i. A function need not be declared, if it is defined before it is called.
   j. The shorthand declaration of parameters in the parameter list is not allowed.
   k. One of the uses of function prototype is in type checking.
   l. If the arguments are passed by reference, the changes made in the values pointed to by the formal parameters in the called function are reflected to the calling function.
   m. A function can return only one value.

3. Programming exercises:
   a. Write a C function that checks whether a given number is even or odd. Illustrate its use.
   b. Write a C function that checks whether a given number is perfect or not. Illustrate its use.
   c. Write a recursive C function to find the sum of individual digits of a given positive integer number.
   d. Write a C function that finds the reverse of a given number.
   e. Write a C function that checks whether a given number is a palindrome or not.
   f. Write a C function that checks whether a given number is an Armstrong number or not.
   g. Write an iterative C function to print the first n terms of a Fibonacci series. Get the value of n from the user.
   h. Write a recursive C function to print the first n terms of a Fibonacci series. Get the value of n from the user. Illustrate its use.

i. Write an iterative C function that finds the value of x^n. Get the values of x and n from the user. Illustrate its use.

j. Write a recursive C function that finds the value of x^n. Get the values of x and n from the user. Illustrate its use.

k. Write a recursive C function that implements linear search. Given a list of n elements and a key. Using the developed function, check whether the given key exists in the list or not. If yes, print the position at which it exists in the list.

l. Write an iterative C function that finds the factorial of a given integer. Use this function to find ${}^{n}_{r}C = \frac{n!}{r!(n-r)!}$

m. Write a recursive C function that finds the factorial of a given integer. Use this function to find ${}^{n}_{r}C = \frac{n!}{r!(n-r)!}$

n. Write a C function to evaluate the following series. Use a function to compute the factorials. Get the value x and the number of terms in the series from the user:

i. $\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots \infty$

ii. $\cosh(x) = 1 + \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots \infty$

iii. $\exp(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots \infty$

iv. $e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots + \frac{1}{n!}$

o. Write a C function that finds the sum of all the elements of a matrix. Illustrate the use of this function.

p. Write a C function that checks whether a given matrix is symmetric or not. Illustrate its use.

q. Write a C function that finds the sum of elements of the principal diagonal of a matrix. Illustrate the use of this function.

r. Write a C program that extracts the lower-triangular matrix from a square matrix. Illustrate the use of the developed function in a program.

s. Write a C function that finds the largest and the smallest element in a matrix. Illustrate the use of the developed function in a program.

t. Write a C function that swaps the contents of two one-dimensional arrays. Do not use any additional storage space. Illustrate the use of the developed function in a program.

u. Given n boolean variables $x_1, x_2, x_3, \ldots x_n$. We wish to print all the possible combinations of the truth values that they can assume. For instance, if n is equal to 2, there are four possibilities 11, 01, 10 and 11. Write a C program to accomplish this task.

v. Write a C program to implement ternary search. The ternary search works on the following strategy:
Given a sorted list of n elements in ascending order. First, test the element at the location n/3 for equality with the given key x. If they are found to be equal, print that the given key is found at the location n/3, else compare it with the element at the location 2n/3. If they are found to be equal, print that the given key is found at location 2n/3, else reduce the size of the list to one-third and search the given key in the reduced list.

6

STRINGS AND CHARACTER ARRAYS

Learning Objectives

In this chapter, you will learn about:

- Strings
- How strings are represented in C language
- The usage of character arrays to store strings
- Null character and its importance in string representation
- How to read strings from the keyboard
- How to print strings on the screen
- Various string operations like copy, compare, concatenate, etc.
- String library functions
- How to store and work with a list of strings
- Command line arguments

6.1 Introduction

The character string is one of the most useful and important data types. You have used the character strings all the way in the previous chapters, but there is still much to learn about them. The C string library provides a wide range of functions for strings like reading, writing, copying, comparing, combining, searching, etc. This chapter will add these capabilities to your programming skills.

6.2 Strings

A **character string literal constant** or just a **string literal** is a sequence of zero or more characters enclosed within double quotes. For example, "GOD Bless!!" is a string literal constant. Knowingly or unknowingly, you have used strings in abundance with the printf function in previous chapters.

The important points about the string literal constants are as follows:

1. String literals are enclosed within double quotes, whereas character literals are enclosed within single quotes, e.g. "A" is a string literal constant while 'A' is a character literal constant.
2. The used double quotes are not part of the string literal and are used only to delimit it.
3. Every string literal constant is automatically terminated by the **null character**, i.e. '\0'.

> The character constant with an ASCII value of zero is known as a **null character** and is written as '\0'.

4. Like other literal constants, string literal constants are also stored in the memory. The characters enclosed within double quotes and the terminating null character are stored in the contiguous memory locations in a similar manner as arrays are stored in the memory. Thus, a string literal constant "GOD Bless!!" will be stored in the memory as shown in Figure 6.1.

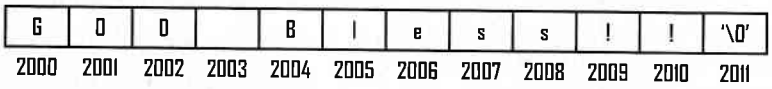


Figure 6.1 | Storage of string literal constant "GOD Bless!!"

5. Unlike other literal constants, the amount of the memory space required for storing a string literal constant is not fixed and depends upon the number of characters present in a string literal.
6. The number of bytes required to store a string literal constant is one more than the number of characters present in it. The additional byte is required for storing the terminating null character. For example, the memory required to store the string literal "xyz" is 4 bytes. The code snippet in Program 6-1 illustrates this fact.

Line	Prog 6-1.c	Output window
1	//Memory requirement of string literal	Memory requirement of "xyz" is 4 bytes **Remarks:** • Escape sequence \" is used to print double quotes • The additional byte is required to store the terminating null character
2	#include<stdio.h>	
3	main()	
4	{	
5	printf("Memory requirement of \"xyz\" is %d bytes",sizeof("xyz"));	
6	}	

Program 6-1 | A program to illustrate that the memory space required by a string literal constant is one more than the number of characters in it

7. The **length of a string** is defined as the number of characters present in it. The terminating null character is not counted while determining the length of a string. For example, the length of the string literal "xyz" is 3. The code snippet in Program 6-2 verifies this fact.

Line	Prog 6-2.c	Output window
1	//Length of string literal	Length of string literal "xyz" is 3 characters **Remarks:** • The terminating null character is not counted while determining the length of a string • strlen is a string library function that determines the length of a string • The prototype of the strlen function is present in the header file string.h
2	#include<stdio.h>	
3	//string.h header file is to be included for using string library functions	
4	#include<string.h>	
5	main()	
6	{	
7	printf("Length of string literal \"xyz\" is %d characters",strlen("xyz"));	
8	}	

Program 6-2 | A program to find the length of a string

8. A string literal constant of zero length is known as an **empty string**. The empty string is written as "", i.e. no character enclosed within double quotes. Although an empty string is of zero length, it still takes 1 byte in the memory for the storage of a null character.
9. In C language, **string type** is not separately available, and **character pointers** are used to represent strings. Thus, the type of string literal (e.g. "xyz") is **const char***. The constant pointer refers to the address of the first element of the string. The strings represented and interpreted in this way are known as **C-style character strings**. The code snippet in Program 6-3 illustrates that a string literal decomposes into a pointer (const char*) pointing to the first character of the string.

Line	Prog 6-3.c	Output window
1	//C-style character strings are represented by const char*	The first character of string literal "xyz" is x The second character of string literal "xyz" is y **Remarks:** • The type of string literals is const char* • "xyz" refers to the address of the first element of the string, i.e. the address of x • Hence, dereferencing "xyz" outputs x
2	#include<stdio.h>	
3	main()	
4	{	
5	printf("The first character of string literal \"xyz\" is %c\n",*"xyz");	
6	printf("The second character of string literal \"xyz\" is %c",*("xyz"+1));	
7	}	

Program 6-3 | A program to illustrate that the string literal constant refers to the address of its first element

10. Since a string literal constant refers to a constant character pointer and does not have a modifiable l-value, only the operations that can be applied on constant pointers can be applied on C-style character strings. The application of any other operator on string literals that cannot be applied on constant pointers leads to 'L-value required' compilation error. The code snippet in Program 6-4 illustrates this fact.

Line	Prog 6-4.c	Output window
1	//String literal refers to constant character pointer	Compilation error "L-value required"
2	#include<stdio.h>	**Remarks:**
3	main()	• The expression *"xyz"++ will be interpreted as *("xyz"++)
4	{	• The application of the post-increment operator on "xyz" leads to the compilation error as "xyz" does not have a modifiable l-value
5	printf("The first character of string literal \"xyz\" is %c",*"xyz"++);	
6	}	

Program 6-4 | A program to illustrate that a string literal constant refers to a constant pointer and does not have a modifiable l-value

11. Since C-style character string is of const char* type, it can be assigned to or initialized to a character pointer variable. The following statements are valid:

```
char *string="Strings!!!";
string="Trings!!!";
```

12. Adjacent string literal constants are concatenated. This concatenation is carried out during the preprocessing phase. The code snippet in Program 6-5 illustrates this fact.

Line	Prog 6-5.c	Output window
1	//Adjacent string literal constants get concatenated	GOD Bless us!!!
2	#include<stdio.h>	**Remark:**
3	main()	• Adjacent string literal constants in line number 5 are concatenated and then printed
4	{	
5	printf("GOD Bless " "us" "!!!");	
6	}	

Program 6-5 | A program to illustrate that the adjacent string literal constants get concatenated

Forward Reference: Preprocessor directives (Chapter 8).

6.3 Character Arrays

An integer variable can store the value of an integer constant. For example, statement int a=10; creates a variable a to store an integer constant 10. Similarly, float variables can store floating point constants, and character variables can be used to store character constants. Now, the question that arises here is: 'Can we create a variable that can be used to store a string literal constant?' The answer to this question is YES! We can create variables of type char[] (i.e. **character arrays**) to store string constants.

The general form of a **string variable** or **a character array** declaration is:

<storage_class_specf><type_qualifier><type_modifier>**char identifier[**<size_specifier>**]**<=initialization_list OR string literal>;

The important points about string variable declarations are as follows:

1. The terms enclosed within angular brackets (i.e. <>) are optional and might not be present in a declaration statement. The terms shown in **bold** are the mandatory parts of a string variable declaration.
2. Since a string variable is a character array, all the syntactic rules discussed in Chapter 4 for declaring arrays are applicable for declaring string variables as well.
3. The size specification is optional if a string variable is explicitly initialized.
4. The string variable or character array can be initialized in two different ways:
   a. **By using string literal constant:** In the declaration statement `char str[6]="Hello";` the character array or string variable `str` is initialized with a string literal constant "Hello". It will be stored in the memory as shown in Figure 6.2.
   b. **By using initialization list:** The alternate way to initialize a character array is by using a list of character initializers. The declaration statement `char str[6]={'H','e','l','l','o','\0'};` initializes the locations of the character array `str` with character initializers. The character array `str` will be stored in the memory in the same way as shown in Figure 6.2.

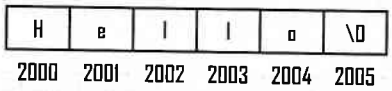


Figure 6.2 | Two different ways to initialize a string variable or a character array

When a character array is initialized with a list of character initializers, the terminating null character is to be explicitly placed but when it is initialized with a string literal constant, the terminating null character is automatically placed (if the size of the character array is one more than the length of the string literal constant).

Forward Reference: Storage class specifier (Chapter 7).

6.4 Reading Strings from the Keyboard

The user can enter strings and store them in character arrays at the run time in a similar manner as the string literal constants can be stored in the character arrays at the compile time. The methods that can be used to read strings from the user at the run time are as follows:

1. **Using scanf function:** The `scanf` function with `%s` format specification can be used to read a string from the user and store it in a character array. The code snippet in Program 6-6 illustrates the use of the `scanf` function to read a string from the user.

Line	Prog 6-6.c	Output window
1	//Reading strings using the scanf function	Enter your name Sam Your name is Sam **Remark:** • The scanf function automatically terminates the input string with a null character, and therefore the character array should be large enough to hold the input string plus the terminating null character
2	#include<stdio.h>	
3	main()	
4	{	
5	char name[20];	
6	printf("Enter your name\t");	
7	scanf("%s",name);	
8	printf("Your name is %s",name);	
9	}	

Program 6-6 | A program to illustrate the use of scanf function to read a string from the user at the run time

The important points about the use of scanf function for reading strings are as follows:

a. The scanf function with %s specifier reads all the characters up to, but not including, the white-space character. For example, in Program 6-6, instead of entering the first name, enter the full name, e.g. "Sam Mine". Even on entering the full name, the output of the program would be "Your name is Sam". This happens because the scanf function reads the characters only up to the first white-space character.
Thus, scanf function with %s specifier can be used to read single word strings like "Sam" but cannot be used to read multi-word strings like "Sam Mine".

b. The scanf function can be used to read a specific number of characters by specifying the field width. The code snippet in Program 6-7 illustrates the use of a field width specifier.

Line	Prog 6-7.c	Output window
1	//Field width specifier and scanf function	Enter your name Samuel Your name is Sam **Remarks:** • If the length of the entered string is more than the specified field width, the number of characters read will be at most equal to the field width • The scanf function reads all characters up to, but not including, the white-space character even if the value of field width specification is more than the position of first white-space character
2	#include<stdio.h>	
3	main()	
4	{	
5	char name[20];	
6	printf("Enter your name\t");	
7	scanf("%3s",name);	
8	printf("Your name is %s",name);	
9	}	

Program 6-7 | A program to illustrate the use of a field width specifier and the scanf function

c. The scanf function can also be used to read selected characters by making use of search sets. A **search set** defines a set of possible characters that can make up the string. The rules to write search sets are as follows:

i. The possible set of characters making up the search set is enclosed within square brackets, e.g. [abcd]. The scanf function reads all the characters up to but not including the one that does not appear in a search set. If a search set [abcd] is used, the scanf function reads the input characters and stops when a character except a, b, c or d is encountered. The code snippet in Program 6-8 illustrates this fact.

Line	Prog 6-8.c	Output window
1	//Search set and scanf function	Enter your name daman Your name is da **Remarks:** • Search sets are case sensitive • If the specified search set is [abcd] and the entered string is Daman, no character will be read, as the character D does not belong to the search set
2	#include<stdio.h>	
3	main()	
4	{	
5	char name[20];	
6	printf("Enter your name\t");	
7	scanf("%[abcd]",name);	
8	printf("Your name is %s",name);	
9	}	

Program 6-8 | A program to illustrate the use of a search set and the scanf function

ii. If the first character in the bracket is a caret (i.e. ^), the search set is inverted to include all the characters (even white-space characters) except those between the brackets. For example, the search set [^abcd] searches the input for any character except a, b, c and d. The scanf function reads the input characters and stops when the characters a, b, c or d are encountered. The code snippet in Program 6-9 illustrates this fact.

Line	Prog 6-9.c	Output window
1	//Inverted search set and scanf function	Enter your name Neha Your name is Neh **Caution:** • The input will only terminate when any character specified within the brackets is encountered • Matching process is case sensitive • Re-execute the code and enter the name in uppercase, i.e. NEHA. The input will not terminate even on pressing enter. Enter character 'a' and then press enter. The input will terminate
2	#include<stdio.h>	
3	main()	
4	{	
5	char name[20];	
6	printf("Enter your name\t");	
7	scanf("%[^abcd]",name);	
8	printf("Your name is %s",name);	
9	}	

Program 6-9 | A program to illustrate the use of inverted search set and the scanf function

The inverted search set can be used with the scanf function to **read a line of text**. The code snippet in Program 6-10 illustrates the use of an inverted search set to read a line of text.

Line	Prog 6-10.c	Output window
1	//Reading a line of text using inverted search set	Enter a line of text: We can change our destiny!! The text you entered is: We can change our destiny!! **Remark:** • The inverted search set [^\n] can be used to read the characters till the new line character is encountered
2	#include<stdio.h>	
3	main()	
4	{	
5	char line[50];	
6	printf("Enter a line of text:\n");	
7	scanf("%[^\n]",line);	
8	printf("The text you entered is:\n%s",line);	
9	}	

Program 6-10 | A program to illustrate the use of an inverted search set to read a line of text

iii. The search set can be used for including the characters that lie within a particular range. For example, the search set %[d-f] searches the input for any character that lies in the range d to f, i.e. d, e and f.

d. The scanf function automatically terminates the input string with a null character and therefore the character array should be large enough to hold the input string plus the terminating null character.

e. It is not mandatory to use ampersand, i.e. address-of operator (&) with string variable names while reading strings using the scanf function. The reason behind this relaxation is that the scanf function requires an l-value as an argument where it can store the input. Since the string variable is a character array and the name of an array refers to the address of the first element of the array, the string variable name itself refers to the l-value. However, if an address-of operator is used with the string variable name, there will be no problem since it also refers to the same address. The code snippet in Program 6-11 illustrates this fact.

Line	Prog 6-11.c
1	//Usage of address-of operator with
2	//string variable is not mandatory
3	#include<stdio.h>
4	main()
5	{
6	char name[5];
7	printf("Enter your name\t");
8	scanf("%s",name);
9	printf("Your name is %s\n",name);
10	printf("Enter your name again\t");
11	scanf("%s",&name);
12	printf("Your name is %s\n",name);
13	}

Memory contents

name

A	j	a	y	\0
4000	4001	4002	4003	4004

Output window

```
Enter your name    Ajay
Your name is Ajay
Enter your name again    Ajay
Your name is Ajay
```

Remarks:

- Usage of address-of operator while using a string variable with the scanf function is not mandatory
- Both name and &name refer to the same memory address, i.e. 4000

Remember:

- The difference between name and &name is that the type of name is char* while that of &name is char(*)[5]

Program 6-11 | A program to illustrate that the usage of address-of operator with a string variable is not mandatory

2. **Using getchar function:** The getchar function is used to read a character from the terminal, i.e. keyboard. The prototype of the getchar function is int getchar(void); and is available in the stdio.h header file. The getchar function reads a character from the keyboard and returns the ASCII code of the read character. Since a string is a sequence of characters, the getchar function can be called repeatedly to read a string. The code snippet in Program 6-12 illustrates the use of the getchar function to read a string.
3. **Using gets function:** Another convenient way to accept a string from the user at the run time is by using the gets library function. The prototype of the gets function is char* gets(char*); and is available in the stdio.h header file. The gets function accepts a character array or a character pointer as an argument, reads characters from the keyboard until a new line character is encountered, stores them in a character array or

in the memory location pointed by the character pointer, appends a null character to the string and returns the starting address of the location where the string is stored. The code snippet in Program 6-13 illustrates the use of the gets function.

