Arrays, Pointers and Structures

2

2.1 Arrays

An array is a collection of similar type of data items and each data item is called an element of the array. Th data type of the elements may be any valid data type like char, int or float. The elements of array share the sam variable name but each element has a different index number known as subscript.

Consider a situation when we want to store and display the age of 100 employees. We can take an arra variable `age` of type int. The size of this array variable is 100 so it is capable of storing 100 integer values. Th individual elements of this array are-

`age[0], age[1], age[2], age[3], age[4],.....age[98], age[99]`

In C, the subscripts start from zero, so `age[0]` is the first element, `age[1]` is the second element of array and s on.

Arrays can be single dimensional or multidimensional. The number of subscripts determines the dimensio of array. A one-dimensional array has one subscript; two-dimensional array has two subscripts and so on. Th two-dimensional arrays are known as matrices.

2.1.1 One Dimensional Array

2.1.1.1 Declaration of 1-D Array

Like other simple variables, arrays should also be declared before they are used in the program. The syntax fo declaration of an array is-

```
data_type array_name[size];
```

Here `array_name` denotes the name of the array and it can be any valid C identifier, `data_type` is the data type of the elements of array. The size of the array specifies the number of elements that can be stored in the array. It may be a positive integer constant or constant integer expression. Here are some examples of array declarations-

```
int age[100];
float salary[15];
char grade[20];
```

Here `age` is an integer type array, which can store 100 elements of integer type. The array `salary` is a float type array of size 15, can hold float values and third one is a character type array of size 20, can hold characters. The individual elements of the above arrays are-

```
age[0], age[1], age[2].....age[99]
salary[0], salary [1], salary [2].....salary [14]
grade[0], grade[1], grade[2].....grade[19]
```

When the array is declared, the compiler allocates space in memory sufficient to hold all the elements of the array, so the size of array should be known at the compile time. Hence we can't use variables for specifying the size of array in the declaration. The symbolic constants can be used to specify the size of array. For example-

```
#define SIZE 10
main()
{
        int size = 15;
        float sal[SIZE];    /*Valid*/
        int marks[size];    /*Not valid*/
        ..............
        ..............
}
```

The use of symbolic constant to specify the size of array makes it convenient to modify the program if the size of array is to be changed later, because the size has to be changed only at one place, in the `#define` directive.

2.1.1.2 Accessing 1-D Array Elements

The elements of an array can be accessed by specifying the array name followed by subscript in brackets. In C, the array subscripts start from 0. Hence if there is an array of size 5 then the valid subscripts will be from 0 to 4. The last valid subscript is one less than the size of the array. This last valid subscript is known as the upper bound of the array and 0 is known as the lower bound of the array. Let us take an array-

```
int arr[5];     /*Size of array arr is 5, can hold five integer elements*/
```

The elements of this array are-

```
arr[0], arr[1], arr[2], arr[3], arr[4]
```

Here 0 is the lower bound and 4 is the upper bound of the array.

The subscript can be any expression that yields an integer value. It can be any integer constant, integer variable, integer expression or return value(int) from a function call. For example, if `i` and `j` are integer variables then these are some valid subscripted array elements-

```
arr[3], arr[i], arr[i+j], arr[2*j], arr[i++]
```

A subscripted array element is treated as any other variable in the program. We can store values in them, print their values or perform any operation that is valid for any simple variable of the same data type. For example if `arr` and `sal` are two arrays of sizes 5 and 10 respecively, then these are valid statements-

```
int arr[5];
float sal[10];
int i;
scanf("%d",&arr[1]);    /*input value into arr[1]*/
printf("%f",sal[3]);    /*print value of sal[3]*/
arr[4] = 25;            /*assign a value to arr[4]*/
arr[4]++;               /*Increment the value of arr[4] by 1*/
sal[5]+=200;            /*Add 200 to sal[5]*/
sum = arr[0]+arr[1]+arr[2]+arr[3]+arr[4];/*Add all the values of array arr[5]*/
i=2;
scanf("%f",&sal[i]);    /*Input value into sal[2]*/
printf("%f",sal[i]);    /*Print value of sal[2]*/
printf("%f",sal[i++] ); /*Print value of sal[2] and increment the value of i*/
```

There is no check on bounds of the array. For example, if we have an array `arr` of size 5, the valid subscripts are only 0, 1, 2, 3, 4 and if someone tries to access elements beyond these subscripts, like `arr[5]` , `arr[10]`, the compiler will not show any error message but this may lead to run time errors. So it is the responsibility of programmer to provide array bounds checking wherever needed.

2.1.1.3 Processing 1-D Arrays

For processing arrays we generally use a for loop and the loop variable is used at the place of subscript. The initial value of loop variable is taken 0 since array subscripts start from zero. The loop variable is increased by 1 each time so that we can access and process the next element in the array. The total number of passes in the

loop will be equal to the number of elements in the array and in each pass we will process one element. Suppose arr is an array of type int-

(i) Reading values in arr

```
for(i=0; i<10; i++)
        scanf("%d",&arr[i]);
```

(ii) Displaying values of arr

```
for(i=0; i<10; i++)
     printf("%d ",arr[i]);
```

(iii) Adding all the elements of arr

```
sum=0;
for(i=0; i<10; i++)
        sum+=arr[i];
```

```
/*P2.1 Program to input values into an array and display them*/
#include<stdio.h>
main()
{
        int arr[5], i;
        for(i=0; i<5; i++)
        {
                printf("Enter the value for arr[%d] : ",i);
                scanf("%d",&arr[i]);
        }
        printf("The array elements are : \n");
        for(i=0; i<5; i++)
                printf("%d\t",arr[i]);
        printf("\n");
}
```

2.1.1.4 Initialization of 1-D Array

After declaration, the elements of a local array have garbage value while the elements of global and static arrays are automatically initialized to zero. We can explicitly initialize arrays at the time of declaration. The syntax for initialization of an array is-

```
data_type array_name[size]={value1, value2.....valueN};
```

Here array_name is the name of the array variable, size is the size of the array and value1, value2, ...valueN are the constant values known as initializers, which are assigned to the array elements one after another. These values are separated by commas and there is a semicolon after the ending braces. For example-

```
int marks[5] = {50, 85, 70, 65, 95};
```

The values of the array elements after this initialization are-

```
marks[0]:50, marks[1]:85, marks[2]:70, marks[3]:65, marks[4]:95
```

While initializing a 1-D array, it is optional to specify the size of the array. If the size is omitted during initialization then the compiler assumes the size of array equal to the number of initializers. For example-

```
int marks[] = {99, 78, 50, 45, 67, 89};
float sal[] = {25.5, 38.5, 24.7};
```

Here the size of array marks is assumed to be 6 and that of sal is assumed to be 3.

If during initialization the number of initializers is less than the size of array, then all the remaining elements of array are assigned value zero. For example-

```
int marks[5] = {99, 78};
```

Here the size of array is 5 while there are only 2 initializers. After this initialization the value of the elements are-

```
marks[0]:99,  marks[1]:78,  marks[2]:0,  marks[3]:0,  marks[4]:0
```

So if we initialize an array like this-

```
int arr[100] = {0};
```

then all the elements of arr will be initialized to zero.

If the number of initializers is more than the size given in brackets then compiler will show an error. For example-

```
int arr[5] = {1,2,3,4,5,6,7,8};        /*Error*/
```

We can't copy all the elements of an array to another array by simply assigning it to the other array. For example if we have two arrays a and b, then-

```
int a[5] = {1,2,3,4,5};
int b[5];
b = a;  /*Not valid*/
```

We will have to copy all the elements of array one by one, using a for loop.

```
for(i=0; i<5; i++)
        b[i] = a[i];
```

In the following program we will find out the largest and smallest number in an integer array.

```
/*P2.2 Program to find the largest and smallest number in an array*/
#include<stdio.h>
main()
{
        int i,arr[10]={2,5,4,1,8,9,11,6,3,7};
        int small,large;
        small = large = arr[0];
        for(i=1; i<10; i++)
        {
                if(arr[i] < small)
                        small = arr[i];
                if(arr[i] > large)
                        large = arr[i];
        }
        printf("Smallest = %d, Largest = %d\n",small,large);
}
```

We have taken the value of first element as the initial value of small and large. Inside the for loop, we will start comparing from second element onwards so this time we have started the loop from 1 instead of 0.

The following program will reverse the elements of an array.

```
/*P2.3 Program to reverse the elements of an array*/
#include<stdio.h>
main()
{
        int i,j,temp,arr[10] = {1,2,3,4,5,6,7,8,9,10};
        for(i=0,j=9; i<j; i++,j--)
        {
                temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
        }
        printf("After reversing, the array is : ");
        for(i=0; i<10; i++)
                printf("%d  ",arr[i]);
        printf("\n");
}
```

In the for loop we have used comma operator and taken two variables i and j. The variable i is initialized with the lower bound and j is initialized with upper bound. After each pass of the loop, i is incremented while j is decremented. Inside the loop, a[i] is exchanged with a[j]. So a[0] will be exchanged with a[9], a[1] with a[8], a[2] with a[7] and so on.

2.1.1.5 1-D Arrays and Functions

2.1.1.5.1 Passing Individual Array Elements to a Function

We know that an array element is treated as any other simple variable in the program. So like other simple variables, we can pass individual array elements as arguments to a function.

```
/*P2.4 Program to pass array elements to a function*/
#include<stdio.h>
void check(int num);
main()
{
        int arr[10],i;
        printf("Enter the array elements : ");
        for(i=0; i<10; i++)
        {
                scanf("%d",&arr[i]);
                check(arr[i]);
        }
}
void check(int num)
{
        if(num%2 == 0)
                printf("%d is even\n",num);
        else
        printf("%d is odd\n",num);
}
```

2.1.1.5.2 Passing whole 1-D Array to a Function

We can pass whole array as an actual argument to a function. The corresponding formal argument should be declared as an array variable of the same data type.

```
main()
{
        int arr[10]
        .....
        .....
        func(arr); /*In function call, array name is specified without brackets*/
}
func(int val[10])
{
        .....
        .....
}
```

It is optional to specify the size of the array in the formal argument, for example we may write the function definition as-

```
func(int val[])
{
  .....
  .....
}
```

The mechanism of passing an array to a function is quite different from that of passing a simple variable. In the case of simple variables, the called function creates a copy of the variable and works on it, so any changes made in the function do not affect the original variable. When an array is passed as an actual argument, the called function actually gets access to the original array and works on it, so any changes made inside the function affect the original array. Here is a program in which an array is passed to a function.

```
/*P2.5 Program to pass an array to a function*/
#include<stdio.h>
```

```
void func(int val[]);
main()
{
        int i,arr[6] = {1,2,3,4,5,6};
        func(arr);
        printf("Contents of array are now : ");
        for(i=0; i<6; i++)
                printf("%d ",arr[i]);
        printf("\n");
}
void func(int val[])
{
        int sum=0,i;
        for(i=0;i<6;i++)
        {
                val[i] = val[i]*val[i];
                sum += val[i];
        }
        printf("The sum of squares = %d\n",sum);
}
```

Output:
The sum of squares = 91
Contents of array are now : 1 4 9 16 25 36

Here we can see that the changes made to the array inside the called function are reflected in the calling function. The name of the formal argument is different but it refers to the original array.

2.1.2 Two Dimensional Arrays

2.1.2.1 Declaration and Accessing Individual Elements of a 2-D array

The syntax of declaration of a 2-D array is similar to that of 1-D arrays, but here we have two subscripts.

```
data_type array_name[rowsize][columnsize];
```

Here rowsize specifies the number of rows and columnsize represents the number of columns in the array. The total number of elements in the array are rowsize * columnsize. For example-

```
int arr[4][5];
```

Here `arr` is a 2-D array with 4 rows and 5 columns. The individual elements of this array can be accessed by applying two subscripts, where the first subscript denotes the row number and the second subscript denotes the column number. The starting element of this array is `arr[0][0]` and the last element is `arr[3][4]`. The total number of elements in this array is 4*5 = 20.