Line	Prog 6-12.c	Output window
1	//Iterative use of getchar function to read a string	Enter a line of text:
2	#include<stdio.h>	We can change our destiny!!
3	main()	The text you entered is:
4	{	We can change our destiny!!
5	char ch, line[50];	
6	int loc=0;	
7	printf("Enter a line of text:\n");	
8	while((ch=getchar())!='\n')	
9	line[loc++]=ch;	
10	line[loc]='\0';	
11	printf("The text you entered is:\n%s",line);	
12	}	

Program 6-12 | A program to illustrate the use of the getchar function to read a string

Line	Prog 6-13.c	Output window
1	//Use of gets function to read a string	Enter name of a programming language
2	#include<stdio.h>	Visual Basic
3	main()	First programming language is Visual Basic
4	{	Enter name of another programming language
5	char plang[50];	Visual C#
6	printf("Enter name of a programming language\n");	Second programming language is Visual C#
7	gets(plang);	**Remarks:**
8	printf("First programming language is %s\n",plang);	• The gets function can be used to read multi-word strings.
9	printf("Enter name of another programming language\n");	• Since the gets function returns the pointer to the input string, it can be used as an argument within the printf function (as done in line number 10)
10	printf("Second programming language is %s\n",gets(plang));	
11	}	

Program 6-13 | A program to illustrate the use of the gets function to read a string

The important points about the gets function are as follows:

a. Unlike the scanf function, the gets function reads the entire line of text until a new line character is encountered and does not stop upon encountering any other white-space character.
b. Thus, the gets function is suited for reading multi-word strings.

The important points about the input functions mentioned above are as follows:

a. The input functions are categorized into **buffered input functions** and **unbuffered input functions.**
b. In buffered input, the input given is kept in a temporary memory area known as the **buffer** and is transmitted to the program when the Enter key is pressed. The pressed Enter key is also transmitted to the program in the form of a new line character, which the program must handle.

c. In unbuffered input, the given input is immediately transferred to the program without waiting for the Enter key to be pressed.
d. The difference between buffered and unbuffered input is depicted in Figure 6.3.

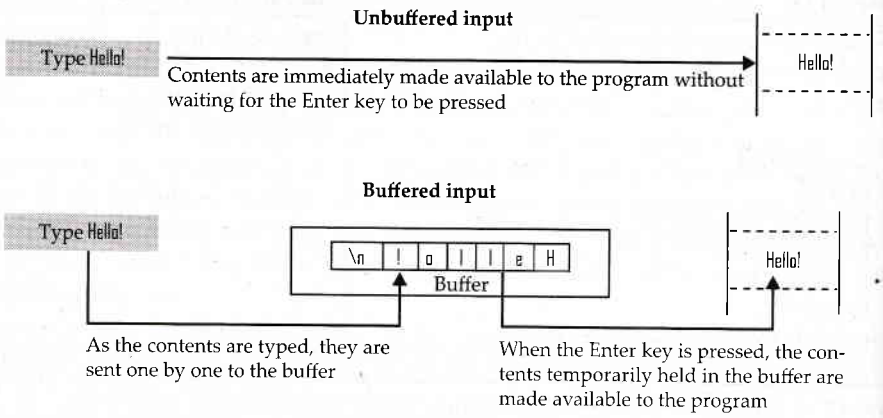


Figure 6.3 | Unbuffered and buffered input

e. The examples of buffered input functions are scanf, getchar and gets function.
f. The examples of unbuffered input functions are getch and getche function.
g. Program 6-12 can be rewritten using the unbuffered input function getche as in Program 6-14.

Line	Prog 6-14.c	Output window
1	//Iterative use of getche function to read a string	Enter a line of text:
2	#include<stdio.h>	We can change our destiny!!
3	#include<conio.h>	The text you entered is:
4	main()	We can change our destiny!!
5	{	**Remarks:**
6	char ch, line[50];	• The prototype of the getche function is present in the header file conio.h
7	int loc=0;	• The sentinel value to be used for unbuffered input functions like getche is '\r', i.e. carriage return character instead of '\n', i.e. new line character that is used for buffered input functions
8	printf("Enter a line of text:\n");	
9	while((ch=getche())!='\r')	
10	line[loc++]=ch;	
11	line[loc]='\0';	
12	printf("\nThe text you entered is:\n%s",line);	
13	}	

Program 6-14 | A program to illustrate the use of the getche function to read a string

Forward Reference: Refer Question number 15 and its answer to know how a program can handle the transmitted new line character.

6.5 Printing Strings on the Screen

The methods that can be used to print strings on the screen are as follows:

1. **Using printf function:** The printf function can be used to print a string literal constant, the contents of a character array and the contents of the memory locations pointed by a character pointer on the screen in two different ways:
   a. **Without using format specifier:** The printf function can print strings onto the screen without using any format specifier. The code snippet in Program 6-15 illustrates this use.

Line	Prog 6-15.c	Output window
1	//Printing a string with the help of printf function without using any	HelloDearReaders!!
2	//format specifier	**Remarks:**
3	#include<stdio.h>	• The first argument of the printf function must be of type const char*
4	main()	• A string literal constant and a string variable name refer to const char*
5	{	• Hence, the usage of the printf function as done in line numbers 8, 9 and 10 is perfectly valid
6	char str[20]="Readers!!"; //Array holding string	
7	char* ptr="Dear"; //Character pointer pointing to a string	
8	printf("Hello"); // Printing string literal constant	
9	printf(ptr) // Printing string pointed to by a character pointer	
10	printf(str); // Printing contents of character array	
11	}	

Program 6-15 | A program to illustrate the use of the printf function without a format specifier to print strings

The important points about this type of usage are as follows:

i. The first argument of the printf function must be of const char* type. Since the string variable name and the string literal constant implicitly decompose into const char*, this type of usage is perfectly valid.

ii. This type of usage however has a limitation that the contents of only one character array or the contents pointed by only one character pointer can be printed at a time.

   b. **Using %s format specifier:** The second way to print the strings on the screen is by using the printf function along with the %s format specifier. The code snippet in Program 6-16 illustrates this use.

Line	Prog 6-16.c	Output window
1	//Printing strings by using printf function along with %s	HelloDearReaders!!
2	//format specifier	**Remarks:**
3	#include<stdio.h>	• %s specifier is used to print a string literal, the contents of a character array and a string literal pointed to by a character pointer
4	main()	• Two or more strings can be printed by a single call to the printf function having multiple %s specifiers
5	{	
6	char str[20]="Readers!!";	
7	char* ptr="Dear";	
8	printf("%s%s%s","Hello", ptr,str);	
9	}	

Program 6-16 | A program to illustrate the use of the printf function the along with %s specifier to print strings

This type of usage has an advantage that two or more strings can be printed by a single call to the printf function having multiple %s specifiers.

2. **Iteratively printing a string's constituent characters:** A string can be printed by iteratively printing its constituent characters. They can be printed either by using the putchar function or by using the putch function. The prototype of the putchar function is int putchar(int c); and is present in the header file stdio.h. The prototype of the putch function is int putch(int c); and is present in the header file conio.h. The code snippet in Program 6-17 prints the strings by using these functions.

Line	Prog 6-17.c	Output window
1	//Printing string by iteratively printing its constituent characters	HelloDearReaders!!
2	#include<stdio.h>	**Remark:**
3	#include<conio.h>	• A character can also be printed by using the printf function and the %c format specifier as shown in line number 10
4	main()	
5	{	
6	char str[20]="Hello";	
7	char *ptr="Dear";	
8	int i=0, j=0 ;	
9	while(str[i]!='\0')	
10	printf("%c",str[i++]);	
11	while(*ptr!='\0')	
12	putch(*ptr++);	
13	while(*("Readers!!"+j)!='\0')	
14	{	
15	putchar(*("Readers!!"+j));	
16	j++;	
17	}	
18	}	

Program 6-17 | A program to illustrate the printing of a string by printing its constituent characters

3. **Using puts function:** Another convenient way to print the strings on the screen is by using the puts function. The prototype of the puts function is int puts(const char*); and is available in the stdio.h header file. The puts function prints the string on the screen and returns the number of characters printed. The code snippet in Program 6-18 illustrates the use of the puts function to print strings.

Line	Prog 6-18.c	Output window
1	//Use of puts function to print a string	Hello
2	#include<stdio.h>	Dear
3	main()	Readers!!
4	{	**Remarks:**
5	char str[20]="Readers!!";	• The argument to the puts function can be a string literal constant or a character array or a character pointer pointing to a string
6	char* ptr="Dear";	• The puts function prints the string and places a new line character after the string
7	puts("Hello");	
8	puts(ptr);	
9	puts(str);	
10	}	

Program 6-18 | A program to illustrate the use of the puts function to print a string

The important points about the usage of the puts function are as follows:

i. It has a limitation that only one string can be printed at one time.
ii. The difference between the puts function and the printf function is that the puts function places a new line character after printing the string, whereas the printf function does not. Compare the outputs of Programs 6-15 and 6-18.

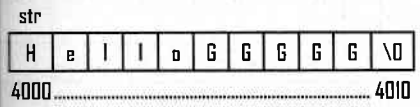
6.6 Importance of Terminating Null Character

The terminating null character in strings is very important. Every string operation checks the presence of the null character to determine the end of a string. Consider the piece of code snippet in Program 6-19 that illustrates the importance of terminating a null character in strings.

Prog 6-19.c

```
1  //Importance of terminating null character
2  #include<stdio.h>
3  #include<string.h>
4  main()
5  {
6      char str[5]={'H','e','l','l','o'};
7      printf("The string is\t");
8      puts(str);
9      printf("Its length is %d",strlen(str));
10 }
```

Memory contents



Output window

```
The string is    Hello□¥¤§¶
Its length is 10
```

Remarks:

- The printf function and the puts function print the characters starting from the memory location pointed to by its argument till the terminating null character is encountered
- In line number 6, the character array str is initialized with a list of characters and the null character is not explicitly placed at its end
- If a character array is not terminated with a null character, the output of the strlen function would be indeterminate and depends upon where the null character is present in the memory
- Thus, the puts function in line number 8 while printing str gives garbage (any arbitrary value) as it starts printing from the memory location pointed to by its argument (i.e. 4000) and keeps on printing till a terminating null character is encountered
- The number of garbage characters in the output depends upon where the first null character is encountered in the memory
- Executing the same code at different times or on different machines may give different outputs (i.e. Hello followed by different and/or different number of garbage characters)
- The strlen function determines the length of the string by counting the number of characters in the string starting from the memory location pointed to by its argument till the null character is encountered. The terminating null character is not counted while determining the length of a string

(Contd...)

Line	Prog 6-19.c	Output window
		• Thus, it is very important to explicitly place the null character at the end when a character array is initialized with the character initializers or when its content are manipulated • The null character is automatically placed at the end of a character array when it is initialized with a string literal constant or when scanf and gets functions are used to read a string from the user

Program 6-19 | A program to illustrate the importance of the terminating null character in the strings

6.7 String Library Functions

The C string library provides a large number of functions that can be used for string manipulations. The commonly used C string library functions are given in Table 6.1.

Table 6.1 | C string library functions

S. No	Function name	Prototype	Role
1.	strlen	int strlen(const char* s);	Calculates the length of a string s
2.	strcpy	char* strcpy(char* dest, const char* src);	Copies the source string str to the destination string dest
3.	strcat	char* strcat(char *dest, const char*src);	Appends a copy of the string src to the end of the string dest
4.	strcmp	int strcmp(const char*s1, const char* s2);	Compares two strings
5.	strcmpi	int strcmpi(const char*s1, const char* s2);	Compares two strings without case sensitivity
6.	strrev	char* strrev(char* s);	Reverses the content of a string s
7.	strlwr	char* strlwr(char* s);	Converts the string to lowercase
8.	strupr	char* strupr(char* s);	Converts the string to uppercase
9.	strset	char* strset(char* s, int ch);	Set all characters in a string s to the character ch
10.	strchr	char* strchr(const char* s, int c);	Scans a string for the first occurrence of a given character
11.	strrchr	char* strrchr(const char* s, int c);	Finds the last occurrence of a character in the string s
12.	strstr	char* strstr(const char* s1, const char* s2);	Finds the first occurrence of a substring (i.e. s2) in another string (i.e. s1)
13.	strncpy	char* strncpy(char* dest, const char* src, int n);	Copies at the most n characters of the string src to the string dest
14.	strncat	char* strncat(char* dest, const char* src, int n);	Appends at the most n characters of the string src to the string dest

(Cont...

	strncmp	int strncmp(const char* s1, const char* s2, int n);	Compares at the most n characters of two strings s1 and s2
	strncmpi	int strncmpi(const char* s1, const char* s2, int n);	Compares at the most n characters of two strings s1 and s2 without case sensitivity
	strnset	char* strnset(char* s, int ch, int n);	Sets the first n characters of the string s to the character ch

The following sub-sections illustrate the use of the above-mentioned string library functions along with the development of user-defined functions with the same functionality.

6.7.1 strlen Function

The strlen function is used to find the length of a string.

The input to the strlen function can be a string literal constant or a character array holding a string or a character pointer pointing to a string.

The strlen function returns the length of the string. The terminating null character is not counted while determining the length of the string.

The code snippets in Program 6-20 illustrate the use of the strlen function and the development of the strlen functionality.

Prog 6-20a.c
Using library function

```
//Finding length of a string
#include<stdio.h>
#include<string.h>
main()
{
    char *ptr="Dear";
    char name[50]="Reader";
    printf("The length of strings:\n");
    printf("Hello is %d\n",strlen("Hello"));
    printf("Dear is %d\n",strlen(ptr));
    printf("Reader is %d\n",strlen(name));
}
```

Prog 6-20b.c
Using user-defined function

```
//Finding length of a string
#include<stdio.h>
int mystrlen(char* s);
main()
{
    char *ptr="Dear";
    char name[50]="Reader";
    printf("The length of strings:\n");
    printf("Hello is %d\n",mystrlen("Hello"));
    printf("Dear is %d\n",mystrlen(ptr));
    printf("Reader is %d\n",mystrlen(name));
}
int mystrlen(char *s)
{
int i=0;
while(*(s+i)!='\0')
    i++;
return i;
}
```

Output window

```
The length of strings:
Hello is 5
Dear is 4
Reader is 6
```

Remarks:

- The strlen function returns the number of characters that precede the terminating null character
- If a terminating null character is not present at the end of a string, the strlen function gives an arbitrary result

Program 6-20 | A program to find the length of a string (a) using a library function and (b) using a user-defined function

6.7.2 strcpy Function

The strcpy function copies the source string to the destination string.

A source string and a destination string. The source string can be a string literal or a character array or a character pointer pointing to a string. The destination

should be a character array or a character pointer to the memory location which the source string is to be copied.

Output: The strcpy function copies the source string to the destination and returns a pointer to the destination string.

Usage: The code snippets in Program 6-21 illustrate the use of the strcpy function and the development of the strcpy functionality.

Line	Prog 6-21a.c Using library function	Prog 6-21b.c Using user-defined function	Output window
1	//Copying one string to another	//Copying one string to another	Source string is Hello Destination string is Hello **Remark:** • If the number of characters in the source string is more than the number of characters that the destination can hold, a memory exception may arise
2	#include<stdio.h>	#include<stdio.h>	
3	#include<string.h>	char* mystrcpy(char* dest, const char* src);	
4	main()	main()	
5	{	{	
6	char src[50]="Hello";	char src[50]="Hello";	
7	char dest[50];	char dest[50];	
8	puts("Source string is");	puts("Source string is");	
9	puts(src);	puts(src);	
10	strcpy(dest,src);	mystrcpy(dest,src);	
11	puts("Destination string is");	puts("Destination string is");	
12	puts(dest);	puts(dest);	
13	}	}	
14		char* mystrcpy(char* dest, const char* src)	
15		{	
16		int i=0;	
17		while(src[i]!='\0')	
18		{	
19		dest[i]=src[i];	
20		i++;	
21		}	
22		//Null character should be explicitly placed at	
23		//the end of the string.	
24		dest[i]='\0';	
25		return dest;	
26		}	

Program 6-21 | A program to copy a string (a) using a library function and (b) using a user-defined function

The destination character array or the destination memory block to which the character pointer points should be big enough to hold the source string. If they are not big enough, a run time exception may occur. Refer Question number 12 and its answer for more details.

6.7.3 strcat Function

Role: The strcat function concatenates one string with another. It appends a source string to the destination string.

Inputs: The source string to be appended and the destination string to which the source string is to be appended. The first argument of the function strcat can be a character array or a character pointer but should not be a string literal constant.

Output: The strcat function appends a source string to the destination string and returns a pointer to the destination string.

Usage: The code snippets in Program 6-22 illustrate the use of the strcat function and the development of the strcat functionality.

Prog 6-22a.c Using library function	Prog 6-22b.c Using user-defined function	Output window

//Concatenating a string with another
#include<stdio.h>
#include<string.h>
main()
{
char dest[50]="Hello";
char src[50]="Readers!!";
puts("The strings are::");
puts(dest);
puts(src);
strcat(dest,src);
puts("After concatenation::");
puts(dest);
}
	//Concatenating a string with another
	#include<stdio.h>
	char* mystrcat(char* dest, const char* src);
	main()
	{
	char dest[50]="Hello";
	char src[50]="Readers!!";
	puts("The strings are::");
	puts(dest);
	puts(src);
	mystrcat(dest,src);
	puts("After concatenation::");
	puts(dest);
	}
	char* mystrcat(char* dest, const char* src)
	{
	int i=0, j=0;
	while(dest[i]!='\0')
	i++;
	while(src[j]!='\0')
	{
	dest[i]=src[j];
	i++;j++;
	}
	dest[i]='\0';
	return dest;
	}
		The strings are::
		Hello
		Readers!!
		After concatenation::
		HelloReaders!!
		Remarks:
		• The length of the destination string after concatenation = the length of the destination string before concatenation plus the length of the source string
		• The destination should be big enough to hold the destination string plus the source string
		• If it is not big enough, the characters of the resulting string would be placed in unreserved memory and may lead to memory violation. Hence memory exception may occur

Program 6-22 | A program to concatenate a string with another (a) using a library function and (b) using a user-defined function

6.7.4 strcmp Function

Role: The strcmp function compares two strings.

Inputs: Two strings str1 and str2 that are to be compared in the form of string literal constants or character arrays or character pointers to the memory locations in which str1 and str2 are stored.

Output: The strcmp function performs the comparison of str1 and str2 character by character, starting with the first character in each string and continuing with subsequent characters until the corresponding characters differ or until the end of the strings is reached. It returns the ASCII difference of the first dissimilar corresponding characters or zero if none of the corresponding characters in both the strings are different.