	Col 0	Col 1	Col 2	Col 3	Col 4
Row 0	arr[0][0]	arr[0][1]	arr[0][2]	arr[0][3]	arr[0][4]
Row 1	arr[1][0]	arr[1][1]	arr[1][2]	arr[1][3]	arr[1][4]
Row 2	arr[2][0]	arr[2][1]	arr[2][2]	arr[2][3]	arr[2][4]
Row 3	arr[3][0]	arr[3][1]	arr[3][2]	arr[3][3]	arr[3][4]

2.1.2.2 Processing 2-D Arrays

For processing 2-D arrays, we use two nested for loops. The outer `for` loop corresponds to the row and the inner `for` loop corresponds to the column.

```
int arr[4][5];
```

(i) Reading values in `arr`

```
for(i=0; i<4; i++)
        for(j=0; j<5; j++)
                scanf("%d",&arr[i][j]);
```

(ii) Displaying values of `arr`

```
        for(i=0; i<4; i++)
                for(j=0; j<5; j++)
                        printf("%d ",arr[i][j]);
```

This will print all the elements in the same line. If we want to print the elements of different rows on different lines then we can write like this-

```
for(i=0; i<4; i++)
{
        for(j=0; j<5; j++)
                printf("%d ",arr[i][j]);
        printf("\n");
}
```

Here the `printf("\n")` statement causes the next row to begin from a new line.

```
/*P2.6 Program to input and display a matrix*/
#define ROW 3
#define COL 4
#include<stdio.h>
main()
{
        int mat[ROW][COL], i, j;
        printf("Enter the elements of the matrix(%dx%d) row-wise :\n",ROW,COL);
        for(i=0; i<ROW; i++)
                for(j=0; j<COL; j++)
                        scanf("%d",&mat[i][j]);

        printf("The matrix that you have entered is :\n");
        for(i=0; i<ROW; i++)
        {
                for(j=0; j<COL; j++)
                        printf("%5d",mat[i][j]);
                printf("\n");
        }
        printf("\n");
}
```

2.1.2.3 Initialization of 2-D Arrays

2-D arrays can be initialized in a way similar to that of 1-D arrays. For example-

```
int mat[4][3] = {11,12,13,14,15,16,17,18,19,20,21,22};
```

These values are assigned to the elements row-wise, so the values of elements after this initialization are-

mat[0][0] : 11	mat[0][1] : 12	mat[0][2] : 13
mat[1][0] : 14	mat[1][1] : 15	mat[1][2] : 16
mat[2][0] : 17	mat[2][1] : 18	mat[2][2] : 19
mat[3][0] : 20	mat[3][1] : 21	mat[3][2] : 22

While initializing we can group the elements row-wise using inner braces. For example-

```
int mat[4][3] = { {11,12,13}, {14,15,16}, {17,18,19}, {20,21,21} };
int mat[4][3] = {
                        {11,12,13},     /*Row 0*/
                        {14,15,16},     /*Row 1*/
                        {17,18,19},     /*Row 2*/
                        {20,21,21}      /*Row 3*/
                };
```

Here the values in the first inner braces will be the values of Row 0, values in the second inner braces will be values of Row 1 and so on. Now consider this array initialization-

```
int mat[4][3] = {
                        {11},           /*Row 0*/
                        {12,13},        /*Row 1*/
                        {14,15,16},     /*Row 2*/
                        {17}            /*Row 3*/
                };
```

The remaining elements in each row will be assigned values 0, so the values of elements will be-

mat[0][0] : 11	mat[0][1] : 0	mat[0][2] : 0
mat[1][0] : 12	mat[1][1] : 13	mat[1][2] : 0
mat[2][0] : 14	mat[2][1] : 15	mat[2][2] : 16
mat[3][0] : 17	mat[3][1] : 0	mat[3][2] : 0

In 2-D arrays, it is optional to specify the first dimension while initializing but the second dimension should always be present. For example-

```
int mat[][3] = {
                    {1,10},
                    {2,20,200},
                    {3},
                    {4,40,400}
               };
```

Here first dimension is taken 4 since there are 4 rows in initialization list.

A 2-D array is also known as a matrix. The next program adds two matrices; the order of both the matrices should be same.

```
/*P2.7 Program for addition of two matrices.*/
#define ROW 3
#define COL 4
#include<stdio.h>
main()
{
        int i,j,mat1[ROW][COL],mat2[ROW][COL],mat3[ROW][COL];
        printf("Enter matrix mat1(%dx%d)row-wise :\n",ROW,COL);
        for(i=0; i<ROW; i++)
                for(j=0; j<COL; j++)
                        scanf("%d",&mat1[i][j]);
        printf("Enter matrix mat2(%dx%d)row-wise :\n",ROW,COL);
        for(i=0; i<ROW; i++)
                for(j=0; j<COL; j++)
                        scanf("%d",&mat2[i][j] );
        /*Addition*/
        for(i=0; i<ROW; i++)
                for(j=0; j<COL; j++)
                        mat3[i][j] = mat1[i][j] + mat2[i][j];
        printf("The resultant matrix mat3 is :\n");
        for(i=0; i<ROW; i++)
        {
                for(j=0; j<COL; j++)
                        printf("%5d",mat3[i][j]);
                printf("\n");
        }
}
```

Now we will write a program to multiply two matrices. Multiplication of matrices requires that the number of columns in first matrix should be equal to the number of rows in second matrix. Each row of first matrix is multiplied with the column of second matrix then added to get the element of resultant matrix. If we multiply two matrices of order m x n and n x p then the multiplied matrix will be of order m x p. For example-

$$A_{2x2} = \begin{bmatrix} 4 & 5 \\ 3 & 2 \end{bmatrix} \qquad B_{2x3} = \begin{bmatrix} 2 & 6 & 3 \\ -3 & 2 & 4 \end{bmatrix}$$

$$C_{2x3} = \begin{bmatrix} 4*2+5*(-3) & 4*6+5*2 & 4*3+5*4 \\ 3*2+2*(-3) & 3*6+2*2 & 3*3+2*4 \end{bmatrix} = \begin{bmatrix} -7 & 34 & 32 \\ 0 & 22 & 17 \end{bmatrix}$$

```
/*P2.8 Program for multiplication of two matrices*/
#include<stdio.h>
#define ROW1 3
```

```
#define COL1 4
#define ROW2 COL1
#define COL2 2
main()
 {
        int mat1[ROW1][COL1],mat2[ROW2][COL2],mat3[ROW1][COL2];
        int i,j,k;
        printf("Enter matrix mat1(%dx%d)row-wise :\n",ROW1,COL1);
        for(i=0; i<ROW1; i++)
                for(j=0; j<COL1; j++)
                        scanf("%d",&mat1[i][j]);
         printf("Enter matrix mat2(%dx%d)row-wise :\n",ROW2,COL2);
         for(i=0; i<ROW2; i++)
                 for(j=0; j<COL2; j++)
                         scanf("%d",&mat2[i][j] );
          /*Multiplication*/
          for(i=0; i<ROW1; i++)
                  for(j=0; j<COL2; j++)
                  {
                          mat3[i][j] = 0;
                          for(k=0; k<COL1; k++)
                                  mat3[i][j] += mat1[i][k] * mat2[k][j];
                  }
           printf("The Resultant matrix mat3 is :\n");
           for(i=0; i<ROW1; i++)
           {
                   for(j=0; j<COL2; j++)
                           printf("%5d",mat3[i][j]);
                   printf("\n");
           }
}
```

2.2 Pointers

A pointer is a variable that stores memory address. Like all other variables it also has a name, has to be declared and occupies some space in memory. It is called pointer because it points to a particular location in memory by storing the address of that location. The use of pointers makes the code more efficient and compact. Some of the uses of pointers are-

(i) Accessing array elements.
(ii) Returning more than one value from a function.
(iii) Accessing dynamically allocated memory
(iv) Implementing data structures like linked lists, trees, and graphs.

2.2.1 Declaration of a Pointer Variable

Like other variables, pointer variables should also be declared before being used. The general syntax of declaration is-

```
data_type *pname;
```

Here `pname` is the name of the pointer variable, which should be a valid C identifier. The asterisk '*' preceding this name informs the compiler that the variable is declared as a pointer. Here `data_type` is known as the base type of pointer. Let us take some pointer declarations-

```
int *iptr;
float *fptr;
char *cptr,ch1,ch2;
```

Here `iptr` is a pointer that should point to variable of type `int`, similarly `fptr` and `cptr` should point to variables of `float` and `char` type respectively. Here type of variable `iptr` is 'pointer to int' or (int *), or we can say that base type of `iptr` is `int`. We can also combine the declaration of simple variables and pointer

variables as we have done in the third declaration statement where `ch1` and `ch2` are declared as variables of type `char`.

Pointers are also variables so compiler will reserve space for them and they will also have some address. All pointers irrespective of their base type will occupy same space in memory since all of them contain addresses only. The size of a pointer depends on the architecture and may vary on different machines; in our discussion we will take the size of pointer to be 4 bytes.

2.2.2 Assigning Address to a Pointer Variable

When we declare a pointer variable it contains garbage value i.e. it may be pointing anywhere in the memory. So we should always assign an address before using it in the program. The use of an unassigned pointer may give unpredictable results and even cause the program to crash. Pointers may be assigned the address of a variable using assignment statement. For example-

```
int *iptr,age = 30;
float *fptr,sal = 1500.50;
iptr = &age;
fptr = &sal;
```

Now `iptr` contains the address of variable `age` i.e. it points to variable age, similarly `fptr` points to variable `sal`. Since `iptr` is declared as a pointer of type `int`, we should assign address of only integer variables to it. If we assign address of some other data type then compiler won't show any error but the output will be incorrect.

We can also initialize the pointer at the time of declaration, but in this case the variable should be declared before the pointer. For example-

```
int age=30,*iptr=&age;
float sal=1500.50,*fptr=&sal;
```

It is also possible to assign the value of one pointer variable to the other, provided their base type is same. For example if we have an integer pointer `p1` then we can assign the value of `iptr` to it as-

```
p1 = iptr;
```

Now both pointer variables `iptr` and `p1` contain the address of variable `age` and point to the same variable `age`.

We can assign constant zero to a pointer of any type. A symbolic constant NULL is defined in the header file stdio.h, which denotes the value zero. The assignment of NULL to a pointer guarantees that it does not point to any valid memory location. This can be done as-

```
ptr = NULL;
```

2.2.3 Dereferencing Pointer Variables

We can access a variable indirectly using pointers. For this we will use the indirection operator(*). By placing the indirection operator before a pointer variable, we can access the variable whose address is stored in the pointer. Let us take an example-

```
int a = 87
float b = 4.5;
int *p1 = &a;
float *p2 = &b;
```

In our program, if we place '*' before `p1` then we can access the variable whose address is stored in `p1`. Since `p1` contains the address of variable `a`, we can access the variable `a` by writing `*p1`. Similarly we can access variable `b` by writing `*p2`. So we can use `*p1` and `*p2` in place of variable names `a` and `b` anywhere in our program. Let us see some examples-

`*p1= 9;`	is equivalent to	`a = 9;`
`(*p1)++;`	is equivalent to	`a++;`
`x = *p2 + 10 ;`	is equivalent to	`x = b + 10;`

```
printf("%d %f",*p1,*p2);      is equivalent to      printf("%d %f",a,b);
scanf("%d%f",p1,p2);          is equivalent to      scanf("%d%f",&a,&b);
```

The indirection operator can be read as 'value at the address'. For example *p1 can be read as 'value at the address p1'. This indirection operator (*) is different from the asterisk that was used while declaring the pointer variable.

```
/*P2.9 Program to dereference pointer variables*/
#include<stdio.h>
main()
{
        int a = 87;
        float b = 4.5;
        int *p1 = &a;
        float *p2 = &b;
        printf("Value of p1 = Address of a = %p\n",p1);
        printf("Value of p2 = Address of b = %p\n",p2);
        printf("Address of p1 = %p\n",&p1);
        printf("Address of p2 = %p\n",&p2);
        printf("Value of a = %d %d %d\n",a,*p1,*(&a));
        printf("Value of b = %f %f %f\n",b,*p2,*(&b));
}
```