Usage: The code snippets in Program 6-23 illustrate the use of the strcmp function and the development of the strcmp functionality.

Line	Prog 6-23a.c Using library function	Prog 6-23b.c Using user-defined function	Output window
1	//Comparing two strings	//Comparing two strings	Enter string 1:
2	#include<stdio.h>	#include<stdio.h>	Hello
3	#include<string.h>	int mystrcmp(const char* s1, const char* s2);	Enter string 2:
4	main()	main()	Hi
5	{	{	Strings are not equal
6	char str1[20],str2[20];	char str1[20],str2[20];	**Output window (second execution)**
7	int res;	int res;	Enter string 1:
8	puts("Enter string 1:");	puts("Enter string 1:");	Hello
9	gets(str1);	gets(str1);	Enter string 2:
10	puts("Enter string 2:");	puts("Enter string 2:");	Hello
11	gets(str2);	gets(str2);	Strings are equal
12	res=strcmp(str1,str2);	res=mystrcmp(str1,str2);	**Output window (third execution)**
13	if(res==0)	if(res==0)	Enter string 1:
14	puts("Strings are equal");	puts("Strings are equal");	hello
15	else	else	Enter string 2:
16	puts("Strings are not equal");	puts("Strings are not equal");	HELLO
17	}	}	Strings are not equal
18		int mystrcmp(const char* s1, const char* s2)	**Remarks:**
19		{	• **strcmp(str1,str2)** returns a value:
20		int i=0;	• 0 if str1 and str2 are equal, or
21		while(s1[i]!='\0' || s2[i]!='\0')	• >0 if str1 is greater than str2, i.e. str1 comes after str2 in lexicographic order (i.e. dictionary order), or
22		{	• <0 if str1 is lesser than str2, i.e. str1 comes before str2 in lexicographic order
23		if(s1[i]!=s2[i])	
24		return(s1[i]-s2[i]);	
25		i++;	
26		}	
27		return 0;	
28		}	

Program 6-23 | A program to compare two strings (a) using a library function and (b) using a user-defined function

Output: The strcmp function performs the comparison of str1 and str2 character by character, starting with the first character in each string and continuing with the subsequent characters until the corresponding characters differ or until the end of the strings is reached. It returns the ASCII difference of the first dissimilar corresponding characters or zero if none of the corresponding characters in both the strings are different.

Usage: The code snippets in Program 6-23 illustrate the use of the strcmp function and the development of the strcmp functionality.

Line	Prog 6-23a.c Using library function	Prog 6-23b.c Using user-defined function	Output window
1	//Comparing two strings	//Comparing two strings	Enter string 1:
2	#include<stdio.h>	#include<stdio.h>	Hello
3	#include<string.h>	int mystrcmp(const char* s1, const char* s2);	Enter string 2:
4	main()	main()	Hi
5	{	{	Strings are not equal
6	char str1[20],str2[20];	char str1[20],str2[20];	**Output window (second execution)**
7	int res;	int res;	Enter string 1:
8	puts("Enter string 1:");	puts("Enter string 1:");	Hello
9	gets(str1);	gets(str1);	Enter string 2:
10	puts("Enter string 2:");	puts("Enter string 2:");	Hello
11	gets(str2);	gets(str2);	Strings are equal
12	res=strcmp(str1,str2);	res=mystrcmp(str1,str2);	**Output window (third execution)**
13	if(res==0)	if(res==0)	Enter string 1:
14	puts("Strings are equal");	puts("Strings are equal");	hello
15	else	else	Enter string 2:
16	puts("Strings are not equal");	puts("Strings are not equal");	HELLO
17	}	}	Strings are not equal
18		int mystrcmp(const char* s1, const char* s2)	**Remarks:**
19		{	• **strcmp(str1,str2)** returns a value:
20		int i=0;	• 0 if str1 and str2 are equal, or
21		while(s1[i]!='\0' || s2[i]!='\0')	• >0 if str1 is greater than str2, i.e. str1 comes after str2 in lexicographic order (i.e. dictionary order), or
22		{	• <0 if str1 is lesser than str2, i.e. str1 comes before str2 in lexicographic order
23		if(s1[i]!=s2[i])	
24		return(s1[i]-s2[i]);	
25		i++;	
26		}	
27		return 0;	
28		}	

Program 6-23 | A program to compare two strings (a) using a library function and (b) using a user-defined function

7.5 strcmpi Function

Role: The strcmpi function compares two strings without case sensitivity. The suffix character 'i' in strcmpi stands for ignore case.

Inputs: Two strings str1 and str2 that are to be compared, in the form of string literal constants or character arrays or character pointers to the memory locations in which str1 and str2 are stored.

Output: The strcmpi function performs a comparison of strings str1 and str2 without case sensitivity. It returns the ASCII difference of the first different corresponding characters or zero if none of the corresponding characters in both the strings are different.

Usage: The code snippets in Program 6-24 illustrate the use of the strcmpi function and the development of the strcmpi functionality.

Prog 6-24a.c — Using library function

```
1  //Comparing two strings without
2  //case sensitivity
3  #include<stdio.h>
4  #include<string.h>
5  main()
6  {
7      char str1[20],str2[20];
8      int res;
9      puts("Enter string 1:");
10     gets(str1);
11     puts("Enter string 2:");
12     gets(str2);
13     res=strcmpi(str1,str2);
14     if(res==0)
15         puts("Strings are equal");
16     else
17         puts("Strings are not equal");
18 }
```

Prog 6-24b.c — Using user-defined function

```
1  //Comparing two strings without
2  //case sensitivity
3  #include<stdio.h>
4  int mystrcmpi(const char* s1, const char* s2);
5  main()
6  {
7      char str1[20],str2[20];
8      int res;
9      puts("Enter string 1:");
10     gets(str1);
11     puts("Enter string 2:");
12     gets(str2);
13     res=mystrcmpi(str1,str2);
14     if(res==0)
15         puts("Strings are equal");
16     else
17         puts("Strings are not equal");
18 }
19 int mystrcmpi(const char* s1, const char* s2)
20 {
21 int i=0;
22 while(s1[i]!='\0' || s2[i]!='\0')
23 {
24 if((s1[i]==s2[i])||(s1[i]-s2[i])==32||(s1[i]-s2[i])==-32)
25     i++;
26 else
27     return(s1[i]-s2[i]);
28 }
29 return 0;
30 }
```

Output window

```
Enter string 1:
HELLO
Enter string 2:
hello
Strings are equal
```

Output window (second execution)

```
Enter string 1:
Hello
Enter string 2:
Hi
Strings are not equal
```

Output window (third execution)

```
Enter string 1:
hello
Enter string 2:
HELLO
Strings are equal
```

Remarks:

- The difference between the ASCII values of lowercase letters and their uppercase counterparts is 32
- For example, 'a' has an ASCII value of 97 while 'A' has an ASCII value of 65

Program 6-24 | A program to compare two strings without case sensitivity (a) using a library function and (b) using a user-defined function

6.7.6 strrev Function

Role: The strrev function reverses all the characters of a string except the terminating null character.

Input: A string in the form of a character array or a character pointer or a string literal constant.

Output: The strrev function reverses the string and returns a pointer to the reversed string.

Usage: The code snippets in Program 6-25 illustrate the use of the strrev function and the development of the strrev functionality.

Line	Prog 6-25a.c Using library function	Prog 6-25b.c Using user-defined function	Output window
1	//Reversing the contents of a string	//Reversing the contents of a string	Enter a string: Hello After reversal, the string is: olleH
2	#include<stdio.h>	#include<stdio.h>	
3	#include<string.h>	char* mystrrev(char* s);	
4	main()	main()	
5	{	{	
6	char str[20];	char str[20];	**Output window (second execution)**
7	puts("Enter a string:");	puts("Enter a string:");	
8	gets(str);	gets(str);	Enter a string: Hello Readers After reversal, the string is: sredaeR olleH
9	strrev(str);	mystrrev(str);	
10	puts("After reversal, the string is:");	puts("After reversal, the string is:");	**Remarks:**
11	puts(str);	puts(str);	• The strrev function can also be applied on the string literals, i.e. strrev("Hello")="olleH"
12	}	}	• strrev(strrev("String"))="String"
13		char* mystrrev(char* s)	• Reversal of reverse of a string is the string itself
14		{	
15		int i=0, j=0;	
16		char temp;	
17		while(s[i]!='\0')	
18		i++;	
19		i--;	
20		while(i>j)	
21		{	
22		temp=s[i];	
23		s[i]=s[j];	
24		s[j]=temp;	
25		j++;i--;	
26		}	
27		return s;	
28		}	

Program 6-25 | A program that reverses contents of a string (a) using a library function and (b) using a user-defined function

6.7.7 strlwr Function

Role: The strlwr function converts all the letters in a string to lowercase.

Input: A string in the form of a character array or a character pointer or a string literal constant.

Output: It returns a pointer to the converted string.
Usage: The code snippets in Program 6-26 illustrate the use of the strlwr function and the development of the strlwr functionality.

Line	Prog 6-26a.c Using library function	Prog 6-26b.c Using user-defined function	Output window
1	//Converting all the characters of a	//Converting all the characters of a	Enter a string: HELLO Lowercase string is: hello
2	//string to lower case	//string to lower case	
3	#include<stdio.h>	#include<stdio.h>	
4	#include<string.h>	char* mystrlwr(char* s);	
5	main()	main()	
6	{	{	
7	char str[20];	char str[20];	
8	puts("Enter a string:");	puts("Enter a string:");	
9	gets(str);	gets(str);	
10	strlwr(str);	mystrlwr(str);	
11	puts("Lowercase string is:");	puts("Lowercase string is:");	
12	puts(str);	puts(str);	
13	}	}	
14		char* mystrlwr(char* s)	
15		{	
16		int i=0;	
17		while(s[i]!='\0')	
18		{	
19		if(s[i]>=65 && s[i]<=90)	
20		s[i]=s[i]+32;	
21		i++;	
22		}	
23		return s;	
24		}	

Output window (second execution)

Enter a string:
HELLO READERS!!
Lowercase string is:
hello readers!!

Remarks:

- Digits, special characters and white-space characters within the string remain unchanged

Program 6-26 | A program that converts all the characters of a string to lowercase (a) using a library function and (b) using a user-defined function

6.7.8 strupr Function

Role: The strupr function converts all the letters in a string to uppercase.
Input: A string in the form of a character array or a character pointer or a string literal constant.
Output: It returns a pointer to the converted string.
Usage: The code snippets in Program 6-27 illustrate the use of the strupr function and the development of the strupr functionality.

Line	Prog 6-27a.c Using library function	Prog 6-27b.c Using user-defined function	Output window
1	//Converting all the characters of a	//Converting all the characters of a	Enter a string:
2	//string to uppercase	//string to uppercase	hello
3	#include<stdio.h>	#include<stdio.h>	Uppercase string is:
4	#include<string.h>	char* mystrupr(char* s);	HELLO
5	main()	main()	**Output window (second execution)**
6	{	{	Enter a string:
7	char str[20];	char str[20];	hello readers!!
8	puts("Enter a string:");	puts("Enter a string:");	Uppercase string is:
9	gets(str);	gets(str);	HELLO READERS!!
10	strupr(str);	mystrupr(str);	**Remark:**
11	puts("Uppercase string is:");	puts("Uppercase string is:");	• Digits, special characters and white-space characters within a string remain unchanged
12	puts(str);	puts(str);	
13	}	}	
14		char* mystrupr(char* s)	
15		{	
16		int i=0;	
17		while(s[i]!='\0')	
18		{	
19		if(s[i]>=97 && s[i]<=122)	
20		s[i]=s[i]-32;	
21		i++;	
22		}	
23		return s;	
24		}	

Program 6-27 | A program that converts all the characters of a string to uppercase (a) using a library function and (b) using a user-defined function

6.7.9 strset Function

Role: The strset function sets all characters in a string to a specific character.

Inputs: A string and a character. The string can be in the form of a character array or a character pointer or a string literal constant.

Output: The strset function sets all the characters in the string to the given character and returns a pointer to the string.

Usage: The code snippets in Program 6-28 illustrate the use of the strset function and the development of the strset functionality.

Line	Prog 6-28a.c Using library function	Prog 6-28b.c Using user-defined function	Output window
1	//Setting all the characters of a string to	//Setting all the characters of a string to	Before using strset(), string is:
2	//a specific character	//a specific character	123456789
3	#include<stdio.h>	#include<stdio.h>	After using strset(), string is:
4	#include<string.h>	char* mystrset(char* s, int ch);	cccccccc
5	main()	main()	

(Contd...)

```
{
    char str[10]="123456789";
    char ch='c';
    puts("Before using strset(), string is:");
    puts(str);
    strset(str,ch);
    puts("After using strset(), string is:");
    puts(str);
}
```

```
{
    char str[10]="123456789";
    char ch='c';
    puts("Before using strset(), string is:");
    puts(str);
    mystrset(str,ch);
    puts("After using strset(), string is:");
    puts(str);
}
char* mystrset(char* s, int ch)
{
int i=0;
while(s[i]!='\0')
{
    s[i]=ch;
    i++;
}
return s;
}
```

Remark:

- All the characters (letters, digits, special characters and white-space characters) within a string are set to a specific character

Program 6-28 | A program that sets all the characters of a string to a specific character (a) using a library function and (b) using a user-defined function

6.7.10 strchr Function

Role: The strchr function scans a string for the first occurrence of a given character.

Inputs: A string and a character to be found in the string. The string can be in the form of a character array or a character pointer or a string literal constant.

Output: The strchr function scans the input string in the forward direction, looking for the specific character. If the character is found, it returns a pointer to the first occurrence of the character in the given string. If the character is not found it returns NULL.

Usage: The code snippets in Program 6-29 illustrate the use of the strchr function and the development of the strchr functionality.

Line	Prog 6-29a.c Using library function	Prog 6-29b.c Using user-defined function	Output window
1	//Scans a string for the first occurrence	//Scans a string for the first occurrence	Enter a string: Hello Enter a character to be found: e Located at the index 1
2	//of a given character	//of a given character	
3	#include<stdio.h>	#include<stdio.h>	
4	#include<string.h>	char* mystrchr(const char* s, int c);	
5	main()	main()	
6	{	{	**Output window (second execution)**
7	char str[20], ch;	char str[20], ch;	Enter a string: Hello
8	char* ptr;	char* ptr;	
9	puts("Enter a string:");	puts("Enter a string:");	
10	gets(str);	gets(str);	

(Contd...)

Line	Prog 6-29a.c Using library function	Prog 6-29b.c Using user-defined function	Output window
11	puts("Enter a character to be found:");	puts("Enter a character to be found:");	Enter a character to be found
12	scanf("%c",&ch);	scanf("%c",&ch);	y
13	ptr=strchr(str,ch);	ptr=mystrchr(str,ch);	Character not found
14	if(ptr==NULL)	if(ptr==NULL)	**Remark:**
15	puts("Character not found");	puts("Character not found");	• The terminating null character is also considered to be a part of the string
16	else	else	
17	printf("Located at the index %d",ptr-str);	printf("Located at the index %d",ptr-str);	
18	}	}	
19		char* mystrchr(const char* s, int c)	
20		{	
21		int i=0;	
22		while(s[i]!='\0')	
23		{	
24		if(s[i]==c)	
25		return((char*)s+i);	
26		i++;	
27		}	
28		return NULL;	
29		}	

Program 6-29 | A program that scans a string for the first occurrence of a given character (a) using a library function and (b) using a user-defined function

6.7.11 strrchr Function

Role: The strrchr function locates the last occurrence of a character in a given string.

Inputs: A string and a character to be found in the string. The string can be in the form of a character array or a character pointer or a string literal constant.

Output: The strrchr function scans the input string in the reverse direction, looking for a specific character. If the character is found, it returns a pointer to the first occurrence of the character in the given string. If the character is not found, it returns NULL.

Usage: The code snippets in Program 6-30 illustrate the use of the strrchr function and the development of the strrchr functionality.

Line	Prog 6-30a.c Using library function	Prog 6-30b.c Using user-defined function	Output window
1	//Scans a string in the reverse direction	//Scans a string in the reverse direction	Enter a string:
2	//for the first occurrence of a given	//for the first occurrence of a given	Hello
3	//character	//character	Enter a character to be found
4	#include<stdio.h>	#include<stdio.h>	o
5	#include<string.h>	char* mystrrchr(const char* s, int c);	Located at the index 4
6	main()	main()	
7	{	{	
8	char str[20], ch;	char str[20], ch;	
9	char* ptr;	char* ptr;	
10	puts("Enter a string:");	puts("Enter a string:");	

(Contd...)

```
    gets(str);
    puts("Enter a character to be found:");
    scanf("%c",&ch);
    ptr=strrchr(str,ch);
    if(ptr==NULL)
    puts("Character not found");
    else
    printf("Located at the index %d",ptr-str);
}
```

```
    gets(str);
    puts("Enter a character to be found:");
    scanf("%c",&ch);
    ptr=mystrrchr(str,ch);
    if(ptr==NULL)
    puts("Character not found");
    else
    printf("Located at the index %d",ptr-str);
}
char* mystrrchr(const char* s, int c)
{
int i=0;
while(s[i]!='\0')
    i++;
i--;
while(i>=0)
{
    if(s[i]==c)
        return((char*)s+i);
    i--;
}
return NULL;
}
```

Output window (second execution)

```
Enter a string:
Hello
Enter a character to be found:
y
Character not found
```

Output window (third execution)

```
Enter a string:
Hello
Enter a character to be found:
l
Located at the index 3
```

Remark:

- The terminating null character is also considered to be a part of the string

Program 6-30 | A program that scans a string in the reverse direction for the first occurrence of a given character (a) using a library function and (b) using a user-defined function

6.7.12 strstr Function

Role: The strstr function finds the first occurrence of a string in another string.

Inputs: Two strings str1 and str2. The strings can be in the form of a character array or a character pointer or a string literal constant.

Output: The strstr function finds the first occurrence of the string (i.e. str2) in the string (i.e. str1). If the string str2 is found, it returns a pointer to the position from where the string starts. If the string str2 is not found in the string str1, it returns NULL.

Usage: The code snippets in Program 6-31 illustrate the use of the strstr function and the development of the strstr functionality.