2.2.4 Pointer to Pointer

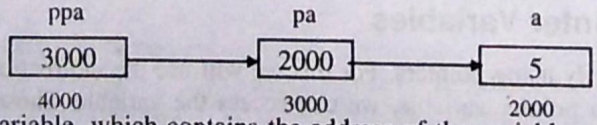
We know that pointer is a variable that can contain memory address. The pointer variable takes some space in memory and therefore it also has an address. We can store the address of a pointer variable in some other variable, which is known as a pointer to pointer variable. Similarly we can have a pointer to pointer to pointer variable and this concept can be extended to any limit, but in practice only pointer to pointer is used. Pointer to pointer is generally used while passing pointer variables to functions. The syntax of declaring a pointer to pointer is as-

```
data_type **pptr;
```

Here variable pptr is a pointer to pointer and it can point to a pointer pointing to a variable of type data_type. The double asterisk used in the declaration informs the compiler that a pointer to pointer is being declared. Now let us take an example-

```
int a = 5;
int *pa = &a;
int **ppa = &pa;
```

Here type of variable a is int, type of variable pa is (int *) or pointer to int, and type of variable ppa is (int **) or pointer to pointer to int.



Here pa is a pointer variable, which contains the address of the variable a and ppa is a pointer to pointer variable, which contains the address of the pointer variable pa.

We know that *pa gives value of a, similarly *ppa will give the value of pa. Now let us see what value will be given by **ppa.

```
   **ppa
→ *(*ppa)
→ *pa  (Since *ppa gives pa)
→ a    (Since *pa gives a)
```

Hence we can see that **ppa will give the value of a. So to access the value indirectly pointed to by a pointer to pointer, we can use double indirection operator. The table given next will make this concept clear.

Value of a	a	*pa	**ppa	5
Address of a	&a	pa	*ppa	2000
Value of pa	&a	pa	*ppa	2000
Address of pa		&pa	ppa	3000
Value of ppa		&pa	ppa	3000
Address of ppa			&ppa	4000

```
/*P2.10 Program to understand pointer to pointer*/
#include<stdio.h>
main()
{
        int a = 5;
        int *pa;
        int **ppa;
        pa = &a;
        ppa = &pa;
        printf("Address of a = %p\n",&a);
        printf("Value of pa = Address of a = %p\n",pa);
        printf("Value of *pa = Value of a = %d\n",*pa);
        printf("Address of pa = %p\n",&pa);
        printf("Value of ppa = Address of pa = %p\n",ppa);
        printf("Value of *ppa = Value of  pa = %p\n",*ppa);
        printf("Value of **ppa = Value of a = %d\n",**ppa);
        printf("Address of ppa = %p\n",&ppa);
}
```

2.2.5 Pointers and One Dimensional Arrays

The elements of an array are stored in contiguous memory locations. Suppose we have an array `arr` of type int.

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

Suppose it is stored in memory at address 5000

arr[0]	arr[1]	arr[2]	arr[3]	arr[4]
1	2	3	4	5
5000	5004	5008	5012	5016

Here 5000 is the address of first element, and since each element (type `int`) takes 4 bytes, address of next element is 5004, and so on. The address of first element of the array is also known as the base address of the array. Thus it is clear that the elements of an array are stored sequentially in memory one after another.

In C language, pointers and arrays are closely related. We can access the array elements using pointer expressions. Actually the compiler also accesses the array elements by converting subscript notation to pointer notation. Following are the main points for understanding the relationship of pointers with arrays.

1. Elements of an array are stored in consecutive memory locations.
2. The name of an array is a constant pointer that points to the first element of the array, i.e. it stores the address of the first element, also known as the base address of array.
3. According to pointer arithmetic, when a pointer variable is incremented, it points to the next location of its base type.

For example-

```
int arr[5] = {5,10,15,20,25}
```

Here `arr` is an array that has 5 elements each of type `int`.

arr[0]	arr[1]	arr[2]	arr[3]	arr[4]
5	10	15	20	25
2000	2004	2008	2012	2016

We can get the address of an element of array by applying & operator in front of subscripted variable name. Hence `&arr[0]` gives address of 0^{th} element, `&arr[1]` gives the address of first element and so on. Since array subscripts start from 0, we'll refer to the first element of array as 0^{th} element and so on. The following program shows that the elements of an array are stored in consecutive memory locations.

```
/*P2.11 Program to print the value and address of the elements of an array*/
#include<stdio.h>
main()
{
        int arr[5] = {5,10,15,20,25};
        int i;
        for(i=0; i<5; i++)
        {
                printf("Value of arr[%d] = %d\t",i,arr[i]);
                printf("Address of arr[%d] = %p\n",i,&arr[i]);
        }
}
```

The name of the array 'arr' denotes the address of 0^{th} element of array which is 2000. The address of 0^{th} element can also be given by `&arr[0]`, so `arr` and `&arr[0]` represent the same address. The name of an array is a constant pointer, and according to pointer arithmetic when an integer is added to a pointer then we get the address of next element of same base type. Hence `(arr+1)` will denote the address of the next element `arr[1]`. Similarly `(arr+2)` denotes the address of `arr[2]` and so on. In other words we can say that the pointer expression `(arr+1)` points to 1^{st} element of array, `(arr+2)` points to 2^{nd} element of array and so on.

arr	→	Points to 0^{th} element	→	&arr[0]	→	2000
arr+1	→	Points to 1^{st} element	→	&arr[1]	→	2004
arr+2	→	Points to 2^{nd} element	→	&arr[2]	→	2008
arr+3	→	Points to 3^{rd} element	→	&arr[3]	→	2012
arr+4	→	Points to 4^{th} element	→	&arr[4]	→	2016

In general we can write-

The pointer expression `(arr+i)` denotes the same address as `&arr[i]`.

Now if we dereference `arr`, then we get the 0^{th} element of array. i.e. expression `*arr` or `*(arr+0)` represents 0^{th} element of array. Similarly on derferencing `(arr+1)` we get the 1^{st} element and so on.

*arr	→	Value of 0^{th} element	→	arr[0]	→	5
*(arr+1)	→	Value of 1^{st} element	→	arr[1]	→	10
*(arr+2)	→	Value of 2^{nd} element	→	arr[2]	→	15
*(arr+3)	→	Value of 3^{rd} element	→	arr[3]	→	20
*(arr+4)	→	Value of 4^{th} element	→	arr[4]	→	25

In general we can write-

`*(arr+i)` → `arr[i]`

So in subscript notation the address of an array element is `&arr[i]` and its value is `arr[i]`, while in pointer notation the address is `(arr+i)` and the element is `*(arr+i)`.

```
/*P2.12 Program to print the value and address of elements of an array using pointer
notation*/
#include<stdio.h>
main()
{
        int i, arr[5] = {5,10,15,20,25};
        for(i=0; i<5; i++)
        {
                printf("Value of arr[%d] = %d\t",i,*(arr+i));
                printf("Address of arr[%d] = %p\n",i,arr+i);
        }
}
```

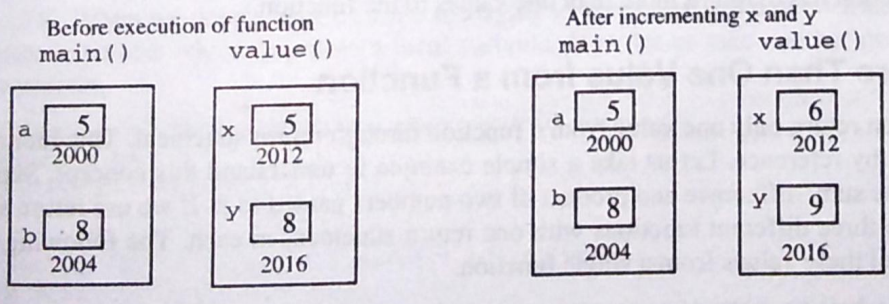
2.2.6 Pointers and Functions

The arguments to the functions can be passed in two ways-
(i) Call by value (ii) Call by reference
In call by value, only the values of arguments are sent to the function while in call by reference, addresses of arguments are sent to the function. In call by value method, any changes made to the formal arguments do not change the actual argument. In call by reference method, any changes made to the formal arguments changes the actual arguments also. C uses only call by value when passing arguments to a function, but we can simulate call by reference by using pointers. All the functions that we had written so far used call by value method. Here is another simple program that uses call by value-

```
/*P2.13 Call by value*/
#include<stdio.h>
void value(int x,int y);
main()
{
        int a=5,b=8;
        printf("Before calling the function,a = %d and b = %d\n",a,b);
        value(a,b);
        printf("After calling the function,a = %d and b = %d\n",a,b);
}
void value(int x,int y)
{
        x++;
        y++;
        printf("Inside function x = %d,y = %d\n",x,y);
}
```

Here a and b are variables declared in the function `main()` while x and y are declared in the function `value()`.These variables reside at different addresses in memory. Whenever the function `value()` is called, two variables are created named x and y and are initialized with the values of variables a and b. This type of parameter passing is called call by value since we are only supplying the values of actual arguments to the calling function. Any operation performed on variables x and y in the function `value()`, will not affect variables a and b.

Before calling the function `value()`, the value of a is 5 and value of b is 8. The values of a and b are copied into x and y. Since the memory locations of x, y and a, b are different, when the values of x and y are incremented, there will be no affect on the values of a and b. Therefore after calling the function, a and b are same as before calling the function and have the values 5 and 8.



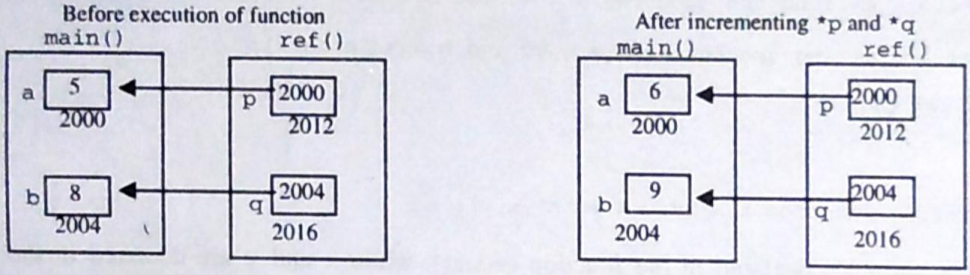
Although C does not use call by reference, we can simulate it by passing addresses of variables as arguments to the function. To accept the addresses inside the function, we will need pointer variables. Here is a program that simulates call by reference by passing addresses of variables a and b.

```
/*P2.14 Call by reference*/
#include<stdio.h>
void ref(int *p,int *q);
main()
{
        int a = 5;
```

```
        int b = 8;
        printf("Before calling the function,a = %d and b = %d\n",a,b);
        ref(&a,&b);
        printf("After calling the function,a = %d and b = %d\n",a,b);
}
void ref(int *p,int *q)
{
        (*p)++;
        (*q)++;
        printf("Inside function *p = %d,*q = %d\n",*p,*q);
}
```

Here we are passing addresses of variables a and b in the function call. So the receiving formal arguments in the function declaration should be declared of pointer type. Whenever function ref() is called, two pointer to int variables, named p and q are created and they are initialized with the addresses of a and b. Now if we dereference pointers p and q, we will be able to access variables a and b from function ref().

The main() accesses the memory locations occupied by variables a and b by writing their names, while ref() accesses the same memory locations indirectly by writing *p, *q.



Before calling the function ref(), the value of a is 5 and value of b is 8. The value of actual arguments are copied into pointer variables p and q, and here the actual arguments are addresses of variables a and b. Since p contains address of variable a, we can access variable a inside ref() by writing *p, similarly variable b can be accessed by writing *q.

Now (*p)++ means value at address 2000 (which is 5) is incremented. Similarly (*q)++ means value at address 2004 (which is 8) is incremented. Now the value of *p = 6 and *q = 9. When we come back to main(), we see that the values of variable a and b have changed. This is because the function ref() knew the addresses of a and b, so it was able to access them indirectly.

So in this way we could simulate call by reference by passing addresses of arguments. This method is mainly useful when the called function has to return more than one values to the function.