Prog 6-31a.c — Using library function

```
//Finding string within a string
#include<stdio.h>
#include<string.h>
main()
{
    char* ptr;
    char str1[20];
    char str2[20]
    puts("Enter a string:");
```

Prog 6-31b.c — Using user-defined function

```
//Finding string within a string
#include<stdio.h>
char* mystrstr(const char* s1, const char* s2);
main()
{
    char* ptr;
    char str1[20];
    char str2[20];
    puts("Enter a string:");
```

Output window

```
Enter a string:
Hello Readers!!
Enter the string to be found:
Read
Found at the index 6
Found in Readers!!
```

(Contd...)

```
10      gets(str1);
11      puts("Enter the string to be found:");
12      gets(str2);
13      ptr=strstr(str1,str2);
14      if(ptr==NULL)
15          puts("String not found");
16      else
17      {
18      printf("Found at the index %d\n",ptr-str1);
19      printf("Found in %s",ptr);
20      }
21  }
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
```

```
    gets(str1);
    puts("Enter the string to be found:");
    gets(str2);
    ptr=mystrstr(str1,str2);
    if(ptr==NULL)
        puts("String not found");
    else
    {
    printf("Found at the index %d\n",ptr-str1);
    printf("Found in %s",ptr);
    }
}
char* mystrstr(const char* s1, const char* s2)
{
int i=0,j=0,k;
while(s1[i]!='\0')
{
    k=i;
    while(s2[j]!='\0')
    {
        if(s1[k]!=s2[j])
            break;
        k++;j++;
    }
    if(s2[j]=='\0')
        return (char*)s1+i;
    else
        i++;j=0;
}
return NULL;
}
```

Output window (second execution)

```
Enter a string:
Hello Readers!!
Enter the string to be found:
Student
String not found
```

Program 6-31 | A program that finds a string within a string (a) using a library function and (b) using a user-defined function

6.7.13 strncpy Function

Role: The strncpy function copies at the most n characters of a source string to the destination string.

Inputs: A character array or a character pointer to the memory location where the source string is to be copied (i.e. destination), the source string that is to be copied and an integer value that specifies the number of characters of the source string that is to be copied.

Output: The strncpy function copies at the most n characters of the source string to the destination and returns a pointer to the destination string.

Usage: The code snippets in Program 6-32 illustrate the use of the strncpy function and the development of the strncpy functionality.

Prog 6-32a.c Using library function	Prog 6-32b.c Using user-defined function	Output window

//Copying at the most n characters of
//a source string to the destination
//string
#include<stdio.h>
#include<string.h>
main()
{
char src[50];
char dest[50];
int n;
puts("Enter source string:");
gets(src);
puts("Enter the value of n:");
scanf("%d",&n);
puts("Source string is:");
puts(src);
strncpy(dest,src,n);
dest[n]='\0';
puts("Destination string is:");
puts(dest);
}
	//Copying at the most n characters of
	//a source string to the destination
	//string
	#include<stdio.h>
	char* mystrncpy(char* dest, const char* src, int n);
	main()
	{
	char src[50];
	char dest[50];
	int n;
	puts("Enter source string:");
	gets(src);
	puts("Enter the value of n:");
	scanf("%d",&n);
	puts("Source string is:");
	puts(src);
	mystrncpy(dest,src,n);
	dest[n]='\0';
	puts("Destination string is:");
	puts(dest);
	}
	char* mystrncpy(char* dest, const char* src, int n)
	{
	int i=0;
	while(i<n)
	{
	if(src[i]=='\0')
	{
	dest[i]='\0';
	break;
	}
	else
	{
	dest[i]=src[i];
	i++;
	}
	}
	return dest;
	}
		Enter source string:
		Hello Readers!!
		Enter the value of n:
		5
		Source string is:
		Hello Readers!!
		Destination string is:
		Hello
		Remarks:
		• If the source string contains more than n characters, n characters are copied and the null character is not placed at the end. The terminating null character is to be explicitly placed as done in line number 18
		• If the source string is shorter than n characters, the terminating null character is copied into the destination string
		Try:
		• Comment line number 18
		• Execute the code with the same input and observe the garbage characters in the output in some of the executions

Program 6-32 | A program that copies at most n characters of a source string to a destination string (a) using a library function and (b) using a user-defined function

6.7.14 strncat Function

Role: The strncat function concatenates a portion of one string with another. It appends at the most n characters of a source string to a destination string.

Inputs: The source string to be appended, the destination string to which the source string is to be appended and the number of characters to be appended. The destination string should be a character array or a character pointer but should not be a string literal constant.

Output: The strncat function appends at the most n characters of the source string to the destination string and returns a pointer to the destination string.

Usage: The code snippets in Program 6-33 illustrate the use of the strncat function and the development of the strncat functionality.

Line	Prog 6-33a.c Using library function	Prog 6-33b.c Using user-defined function	Output window
1	//String Concatenation	//String Concatenation	Enter strings:
2	#include<stdio.h>	#include<stdio.h>	Hello
3	#include<string.h>	char* mystrncat(char* dest, const char* src, int n);	Readers!!
4	main()	main()	Enter the value of n:
5	{	{	7
6	char dest[50], src[50];	char dest[50], src[50];	The strings are:
7	int n;	int n;	Hello
8	puts("Enter strings:");	puts("Enter strings:");	Readers!!
9	gets(dest);	gets(dest);	After concatenation:
10	gets(src);	gets(src);	HelloReaders
11	puts("Enter the value of n:");	puts("Enter the value of n:");	**Remarks:**
12	scanf("%d",&n);	scanf("%d",&n);	• Unlike strncpy, a terminating null character is always appended to the result
13	puts("The strings are:");	puts("The strings are:");	• The maximum number of characters in the destination string after the execution of strncat would be the number of characters in the dest (before execution of strncat)+n
14	puts(dest);	puts(dest);	
15	puts(src);	puts(src);	
16	strncat(dest,src,n);	mystrncat(dest,src,n);	
17	puts("After concatenation:");	puts("After concatenation:");	
18	puts(dest);	puts(dest);	
19	}	}	
20		char* mystrncat(char* dest, const char* src,int n)	
21		{	
22		int i=0, j=0,k=1;	
23		while(dest[i]!='\0')	
24		i++;	
25		while(src[j]!='\0' && k<=n)	
26		{	
27		dest[i]=src[j];	
28		i++;j++;k++;	
29		}	
30		dest[i]='\0';	
31		return dest;	
32		}	

Program 6-33 | A program that concatenates at the most n characters of a source string with the destination string (a) using a library function and (b) using a user-defined function

6.7.15 strncmp Function

Title: The strncmp function compares a portion of two strings.

Inputs: Two strings str1 and str2 and the value of n, i.e. the number of characters to be compared.

Output: The strncmp function performs the comparison of str1 and str2, starting with the first character in each string and continuing with the subsequent characters until the corresponding characters differ or until the end of strings is reached or n characters have been compared. It returns the ASCII difference of the first dissimilar corresponding characters or zero if none of the corresponding n characters in both the strings are different.

Usage: The code snippets in Program 6-34 illustrate the use of the strncmp function and the development of the strncmp functionality.

Line	Prog 6-34a.c Using library function	Prog 6-34b.c Using user-defined function	Output window
1	//Comparing a portion of two strings	//Comparing a portion of two strings	Enter string 1: Hello Enter string 2: Hi Enter the value of n: 1 String portions are equal
2	#include<stdio.h>	#include<stdio.h>	
3	#include<string.h>	int mystrncmp(const char* s1, const char* s2, int n);	
4	main()	main()	
5	{	{	
6	char str1[20],str2[20];	char str1[20],str2[20];	
7	int res, n;	int res,n;	
8	puts("Enter string 1:");	puts("Enter string 1:");	**Output window (second execution)**
9	gets(str1);	gets(str1);	
10	puts("Enter string 2:");	puts("Enter string 2:");	Enter string 1: Hello Enter string 2: Hello Enter the value of n: 4 String portions are equal
11	gets(str2);	gets(str2);	
12	puts("Enter the value of n:");	puts("Enter the value of n:");	
13	scanf("%d",&n);	scanf("%d",&n);	
14	res=strncmp(str1,str2,n);	res=mystrncmp(str1,str2,n);	
15	if(res==0)	if(res==0)	
16	puts("String portions are equal");	puts("String portions are equal");	
17	else	else	
18	puts("String portions are not equal");	puts("String portions are not equal");	**Output window (third execution)**
19	}	}	
20		int mystrncmp(const char* s1, const char* s2,int n)	Enter string 1: hello Enter string 2: HELLO Enter the value of n: 3 String portions are not equal
21		{	
22		int i=0;	
23		while((s1[i]!='\0' || s2[i]!='\0') && i<n)	
24		{	
25		if(s1[i]!=s2[i])	
26		return(s1[i]-s2[i]);	
27		i++;	
28		}	
29		return 0;	
30		}	

Program 6-34 | A program that compares a portion of two strings (a) using a library function and (b) using a user-defined function

6.7.16 strncmpi Function

Role: The strncmpi function compares a portion of two strings without case sensitivity.

Inputs: Two strings str1 and str2 and the value of n, i.e. the number of characters to be compared.

Output: The strncmpi function performs the comparison of str1 and str2 without case sensitivity, starting with the first character in each string and continuing with the subsequent characters until the corresponding characters differ or until the end of strings is reached or n characters have been compared. It returns the ASCII difference of the first different corresponding characters or zero if none of the corresponding n characters in both the strings are different.

Usage: The code snippets in Program 6-35 illustrate the use of the strncmpi function and the development of the strncmpi functionality.

Line	Prog 6-35a.c Using library function	Prog 6-35b.c Using user-defined function	Output window
1	//Comparing a portion of strings	//Comparing a portion of strings without	Enter string 1:
2	//without case sensitivity	//case sensitivity	Hello
3	#include<stdio.h>	#include<stdio.h>	Enter string 2:
4	#include<string.h>	int mystrncmpi(const char* s1, const char* s2, int n);	Hi
5	main()	main()	Enter the value of n:
6	{	{	2
7	char str1[20],str2[20];	char str1[20],str2[20];	String portions are not equal
8	int res, n;	int res,n;	**Output window (second execution)**
9	puts("Enter string 1:");	puts("Enter string 1:");	Enter string 1:
10	gets(str1);	gets(str1);	Hello
11	puts("Enter string 2:");	puts("Enter string 2:");	Enter string 2:
12	gets(str2);	gets(str2);	Hello
13	puts("Enter the value of n:");	puts("Enter the value of n:");	Enter the value of n:
14	scanf("%d",&n);	scanf("%d",&n);	5
15	res=strncmpi(str1,str2,n);	res=mystrncmpi(str1,str2,n);	String portions are equal
16	if(res==0)	if(res==0)	**Output window (third execution)**
17	puts("String portions are equal");	puts("String portions are equal");	Enter string 1:
18	else	else	hello
19	puts("String portions are not equal");	puts("String portions are not equal");	Enter string 2:
20	}	}	HELLO
21		int mystrncmpi(const char* s1, const char* s2,int n)	Enter the value of n:
22		{	4
23		int i=0;	String portions are equal
24		while((s1[i]!='\0' || s2[i]!='\0') && i<n)	
25		{	
26		if((s1[i]==s2[i])|| (s1[i]-s2[i])==32|| (s1[i]-s2[i])==-32)	
27		i++;	
28		else	
29		return(s1[i]-s2[i]);	
30		}	
31		return 0;	
32		}	

Program 6-35 | A program that compares a portion of two strings without case sensitivity (a) using a library function and (b) using a user-defined function

6.7.17 strnset Function

Role: The strnset function sets the first n characters in a string to a specific character.

Inputs: A string, a character and an integer value n.

Output: The strnset function sets the first n characters in a string to the given character and returns a pointer to the string.

Usage: The code snippets in Program 6-36 illustrate the use of the strnset function and the development of the strnset functionality

Line	Prog 6-36a.c Using library function	Prog 6-36b.c Using user-defined function	Output window
1	//Setting the first n characters of a string	//Setting the first n characters of a	Enter the string:
2	//to a specific character	//string to a specific character	Hello Readers!!
3	#include<stdio.h>	#include<stdio.h>	Enter the character:
4	#include<string.h>	char* mystrnset(char* s, int ch, int n);	X
5	main()	main()	Enter the value of n:
6	{	{	6
7	char str[20], ch;	char str[20], ch;	Before using strnset(), string is:
8	int n;	int n;	Hello Readers!!
9	puts("Enter the string:");	puts("Enter the string:");	After using strnset(), string is:
10	gets(str);	gets(str);	XXXXXXReaders!!
11	puts("Enter the character:");	puts("Enter the character:");	**Remark:**
12	scanf("%c",&ch);	scanf("%c",&ch);	• If the length of the string is less than the value of n then the strnset function sets all the characters of the string to the specific character
13	puts("Enter the value of n:");	puts("Enter the value of n:");	
14	scanf("%d",&n);	scanf("%d",&n);	
15	puts("Before using strnset(), string is:");	puts("Before using strnset(), string is:");	
16	puts(str);	puts(str);	
17	strnset(str,ch,n);	mystrnset(str,ch,n);	
18	puts("After using strnset(), string is:");	puts("After using strnset(), string is:");	
19	puts(str);	puts(str);	
21	}	}	
22		char* mystrnset(char* s, int ch, int n)	
23		{	
24		int i=0;	
25		while(s[i]!='\0' && i<n)	
26		{	
27		s[i]=ch;	
28		i++;	
29		}	
30		return s;	
31		}	

Program 6-36 | A program that sets the first n characters of a string to a specific character (a) using a library function and (b) using a user-defined function

6.8 List of Strings

In the previous sections, we have seen how to store the strings in character arrays and the functions that can be used to manipulate them. However, real-time applications often require

storage and manipulation of a number of strings (i.e. **list of strings**) and not only a single string. A list of strings can be stored in two ways:

1. Using an array of strings
2. Using an array of character pointers

6.8.1 Array of strings

If an application requires the storage of multiple strings, an array of strings can be used to store them. Since a string itself is stored in a one-dimensional character array, the list of strings can be stored by creating an array of one-dimensional character arrays, i.e. two-dimensional character array. Figure 6.4 depicts an array of strings.

A 2-D char array → (Array of strings)								
	R	a	m	a	n	\0		←1st string
	S	a	m	\0				←2nd string
	V	i	s	h	a	l	\0	←3rd string
	N	e	h	a	\0			←4th string

Figure 6.4 | Array of strings

6.8.1.1 Declaration of Array of strings

The general form of an **array of strings declaration** is:

<sclass_specifier><type_qualifier><type_modifier>**char identifier[**<row_specifier>**][**column_specifier**]**<=initialization_list>

The important points about an array of strings declaration are as follows:

1. Array of strings declaration consists of char type specifier, an identifier name, row size specifier and column size specifier. The following declarations are valid:

```
char array1[2][30];    //←array1 can store 2 strings of maximum 30 characters each
char array2[5][5];     //←array2 can store 5 strings of maximum 5 characters each
```

2. All the syntactic rules discussed in Chapter 4 for declaring two-dimensional arrays are applicable for declaring arrays of strings as well.
3. **Initialization of array of strings:** Array of strings can be initialized in two ways:
   a. **Using string literal constants:** Using string literal constants, an array of strings can be initialized as:

```
char str[][20]={
                "Raman",
                "Sam",
                "Vishal",
                "Neha"
                };
```

   b. **Using a list of character initializers:** Using a list of character initializers, an array of strings can be initialized as:

```
char str[][20]={
                    {'R','a','m','a','n','\0'},
                    {'S','a','m','\0'},
                    {'V','i','s','h','a','l','\0'},
                    {'N','e','h','a','\0'}
                };
```

6.8.1.2 Reading List of Strings from the Terminal

A list of strings can be read from the terminal by iteratively calling the gets or scanf function. Program 6-37 reads a list of strings from the terminal and stores them in an array of strings.

Line	Prog 6-37.c	Output window
1	//Reading a list of strings from the terminal	Enter names of students and their marks:
2	#include<stdio.h>	Praveen 89
3	main()	Do you want to enter more(Y/N) Y
4	{	Ashok 80
5	int i=0,j=0, marks[10], max;	Do you want to enter more(Y/N) Y
6	char students[10][20], ch;	Manish 90
7	printf("Enter names of students and their marks:\n");	Do you want to enter more(Y/N) Y
8	while(1)	Ameet 85
9	{	Do you want to enter more(Y/N) N
10	scanf("%s %d",students[i], &marks[i]);	
11	printf("Do you want to enter more(Y/N)\t");	Manish got maximum marks
12	flushall();	**Remarks:**
13	scanf("%c",&ch);	• List of strings can be read by iteratively using the gets or scanf function
14	if(ch=='Y'||ch=='y')	• The role of the flushall function is to flush (i.e. clear) the contents of all the streams
15	i=i+1;	• Refer Question number 15 for a description on streams and the flushall function
16	else	
17	break;	
18	if(i==10)	
19	{	
20	printf("Cannot hold more names\n");	
21	break;	
22	}	
23	}	
24	max=0;	
25	for(j=0;j<i;j++)	
26	if(marks[j]>marks[max])	
27	max=j;	
28	printf("\n%s got maximum marks",students[max]);	
29	}	

Program 6-37 | A program that demonstrates a method to read a list of strings

6.8.2 Array of Character Pointers

An array of strings can also be stored by using an array of character pointers. The starting addresses of strings are stored in an array of character pointers as shown in Figure 6.5.

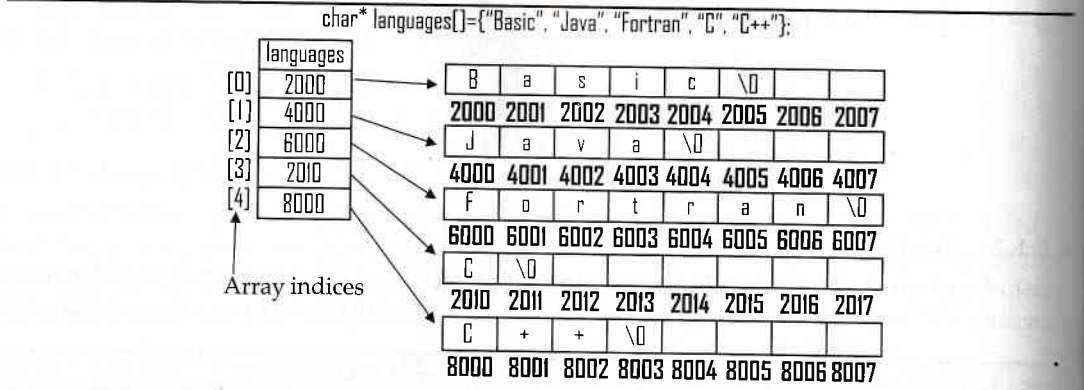


Figure 6.5 | Storing a list of strings using an array of character pointers

6.8.2.1 Use of Array of Character Pointers

Program 6-38 demonstrates the use of an array of character pointers to store a list of strings.