2.2.7 Returning More Than One Value from a Function

We have studied that we can return only one value from a function through return statement. This limitation can be overcome by using call by reference. Let us take a simple example to understand this concept. Suppose we want a function to return the sum, difference and product of two numbers passed to it. If we use return statement then we will have to make three different functions with one return statement in each. The following program shows how we can return all these values from a single function.

```
/*P2.15 Program to show how to return more than one value from a function using call by
reference*/
#include<stdio.h>
func(int x,int y,int *ps,int *pd,int *pp);
main()
{
        int a,b,sum,diff,prod;
        a = 6;
        b = 4;
        func(a,b,&sum,&diff,&prod);
        printf("Sum = %d, Difference = %d, Product = %d\n",sum,diff,prod);
```

```
}
func(int x,int y,int *ps,int *pd,int *pp)
{
        *ps = x+y;
        *pd = x-y;
        *pp = x*y;
}
```

In `func()`, variables a and b are passed by value while variables `sum, diff, prod` are passsed by reference. The function `func()` gets the addresses of variables `sum, diff` and `prod`, so it accesses these variables indirectly using pointers and changes their values.

2.2.8 Function Returning Pointer

We can have a function that returns a pointer. The syntax of declaration of such type of function is -

```
type *func(type1,type 2, ...);
```

For example-

```
float *func1(int,char);   /*This function returns a pointer to float*/
int *func2(int,int);      /*This function returns a pointer to int*/
```

While returning a pointer, make sure that the memory address returned by the pointer will exist even after the termination of function. For example a function of this form should not be written.

```
main()
{
        int *ptr;
        ptr = func();
        ..........
}
int *func()
{
        int x = 5;
        int *p = &x;
        ..........
        return p;
}
```

Here we are returning a pointer which points to a local variable. We know that a local variable exists only inside the function. Suppose the variable x is stored at address 2500, the value of p will be 2500 and this value will be returned by the function `func()`. As soon as `func()` terminates, the local variable x will cease to exist.

The address returned by `func()` is assigned to pointer variable `ptr` inside `main()`, so now `ptr` will contain address 2500. When we dereference `ptr`, we are trying to access the value of a variable that no longer exists. So never return a pointer which points to a local variable. Now let us take another program that uses a function returning pointer.

```
/*P2.16 Program to show a function that returns pointer*/
#include<stdio.h>
int *fun(int *p, int n);
main()
{
        int arr[10] = {1,2,3,4,5,6,7,8,9,10},n,*ptr;
        n = 5;
        ptr = fun(arr,n);
        printf("arr = %p,ptr = %p,*ptr = %d\n",arr,ptr,*ptr);
}
int *fun(int *p,int n)
{
        p = p+n;
        return p;
}
```

2.2.9 Passing a 1-D Array to a Function

When an array is passed to a function, the changes made inside the function affect the original array. This is because the function gets access to the original array. The following program shows this-

```
/*P2.17 Program to show that changes to the array made inside the function affect the
original array*/
#include<stdio.h>
void func(int a[]);
main()
{
        int i, arr[5] = {3,6,2,7,1};
        func(arr);
        printf("Inside main() : ");
        for(i=0; i<5; i++)
                printf("%d  ",arr[i]);
        printf("\n");
}
void func(int a[])
{
        int i;
        printf("Inside func() : ");
        for(i=0; i<5; i++)
        {
                a[i] = a[i] + 2;
                printf("%d  ",a[i]);
        }
        printf("\n");
}
```

Now let us see what actually happens when an array is passed to a function. There are three ways of declaring the formal parameter, which has to receive the array. We can declare it as an unsized or sized array or we can declare it as a pointer.

```
func(int a[])
{
        .....
}
func(int a[5]);
{
        .....
}
func(int *a);
{
        .....
}
```

In all the three cases the compiler reserves space only for a pointer variable inside the function. In the function call, the array name is passed without any subscript or address operator. Since array name represents the address of first element of array, this address is assigned to the pointer variable in the function. So inside the function we have a pointer that contains the base address of the array. In the above program, the argument a is declared as a pointer variable whose base type is `int`, and it is initialized with the base address of array arr. We have studied that if we have a pointer variable containing the base address of an array, then we can access any array element either by pointer notation or subscript notation. So inside the function, we can access any element of arr by writing `*(a+i)` or `a[i]`. Since we are directly accessing the original array, all the changes made to the array in the called function are reflected in the calling function. The following program will illustrate the point that we have discussed.

```
/*P2.18 When an array is passed to a function, the receiving argument is declared as a
pointer*/
#include<stdio.h>
func(float f[],int *i,char c[5]);
main()
```

```
{
        float f_arr[5] = {1.4,2.5,3.7,4.1,5.9};
        int i_arr[5] = {1,2,3,4,5};
        char c_arr[5] = {'a','b','c','d','e'};
        printf("Inside main(): ");
        printf("Size of f_arr = %u\t",sizeof(f_arr));
        printf("Size of i_arr = %u\t",sizeof(i_arr));
        printf("Size of c_arr = %u\n",sizeof(c_arr));
        func(f_arr,i_arr,c_arr);
}
func(float f[],int *i,char c[5])
{
        printf("Inside func(): ");
        printf("Size of f = %u\t",sizeof(f));
        printf("Size of i = %u\t",sizeof(i));
        printf("Size of c = %u\n",sizeof(c));
}
```

Inside the function `func()`, variables `f`, `i` and `c` are declared as pointers.

2.2.10 Array of Pointers

We can declare an array that contains pointers as its elements. Every element of this array is a pointer variable that can hold address of any variable of appropriate type. The syntax of declaring an array of pointers is similar to that of declaring arrays except that an asterisk is placed before the array name.

```
datatype *arrayname[size];
```

An array of size 10 containing integer pointers can be declared as-

```
int *arrp[10];
```

```
/*P2.19 Array of pointers*/
#include<stdio.h>
main()
{
        int *pa[3];
        int i,a=5,b=10,c=15;
        pa[0] = &a;
        pa[1] = &b;
        pa[2] = &c;
        for(i=0; i<3; i++)
        {
                printf("pa[%d] = %p\t",i,pa[i]);
                printf("*pa[%d] = %d\n",i,*pa[i]);
        }
}
```

Here `pa` is declared as an array of pointers. Every element of this array is a pointer to an integer. `pa[i]` gives the value of the i^{th} element of `pa` which is an address of any `int` variable and `*pa[i]` gives the value of that `int` variable. The array of pointers can also contain addresses of elements of another array.