Line	Prog 6-38.c
1	//Use of array of character pointers
2	#include<stdio.h>
3	main()
4	{
5	int i,a[4];
6	char* states[]={"Punjab", "Bihar", "Rajasthan", "Gujarat"} ;
7	char* capitals[]={"Gandhinagar", "Chandigarh", "Jaipur", "Patna"};
8	printf("States\t\t\tCapitals\n");
9	printf("----------\n");
10	for(i=0;i<4;i++)
11	printf("%d. %-10s\t\t%d. %s\n",i+1,states[i],i+1,capitals[i]);
12	printf("\nMatch states in Col. 1 with capitals in Col. 2\n");
13	printf("(Enter only Sr. Nos.)\n");
14	printf("----------\n");
15	for(i=0;i<4;i++)
16	{
17	printf("Capital of state %d is at\t",i+1);
18	scanf("%d",&a[i]);
19	}
20	printf("----------\n");
21	for(i=0;i<4;i++)
22	printf("%-11s is capital of %s\n",capitals[a[i]-1],states[i]);
23	}

Output window

```
States          Capitals
----------
1. Punjab       1. Gandhinagar
2. Bihar        2. Chandigarh
3. Rajasthan    3. Jaipur
4. Gujarat      4. Patna

Match states in Col. 1 with capitals in Col. 2
(Enter only Sr. Nos.)
----------
Capital of state 1 is at    2
Capital of state 2 is at    4
Capital of state 3 is at    3
Capital of state 4 is at    1
----------
Chandigarh  is capital of Punjab
Patna       is capital of Bihar
Jaipur      is capital of Rajasthan
Gandhinagar is capital of Gujarat
```

Remark:

- Lists of strings are stored using arrays of character pointers in line number 6 and 7

Program 6-38 | A program that illustrates the use of an array of character pointers

6.9 Command Line Arguments

In the previous chapter, we have seen that inputs are given to the functions by means of arguments. main is also a function. Therefore, can we give inputs to the function main also by supplying arguments? The answer to this question is YES! Inputs to the function main are given by making use of special arguments known as **command line arguments**.

If you have used DOS, you must have used copy command. The copy command looks like:

copy source_file.txt dest_file.txt

To the copy program, the name of the source file (i.e. source_file.txt) and the name of the destination file (i.e. dest_file.txt) are given as inputs. These inputs are given at the **command line** or **command prompt** and are known **command line arguments.**

C provides a fairly simple mechanism for retrieving command line arguments entered by the user at the command line. To retrieve the command line arguments, the function main should be defined as:

```
main(int argc, char* argv[])        //←Header of the function main
{
  Statements.....
  Body.....
  Statements.....
}
```

In the header of the function main, two parameters are given, namely:

1. argc: The parameter argc stands for **argument count** and is of integer type.
2. argv: The parameter argv stands for **argument vector** and is an array of character pointers.

The names of parameters are dummy and can be anything like abc, xyz, etc. but generally the names argc and argv are used.

Suppose that on the command prompt, the user has entered:

```
prog opt1 opt2 sfile dfile
```

The important points about the given input are as follows:

1. The command line arguments are separated by blank spaces. In the given input, there are five arguments. The name of the program file (actually executable file) will also be counted while determining the argument count.
2. The parameter argc will receive a value equal to the number of arguments specified on the command prompt. In the given example, argc will have the value 5.
3. The first argument is the name of the program file (actually executable file). The file prog.exe should be present in the current working directory.
4. The contents of the parameter argv will be:

```
argv[0]="prog"
argv[1]="opt1"
argv[2]="opt2"
argv[3]="sfile"
argv[4]="dfile"
```

The contents of the array argv are shown in Figure 6.6.

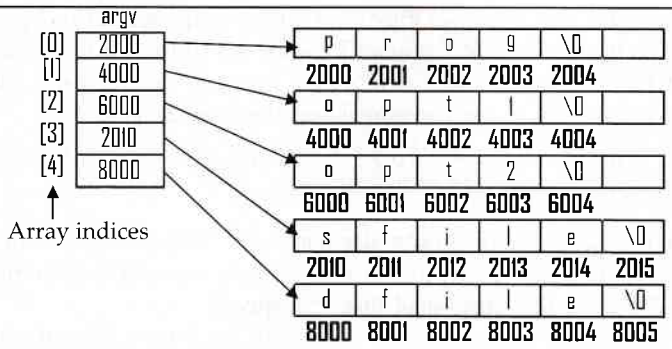


Figure 6.6 | Contents of the array argv

Program 6-39 illustrates the use of command line arguments.

Line	Prog 6-39 mycopy.c	Command prompt
1	//Command line arguments	c:\tc\bin>mycopy source.txt dest.txt
2	#include<stdio.h>	The number of arguments are 3
3	main(int argc, char* argv[])	Arguments are:
4	{	c:\tc\bin>mycopy.exe
5	int i=0;	source.txt
6	printf("The number of arguments are %d\n", argc);	dest.txt
7	printf("Arguments are:\n");	
8	for(i=0;i<argc;i++)	
9	printf("%s\n",argv[i]);	
10	}	

Program 6-39 | A program that illustrates the use of command line arguments

To execute Program 6-39, follow these steps:

1. Save the program with .c extension. Suppose the name given to the program file is mycopy.c.
2. Compile the program and check for compilation errors.
3. If there are no errors, build an executable file by invoking Make or Build all option in the Compile Menu of Turbo C 3.0 or by invoking Make all or Build all option in the Project menu, if using Turbo C 4.5. By default, the name of the executable file would be the same as the name of the program file. However, if a different name is given to the executable file, note it.
4. Observe the name and path of the directory in which the executable file is created.
5. Invoke the command prompt. Change the directory and make the current working directory the same as the directory in which the executable file was created.

6. Execute the program by writing the name of the executable file followed by blank separated arguments, e.g. `mycopy source.txt dest.txt`
7. If using Turbo C 3.0, the other way to execute Program 6-39 is by providing arguments from the IDE. Invoke Arguments... option is available in the Run Menu. Provide the arguments and execute the program. Note that if using this option, all the arguments except the name of the program file are to be provided. The name of the program is used by default and should not be specified.

Practically, the command line arguments are used in the applications that involve file handling.

Forward Reference: Files and file handling (Chapter 10).

6.10 Summary

1. A string literal is a sequence of zero or more characters enclosed within double quotes.
2. A string literal is automatically terminated by a null character.
3. A null character has an ASCII value of 0 and is written as '\0'.
4. Due to this additional null character, a string constant takes 1 byte more than the number of characters present in the string.
5. String literals are stored in character arrays.
6. In C language, string type is not separately available and character pointers are used to represent a string.
7. The type of string literal constants is `const char*`. The constant pointer refers to the address of the first character of the string.
8. Strings can be read from the keyboard by using `scanf` and `gets` functions.
9. The `scanf` function is used for reading single-word strings while the `gets` function can be used for reading multi-word strings.
10. Strings are printed on the screen by using `printf` and `puts` functions.
11. The `printf` function does not place a new line character after printing the string but the `puts` function places a new line character after printing the string.
12. The C string library provides a rich set of functionality to manipulate strings in the form of library functions like `strcpy`, `strcmp`, `strcat`, `strrev`, etc.
13. Real-time applications often require storage and processing of a number of strings at a time. A list of strings can be stored by using an array of strings or by using an array of character pointers.
14. The `main` function can also take string inputs from the command line. The arguments given to the function `main` from the command line are known as command line arguments.

Exercise Questions

Conceptual Questions and Answers

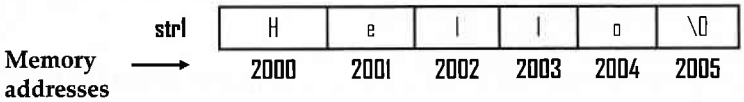
1. *What is a null character?*
   A character constant with an ASCII value of zero is known as the null character and is written as '\0'.
2. *What is a character string literal constant? How is it written and stored in the memory?*

Backward Reference: Refer Section 6.2 for a description on character string literal constants.

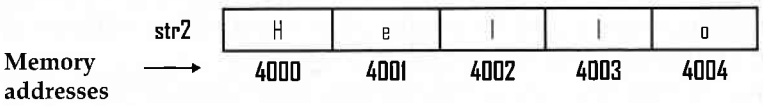
3. *What can the maximum number of characters in a character literal constant be?*
   The character constant can be one (e.g. 'A') or two (e.g. '\n') characters long. Hence, the maximum number of characters in a character literal constant can be two.
4. *What would be the size of the following arrays:*

```
char str1[]= "Hello";
char str2[]={'H','e','l','l','o'};
```

The character array str1 is initialized with a character string literal constant "Hello". Since a character string literal constant is terminated by a null character '\0', the contents stored in the character array str1 will be (say array is allocated at 2000):



The character array str2 is initialized with the five initializers in the initialization list. Hence, the contents of str2 will be (say array is allocated at 4000):



Therefore, the size of array str1 is 6 and that of str2 is 5.

5. *What are the different ways to print character arrays?*
   The following code illustrates four different ways to print character arrays:

```
main()
{
    char character_array[]="Example";
    int i;
    printf(character_array);                    //← Way 1
    printf("\n%s\n",character_array);           //← Way 2
    puts(character_array);                      //← Way 3
    for(i=0;character_array[i]!='\0';i++)       //← Way 4
        printf("%c",character_array[i]);
}
```

In way 1, the character array is printed without using any format specifier. The first argument of the printf function must be of type const char* and the array name character_array is implicitly

converted to pointer type char*. Since the types const char* and char* are compatible, the compiler implicitly converts char* to const char*. Therefore, this usage is perfectly valid. This type of usage however has a limitation that only one character array can be printed at a time.

In way 2, the character array is printed by using a %s format specifier. This type of usage has an advantage that many character arrays can be printed by a single call to the printf function by using multiple %s specifiers. For example:

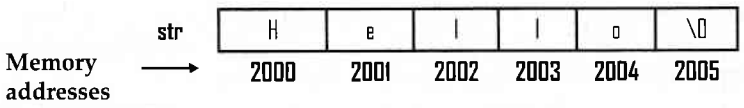
```
main()
{
    char character_array1[]="Hello";
    char character_array2[]="Readers";
    printf("%s %s",character_array1,character_array2);
}
```

In way 3, the puts function is used to print the character array. The difference between the puts and printf function is that the puts function places a new line character at the end, while the printf function does not do so.

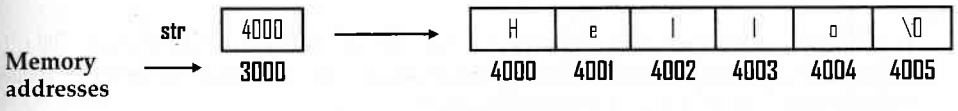
In way 4, a for loop is used to print all the characters of the array character_array one by one.

6. *Is the declaration* char str[6]="Hello" *same as* char *str="Hello"*?*

No, the declaration char str[6]="Hello"; is not the same as char *str="Hello";. The first declaration statement declares str to be a character array of size six and initializes the elements of array str with the characters of the string literal constant "Hello". However, the second declaration statement declares str to be a pointer to the character type and initializes it with the base address of string "Hello". The difference between the two declarations is shown in the figure below:



(a) char str[6]="Hello";



(b) char *str="Hello";

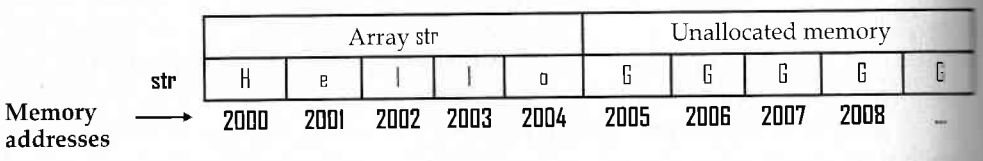
Another difference is that the first declaration statement allocates six bytes of the memory space to str, while the second declaration allocates two bytes to str (since it is a pointer).

It is very important to note that arrays are not pointers, although they are very closely related and sometimes have similar usage. For example, it is valid to write puts(str) and printf(str), where str is either declared by (a) or (b) as shown in the figure above.

7. *The following piece of code on execution gives some garbage. Why?*

```
main()
{
    char str[5]="Hello";
    puts(str);
}
```

The puts function outputs a sequence of characters (i.e. a string) on the screen. The output starts from the character pointed to by the pointer argument and is carried out till the null character is encountered.
The declaration char str[5]="Hello"; creates a character array of five locations and initializes the locations with the characters 'H', 'e', 'l', 'l' and 'o'. The array does not have the space to accommodate the null character. The array allocation and the memory contents are shown in the figure below:

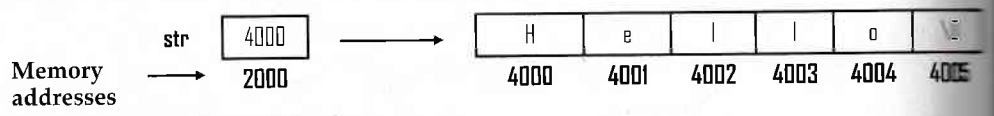


The function call puts(str) prints the characters starting from location 2000 till the null character is encountered. Since the character array str does not have the terminating null character, the output will be **Hello followed by some garbage characters**. The number of garbage characters depends upon where the null character (i.e. 0 value) is encountered in the memory. Execution of the same code at different times or on different machines may give a different number of garbage characters.

8. *Will the following piece of code also give some garbage as the previous code does?*

```
main()
 {
     char *str="Hello";
     puts(str);
 }
```

No, the mentioned piece of code outputs Hello and does not give any garbage character. The declaration char* str="Hello"; creates str as a 'pointer to character' and initializes it with the base address of string literal constant "Hello". This can be depicted as:



The function call puts(str) starts printing the characters from the memory location 4000 till the null character is encountered. Since there is a null character '\0' available at the memory location 4005, the output will be Hello only without any garbage.

9. *Why does the following piece of code not work? Rectify it.*

```
main()
{
    char string[15]="Hello Readers";
    strcat(string,'!');
    puts(string);
}
```

The following piece of code on compilation gives 'Cannot convert char to char*' error. The error is due to the fact that the strcat function expects two arguments of type char* (i.e. both the arguments should be strings). In the function call, strcat(string,'!'); the second argument is a character (i.e. of type char) and is not a string (i.e. of type char*). The conversion from type 'char to char*' is not a standard conversion, hence, the compiler will not carry it out implicitly and flags it as an error. The rectified call to the strcat function is strcat(string,"!");.

10. *What is the difference between* strchr *and* strrchr *functions?*

Backward Reference: Refer Sections 6.7.10 and 6.7.11 for a description on strchr and strrchr functions.

11. *Describe the behavior of the* scanf *function when applied on strings.*

Backward Reference: Refer Section 6.4 for a description on the behavior of the scanf function when applied on strings.

12. *The following piece of code compiles successfully. However, on execution gives an exception. Why? Rectify it.*

```
main()
{
    char *str;
    printf("Enter a string\t");
    gets(str);
    printf("The string entered was\t");
    puts(str);
}
```

The given code on execution gives an exception because before calling the gets function we have not allocated sufficient memory space to store the string entered by the user. There will be no compilation error because the gets function has no way to check whether the memory space pointed to by str is allocated or not.

The following are the rectified pieces of equivalent code:

```
#include<stdio.h>
main()
{
    char str[10];
    printf("Enter a string\t");
    gets(str);
    printf("The entered string was\t");
    puts(str);
}
```

Rectified code 1

```
#include<stdio.h>
#include<alloc.h>
main()
{
    char *str=(char*)malloc(10);
    printf("Enter a string\t");
    gets(str);
    printf("The entered string was\t");
    puts(str);
}
```

Rectified code 2

In the rectified code 1, str has been declared to be a character array of size 10. Hence, 10 bytes are allocated to str at the compile time. In the rectified code 2, the **malloc** (i.e. memory allocate) function is used to allocate 10 bytes of memory. malloc function returns a void pointer to the allocated memory space. The void* is type casted to char* and is assigned to str, i.e. str is made to point to the allocated memory space.

Some of the compilers like GNU GCC compiler, Borland Turbo C 3.0, etc. may not generate an exception, if the uninitialized pointer like str is used with the gets function.

13. *What would be the output of the following piece of code?*

```
main()
{
    char str[10]="ab\n\tcd";
```

```
    printf("Size of string is %d",strlen(str));
}
```

The given piece of code on execution outputs:

Size of string is 6

Character sequences like \n are interpreted at the compile time. When a backslash and an adjacent character n appear in a character constant or a string literal constant, they are immediately translated into a single new line character, i.e. one token. Similar translations also occur for other character escape sequences like \t, \b, \r, etc.

Hence, the string literal constant "ab\n\tcd" has six characters namely 'a', 'b','\n', i.e. new line character, '\t,' i.e. tab character, 'c' and 'd'.

14. *Consider the following piece of code:*

```
main()
{
    char str[10];
    gets(str);
    printf("Size of string is %d",strlen(str));
}
```

What would the output of the mentioned piece of code be, if the user entered the same string as in the previous question, i.e. "ab\n\tcd"*?*

On execution of the code, if the user enters the string "ab\n\tcd", the output of the code would be:

Size of string is 8

The output of this code is different from the output of the previous question because of the fact that when strings are taken from the user or read from a file at the run time, no interpretation of character sequences like \n, \t, etc. is performed. '\' and 'n' are treated as separate characters and are not transformed into single characters. The same is true for other escape sequences.

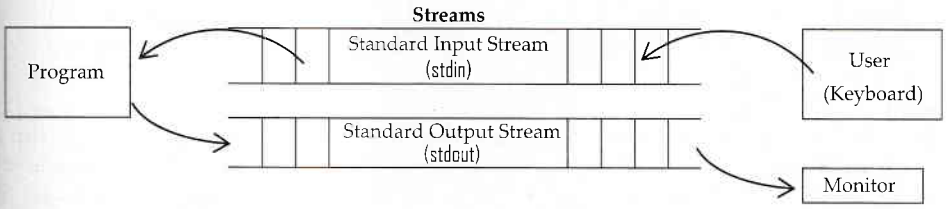
Hence, the string "ab\n\tcd" entered by the user at the run time has eight characters namely 'a', 'b', 'n', '\', 't', 'c' and 'd'.

15 *Consider the following piece of code:*

```
main()
{
    char str1[20],str2[20];
    printf("This code demonstrates two different ways to read strings\n");
    printf("Enter string 1\t");
    scanf("%s",str1);
    printf("Enter string 2\t");
    gets(str2);
    printf("\nThe strings entered were\n");
    puts(str2);
    puts(str1);
}
```

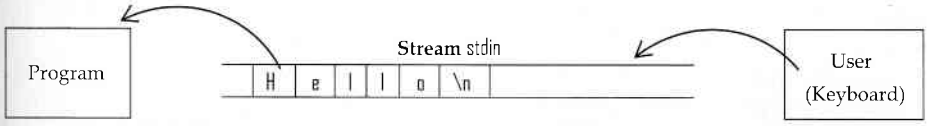
On execution, the code does not use the prompt to enter string 2 and directly starts printing the strings. Why?

The reason behind this behavior can be understood by learning how input and output are done in C. All the input and output in C are done with **streams.** A **stream** can be thought of as a buffer from which a sequence of data elements is made available during input or to which a sequence of data elements is written during output. The figure shown below depicts how input and output are done by means of input and output streams.



All the input functions like scanf, gets, getc etc. read from the **standard input stream** stdin and prompt the user to enter the data only if the stream is empty. If the stream already contains data or some characters, the input function will not prompt the user and silently retrieves the already available characters from the stream.

Suppose on execution of the given code, the user typed Hello and pressed the Enter key. The contents entered into the standard input stream are shown in the figure below.



The scanf function retrieves all the characters from stdin up to but not including the white-space character. Hence, after the execution of the function call scanf("%s",str1);, Hello is removed from stdin and is stored in str1 but the new line character still remains in the stream stdin. The call to the function gets(str2); finds the new line character in the stream. That is why it does not prompt the user to make input. It silently removes that new line character from stdin and stores it in str2.

This problem can be solved by removing the new line character from the stdin stream before giving call to the gets function. This can be done either by calling function flushall(); or fflush(stdin);. The rectified piece of code is as follows:

```
main()
{
    char str1[20],str2[20];
    printf("This code demonstrates two different ways to read strings\n");
    printf("Enter string 1\t");
    scanf("%s",str1);
    printf("Enter string 2\t");
    flushall();   //flushall(); flushes all the streams
                  //or fflush(stdin); flushes only stdin stream.
    gets(str2);
    printf("\nThe strings entered were\n");
    puts(str2);
    puts(str1);
}
```

Code Snippets

Determine the output of the following code snippets. Assume that the inclusion of the required header files has been made and there is no prototyping error due to them.

16.
```
main()
{
    char str1[]= "Strings";
    char str2[]={'S','t','r','i','n','g','s'};
```

```
        puts(str1);
        puts(str2);
    }
17. main()
    {
        char str[]="Strings";
        int i;
        for(i=0;str[i];i++)
        printf("%c",str[i]);
    }
18. main()
    {
        printf("%d %d",sizeof('A'),sizeof("A"));
    }
19. main()
    {
        char str1[]="Hello";
        char *str2="Hello";
        printf("%d %d\n",sizeof(str1),sizeof(str2));
        printf("%d %d",sizeof(*str1),sizeof(*str2));
    }
20. main()
    {
        char str[]="Characters";
        printf("%d %d",strlen(str) ,sizeof(str));
    }
21. main()
    {
        char str1[]="Hello";
        char str2[]="Readers!";
        printf("Hello ""Readers!""\n");
        puts("Hello ""Readers!");
        printf("%s %s",str1,str2);
    }
22. main()
    {
        char str1[]="Hello";
        char str2[]="Readers!",
        puts(str1,str2);
    }
23. main()
    {
        char str1[]="Hello";
        char str2[]="Readers!";
        puts((str1,str2));
    }
24. main()
    {
        char *str;
```

```
    str="Hello","Readers!";
    puts(str);
}
25. main()
{
    char *str;
    str=("Hello","Readers!");
    puts(str);
}
26. main()
{
    char str1[]="Hello";
    char str2[]="Readers!";
    printf(str1,str2);
}
27. main()
{
    char str[]="HelloReaders!";
    printf("%s %s %s",&str[5],&5[str],str+5);
}
28. main()
{
    char str[]="Hello Readers!";
    printf("%c %c %c",str[6],6[str],*(str+6));
}
29. main()
{
    printf("Hello Readers!"+6);
}
30. main()
{
    putchar("Hello Readers!"[6]);
    putchar(6["Hello Readers!"]);
}
31. main()
{
    printf("The size of string is %d\n",sizeof("Hello Readers!"));
    printf("The string is allocated memory starting at %p",&"Hello Readers!");
}
32. main()
{
    char str1[]="Strings!";
    char str2[]="Strings!";
    if(str1==str2)
        printf("Strings are same!!");
    else
        printf("Strings are different!!");
}
```

33.
```
main()
{
    char str1[]="Strings!";
    char str2[]="Strings!";
    if(strcmp(str1,str2)==0)
        printf("Strings are same!!");
    else
        printf("Strings are different!!");
}
```

34.
```
main()
{
    char str1[]="strings!";
    char str2[]="STRINGS!";
    if(strcmp(str1,str2)==0)
        printf("Strings are same!!");
    else
        printf("Strings are different!!");
}
```

35.
```
main()
{
    char str1[]="strings!";
    char str2[]="STRINGS!";
    if(strcmpi(str1,str2)==0)
        printf("Strings are same!!");
    else
        printf("Strings are different!!");
}
```

36.
```
main()
{
    if(strcmp("Strings","Strings\0"))
        printf("Strings are different!!");
    else
        printf("Strings are same!!");
}
```

37.
```
main()
{
    char str1[]={'S','t','r','i','n','g','s'};
    char str2[]="Strings";
    if(strcmp(str1,str2))
        printf("Strings are different!!");
    else
    printf("Strings are same!!");
}
```

38.
```
main()
{
    char format[]="%d\n";
    format[1]='c';
```

```
    printf(forma
}
39. main()
{
    char format
    printf(forma
}
40. main()
{
    char str1[]=
    char str2[1
    str2=str1;
    puts(str1);
    puts(str2);
}
41. main()
{
    char src[]=
    char dest[1
    strcpy(dest
    puts(src);
    puts(dest);
}
42. main()
{
    char dest[
    char src[]=
    puts(strcp
}
43. main()
{
    char dest[
    char src[]
    strcpy(&de
    puts(dest)
}
44. main()
{
    char dest[
    char src[]
    strcpy(&d
    puts(dest)
}
45. main()
{
    if(printf("
        printf(
    else
```

```
    printf(format,65);
}
39. main()
{
    char format[]={37,111,32,37,120,0};
    printf(format,format[0],format[1]);
}
40. main()
{
    char str1[]="Strings";
    char str2[10];
    str2=str1;
    puts(str1);
    puts(str2);
}
41. main()
{
    char src[]="Strings";
    char dest[10];
    strcpy(dest,src);
    puts(src);
    puts(dest);
}
42. main()
{
    char dest[]="Visual Basic";
    char src[]="C++";
    puts(strcpy(&dest[7],src));
}
43. main()
{
    char dest[]="Visual Basic";
    char src[]="C++";
    strcpy(&dest[7],src);
    puts(dest);
}
44. main()
{
    char dest[]="Visual Basic";
    char src[]="Visual C++";
    strcpy(&dest[7],&src[7]);
    puts(dest);
}
45. main()
{
    if(printf("\0"))
        printf("Characters");
    else
```

```
        printf("Strings");
    }
46. main()
    {
        char cities[][11]={"Delhi","Chandigarh","Noida"};
        int i;
        for(i=0;i<3;i++)
            puts(cities[i]);
    }
47. main()
    {
        char languages[5][20]={"Visual Basic","Java","Fortran","C","C++"};
        int i; char *t;
        t=languages[3];
        languages[3]=languages[4];
        languages[4]=t;
        for (i=0;i<=4;i++)
            printf("%s\n",languages[i]);
    }
48. main()
    {
        char *languages[]={"Basic","Java","Fortran","C","C++"};
        int i; char *t;
        t=languages[3];
        languages[3]=languages[4];
        languages[4]=t;
        for (i=0;i<=4;i++)
            printf("%s\n",languages[i]);
    }
49. main()
    {
        char lang[5][20]={"Visual Basic","Java","Fortran","C","C++"};
        int i; char *t;
        t=lang[0];
        while(*t++!=32);
            for(i=0;i<5;i++)
            {
                puts(lang[0]);
                strcpy(t,lang[i+1]);
            }
    }
50. main()
    {
        int i,len;
        char *ptr="String";
        len=strlen(ptr);
        for(i=0;i<len;i++)
```

```
    {
        puts(ptr);
        ptr++;
    }
}
51. string_manipulation(char[][]);
main()
{
    char arr[][10]={"Hello","Students"};
    string_manipulation(arr);
    printf("%s %s",arr[0],arr[1]);
}
string_manipulation(char arr[][])
{
    strcpy(arr[1],"Readers!!");
}
52. string_manipulation(char(*)[10]);
main()
{
    char arr[][10]={"Hello","Students"};
    string_manipulation(arr);
    printf("%s %s",arr[0],arr[1]);
}
string_manipulation(char (*arr)[10])
{
    strcpy(arr[1],"Readers!!");
}
53. string_manipulation(char[][10]);
main()
{
    char arr[][10]={"Hello","Students"};
    string_manipulation(arr);
    printf("%s %s",arr[0],arr[1]);
}
string_manipulation(char arr[][10])
{
    strcpy(arr[1],"Readers!!");
}
54. char[20] print_string()
{
    char str[20]="Strings!!";
    return str;
}
main()
{
 puts(print_string());
}
```

55.
```
main(int argc,char*argv[])
{
    int i;
    printf("The argument count is %d\n",argc);
    printf("The content of argument vector i.e. array is\n");
    for(i=0;i<argc;i++)
        printf("%s\n",argv[i]);
}
```

Suppose the name of the program file is ques55.c *and the executable file* ques55.exe *is invoked from the command prompt as follows:*

```
c:\>ques55 Hello Readers!!
```

Multiple-choice Questions

56. The maximum number of characters in a character literal constant can be
    a. One
    b. Two
    c. Three
    d. As many as the user likes

57. The size occupied by a string literal constant in the memory is
    a. One more than the number of characters in the string
    b. Same as the number of characters in the string
    c. One less than the number of characters in the string
    d. None of these

58. The value returned by the strlen function when a string literal constant is given to it as an argument is
    a. One more than the number of characters in the string argument
    b. Same as the number of characters in the string argument
    c. One less than the number of characters in the string argument
    d. None of these

59. String literal constants are terminated by
    a. New line character
    b. Carriage return character
    c. Null character
    d. None of these

60. The ASCII code of the null character is
    a. 32
    b. 27
    c. 13
    d. 0

61. The output of the statement printf("%d","123456"[1]); is
    a. 1
    b. 2
    c. 50
    d. None of these

62. The output of the statement printf("%s","123456"+1); is
    a. 123456
    b. 123457
    c. 23456
    d. None of these

63. The correct way to compare two string literal constants "Hello" and "Hi" is
    a. "Hello"="Hi"
    b. "Hello"=="Hi"
    c. strcmp("Hello","Hi")
    d. None of these

The output of the statement puts("\0ABCD\0"); is

a. ABCD
b. No output
c. \0ABCD
d. Compilation error

The result of evaluation of the expression strcmp("Hello","Hi"); will be

a. 0
b. 4
c. −4
d. None of these

The correct statement to copy a string literal constant "Hello" to a character array str is

a. str="Hello";
b. strcpy(str,"Hello");
c. strcpy("Hello",str);
d. None of these

Adjacent string literal constants

a. Are always concatenated
b. Are treated as two separate tokens
c. Leads to compilation error
d. None of these

The invocation of the function call strcat("Hi","Readers!!"); leads to

a. HiReaders!!
b. Compilation error
c. Run-time exception
d. None of these

The invocation of the function call puts("Hi","Readers!!"); leads to

a. HiReaders!!
b. Compilation error
c. Run-time exception
d. None of these

The invocation of the function call puts("Hi""Readers!!"); leads to

a. HiReaders!!
b. Compilation error
c. Run-time exception
d. None of these

The output of the following program file ques71.c, if executed from the command line as ques71 1 2 3, is

```
main(int argc, char* argv[])
{
   int val;
   val=argv[1]+argv[2]+argv[3];
   printf("%d",val);
}
```

a. 6
b. 123
c. Compilation error
d. None of these

The output of the following program file ques72.c, if executed from the command line as ques72 1 2 3, is

```
#include<stdlib.h>          //←atoi function converts string to an integer and its
                            //←prototype is in stdlib.h
main(int argc, char* argv[])
{
   int val;
   val=atoi(argv[1])+atoi(argv[2])+atoi(argv[3]);
   printf("%d",val);
}
```

a. 6
b. 123
c. Compilation error
d. None of these

73. The output of the following program file ques73.c, if executed from the command line as ques73
```
main(int argc, char* argv[])
{
    char str[10];
    strcpy(str,argv[1]);
    strcpy(str,argv[2]);
    printf("%s",str);
}
```
a. 1
b. 2
c. 12
d. None of these

74. The output of the following program file ques74.c, if executed from the command line as ques74
```
main(int argc, char* argv[])
{
    char str[10];
    strcpy(str,argv[1]);
    strcat(str,argv[2]);
    printf("%s",str);
}
```
a. 1
b. 2
c. 12
d. None of these

75. Which of the following is true about argv?
a. It is an array of character pointers
b. It is a pointer to an array of character pointers
c. It is an array of characters
d. None of these

Outputs and Explanations to Code Snippets

16. Strings
Strings! ¥¤§¶

Explanation:

As the character array str1 is initialized with the character string literal constant, str1[7] will be null character. However, as the character array str2 is initialized with a list of characters, i.e. 't', 'r', 'i', 'n', 'g' and 's', no terminating null character is placed in it. Hence, the puts function while printing str2 gives garbage as it prints from the memory location pointed to by its argument till the terminating null character is encountered.

17. Strings

Explanation:

The for loop is used to print the elements of character array str one by one. The loop terminates when the value of i becomes 7 and str[i] evaluates to 0 (i.e. the ASCII value of null character). for(i=0;str[i];i++) is equivalent to writing for(i=0;str[i]!='\0';i++).

18. 1 2

Explanation:

'A' is a character constant and characters take one byte in the memory. "A" is a string literal constant and string literal constants are terminated by a null character, i.e. '\0'. So "A" is actually made up of two characters, i.e. 'A' and '\0'. Hence, sizeof("A") comes out to be 2.

19. 6 2
    1 1

Explanation:

str1 is a character array of 6 locations while str2 is a pointer to character. Hence, size of str1 and str2 would be 6 and 2, respectively (in Borland TC 3.0 for DOS), and 6 and 4 (in Borland TC 4.5 for Windows or Microsoft Visual C++ 6.0). The usage of * with str1 and str2 dereferences them to a character and hence, sizeof(*str1) and sizeof(*str2) would be 1 and 1.

20. 10 11

Explanation:

The strlen function computes the length of a string given to it as an argument. The strlen function does not count the null character while computing the length of the string. It returns the number of characters that precedes the terminating null character. On the other hand, the sizeof function also counts the memory required by the null character while computing the number of memory bytes occupied by the string.

21. Hello Readers!
    Hello Readers!
    Hello Readers!

Explanation:

Adjacent character string literal constants are concatenated. Hence, writing "Hello ""Readers!""\n" is equivalent to writing "Hello Readers!\n". The printf function outputs this character string literal constant onto the screen. The puts function also does the same with a difference that it places the new line character at the end. Hence, there is no requirement of the new line character in the string given to puts. The last call to the printf function uses %s specifiers to output the contents of the character arrays str1 and str2.

22. Compilation error "Extra parameter in call to puts"

Explanation:

The puts function expects only one argument of char* type while in the call puts(str1,str2); two arguments of type char* are provided. Hence, there is an extra parameter in the function call, which is the source of error.

23. Readers!

Explanation:

The argument of the puts function is an expression (str1,str2). Now, the instance of the comma symbol separating str1 and str2 in the expression (str1,str2) is treated as a comma operator. The comma operator guarantees left-to-right evaluation and returns the result of the rightmost sub-expression. Therefore, the expression (str1,str2) evaluates to str2. Hence, the string Readers! gets printed.

24. Hello

Explanation:

The string literal constant refers to the address of its initial element except in the cases when it is an operand of the sizeof operator or the unary & operator. Hence, "Hello" and "Readers!" refer to the starting addresses of the strings "Hello" and "Readers!". In the expression, str="Hello","Readers!", the assignment operator has a higher priority as compared to the comma operator. Hence, the assignment operator is evaluated first and the starting address of the string literal constant "Hello" is assigned to str. In the next statement, the string pointed to by str is printed by the puts function. Hence, "Hello" is the output.

25. Readers!

Explanation:

The use of parentheses makes the comma operator to be evaluated first. The comma operator returns the result of the rightmost sub-expression. Therefore, in the expression str=("Hello","Readers"), the starting address of the string "Readers!" is assigned to str. In the next statement, the string pointed to by str is printed by the puts function. Hence "Readers!" is the output.

26. Hello

Explanation:

On compilation, the given code does not produce a compilation error as the code of Question number 22 does. This is due to the fact that printf is a variable argument function. It can take a variable number of arguments while puts can only take one argument. Examples when the printf function takes 1, 2 and 3 arguments are as follows:

```
printf("Hello Readers");     //← Only one argument
printf("%d",2);              //← Two arguments
printf("%d %d",2,3);         //← Three arguments
```

The following important points should also be remembered:

1. The comma symbol appearing in a printf function call is not treated as a comma operator.
2. The printf function expects the first argument to be a string (commonly known as a format string). It actually prints only the first argument while the other arguments available in the printf function replace the format specifiers in the first string, if they are present. If no format specifiers are present in the format string, the arguments following the first argument are ignored.

Hence, the output of printf(str1, str2); is Hello, as the string str1 does not contain a format specifier.

27. Readers! Readers! Readers!

Explanation:

The declaration statement char str[]="HelloReaders!"; allocates str, say at the memory location 2000. The contents of the array are shown in the figure below:

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]
str	H	e	l	l	o	R	e	a	d	e	r	s	!	\0
	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013

The printf function prints the sequence of characters from the address given as an argument till a null character is encountered. All the expressions &str[5], &5[str] and str+5 evaluate to 2005. Therefore, the printing starts from the character present at the location 2005 and is carried out till a null character is encountered.

28. R R R

Explanation:

The expressions str[6], 6[str] and *(str+6) refer to the seventh character of the array str, i.e. R.

29. Readers!

Explanation:

Suppose the string literal constant "Hello Readers!" is allocated the memory space from the address 2000 to 2014 as depicted in the figure below:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
H	e	l	l	o		R	e	a	d	e	r	s	!	\0
2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014

String literal constant refers to the address of its initial element except in the cases when it is an operand of the sizeof operator or the unary & operator. Hence, the expression "Hello Readers!" evaluates to 2000, and the expression "Hello Readers!"+6 evaluates to 2006. When the expression "Hello Readers!"+6 is given as an argument to the printf function, the printf function starts printing the characters from 2006 till a null character is encountered. Hence, the output comes out to be Readers!.

30. RR

Explanation:

The function putchar outputs the character given to it as an argument on the screen. Suppose the string "Hello Readers!" is allocated the same memory location as in Answer number 29.