2.3 Dynamic Memory Allocation

The memory allocation that we have done till now was static memory allocation. The memory that could be used by the program was fixed i.e. we could not increase or decrease the size of memory during the execution of a program. In many applications it is not possible to predict how much memory would be needed by the program at run time. Suppose we declare an array of integers-

```
int emp_no[200];
```

In an array, it is must to specify the size of array while declaring, so the size of this array will be fixed during runtime. Now two types of problems may occur. The first type of problem is that the number of values to be stored is less than the size of array and hence there is wastage of memory. For example if we have to store only

50 values in the above array, then space for 150 values(600 bytes) is wasted. The second type of problem is that, our program fails if we want to store more values than the size of array, for example when there is need to store 205 values in the above array.

To overcome these problems we should be able to allocate memory at run time. The process of allocating memory at the time of execution is called dynamic memory allocation. The allocation and release of this memory space can be done with the help of some built-in-functions whose prototypes are found in and stdlib.h header file. These functions allocate memory from a memory area called heap and release this memory whenever not required, so that it can be used again for some other purpose.

Pointers play an important role in dynamic memory allocation because we can access the dynamically allocated memory only through pointers.

2.3.1 malloc( )

Declaration : `void *malloc(size_t size);`

This function is used to allocate memory dynamically. The argument `size` specifies the number of bytes to be allocated. The type `size_t` is defined in stdlib.h as unsigned int. On success, `malloc()` returns a pointer to the first byte of allocated memory. The returned pointer is of type `void`, which can be type cast to appropriate type of pointer. It is generally used as-

```
ptr = (datatype *)malloc(specified size);
```

Here `ptr` is a pointer of type `datatype`, and specified size is the size in bytes required to be reserved in memory. The expression `(datatype *)` is used to typecast the pointer returned by `malloc()`. For example-

```
int *ptr;
ptr = (int *)malloc(12);
```

This allocates 12 contiguous bytes of memory space and the address of first byte is stored in the pointer variable `ptr`. The allocated memory contains garbage value. We can use `sizeof` operator to make the program portable and more readable.

```
ptr = (int *)malloc(5*sizeof(int));
```

This allocates the memory space to hold five integer values.

If there is not sufficient memory available in heap then `malloc()` returns NULL. So we should always check the value returned by `malloc()`.

```
ptr = (float *)malloc(10*sizeof(float));
if(ptr==NULL)
        printf("Sufficient memory not available");
```

Unlike memory allocated for variables and arrays, dynamically allocated memory has no name associated with it. So it can be accessed only through pointers. We have a pointer which points to the first byte of the allocated memory and we can access the subsequent bytes using pointer arithmetic.

```
/*P2.20 Program to understand dynamic allocation of memory*/
#include<stdio.h>
#include<stdlib.h>
main()
{
        int *p,n,i;
        printf("Enter the number of integers to be entered : ");
        scanf("%d",&n);
        p = (int *)malloc(n*sizeof(int));
        if(p==NULL)
        {
                printf("Memory not available\n");
                exit(1);
        }
        for(i=0; i<n; i++)
        {
```

```
            printf("Enter an integer : ");
            scanf("%d",p+i);
    }
    for(i=0; i<n; i++)
            printf("%d\t",*(p+i));
}
```

The function `malloc()` returns a void pointer and we have studied that a void pointer can be assigned to any type of pointer without typecasting. But we have used typecasting because it is a good practice to do so and moreover it also ensures compatibility with C++.

2.3.2 calloc( )

Declaration : `void *calloc(size_t n, size_t size);`

The `calloc()` function is used to allocate multiple blocks of memory. It is similar to `malloc()` function except for two differences. The first one is that it takes two arguments. The first argument specifies the number of blocks and the second one specifies the size of each block. For example-

```
ptr=(int *)calloc(5,sizeof(int));
```

This allocates 5 blocks of memory, each block contains 4 bytes and the starting address is stored in the pointer variable `ptr`, which is of type `int *`. An equivalent `malloc()` call would be-

```
ptr=(int *)malloc(5*sizeof(int));
```

Here we have to do the calculation ourselves by multiplying, but `calloc()` function does the calculation for us.

The other difference between `calloc()` and `malloc()` is that the memory allocated by `malloc()` contains garbage value while the memory allocated by `calloc()` is initialized to zero. But this initialization by `calloc()` is not very reliable, so it is better to explicitly initialize the elements whenever there is a need to do so.

Like `malloc()`, `calloc()` also returns NULL if there is not sufficient memory available in the heap.

2.3.3 realloc( )

Declaration : `void *realloc(void *ptr, size_t newsize)`

We may want to increase or decrease the memory allocated by `malloc()` or `calloc()`. The function `realloc()` is used to change the size of the memory block. It alters the size of the memory block without losing the old data. This is known as reallocation of memory.

This function takes two arguments, first is a pointer to the block of memory that was previously allocated by `malloc()` or `calloc()` and second one is the new size for that block. For example-

```
ptr = (int *)malloc(size);
```

This statement allocates the memory of the specified size and the starting address of this memory block is stored in the pointer variable `ptr`. If we want to change the size of this memory block, then we can use `realloc()` as-

```
ptr = (int *)realloc(ptr,newsize);
```

This statement allocates the memory space of `newsize` bytes, and the starting address of this memory block is stored in the pointer variable `ptr`. The `newsize` may be smaller or larger than the old size. If the `newsize` is larger, then the old data is not lost and the newly allocated bytes are uninitialized. The starting address contained in `ptr` may change if there is not sufficient memory at the old address to store all the bytes consecutively. This function moves the contents of old block into the new block and the data of the old block is not lost. On failure, `realloc()` returns NULL.

```
/*P2.21 Program to understand the use of realloc() function*/
#include<stdio.h>
```