In the first call to the function putchar, the argument is an expression "Hello Readers!"[6]. This expression gets converted to the form *("Hello Readers!"+6). The expression *("Hello Readers!"+6) evaluates to *(2000+6), i.e. *(2006), i.e. R (refer to the explanation given in Answer number 29). Similarly, 6["Hello Readers!"] evaluates to R.

Backward Reference: Refer to the explanation given in Answer number 14 (Chapter 4).

31. The size of string is 15
The string is allocated memory starting at 2A4F:00A0

Explanation:

The string literal constant expression does not decompose into the pointer to its initial element when it is an operand of the sizeof operator or the unary & operator.

32. Strings are different!!

Explanation:

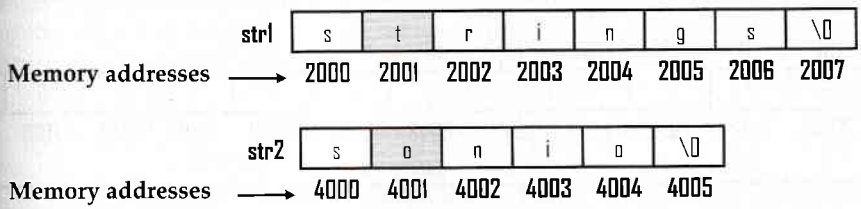
In the expression str1==str2, str1 and str2 are the names of character arrays and refer to the addresses of their first elements. Since the addresses of the first element of two arrays can never be the same, the expression str1==str2 evaluates to false and Strings are different!! is the output.

33. Strings are same!!

Explanation:

strcmp(str1,str2) performs a comparison between str1 and str2, starting with the first character in each string and continuing with the subsequent characters until the corresponding characters differ or until the end of strings is reached. It returns the ASCII difference of the first dissimilar corresponding characters or zero if none of the corresponding characters in both the strings are different.

For example, when strcmp function is applied on the strings str1 (say strings) and str2 (say sonia) shown in the figure below, it returns 116-111 (i.e. ASCII code of 't' – ASCII code of 'o') = 5.



strcmp(str1,str2) returns a value equal to:
- 0 if str1 and str2 are equal, or
- >0 if str1 is greater than str2, i.e. str1 comes after str2 in lexicographic order, or
- <0 if str1 is lesser than str2, i.e. str1 comes before str2 in lexicographic order.

In the given question, str1 and str2 being the same, the expression strcmp(str1,str2)==0 evaluates to true as the strcmp function returns 0. Hence, Strings are same!! is the output.

34. Strings are different!!

Explanation:

The strcmp function when used to compare str1 and str2 returns 115-83 (i.e. ASCII code of 's'-ASCII code of 'S') = 32. The returned value is not equal to zero. Hence, the expression strcmp(str1,str2)== evaluates to false and Strings are different!! is the output. Note that the strcmp function considers the case sensitivity of the characters while comparing the strings.

35. Strings are same!!

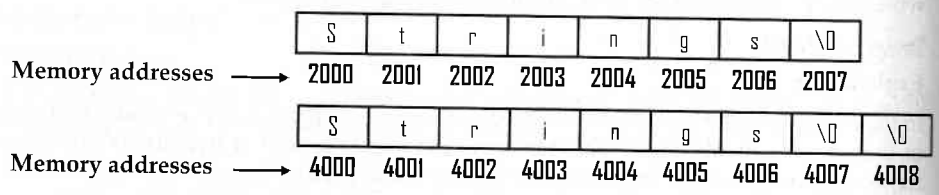
Explanation:

The strcmpi function compares the strings without case sensitivity. The character i in the strcmpi function stands for ignore case.

36. Strings are same!!

Explanation:

Suppose, the character string literal constants "Strings" and "Strings\0" are allocated the memory space at the addresses 2000 and 4000 as shown in the figure below:

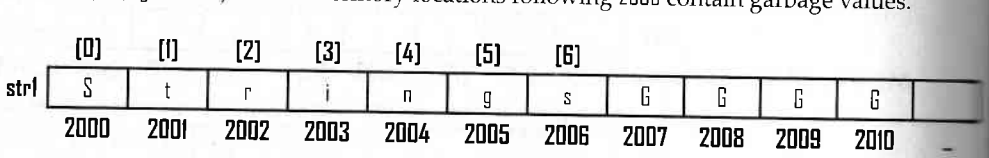


The function call strcmp("Strings","Strings\0") compares characters of both the strings one by one until the corresponding characters differ or until the end of the strings is reached. In the given piece of code, the function call terminates by comparing the null characters located at the locations 2007 and 4007 and returns 0. Hence, Strings are same!! is the output.

37. Strings are different!!

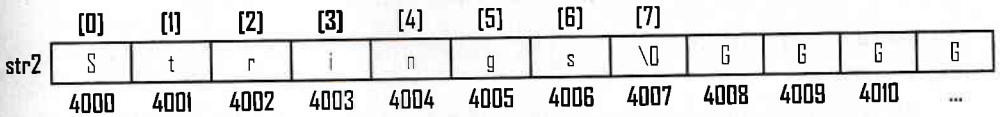
Explanation:

Suppose that the character array str1 gets allocated at the memory address 2000 as shown in the figure below. The first seven elements of the character array str1 are initialized with the characters 'S', 't', 'r', 'i', 'n', 'g' and 's', and the memory locations following 2006 contain garbage values.



G (in the above figure) means garbage value.

The contents of the character array str2 allocated at the memory address 4000 are shown in the figure below:

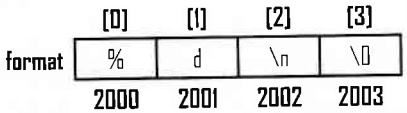


The function strcmp(str1,str2) returns the ASCII difference of the first dissimilar corresponding characters or zero if there is no dissimilarity. The first dissimilarity in str1 and str2 is in the characters located at 2007 and 4007, respectively. Hence, the function strcmp returns the difference between garbage value G and 0 (i.e. the ASCII code of the null character). There is high probability that this garbage value is a non-zero value, and hence Strings are different!! is the output. However, by chance, if the garbage value located at 2007 is 0 (which has lesser probability), then the output will be Strings are same!!.

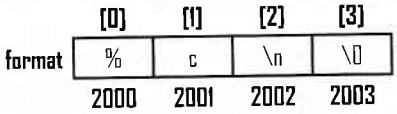
38. A

Explanation:

The contents of the character array format after initialization are shown in the figure below:



After the execution of the assignment statement format[1]='c'; the contents of the character array format become:

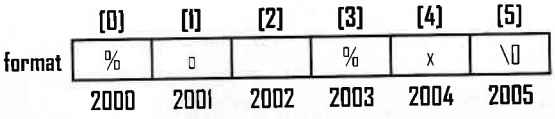


Writing printf(format,65); is equivalent to writing printf("%c\n",65);. Printing of integer value 65 is done according to the %c format specifier; hence A is the output (since 65 is the ASCII value of 'A').

39. 45 Gf

Explanation:

The character array format is initialized with an initialization list consisting of integer values. If the initialization list of a character array consists of integer values, then the locations of the array are initialized with the characters whose ASCII values are equivalent to the integer values in the initialization list. The characters having an ASCII value of 37 is '%', 111 is 'o', 32 is ' ', (i.e. blank space), 120 is 'x' and 0 is '\0', i.e. null character. The initialized contents of the character array format are shown in the figure below:



Now, printf(format,format[0],format[1]); prints the value of format[0] (i.e. 37) and format[1] (i.e. 111) according to %o and %x format specifiers, respectively. Hence, the output comes out to be 45 and 6f as the octal equivalent of 37 is 45 and the hexadecimal equivalent of 111 is 6f.

40. Compilation error "L-value required"

Explanation:

str2 is the name of the character array and is a constant object. It cannot be placed on the left side of an assignment operator. Hence, writing str2=str1 is not valid and gives 'L-value required' error.

41. Strings
Strings

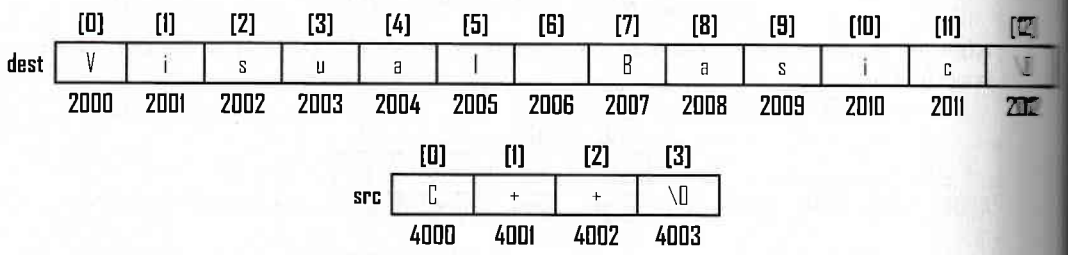
Explanation:

The strcpy(dest,src); copies the string pointed by src into the memory location pointed to by dest. The copying terminates after the terminating null character of src has been copied to dest. The strcpy function returns the starting address of the memory location where the string has been copied. Thus, after the execution of the function call strcpy(dest,src); both str and dest contain "Strings", and their contents are printed using the puts function.

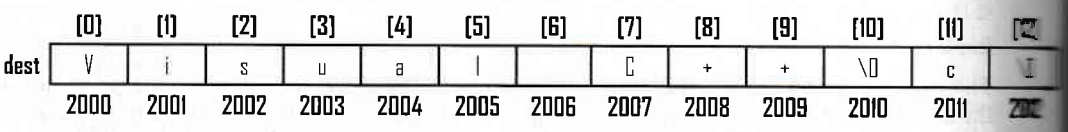
42 C++

Explanation:

The contents of the character array dest and src are shown in the figure below:



The function call strcpy(&dest[7],src); copies the contents of the source string src to the memory locations starting from 2007, i.e. &dest[7]. The contents of dest after the function call are as follows:



The strcpy function returns the address of the memory location where the string has been copied. Hence, strcpy(&dest[7],src); returns 2007. The puts prints the sequence of characters starting from the memory location 2007 till a null character is encountered. Hence, C++ is the output.

43. Visual C++

Explanation:

Backward Reference: Refer to the explanation given in Answer number 42.

44. Visual C++

Explanation:

Backward Reference: Refer to the explanation given in Answer number 42.

45. Strings

Explanation:

The printf function returns an integer value equivalent to the number of characters printed. Printing of the null character using the printf function returns zero. Hence, Strings is the output.

46. Delhi
Chandigarh
Noida

Explanation:

The content of a two-dimensional character array cities is shown in the figure below:

cities	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
[0]	D	e	l	h	i	\0					
	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010
[1]	C	h	a	n	d	i	g	a	r	h	\0
	2011	2012	2013	2014	2015	2016	2017	2018	2019	2020	2021
[2]	N	o	i	d	a	\0					
	2022	2023	2024	2025	2026	2027	2028	2029	2030	2031	2032

Referring to a two-dimensional array with only one subscript gives the starting address of a row. Hence, the expression cities[0] refers to the starting address of the first row, i.e. 2000, and cities[1] refers to the starting address of the second row, i.e. 2011. The function call puts(cities[i]), prints the strings in the first, second and third rows.

47. Compilation error "L-value required"

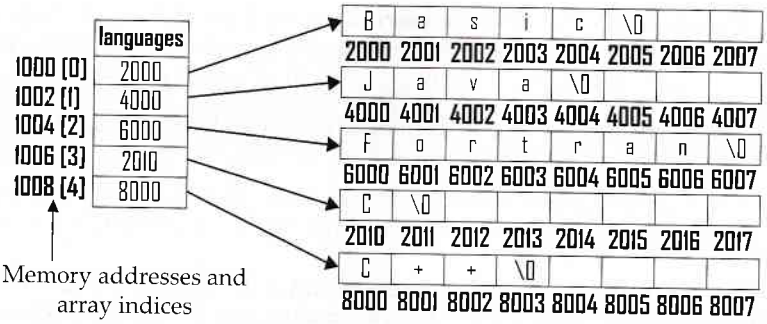
Explanation:

Referring to a two-dimensional array with only one subscript refers to the starting address of the row and is a constant object. The C compiler will not allow its manipulation. Hence, writing languages[3]=languages[4] is not valid and leads to 'l-value required' compilation error.

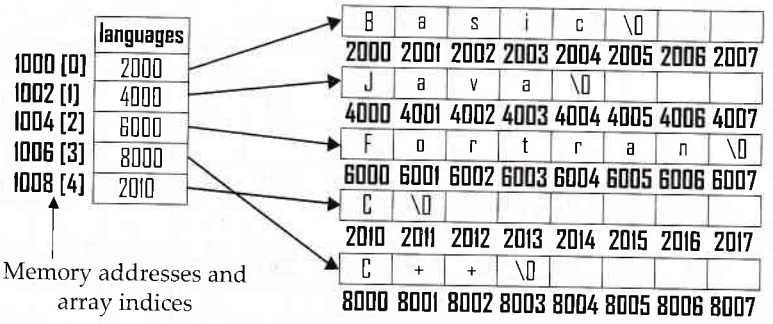
48. Basic
Java
Fortran
C++
C

Explanation:

languages is an array of character pointers and is initialized with the base addresses of the string literal constants "Basic", "Java", "Fortran", "C" and "C++". Contiguous memory (say from the memory address 1000-1009) is allocated to the array languages while the string literal constants are placed randomly in the memory. This is depicted in the figure below:



The statements t=languages[3];, languages[3]=languages[4]; and languages[4]=t; swap the values of languages[3] and language[4]. After the execution of these statements, the contents of array languages are as depicted in the figure below:



Thus, the printing of the strings pointed by the content of the array languages yields the mentioned result.

49. Visual Basic
Visual Java
Visual Fortran
Visual C
Visual C++

Explanation:

The content of the two-dimensional character array lang is shown below:

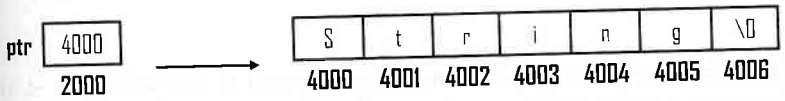
lang	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]
[0]	V	i	s	u	a	l		B	a	s	i	c	\0
	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012
[1]	J	a	v	a	\0								
	2013	2014	2015	2016	2017	2018	2019	2020	2021	2022	2023	2024	2025
[2]	F	o	r	t	r	a	n	\0					
	2026	2027	2027	2028	2029	2030	2031	2032	2033	2034	2035	2036	2037
[3]	C	\0											
	2038	2039	2040	2041	2042	2043	2044	2045	2046	2047	2048	2049	2050
[4]	C	+	+	\0									
	2051	2052	2053	2054	2055	2056	2057	2058	2059	2060	2061	2062	2063

The character pointer t, say, gets allocated at 4000. After the execution of the assignment statement t=lang[0]; t starts pointing to the starting address of the first row of the character array lang. After the execution of while(*t++!=32); statement, t points to the location next to the blank space (ASCII value 32) in the first row of lang, i.e. 2007. Each iteration of the for loop with the loop counter value i prints the content of lang[0] and copies the strings in the row i+1 at the memory location pointed by t, i.e. 2007.

50. String
tring
ring
ing
ng
g

Explanation:

Suppose the character pointer ptr and the character string literal constant "String" are allocated the memory space as shown in the figure below. Since is initialized with the character string literal constant, it points to the starting address of the string literal, i.e. 4000.



Every iteration of the for loop prints a string being pointed by ptr and increments the contents of the pointer ptr.

51. Compilation error

Explanation:

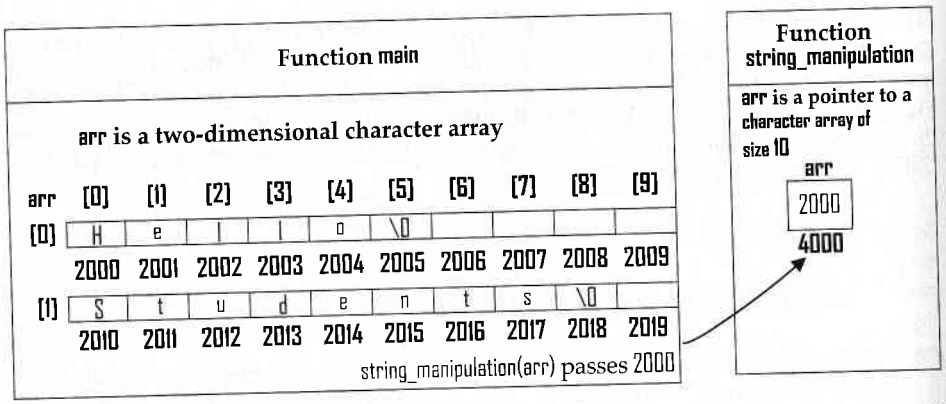
Backward Reference: Refer the explanation given in Answer number 59 (Chapter 4).

- While declaring a two-dimensional array, both the row size and column size cannot be left blank. It is mandatory to mention the column size specifier. Hence, the declaration string_manipulation(char [][]); is not valid. It can be rectified by mentioning the column size specifier as string_manipulation(char [][10]);. The same should also be done in the function header.

52. Hello Readers!!

Explanation:

The two-dimensional character array arr is passed by reference to the string_manipulation function. Suppose the array arr (local to the function main) gets allocated at the memory location 2000. The name arr declared in the function header string_manipulation is of type pointer to the character array of size 10 and is local to the function string_manipulation. Suppose it gets allocated at the memory location say 4000. The name of a two-dimensional array refers to the starting address of the first row of the array. Therefore, the function call string_manipulation(arr); passes 2000, i.e. the starting address of the first row of the array to the function string_manipulation. This is shown in the figure below:



The call to the function strcpy inside the body of function string_manipulation copies the string "Readers!!" at arr[1], i.e., 2010 (because arr is a pointer to a character array of size 10). Thus, the string "Readers!!" overwrites the string "Students" present in the second row of the character array arr. That is why when arr[0] and arr[1] are printed, the output comes out to be Hello Readers!!.

53. Hello Readers!!

Explanation:

The parameter declaration char arr[][10] gets implicitly converted to char(*arr)[10]. Thus, the mentioned code becomes equivalent to the code given in Question number 52.

Backward Reference: Refer to the explanation given in Answer number 52 for the output.

54. Compilation error

Explanation:

The return type of a function cannot be an array type. Since the return type of the function print_string is an array type, i.e. char [20], the compiler issues an error message.

55. The argument count is 3
The content of argument vector i.e. array is
c:\ques55.exe
Hello
Readers!!

Explanation:

Backward Reference: Refer to the explanation given in Section 6.9.

Arguments in command line are separated by blank spaces. Since there are two blank spaces, the total number of command line arguments is three. Note that the name of the program file (actually executable file) is also counted while determining the argument count. argv[0] points to c:\>ques55.exe, argv[1] points to Hello and argv[2] points to Readers!!. Therefore, these strings get printed.

Answers to Multiple-choice Questions

56. b 57. a 58. b 59.c 60. d 61. c 62. c 63. c 64. b 65. c 66. b 67. a 68. c
69. b 70.a 71. c 72. a 73. b 74. c 75. a

Programming Exercises

Program 1 | Input a string and find the number of vowel(s) present in the string

PE 6-1.c

```
1  //Number of vowels in a string
2  #include<stdio.h>
3  main()
4  {
5  char string[200];
6  int count=0, i=0;
7  printf("Enter a string:\n");
8  gets(string);
9  while(string[i]!='\0')
10 {
11     switch(string[i])
12     {
13         case 'A':
14         case 'E':
15         case 'I':
16         case 'O':
17         case 'U':
18         case 'a':
19         case 'e':
20         case 'i':
21         case 'o':
22         case 'u':
23             count++;
24         }
25     i++;
26 }
27 if(count==1)
28  printf("One vowel is present in the string");
29 else
30  printf("%d vowels are present in the string", count);
31 }
```

Output window

```
Enter a string:
There is nothing more beautiful in the world than a healthy wise old man- Yutang
25 vowels are present in the string
```

Program 2 | Input a string and count the number of occurrences of a particular character in the string

PE 6-2.c

```
1  //Count number of occurrences of a particular character in the string
2  #include<stdio.h>
3  main()
4  {
5      char string[200], ch;
6      int count=0, i=0;
7      printf("Enter a string:\n");
8      gets(string);
9      printf("Enter the character:\t");
10     scanf("%c",&ch);
11     while(string[i]!='\0')
12     {
13         if(string[i]==ch)
```

Output window

```
Enter a string:
Nature, time and patience are three great physicians- Bohn
Enter the character: e
In the given string, e occurred 8 times
```

(Contd...)

Line	PE 6-2.c	Output window
14	count++;	
15	i++;	
16	}	
17	printf("In the given string, %c occurred %d times\n",ch, count);	
18	}	

Program 3 | Input a string and count the number of blank spaces in the string

Line	PE 6-3.c	Output window
1	//Number of blank spaces	Enter a string:
2	#include<stdio.h>	People resent a joke if there is some truth in it- Tagore
3	main()	Number of blank spaces in the given string are 11
4	{	
5	char string[200], ch;	
6	int count=0, i=0;	
7	printf("Enter a string:\n");	
8	gets(string);	
9	while(string[i]!='\0')	
10	{	
11	if(string[i]==' ')	
12	count++;	
13	i++;	
14	}	
15	printf("Number of blank spaces in the given string are %d", count);	
16	}	

Program 4- Input two strings of equal length from the user and determine how many times the corresponding positions in two strings hold exactly the same characters

Line	PE 6-4.c	Output window
1	//Number of same characters at the corresponding positions in two strings	Enter two strings of equal length
2	#include<stdio.h>	Enter first string: choice
3	#include<string.h>	Enter second string: chance
4	#include<stdlib.h>	Corresponding positions hold same characters 4 times
5	main()	**Output window (second execution)**
6	{	Enter two strings of equal length
7	char str1[30], str2[30];	Enter first string: very
8	int length1, length2, count=0, i;	Enter second string: much
9	printf("Enter two strings of equal length\n");	Corresponding positions hold same characters 0 times
10	printf("Enter first string:\t");	**Output window (third execution)**
11	gets(str1);	Enter two strings of equal length
12	printf("Enter second string:\t");	Enter first string: life
13	gets(str2);	Enter second string: lovely
14	length1=strlen(str1);	The entered strings are of different lengths
15	length2=strlen(str2);	
16	if(length1!=length2)	
17	{	
18	printf("The entered strings are of different lengths\n");	
19	exit(1);	
20	}	

(Contd...

```
    else
    {
        for(i=0;i<length1;i++)
            if(str1[i]==str2[i])
                count++;
        printf("Corresponding positions hold same characters %d times", count);
    }
}
```

Program 5 | Input a string and display the alternate characters of the string

Line	PE 6-5.c	Output window
1	//Printing alternate characters of a string	Enter a string:
2	#include<stdio.h>	Hatred is preferable to the friendship of fools
3	main()	Alternate characters in the string are:
4	{	Hte speeal otefinsi ffos
5	char str[200], altchars[200];	
6	int i=0, length, j=0;	
7	printf("Enter a string:\n");	
8	gets(str);	
9	length=strlen(str);	
10	while(i<length)	
11	{	
12	altchars[j]=str[i];	
13	i=i+2;	
14	j=j+1;	
15	}	
16	altchars[j]='\0';	
17	printf("Alternate characters in the string are:\n");	
18	puts(altchars);	
19	}	

Program 6 | Input a string and display the alternate characters of the string in the reverse order

Line	PE 6-6.c	Output window
1	//Printing alternate characters of a string in the reverse order	Enter a string:
2	#include<stdio.h>	Harmony in character gains goodwill even from strangers
3	#include<string.h>	Alternate characters of the string in reverse order are:
4	main()	senrsmr eelido na ecrh iyorH
5	{	
6	char str[200], altchars[200];	
7	int i=0, length, j=0;	
8	printf("Enter a string:\n");	
9	gets(str);	
10	length=strlen(str);	
11	i=length-1;	
12	while(i>=0)	
13	{	
14	altchars[j]=str[i];	
15	i=i-2;	

(Contd...)

Line	PE 6-6.c	Output window
16	j=j+1;	
17	}	
18	altchars[j]='\0';	
19	printf("Alternate characters of the string in reverse order are:\n");	
20	puts(altchars);	
21	}	

Program 7 | Input a multi-word string and find out the number of words in the string

Line	PE 6-7.c	Output window
1	//Number of words in a string	Enter a string:
2	#include<stdio.h>	A man should be educated enough to know that education alone is not enou
3	#include<string.h>	Number of words in the string are 14
4	main()	
5	{	
6	char str[200];	
7	int i=0, count=0;	
8	printf("Enter a string:\n");	
9	gets(str);	
10	while(str[i]!='\0')	
11	{	
12	if(str[i]==' ')	
13	count++;	
14	i++;	
15	}	
16	printf("Number of words in the string are %d\n",count+1);	
17	}	

Program 8 | Input a string and check whether the given string is a palindrome or not

Line	PE 6-8.c	Output window
1	//To check whether a given string is a palindrome or not	Enter a string: NITIN
2	#include<stdio.h>	The given string is a palindrome
3	#include<string.h>	**Output window (second execution)**
4	main()	
5	{	Enter a string: Hello
6	char str[200], rev[200];	The given string is not a palindrome
7	printf("Enter a string:\t");	
8	gets(str);	
9	strcpy(rev,str);	
10	rev=strrev(str);	
11	if(strcmp(str,rev)==0)	
12	printf("The given string is a palindrome");	
13	else	
14	printf("The given string is not a palindrome");	
15	}	

Program 9 | Input

Line	PE 6-9.c
1	//Counting the num
2	#include<stdio.h>
3	#include<string.h>
4	main()
5	{
6	char str[200],
7	int i=0, j=0, cou
8	printf("Enter a s
9	gets(str);
10	printf("Enter the
11	gets(word);
12	while(str[i]!='\0
13	{
14	while(str[i]!=
15	{
16	temp[j]=s
17	j++; i++;
18	}
19	temp[j]='\0';
20	if(str[i]!='\0'
21	{
22	i++; j=0;
23	}
24	if(strcmp(tem
25	count++;
26	}
27	if(count==0)
28	printf("The word \"
29	else
30	printf("The word \"%
31	}

Program 10 | Input a

Line	PE 6-10.c
1	//Counting the occurr
2	#include<stdio.h>
3	#include<string.h>
4	main()
5	{
6	char str1[200], str2
7	int i=0, j=0,k=0, cou
8	printf("Enter a strin
9	gets(str1);
10	printf("Enter the str
11	gets(str2);
12	while(str1[i]!='\0')
13	{
14	k=0;
15	while(str2[k]!='\

Program 9 | Input a string and count the number of occurrences of a particular word in the string

Line	PE 6-9.c
1	//Counting the number of occurrences of a particular word in a string
2	#include<stdio.h>
3	#include<string.h>
4	main()
5	{
6	char str[200], word[20], temp[20];
7	int i=0, j=0, count=0;
8	printf("Enter a string:\n");
9	gets(str);
10	printf("Enter the word:\t");
11	gets(word);
12	while(str[i]!='\0')
13	{
14	while(str[i]!=' ' && str[i]!='\0')
15	{
16	temp[j]=str[i];
17	j++; i++;
18	}
19	temp[j]='\0';
20	if(str[i]!='\0')
21	{
22	i++; j=0;
23	}
24	if(strcmp(temp,word)==0)
25	count++;
26	}
27	if(count==0)
28	printf("The word \"%s\" does not exist in the string", word);
29	else
30	printf("The word \"%s\" exists %d times in the string", word, count);
31	}

Output window

```
Enter a string:
Fools are not aware of their own faults although they are known to all
Enter the word:     are
The word "are" exists 2 times in the string
```

Output window (second execution)

```
Enter a string:
You must not expect everything exactly to your taste
Enter the word:     are
The word "are" does not exist in the string
```

Program 10 | Input a string and count the number of occurrences of a particular string in the string

Line	PE 6-10.c
1	//Counting the occurrences of a particular string in the string
2	#include<stdio.h>
3	#include<string.h>
4	main()
5	{
6	char str1[200], str2[200], temp[20];
7	int i=0, j=0,k=0, count=0;
8	printf("Enter a string:\n");
9	gets(str1);
10	printf("Enter the string to be searched:\t");
11	gets(str2);
12	while(str1[i]!='\0')
13	{
14	k=0;
15	while(str2[k]!='\0')

Output window

```
Enter a string:
Try not to become a man of success but rather to be a man of value
Enter the string to be searched:     a man of
String "a man of" exists 2 times
```

Output window (second execution)

```
Enter a string:
Try not to become a man of success but rather to be a man of value
Enter the string to be searched:     civil society
String "civil society" doesnot exist in the given string
```

(Contd...)

Line	PE 6-10.c	Output window
16	{	
17	if(str1[j]==str2[k])	
18	j++, k++;	
19	else	
20	{	
21	j=i+1;	
22	break;	
23	}	
24	if(str2[k]==0)	
25	count++;	
26	}	
27	if(str2[k]==0)	
28	i=j;	
29	else	
30	i++;	
31	}	
32	if(count==0)	
33	printf("String \"%s\" doesnot exist in the given string\n", str2);	
34	else	
35	printf("String \"%s\" exists %d times\n", str2, count);	
36	}	

Program 11 | A class consists of a number of students whose names are entered in a random order. Display the names of all the students that start with a particular character

Line	PE 6-11.c	Output window
1	//Displaying the names of students starting with a particular character	How many students are there in the class: 10
2	#include<stdio.h>	Enter the names of students:
3	#include<string.h>	Abhay Singh
4	main()	Neha Singla
5	{	Jasraj Singh
6	char names[40][30], firstchar;	Aditya Raina
7	int num, i;	Tarun Kumar
8	printf("How many students are there in the class:\t");	Amol Sood
9	scanf("%d",&num);	Joydeep Chandra
10	printf("Enter the names of students:\n");	Tushar Sharma
11	for(i=0;i<num;i++)	Rajini Bansal
12	gets(names[i]);	Sam
13	printf("\nEnter the first character of student's name:\t");	
14	scanf("%c",&firstchar);	Enter the first character of student's name: A
15	printf("Students whose names starts with %c are:\n",firstchar);	Students whose names starts with A are:
16	for(i=0;i<num;i++)	Abhay Singh
17	if(names[i][0]==firstchar)	Aditya Raina
18	puts(names[i]);	Amol Sood
19	}	

Program 12 | A class consists of a number of students whose names are entered in a random order. Display the names in a sorted order

Line	PE 6-12.c
1	//Displaying the names of students in a sorted order
2	#include<stdio.h>
3	#include<string.h>
4	main()
5	{
6	char names[40][30], current[30];
7	int num, i,j;
8	printf("How many students are there in the class:\t");
9	scanf("%d",&num);
10	printf("Enter the names of students:\n");
11	for(i=0;i<num;i++)
12	gets(names[i]);
13	for(i=1;i<num;i++) //← Insertion Sort
14	if(strcmp(names[i],names[i-1])<0) //←equivalent to if(names[i]<names[i-1])
15	{
16	strcpy(current,names[i]); //←equivalent to current=names[i]
17	for(j=i-1;j>=0;j--)
18	{
19	
20	strcpy(names[j+1],names[j]); //←eq. to names[j+1]=names[j]
21	if(j==0||(strcmp(names[j-1],current)<0))
22	break;
23	}
24	strcpy(names[j],current);
25	}
26	printf("\nAfter sorting, names of students are:\n");
27	for(i=0;i<num;i++) //←Print sorted list
28	puts(names[i]);
29	}

Output window

```
How many students are there in the class:    10
Enter the names of students:
Abhay Singh
Neha Singla
Jasraj Singh
Aditya Raina
Tarun Kumar
Amol Sood
Joydeep Chandra
Tushar Sharma
Rajini Bansal
Sam

After sorting, names of students are:
Abhay Singh
Aditya Raina
Amol Sood
Jasraj Singh
Joydeep Chandra
Neha Singla
Rajini Bansal
Sam
Tarun Kumar
Tushar Sharma
```

Program 13 | Chandigarh Housing Board has released a list of successful applicants in the preliminary draw of lots. Find out whether a given name is in the list or not

Line	PE 6-13.c
1	//Searching a name in the list
2	#include<stdio.h>
3	#include<string.h>
4	main()
5	{
6	char applicants[40][30], name[30];
7	int num, i,found=0;
8	printf("The list of draw is of how many applicants?\t");

Output window

```
The list of draw is of how many applicants?    10
Enter the names:
Abhay Singh
Neha Singla
Jasraj Singh
Aditya Raina
Tarun Kumar
Sam
```

(Contd...)

Line	PE 6-13.c	Output window
9	scanf("%d",&num);	Amol Sood Joydeep Chandra Tushar Sharma Rajini Bansal Enter the name to be searched: Sam Name "Sam" appears in the list of successful applicants
10	printf("Enter the names:\n");	
11	for(i=0;i<num;i++)	
12	gets(applicants[i]);	
13	printf("\nEnter name to be searched:\t");	
14	gets(name)	
15	for(i=1;i<num;i++) //← Linear search	
16	if(strcmp(applicants[i],name)==0)	
17	found=1;	
18	if(found==1)	
19	printf("Name \"%s\" appears in the list of successful applicants");	
20	else	
21	printf("Name \"%s\" does not appear in the list of successful applicants");	
22	}	

Program 14 | Count the number of sentences, words and characters in a given paragraph

Line	PE 6-14.c	Output window
1	//Counting the number of sentences, words and characters in a given paragraph	Enter the text: Hello! How are you? Where were you? I have been looking for all these days. Number of sentences in paragraph are 4 Number of words in paragraph are 15 Number of characters in paragraph are 75
2	#include<stdio.h>	
3	main()	
4	{	
5	char paragraph[1000];	
6	int i=0, sentence=0, word=0, chs=0;	
7	printf("Enter the text:\n");	
8	scanf("%[^\n]", paragraph);	
9	while(paragraph[i]!='\0')	
10	{	
11	switch(paragraph[i])	
12	{	
13	case '!':	
14	case '.':	
15	case '?':	
16	sentence++;	
17	chs++;	
18	break;	
19	case ' ':	
20	case '\t':	
21	chs++;	
22	word++;	
23	break;	
24	default:	
25	chs++;	
26	}	
27	i++;	
28	}	
29	printf("\nNumber of sentences in paragraph are %d\n", sentence);	
30	printf("Number of words in paragraph are %d\n", word+1);	
31	printf("Number of characters in paragraph are %d\n", chs);	
32	}	

Test Yourself

1. Fill in the bl
   - a. A string
   - b. Every str
   - c. The amo
   - d. The type
   - e. A string
   - f. Adjacent
   - g. The scanf
   - h. _____
   - i. The _____
   - j. _____
   - k. Inputs to
2. State whethe
   - a. The leng
   - b. The leng
   - c. The amo depends
   - d. The num present i
   - e. It is not while re
   - f. Unlike th ter is enc
   - g. The printf
   - h. If the ch would b
   - i. It is not
   - j. The strin characte
   - k. It is not
   - l. A list of
3. Programmi
   - a. A certain between
   - b. Without string is
   - c. Write a word.
   - d. Write a
   - e. Write a
   - f. Write a tionality substrin

Test Yourself

1. Fill in the blanks in each of the following:
   a. A string literal constant of zero length is called _____.
   b. Every string literal in C is terminated by _____.
   c. The amount of memory taken by an empty string literal is _____.
   d. The type of string literal is _____.
   e. A string literal constant is always enclosed within _____.
   f. Adjacent string literals are _____.
   g. The scanf function uses _____ format specification to read a string from the user.
   h. _____ string library function is used to compare two strings without case sensitivity.
   i. The _____ character is used to invert the search set.
   j. _____ function is used to read a character from the keyboard.
   k. Inputs to function main are given by making use of special arguments known as _____ .

2. State whether each of the following is true or false. If false, explain why.
   a. The length of a string literal constant is equal to the number of characters present in it.
   b. The length of an empty string literal constant is one.
   c. The amount of the memory space required for storing a string literal constant is not fixed and depends upon the number of characters present in the string literal.
   d. The number of bytes required to store a string literal is equal to the number of characters present in it.
   e. It is not mandatory to use ampersand (i.e. address-of operator) with string variable names while reading string using the scanf function.
   f. Unlike the scanf function, the gets function reads the entire line of text until a new line character is encountered and does not stop upon encountering any other white-space character.
   g. The printf function can print a string on the screen without using any format specifier.
   h. If the character array to be printed does not have a terminating null character, the output would be the content of the character array followed by some garbage character.
   i. It is not mandatory to have the first argument of the printf function to be of const char* type.
   j. The string library function strrev reverses all the characters of a string including the null character.
   k. It is not possible to initialize a character array with a string literal constant.
   l. A list of strings can be stored by using a two-dimensional character array.

3. Programming exercises:
   a. A certain piece of text is entered. By mistake, at some places two or more spaces are placed between two words. Write a C program that removes these extra spaces between the words.
   b. Without using inbuilt string library functions, write a C program to check whether a given string is a palindrome or not.
   c. Write a C program to find the longest word in a given string. Also print the length of the word.
   d. Write a C program to read a text and omit all occurrences of a particular word in the text.
   e. Write a C program to read a text and omit all occurrences of a particular string in the text.
   f. Write a C program to read a text. Implement the find and replace functionality. The find functionality will find a given substring in the text, and the replace function will replace the found substring with a given string.